Mathematical Foundations of Deep Learning

Sourangshu Ghosh*

*sourangshug@iisc.ac.in, Department of Civil Engineering, Indian Institute of Science Bangalore

Abstract

Deep learning, as a computational paradigm, fundamentally relies on the synergy of functional approximation, optimization theory, and statistical learning. This work presents an extremely rigorous mathematical framework that formalizes deep learning through the lens of measurable function spaces, risk functionals, and approximation theory. We begin by defining the risk functional as a mapping between measurable function spaces, establishing its structure via Frechet differentiability and variational principles. The hypothesis complexity of neural networks is rigorously analyzed using VC-dimension theory for discrete hypotheses and Rademacher complexity for continuous spaces, providing fundamental insights into generalization and overfitting.

A refined proof of the Universal Approximation Theorem is developed using convolution operators and the Stone-Weierstrass theorem, demonstrating how neural networks approximate arbitrary continuous functions on compact domains with quantifiable error bounds. The depth vs. width trade-off is explored through capacity analysis, bounding the expressive power of networks using Fourier analysis and Sobolev embeddings, with rigorous compactness arguments via the Rellich-Kondrachov theorem.

We extend the theoretical framework to training dynamics, analyzing gradient flow and stationary points, the Hessian structure of optimization landscapes, and the Neural Tangent Kernel (NTK) regime. Generalization bounds are established through PAC-Bayes formalism and spectral regularization, connecting information-theoretic insights to neural network stability. The analysis further extends to advanced architectures, including convolutional and recurrent networks, transformers, generative adversarial networks (GANs), and variational autoencoders, emphasizing their function space properties and representational capabilities.

Finally, reinforcement learning is rigorously examined through deep Q-learning and policy optimization, with applications spanning robotics and autonomous systems. The mathematical depth is reinforced by a comprehensive exploration of optimization techniques, covering stochastic gradient descent (SGD), adaptive moment estimation (Adam), and spectral-based regularization methods. The discussion culminates in a deep investigation of function space embeddings, generalization error bounds, and the fundamental limits of deep learning models.

This work bridges deep learning's theoretical underpinnings with modern advancements, offering a mathematically precise and exhaustive exposition that is indispensable for researchers aiming to rigorously understand and extend the frontiers of deep learning theory.

Contents

1 Mathematical Foundations

Deep learning is a computational paradigm for solving high-dimensional function approximation problems. At its core, it relies on the synergy of:

- **Functional Approximation:** Representing complex, non-linear functions using neural networks. Functional approximation is one of the fundamental concepts in deep learning, and it is integral to how deep learning models, particularly neural networks, solve complex problems. In the context of deep learning, functional approximation refers to the ability of neural networks to represent complex, high-dimensional, and non-linear functions that are often difficult or infeasible to model using traditional mathematical techniques.

- **Optimization Theory:** Solving non-convex optimization problems efficiently. Optimization theory plays a central role in deep learning, as the goal of training deep neural networks is essentially to find the optimal set of parameters (weights and biases) that minimize a predefined objective, often called the loss function. This objective typically measures the difference between the network's predictions and the true values. Optimization techniques guide the training process and determine how well a neural network can learn from data.

- **Statistical Learning Theory:** Understanding generalization behavior on unseen data. Statistical Learning Theory (SLT) provides the mathematical foundation for understanding the behavior of machine learning algorithms, including deep learning models. It offers key insights into how models generalize from training data to unseen data, which is critical for ensuring that deep learning models are not only accurate on the training set but also perform well on new, previously unseen data. SLT helps address fundamental challenges such as overfitting, bias-variance tradeoff, and generalization error.

The problem can be formalized as:

$$\text{Find } f_\theta : X \to Y, \text{ such that } \mathbb{E}_{x,y\sim P}\left[\ell(f_\theta(x), y)\right] \text{ is minimized,} \tag{1}$$

where X is the input space, Y is the output space, P is the data distribution, $\ell(\cdot,\cdot)$ is a loss function, θ are parameters of the neural network. This task involves the composition of several disciplines, each of which is explored in rigorous detail below.

1.1 Problem Definition: Risk Functional as a Mapping Between Spaces

1.1.1 Measurable Function Spaces

A measurable space is a fundamental construct in measure theory, denoted by $(\mathcal{X}, \Sigma)$, where $\mathcal{X}$ is a non-empty set referred to as the underlying set or the sample space, and Σ is a σ-algebra, a specific collection of subsets of $\mathcal{X}$ that encodes the notion of measurability. The σ-algebra $\Sigma \subseteq 2^{\mathcal{X}}$, the power set of $\mathcal{X}$, satisfies three axioms, each ensuring a critical aspect of closure under set operations. First, Σ is closed under complementation, meaning that for any set $A \in \Sigma$, its complement $A^c = \mathcal{X} \setminus A$ is also in Σ. This guarantees the ability to define the "non-occurrence" of measurable events in a mathematically consistent way. Second, Σ is closed under countable unions: for any countable collection $\{A_n\}_{n=1}^{\infty} \subseteq \Sigma$, the union $\bigcup_{n=1}^{\infty} A_n$ is also in Σ, enabling the construction of measurable sets from countably infinite operations. De Morgan's laws then imply closure under countable intersections, as $\bigcap_{n=1}^{\infty} A_n = \left(\bigcup_{n=1}^{\infty} A_n^c\right)^c$, ensuring that the framework accommodates conjunctions of countable collections of events. Finally, the inclusion of the empty set $\emptyset \in \Sigma$ is an axiom that provides a null baseline, ensuring that the σ-algebra is non-empty and can represent the "impossibility" of certain outcomes.

Literature Review: Rao et. al. (2024) [1] investigated approximation theory within Lebesgue measurable function spaces, providing an analysis of operator convergence. They also established a theoretical framework for function approximation in Lebesgue spaces and provided a rigorous study of symmetric properties in function spaces. Mukhopadhyay and Ray (2025) [2] provided a comprehensive introduction to measurable function spaces, with a focus on Lp-spaces and their completeness properties. They also established the fundamental role of Lp-spaces in measure theory and discussed the relationship between continuous function spaces and measurable functions. Szołdra (2024) [3] examined measurable function spaces in quantum mechanics, exploring the role of measurable observables in ergodic theory. They connected functional analysis and measure theory to quantum state evolution and provided a mathematical foundation for quantum machine learning in function spaces. Lee (2025) [10] investigated metric space theory and functional analysis in the context of measurable function spaces in AI models. He formalized how function spaces can model self-referential structures in AI and provided a bridge between measure theory and generative models. Song et. al. (2025) [4] discussed measurable function spaces in the context of urban renewal and performance evaluation. They developed a rigorous evaluation metric using measurable function spaces and explored function space properties in applied data science and urban analytics. Mennaoui et. al. (2025) [5] explored measurable function spaces in the theory of evolution equations, a key concept in functional analysis. They established a rigorous study of measurable operator functions and provided new insights into function spaces and their role in solving differential equations. Pedroza (2024) [6] investigated domain stability in machine learning models using function spaces. He established a formal mathematical relationship between function smoothness and domain adaptation and uses topological and measurable function spaces to analyze stability conditions in learning models. Cerreia-Vioglio and Ok (2024) [7] developed a new integration theory for measurable set-valued functions. They introduced a generalization of integration over Banach-valued functions and established weak compactness properties in measurable function spaces. Averin (2024) [8] applied measurable function spaces to gravitational entropy theory. He provided a rigorous proof of entropy bounds using function space formalism and connected measure theory with relativistic field equations. Potter (2025) [9] investigated measurable function spaces in the context of Fourier analysis and crystallographic structures. He established new results on Fourier transforms of measurable functions and introduced a novel framework for studying function spaces in invariant shift operators.

Measurable spaces are not merely abstract structures but are the backbone of measure theory, probability, and integration. For example, the Borel σ-algebra $\mathcal{B}(\mathbb{R})$ on the real numbers $\mathbb{R}$ is the smallest σ-algebra containing all open intervals (a, b) for $a, b \in \mathbb{R}$. This σ-algebra is pivotal in defining Lebesgue measure, where measurable sets generalize the classical notion of intervals to include sets that are neither open nor closed. Moreover, the construction of a σ-algebra generated by a collection of subsets $\mathcal{C} \subseteq 2^{\mathcal{X}}$, denoted $\sigma(\mathcal{C})$, provides a minimal framework that includes $\mathcal{C}$ and satisfies all σ-algebra properties, enabling the systematic extension of measurability to more complex scenarios. For instance, starting with intervals in $\mathbb{R}$, one can build the Borel σ-algebra, a critical tool in modern analysis.

The structure of a measurable space allows the definition of a measure μ, a function $\mu : \Sigma \to [0, \infty]$ that assigns a non-negative value to each set in Σ, adhering to two key axioms: $\mu(\emptyset) = 0$ and countable additivity, which states that for any disjoint collection $\{A_n\}_{n=1}^{\infty} \subseteq \Sigma$, the measure of their union satisfies $\mu\left(\bigcup_{n=1}^{\infty} A_n\right) = \sum_{n=1}^{\infty} \mu(A_n)$. This property is indispensable in extending intuitive notions of length, area, and volume to arbitrary measurable sets, paving the way for the Lebesgue integral. A function $f : \mathcal{X} \to \mathbb{R}$ is then termed Σ-measurable if for every Borel set $B \in \mathcal{B}(\mathbb{R})$, the preimage $f^{-1}(B)$ belongs to Σ. This definition ensures that the function is compatible with the σ-algebra, a necessity for defining integrals and expectation in probability theory.

In summary, measurable spaces represent a highly versatile and mathematically rigorous framework, underpinning vast areas of analysis and probability. Their theoretical depth lies in their ability to systematically handle infinite operations while maintaining closure, consistency, and flexibility for defining measures, measurable functions, and integrals. As such, the rigorous study of measurable spaces is indispensable for advancing modern mathematical theory, providing a bridge between abstract set theory and applications in real-world phenomena.

Let $(\mathcal{X}, \Sigma_X, \mu_X)$ and $(\mathcal{Y}, \Sigma_Y, \mu_Y)$ be measurable spaces. The true risk functional is defined as:

$$\mathcal{R}(f) = \int_{\mathcal{X}\times\mathcal{Y}} \ell(f(x), y)\, dP(x, y), \tag{2}$$

where:

- f belongs to a hypothesis space $\mathcal{F} \subseteq L^p(\mathcal{X}, \mu_X)$.
- $P(x, y)$ is a Borel probability measure over $\mathcal{X} \times \mathcal{Y}$, satisfying $\int_{\mathcal{X}\times\mathcal{Y}} 1\, dP = 1$.

1.1.2 Risk as a Functional

Literature Review: Wang et. al. (2025) [11] developed a mathematical risk model based on functional variational calculus and introduced a loss functional regularization framework that minimizes adversarial risk in deep learning models. They also proposed a game-theoretic interpretation of functional risk in security settings. Duim and Mesquita (2025) [12] extended the inverse reinforcement learning (IRL) framework by defining risk as a functional over probability spaces and used Bayesian functional priors to model risk-sensitive behavior. They also introduced an iterative regularized risk functional minimization approach. Khayat et. al. (2025) [13] established functional Sobolev norms to quantify risk in adversarial settings and introduced a functional risk decomposition technique using deep neural architectures. They also provided an in-depth theoretical framework for risk estimation in adversarially perturbed networks. Agrawal (2025) [14] developed a variational framework for risk as a loss functional and used adaptive weighting of loss functions to enhance generalization in deep learning. He also provided rigorous convergence analysis of risk functional minimization. Hailemichael and Ayalew (2025) [15] used control barrier function (CBF) theory to develop risk-aware deep learning models and modeled risk as a functional on dynamical systems, optimizing stability and robustness. They also introduced a risk-minimizing constrained optimization formulation. Nguyen et.al. (2025) [16] developed a functional metric learning approach for risk-sensitive deep models and used convex optimization techniques to derive functional risk bounds. They also established semi-supervised loss functions for risk-regularized learning. Luo et. al. (2025) [17] introduced a geometric interpretation of risk functionals in deep learning models and used integral transform techniques to approximate risk in real-world vision systems. They also developed a functional approach to adversarial robustness.

The functional $\mathcal{R} : \mathcal{F} \to \mathbb{R}_+$ is Fréchet-differentiable if:

$$\forall f, g \in \mathcal{F}, \quad \mathcal{R}(f + \epsilon g) = \mathcal{R}(f) + \epsilon \langle \nabla \mathcal{R}(f), g \rangle + o(\epsilon), \tag{3}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product in $L^2(\mathcal{X})$. In the field of risk management and decision theory, the concept of a **risk functional** is a mathematical formalization that captures how risk is quantified for a given outcome or state. A risk functional, denoted as $\mathcal{R}$, acts as a map that takes elements from a given space X (which represents the possible outcomes or states) and returns a real-valued risk measure. This risk measure, $\mathcal{R}(x)$, expresses the degree of risk or the adverse outcome associated with a particular element $x \in X$. The space X may vary depending on the context—this could be a space of random variables, trajectories, or more complex function spaces. The risk functional maps x to $\mathbb{R}$, i.e., $\mathcal{R} : X \to \mathbb{R}$, where each $\mathcal{R}(x)$ reflects the risk associated with the specific realization x. One of the most foundational forms of risk functionals is based on the

expectation of a loss function $L(x)$, where $x \in X$ represents a random variable or state, and $L(x)$ quantifies the loss associated with that state. The risk functional can be expressed as an expected loss, written mathematically as:

$$\mathcal{R}(x) = \mathbb{E}[L(x)] = \int_X L(x)p(x)\,dx \tag{4}$$

where $p(x)$ is the probability density function of the outcome x, and the integration is taken over the entire space X. In this setup, $L(x)$ can be any function that measures the severity or unfavorable nature of the outcome x. In a financial context, $L(x)$ could represent the loss function for a portfolio, and $p(x)$ would be the probability density function of the portfolio's returns. In many cases, a specific form of $L(x)$ is used, such as $L(x) = (x - \mu)^2$, where μ is the target or expected value. This choice results in the risk functional representing the **variance** of the outcome x, expressed as:

$$\mathcal{R}(x) = \int_X (x - \mu)^2 p(x)\,dx \tag{5}$$

This formulation captures the variability or dispersion of outcomes around a mean value, a common risk measure in applications like portfolio optimization or risk management. Additionally, another widely used class of risk functionals arises from **quantile-based risk measures**, such as **Value-at-Risk (VaR)**, which focuses on the extreme tail behavior of the loss distribution. The VaR at a confidence level $\alpha \in [0, 1]$ is defined as the smallest value l such that the probability of $L(x)$ exceeding l is no greater than $1 - \alpha$, i.e.,

$$P(L(x) \leq l) \geq \alpha \tag{6}$$

This defines a threshold beyond which the worst outcomes are expected to occur with probability $1 - \alpha$. Value-at-Risk provides a measure of the worst-case loss under normal circumstances, but it does not provide information about the severity of losses exceeding this threshold. To address this limitation, the **Conditional Value-at-Risk (CVaR)** is introduced, which measures the expected loss given that the loss exceeds the VaR threshold. Mathematically, CVaR at the level α is given by:

$$\mathrm{CVaR}_\alpha(x) = \mathbb{E}[L(x) \mid L(x) \geq \mathrm{VaR}_\alpha(x)] \tag{7}$$

This conditional expectation provides a more detailed assessment of the potential extreme losses beyond the VaR threshold. The CVaR is a more comprehensive measure, capturing the tail risk and providing valuable information about the magnitude of extreme adverse events. In cases where the space X represents **trajectories** or paths, such as in the context of continuous-time processes or dynamical systems, the risk functional is often formulated in terms of integrals over time. For example, consider $x(t)$ as a trajectory in the function space $\mathcal{C}([0, T], \mathbb{R}^n)$, the space of continuous functions on the interval $[0, T]$. The risk functional in this case might quantify the total deviation of the trajectory from a reference or target trajectory over time. A typical example could be the total squared deviation, written as:

$$\mathcal{R}(x) = \int_0^T \|x(t) - \bar{x}(t)\|^2 dt \tag{8}$$

where $\bar{x}(t)$ represents a reference trajectory and $\|\cdot\|$ is a norm, such as the Euclidean norm. This risk functional quantifies the total deviation (or **energy**) of the trajectory from the target path over the entire time interval, and is used in various applications such as control theory and optimal trajectory planning. A common choice for the norm $\|x(t)\|$ might be $\|x(t)\|^2 = \sum_{i=1}^n x_i^2(t)$, where $x_i(t)$ are the components of the trajectory $x(t)$ in $\mathbb{R}^n$. In some cases, the space X of possible outcomes may not be a finite-dimensional vector space, but instead a **Banach space** or a **Hilbert space**, particularly when x represents a more complex object such as a function or a trajectory. For example, the space $\mathcal{C}([0, T], \mathbb{R}^n)$ is a Banach space, and the risk functional may involve the evaluation of integrals over this function space. In such settings, the risk functional can take the form:

$$\mathcal{R}(x) = \int_0^T \|x(t)\|_p^p dt \tag{9}$$

where $\| \cdot \|_p$ is the p-norm, and $p \geq 1$. For $p = 2$, this risk functional represents the total **energy** of the trajectory, but other norms can be used to emphasize different types of risks. For instance, the L^∞-norm would focus on the maximum deviation of the trajectory from the target path. The concept of **convexity** plays a significant role in the theory of risk functionals. Convexity ensures that the risk associated with a **convex combination** of two states x_1 and x_2 is less than or equal to the weighted average of the risks of the individual states. Mathematically, for $\lambda \in [0, 1]$, convexity demands that:

$$\mathcal{R}(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda \mathcal{R}(x_1) + (1 - \lambda)\mathcal{R}(x_2) \tag{10}$$

This property reflects the diversification effect in risk management, where mixing several states or outcomes generally leads to a reduction in overall risk. Convex risk functionals are particularly important in portfolio theory, where they allow for risk minimization through diversification. For example, if $\mathcal{R}(x)$ represents the variance of a portfolio's returns, then the convexity property ensures that combining different assets will result in a portfolio with lower overall risk than the risk of any individual asset. **Monotonicity** is another important property for risk functionals, ensuring that the risk increases as the outcome becomes more adverse. If x_1 is worse than x_2 according to some partial order, we have:

$$\mathcal{R}(x_1) \geq \mathcal{R}(x_2) \tag{11}$$

Monotonicity ensures that the risk functional behaves in a way that aligns with intuitive notions of risk: worse outcomes are associated with higher risk. In financial contexts, this is reflected in the fact that **losses** increase the associated risk measure. Finally, in some applications, the risk functional is derived from perturbation analysis to study how small changes in parameters affect the overall risk. Consider $x(\epsilon)$ as a perturbed trajectory, where ϵ is a small parameter, and the Fréchet derivative of the risk functional with respect to ϵ is given by:

$$\left. \frac{d}{d\epsilon} \mathcal{R}(x(\epsilon)) \right|_{\epsilon=0} \tag{12}$$

This derivative quantifies the sensitivity of the risk to perturbations in the system and is crucial in the analysis of stability and robustness. Such analyses are essential in areas like **stochastic control** and **optimization**, where it is important to understand how small changes in the model's parameters can influence the risk profile.

Thus, the risk functional is a powerful tool for quantifying and managing uncertainty, and its formulation can be adapted to various settings, from random variables and stochastic processes to continuous trajectories and dynamic systems. The risk functional provides a rigorous mathematical framework for assessing and minimizing risk in complex systems, and its flexibility makes it applicable across a wide range of domains.

1.2 Approximation Spaces for Neural Networks

The neural network hypothesis space $\mathcal{F}_\theta$ is parameterized as:

$$\mathcal{F}_\theta = \{f_\theta : \mathcal{X} \to \mathbb{R} \mid f_\theta(x) = \sum_{j=1}^{n} c_j \sigma(a_j \cdot x + b_j),\ \theta = (c, a, b)\}. \tag{13}$$

To analyze its capacity, we rely on:

- **VC-dimension theory** for discrete hypotheses.

- **Rademacher complexity** for continuous spaces:

$$\mathcal{R}_N(\mathcal{F}) = \mathbb{E}_\epsilon \left[\sup_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \epsilon_i f(x_i) \right], \tag{14}$$

  where ϵ_i are i.i.d. Rademacher random variables.

1.2.1 VC-dimension theory for discrete hypotheses

The VC-dimension (Vapnik-Chervonenkis dimension) is a fundamental concept in statistical learning theory that quantifies the capacity of a hypothesis class to fit a range of labelings of a set of data points. The VC-dimension is particularly useful in understanding the generalization ability of a classifier. The theory is important in machine learning, especially when assessing overfitting and the risk of model complexity.

Literature Review: There are several articles that explore the VC-dimension theory for discrete hypotheses very rigorously. N. Bousquet and S. Thomassé (2015) [18] explored in their paper the VC-dimension in the context of graph theory, connecting it to structural properties such as the Erdős-Pósa property. Yıldız and Alpaydin (2009) [19] in their article computed the VC-dimension for decision tree hypothesis spaces, considering both discrete and continuous features. Zhang et. al. (2012) [20] introduced a discretized VC-dimension to bridge real-valued and discrete hypothesis spaces, offering new theoretical tools for complexity analysis. Riondato and Zdonik (2011) [21] adapted VC-dimension theory to database systems, analyzing SQL query selectivity using a theoretical lens. Riggle and Sonderegger (2010) [22] investigated the VC-dimension in linguistic models, focusing on grammar hypothesis spaces. Anderson (2023) [23] provided a comprehensive review of VC-dimension in fuzzy systems, particularly in logic frameworks involving discrete structures. Fox et. al. (2021) [24] proved key conjectures for systems with bounded VC-dimension, offering insights into combinatorial implications. Johnson (2021) [25] discusses binary representations and VC-dimensions, with implications for discrete hypothesis modeling. Janzing (2018) [26] in his paper focuses on hypothesis classes with low VC-dimension in causal inference frameworks. Hüllermeier and Tehrani (2012) [27] in their paper explored the theoretical VC-dimension of Choquet integrals, applied to discrete machine learning models. The book titled “Foundations of Machine Learning” [28] by Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar offers a very good foundational discussion on VC-dimension in the context of statistical learning. Another book titled “Learning Theory: An Approximation Theory Viewpoint” by Felipe Cucker and Ding-Xuan Zhou [29] discusses the role of VC-dimension in approximation theory. Yet another book titled “Understanding Machine Learning: From Theory to Algorithms” by Shai Shalev-Shwartz and Shai Ben-David [30] contains detailed chapters on hypothesis spaces and VC-dimension.

For discrete hypotheses, the VC-dimension theory applies to a class of hypotheses that map a set of input points to binary output labels (typically 0 or 1). The VC-dimension for a hypothesis class refers to the largest set of data points that can be shattered by that class, where ”shattering” means that the hypothesis class can realize all possible labelings of these points.

We shall now discuss the **Formal Mathematical Framework**. Let X be a finite or infinite set called the **instance space**, which represents the input space. Consider a hypothesis class H, where each hypothesis $h \in H$ is a function $h : X \to \{0, 1\}$. The function h classifies each element of X into one of two classes: 0 or 1. Given a subset $S = \{x_1, x_2, \dots, x_k\} \subseteq X$, we say that H **shatters** S if for every possible labeling $\vec{y} = (y_1, y_2, \dots, y_k) \in \{0, 1\}^k$, there exists some $h \in H$ such that for all $i \in \{1, 2, \dots, k\}$:

$$h(x_i) = y_i \tag{15}$$

In other words, a hypothesis class H shatters S if it can produce every possible binary labeling on the set S. The **VC-dimension** $\mathrm{VC}(H)$ is defined as the size of the largest set S that can be shattered by H:

$$\mathrm{VC}(H) = \sup\{k \mid \exists S \subseteq X, |S| = k, S \text{ is shattered by } H\} \tag{16}$$

If no set of points can be shattered, then the VC-dimension is 0. Some Properties of the VC-Dimension are

1. **Shattering Implies Non-empty Hypothesis Class**: If a set S is shattered by H, then H is non-empty. This follows directly from the fact that for each labeling $\vec{y} \in \{0,1\}^k$, there exists some $h \in H$ that produces the corresponding labeling. Therefore, H must contain at least one hypothesis.

2. **Upper Bound on Shattering**: Given a hypothesis class H, if there exists a set $S \subseteq X$ of size k such that H can shatter S, then any set $S' \subseteq X$ of size greater than k cannot be shattered. This gives us the crucial result that:

$$\mathrm{VC}(H) \geq k \quad \text{if} \quad H \text{ can shatter a set of size } k \tag{17}$$

3. **Implication for Generalization** A central result in the theory of **statistical learning** is the connection between VC-dimension and the **generalization error**. Specifically, the **VC-dimension** bounds the ability of a hypothesis class to generalize to unseen data. The higher the VC-dimension, the more complex the hypothesis class, and the more likely it is to **overfit** the training data, leading to poor generalization.

We shall now discuss the VC-Dimension and Generalization Bounds (VC Theorem). The **VC-dimension theorem** (often referred to as **Hoeffding's bound** or the **generalization bound**) provides a probabilistic guarantee on the relationship between the training error and the true error. Specifically, it gives an upper bound on the probability that the generalization error exceeds the empirical error (training error) by more than ϵ.

Let $\mathcal{D}$ be the distribution from which the training data is drawn, and let $\hat{err}(h)$ and $err(h)$ represent the **empirical error** and **true error** of a hypothesis $h \in H$, respectively:

$$\hat{err}(h) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{h(x_i) \neq y_i\}} \tag{18}$$

$$err(h) = \mathbb{P}_{(x,y)\sim\mathcal{D}}\left(h(x) \neq y\right) \tag{19}$$

where $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ are i.i.d. (independent and identically distributed) samples from the distribution $\mathcal{D}$. For a hypothesis class H with **VC-dimension** $d = \mathrm{VC}(H)$, with probability at least $1-\delta$, the following holds for all $h \in H$:

$$|\hat{err}(h) - err(h)| \leq \epsilon \tag{20}$$

where ϵ is bounded by:

$$\epsilon \leq \sqrt{\frac{8}{n}\left(d \log\left(\frac{2n}{d}\right) + \log\left(\frac{4}{\delta}\right)\right)} \tag{21}$$

This result shows that the generalization error (the difference between the true and empirical error) is small with high probability, provided the sample size n is large enough and the VC-dimension d is not too large. The sample complexity n required to guarantee that the generalization error is within ϵ with high probability $1-\delta$ is given by:

$$n \geq \frac{C}{\epsilon^2}\left(d \log\left(\frac{1}{\epsilon}\right) + \log\left(\frac{1}{\delta}\right)\right) \tag{22}$$

where C is a constant depending on the distribution. This bound emphasizes the importance of VC-dimension in controlling the complexity of the hypothesis class. A larger VC-dimension requires a larger sample size to avoid overfitting and ensure reliable generalization. Some Detailed Examples are:

1. **Example 1: Linear Classifiers in $\mathbb{R}^2$**: Consider the hypothesis class H consisting of linear classifiers in $\mathbb{R}^2$. These classifiers are hyperplanes in two dimensions, defined by:

   $$h(x) = \text{sign}(w^T x + b) \tag{23}$$

   where $w \in \mathbb{R}^2$ is the weight vector and $b \in \mathbb{R}$ is the bias term. The **VC-dimension** of linear classifiers in $\mathbb{R}^2$ is 3. This can be rigorously shown by noting that for any set of 3 points in $\mathbb{R}^2$, the hypothesis class H can shatter these points. In fact, any possible binary labeling of the 3 points can be achieved by some linear classifier. However, for 4 points in $\mathbb{R}^2$, it is impossible to shatter all possible binary labelings (e.g., the four vertices of a convex quadrilateral), meaning the VC-dimension is 3.

2. **Example 2: Polynomial Classifiers of Degree d**: Consider a polynomial hypothesis class in $\mathbb{R}^n$ of degree d. The hypothesis class H consists of polynomials of the form:

   $$h(x) = \sum_{i_1,i_2,\ldots,i_n} \alpha_{i_1,i_2,\ldots,i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} \tag{24}$$

   where the $\alpha_{i_1,i_2,\ldots,i_n}$ are coefficients and $x = (x_1, x_2, \ldots, x_n)$. The **VC-dimension** of polynomial classifiers of degree d in $\mathbb{R}^n$ grows as $O(n^d)$, implying that the complexity of the hypothesis class increases rapidly with both the degree d and the dimension n of the input space.

Neural networks, depending on their architecture, can have very high VC-dimensions. In particular, the **VC-dimension** of a neural network with L layers, each containing N neurons, is typically $O(N^L)$, indicating that the VC-dimension grows exponentially with both the number of neurons and the number of layers. This result provides insight into the complexity of neural networks and their capacity to overfit data when the training sample size is insufficient.

The **VC-dimension** of a hypothesis class is a powerful tool in statistical learning theory. It quantifies the complexity of the hypothesis class by measuring its capacity to shatter sets of points, and it is directly tied to the model's ability to generalize. The **VC-dimension theorem** provides rigorous bounds on the generalization error and sample complexity, giving us essential insights into the trade-off between model complexity and generalization. The theory extends to more complex hypothesis classes such as linear classifiers, polynomial classifiers, and neural networks, where it serves as a critical tool for controlling overfitting and ensuring reliable performance on unseen data.

1.2.2 Rademacher complexity for continuous spaces

Literature Review: Truong (2022) [31] in his article explored how Rademacher complexity impacts generalization error in deep learning, particularly with IID and Markov datasets. Gnecco and Sanguineti (2008) [32] developed approximation error bounds in Reproducing Kernel Hilbert Spaces (RKHS) and functional approximation settings. Astashkin (2010) [33] discusses applications of Rademacher functions in symmetric function spaces and their mathematical structure. Ying and Campbell (2010) [34] applies Rademacher complexity to kernel-based learning problems and support vector machines. Zhu et.al. (2009) [35] examined Rademacher complexity in cognitive models and neural representation learning. Astashkin et al. (2020) [36] investigated how the Rademacher system behaves in function spaces and its role in functional analysis. Sachs et.al. (2023) [37] introduced a refined approach to Rademacher complexity tailored to specific machine learning algorithms. Ma and Wang (2020) [38] investigated Rademacher complexity bounds in deep residual networks. Bartlett and Mendelson (2002) [39] wrote a foundational paper on complexity measures, providing fundamental theoretical insights into generalization bounds. Dzahini and Wild (2024) [40] in their paper extended Rademacher-based complexity to stochastic optimization methods. McDonald and Shalizi (2011) [41] showed using sequential Rademacher complexities for I.I.D process how to control the generalization error of time series models wherein past values of the outcome are used to predict future values.

Let $(\mathcal{X}, \Sigma, \mathcal{D})$ represent a probability space where $\mathcal{X}$ is a measurable space, Σ is a sigma-algebra, and $\mathcal{D}$ is a probability measure. The function class $\mathcal{F} \subset L^\infty(\mathcal{X}, \mathbb{R})$ satisfies:

$$\sup_{f \in \mathcal{F}} \|f\|_\infty < \infty, \tag{25}$$

where $\|f\|_\infty = \operatorname{ess\,sup}_{x \in \mathcal{X}} |f(x)|$ denotes the essential supremum. For rigor, $\mathcal{F}$ is assumed measurable in the sense that for every $\epsilon > 0$, there exists a countable subset $\mathcal{F}_\epsilon \subseteq \mathcal{F}$ such that:

$$\sup_{f \in \mathcal{F}} \inf_{g \in \mathcal{F}_\epsilon} \|f - g\|_\infty \leq \epsilon. \tag{26}$$

Given $S = \{x_1, x_2, \ldots, x_n\} \sim \mathcal{D}^n$, the empirical measure $\mathbb{P}_n$ is:

$$\mathbb{P}_n(A) = \frac{1}{n} \sum_{i=1}^{n} \nVdash_{\{x_i \in A\}}, \quad \forall A \in \Sigma. \tag{27}$$

The integral under $\mathbb{P}_n$ for $f \in \mathcal{F}$ approximates the population integral under $\mathcal{D}$:

$$\mathbb{P}_n[f] = \frac{1}{n} \sum_{i=1}^{n} f(x_i), \quad \mathcal{D}[f] = \int_{\mathcal{X}} f(x)\, d\mathcal{D}(x). \tag{28}$$

Let $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$ be independent Rademacher random variables:

$$\mathbb{P}(\sigma_i = +1) = \mathbb{P}(\sigma_i = -1) = \frac{1}{2}, \quad i = 1, \ldots, n. \tag{29}$$

These variables are defined on a probability space $(\Omega, \mathcal{A}, \mathbb{P})$ independent of the sample S. The Duality and Symmetrization of Empirical Rademacher Complexity is also very important. The empirical Rademacher complexity of $\mathcal{F}$ with respect to S is:

$$\hat{\mathfrak{R}}_S(\mathcal{F}) = \mathbb{E}_{\boldsymbol{\sigma}} \left[\sup_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \sigma_i f(x_i) \right], \tag{30}$$

where $\mathbb{E}_{\boldsymbol{\sigma}}$ denotes expectation over $\boldsymbol{\sigma}$. The supremum can be interpreted as a functional dual norm in $L^\infty(\mathcal{X}, \mathbb{R})$, where $\mathcal{F}$ is the unit ball. Using the symmetrization technique, the Rademacher complexity relates to the deviation of $\mathbb{P}_n[f]$ from $\mathcal{D}[f]$:

$$\mathbb{E}_S \sup_{f \in \mathcal{F}} |\mathbb{P}_n[f] - \mathcal{D}[f]| \leq 2\mathfrak{R}_n(\mathcal{F}), \tag{31}$$

where:

$$\mathfrak{R}_n(\mathcal{F}) = \mathbb{E}_S \left[\hat{\mathfrak{R}}_S(\mathcal{F}) \right]. \tag{32}$$

This is derived by first symmetrizing the sample and then invoking Jensen's inequality and the independence of $\boldsymbol{\sigma}$. There are some Complexity Bounds that use Covering Numbers and Entropy that need to be discussed. In Metric Entropy, we let $\|\cdot\|_\infty$ be the metric on $\mathcal{F}$. The covering number $N(\epsilon, \mathcal{F}, \|\cdot\|_\infty)$ satisfies:

$$N(\epsilon, \mathcal{F}, \|\cdot\|_\infty) = \inf\{m \in \mathbb{N} : \exists \{f_1, \ldots, f_m\} \subseteq \mathcal{F}, \forall f \in \mathcal{F}, \exists i, \|f - f_i\|_\infty \leq \epsilon\}. \tag{33}$$

Regarding the Dudley's Entropy Integral, For a bounded function class $\mathcal{F}$ (compact under $\|\cdot\|_\infty$):

$$\mathfrak{R}_n(\mathcal{F}) \leq \inf_{\alpha>0} \left(4\alpha + \frac{12}{\sqrt{n}} \int_\alpha^\infty \sqrt{\log N(\epsilon, \mathcal{F}, \|\cdot\|_\infty)}\, d\epsilon \right). \tag{34}$$

There is also a Relation to Talagrand's Concentration Inequality. Talagrand's inequality provides tail bounds for the supremum of empirical processes:

$$\mathbb{P}\left(\sup_{f\in\mathcal{F}} |\mathbb{P}_n[f] - \mathcal{D}[f]| > \epsilon \right) \leq 2\exp\left(-\frac{n\epsilon^2}{2\|f\|_\infty^2} \right), \tag{35}$$

reinforcing the link between $\mathfrak{R}_n(\mathcal{F})$ and generalization. There are some Applications in Continuous Function Classes. One example is the RKHS with Gaussian Kernel. For $\mathcal{F}$ as the unit ball of an RKHS with kernel $k(x, x')$, the covering number satisfies:

$$\log N(\epsilon, \mathcal{F}, \|\cdot\|_\infty) \sim O\left(\frac{1}{\epsilon^2} \right), \tag{36}$$

yielding:

$$\mathfrak{R}_n(\mathcal{F}) \sim O\left(\frac{1}{\sqrt{n}} \right). \tag{37}$$

For $\mathcal{F} \subseteq H^s(\mathbb{R}^d)$, the covering number depends on the smoothness s and dimension d:

$$\mathfrak{R}_n(\mathcal{F}) \sim O\left(\frac{1}{n^{s/d}} \right). \tag{38}$$

Rademacher complexity is deeply embedded in modern empirical process theory. Its intricate relationship with measure-theoretic tools, symmetrization, and concentration inequalities provides a robust theoretical foundation for understanding generalization in high-dimensional spaces.

1.2.3 Sobolev Embeddings

Literature Review: Abderachid and Kenza (2024) [42] in their paper investigated fractional Sobolev spaces defined using Riemann-Liouville derivatives and studies their embedding properties. It establishes new continuous embeddings between these fractional spaces and classical Sobolev spaces, providing applications to PDEs. Giang et.al. (2024) [43] introduced weighted Sobolev spaces and derived new Pólya-Szegö type inequalities. These inequalities play a key role in establishing compact embedding results in function spaces equipped with weight functions. Ruiz and Fragkiadaki (2024) [44] provided a novel approach using Haar functions to revisit fractional Sobolev embedding theorems and demonstrated the algebra properties of fractional Sobolev spaces, which are essential in nonlinear analysis. Bilalov et.al. (2025) [45] analyzed compact Sobolev embeddings in Banach function spaces, extending the classical Poincaré and Friedrichs inequalities to this setting and provided applications to function spaces used in modern PDE theory. Cheng and Shao (2025) [46] developed the weighted Sobolev compact embedding theorem for function spaces with unbounded radial potentials and used this result to prove the existence of ground state solutions for fractional Schrödinger-Poisson equations. Wei and Zhang (2025) [47] established a new embedding theorem tailored to variational problems arising in Schrödinger-Poisson equations and used Hardy-Sobolev embeddings to study the zero-mass case, an important case in quantum mechanics. Zhang and Qi (2025) [48] examined the compactness of Sobolev embeddings in the presence of small perturbations in quasilinear elliptic equations and proved multiple solution existence results using variational methods. Xiao and Yue (2025) [49] established a Sobolev embedding theorem for fractional Laplacian function spaces and applied the embedding results to image processing, particularly edge detection. Pesce and Portaro (2025) [50] studied intrinsic Hölder spaces and their connection to fractional Sobolev embeddings and established new embedding results for function spaces relevant to ultraparabolic operators.

The Sobolev embedding theorem states that:

$$W^{k,p}(\mathcal{X}) \hookrightarrow C^m(\mathcal{X}), \tag{39}$$

if $k - \frac{d}{p} > m$, ensuring $f_\theta \in C^\infty(\mathcal{X})$ for smooth activations σ. For a function $u \in L^p(\Omega)$, its weak derivative $D^\alpha u$ satisfies:

$$\int_\Omega u(x) D^\alpha \phi(x)\, dx = (-1)^{|\alpha|} \int_\Omega v(x)\phi(x)\, dx \quad \forall \phi \in C_c^\infty(\Omega), \tag{40}$$

where $v \in L^p(\Omega)$ is the weak derivative. This definition extends the classical notion of differentiation to functions that may not be pointwise differentiable. The Sobolev norm encapsulates both function values and their derivatives:

$$\|u\|_{W^{k,p}(\Omega)} = \left(\sum_{|\alpha| \leq k} \|D^\alpha u\|_{L^p(\Omega)}^p \right)^{1/p}. \tag{41}$$

Key properties:

- **Semi-norm Dominance**: The $W^{k,p}$-norm is controlled by the seminorm $|u|_{W^{k,p}}$, ensuring sensitivity to high-order derivatives.
- **Poincaré Inequality**: For Ω bounded, $u - u_\Omega$ satisfies:

$$\|u - u_\Omega\|_{L^p} \leq C \|Du\|_{L^p}. \tag{42}$$

Sobolev spaces $W^{k,p}(\Omega)$ embed into $L^q(\Omega)$ or $C^m(\overline{\Omega})$, depending on k, p, q, and n. These embeddings govern the smoothness and integrability of u and its derivatives. There are several Advanced Theorems on Sobolev Embeddings. They are as follows:

1. **Sobolev Embedding Theorem:** Let $\Omega \subset \mathbb{R}^n$ be a bounded domain with Lipschitz boundary. Then:
    - If $k > n/p$, $W^{k,p}(\Omega) \hookrightarrow C^{m,\alpha}(\overline{\Omega})$ with $m = \lfloor k - n/p \rfloor$ and $\alpha = k - n/p - m$.
    - If $k = n/p$, $W^{k,p}(\Omega) \hookrightarrow L^q(\Omega)$ for $q < \infty$.
    - If $k < n/p$, $W^{k,p}(\Omega) \hookrightarrow L^q(\Omega)$ where $\frac{1}{q} = \frac{1}{p} - \frac{k}{n}$.
2. **Rellich-Kondrachov Compactness Theorem:** The embedding $W^{k,p}(\Omega) \hookrightarrow L^q(\Omega)$ is compact for $q < \frac{np}{n-kp}$. Compactness follows from:
    (a) **Equicontinuity**: $W^{k,p}$-boundedness ensures uniform control over oscillations.
    (b) **Rellich's Selection Principle**: Strong convergence follows from uniform estimates and tightness.

The Proof of Sobolev Embedding starts with the Scaling Analysis.Define $u_\lambda(x) = u(\lambda x)$. Then:

$$\|u_\lambda\|_{L^p(\Omega)} = \lambda^{-n/p} \|u\|_{L^p(\lambda^{-1}\Omega)}. \tag{43}$$

For derivatives:

$$\|D^\alpha u_\lambda\|_{L^p(\Omega)} = \lambda^{|\alpha| - n/p} \|D^\alpha u\|_{L^p(\lambda^{-1}\Omega)}. \tag{44}$$

The scaling relation $\lambda^{k-n/p}$ aligns with the Sobolev embedding condition $k > n/p$. Sobolev norms in $\mathbb{R}^n$ are equivalent to decay rates of Fourier coefficients:

$$\|u\|_{W^{k,p}} \sim \left(\int_{\mathbb{R}^n} |\xi|^{2k} |\hat{u}(\xi)|^2 \, d\xi \right)^{1/2}. \tag{45}$$

For $k > n/p$, Fourier decay implies uniform bounds, ensuring $u \in C^{m,\alpha}$. Interpolation spaces bridge L^p and $W^{k,p}$, providing finer embeddings. Duality: Sobolev embeddings are equivalent to boundedness of adjoint operators in L^q. For $-\Delta u = f$, $u \in W^{2,p}(\Omega)$ ensures $u \in C^{0,\alpha}(\overline{\Omega})$ if $p > n/2$. Sobolev spaces govern variational problems in geometry, e.g., minimal surfaces and harmonic maps. On Ω with fractal boundaries, trace theorems refine Sobolev embeddings.

1.2.4 Rellich-Kondrachov Compactness Theorem

The **Rellich-Kondrachov Compactness Theorem** is one of the most fundamental and deep results in the theory of Sobolev spaces, particularly in the study of functional analysis and the theory of partial differential equations. The theorem asserts the compactness of certain Sobolev embeddings under appropriate conditions on the domain and the function spaces involved. This result is of immense significance in mathematical analysis because it provides a rigorous justification for the fact that bounded sequences in Sobolev spaces, under certain conditions, have strongly convergent subsequences in lower-order normed spaces. In essence, the theorem states that while weak convergence in Sobolev spaces is relatively straightforward due to the Banach-Alaoglu theorem, strong convergence is not always guaranteed. However, under the assumptions of the Rellich-Kondrachov theorem, strong convergence in $L^q(\Omega)$ can indeed be obtained from boundedness in $W^{1,p}(\Omega)$. The compactness property ensured by this theorem is much stronger than mere boundedness or weak convergence and plays a crucial role in proving the existence of solutions to variational problems by ensuring that minimizing sequences possess convergent subsequences in an appropriate function space. The theorem can also be viewed as a generalization of the classical **Arzelà–Ascoli theorem**, extending compactness results to function spaces that involve derivatives.

Literature Review: Lassoued (2026) [51] examined function spaces on the torus and their lack of compactness, highlighting cases where the classical Rellich-Kondrachov result fails. He extended compact embedding results to function spaces with periodic structures. He also discussed trace theorems and regular function spaces in this new context. Chen et.al. (2024) [52] extended the Rellich-Kondrachov theorem to Hörmander vector fields, a class of differential operators that appear in hypoelliptic PDEs. They established a degenerate compact embedding theorem, generalizing previous results in the field. They also provided applications to geometric inequalities, highlighting the role of compact embeddings in PDE theory. Adams and Fournier (2003) [53] in their book provided a complete proof of the Rellich-Kondrachov theorem, along with a discussion of compact embeddings. They also covered function space theory, embedding theorems, and applications in PDEs. Brezis (2010) [54] wrote a highly recommended resource for understanding Sobolev spaces and their compactness properties. The book included applications to variational methods and weak solutions of PDEs. Evans (2022) [55] in his classic PDE textbook includes a discussion of compact Sobolev embeddings, their implications for weak convergence, and applications in variational methods. Maz'ya (2011) [56] provided a detailed treatment of Sobolev space theory, including compact embedding theorems in various settings.

To rigorously state the theorem, we consider a bounded open domain $\Omega \subset \mathbb{R}^n$ with a **Lipschitz boundary**. For $1 \leq p < n$, the theorem asserts that the embedding

$$W^{1,p}(\Omega) \hookrightarrow L^q(\Omega) \tag{46}$$

is **compact** whenever $q \leq \frac{np}{n-p}$. More precisely, this means that if $\{u_k\} \subset W^{1,p}(\Omega)$ is a bounded sequence in the Sobolev norm, i.e., there exists a constant $C > 0$ such that

$$\|u_k\|_{W^{1,p}(\Omega)} = \|u_k\|_{L^p(\Omega)} + \|\nabla u_k\|_{L^p(\Omega)} \leq C, \tag{47}$$

then there exists a **subsequence** $\{u_{k_j}\}$ and a function $u \in L^q(\Omega)$ such that

$$u_{k_j} \to u \quad \text{strongly in } L^q(\Omega), \tag{48}$$

which means that

$$\|u_{k_j} - u\|_{L^q(\Omega)} \to 0 \quad \text{as } j \to \infty. \tag{49}$$

To establish this rigorously, we first recall the fact that **bounded sequences in $W^{1,p}(\Omega)$ are weakly precompact**. Since $W^{1,p}(\Omega)$ is a **reflexive Banach space** for $1 < p < \infty$, the Banach-Alaoglu theorem ensures that any bounded sequence $\{u_k\}$ in $W^{1,p}(\Omega)$ has a subsequence (still denoted by $\{u_k\}$) and a function $u \in W^{1,p}(\Omega)$ such that

$$u_k \rightharpoonup u \quad \text{in } W^{1,p}(\Omega). \tag{50}$$

This means that for all test functions $\varphi \in W^{1,p'}(\Omega)$, where p' is the Hölder conjugate of p satisfying $\frac{1}{p} + \frac{1}{p'} = 1$, we have

$$\int_\Omega u_k \varphi \, dx \to \int_\Omega u\varphi \, dx, \qquad \int_\Omega \nabla u_k \cdot \nabla \varphi \, dx \to \int_\Omega \nabla u \cdot \nabla \varphi \, dx. \tag{51}$$

However, weak convergence alone does not imply compactness. To obtain strong convergence in $L^q(\Omega)$, we need additional arguments. This is accomplished using the **Fréchet-Kolmogorov compactness criterion**, which states that a bounded subset of $L^q(\Omega)$ is compact if and only if it is **tight** and **uniformly equicontinuous**. More formally, compactness follows if

1. The sequence $u_k(x)$ does not oscillate excessively at small scales.
2. The sequence $u_k(x)$ does not escape to infinity in a way that prevents strong convergence.

To quantify this, we invoke the **Sobolev-Poincaré inequality**, which states that for $p < n$, there exists a constant C such that

$$\|u - u_\Omega\|_{L^q(\Omega)} \leq C\|\nabla u\|_{L^p(\Omega)}, \quad u_\Omega = \frac{1}{|\Omega|} \int_\Omega u(x) \, dx. \tag{52}$$

Applying this inequality to $u_k - u$, we obtain

$$\|u_k - u\|_{L^q(\Omega)} \leq C\|\nabla(u_k - u)\|_{L^p(\Omega)}. \tag{53}$$

Since ∇u_k is weakly convergent in $L^p(\Omega)$, we have

$$\|\nabla u_k - \nabla u\|_{L^p(\Omega)} \to 0. \tag{54}$$

Thus,

$$\|u_k - u\|_{L^q(\Omega)} \to 0, \tag{55}$$

which establishes the **strong convergence in $L^q(\Omega)$**, completing the proof. The key insight is that compactness arises because the gradients of u_k provide control over the oscillations of u_k, ensuring that the sequence cannot oscillate indefinitely without converging in norm. The crucial role of Sobolev embeddings is to guarantee that even though $W^{1,p}(\Omega)$ does not embed compactly into itself, it does embed compactly into $L^q(\Omega)$ for $q < \frac{np}{n-p}$. This embedding ensures that weak convergence in $W^{1,p}(\Omega)$ implies strong convergence in $L^q(\Omega)$, proving the theorem.

2 Universal Approximation Theorem: Refined Proof

The Universal Approximation Theorem (UAT) is a fundamental result in neural network theory, stating that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to any desired degree of accuracy, provided that an appropriate activation function is used. This theorem has significant implications in machine learning, function approximation, and deep learning architectures.

Literature Review: Hornik et. al. (1989) [57] in their seminal paper rigorously proved that multilayer feedforward neural networks with a single hidden layer and a sigmoid activation function can approximate any continuous function on a compact set. It extends prior results and lays the foundation for the modern understanding of UAT. Cybenko (1989) [58] provided one of the first rigorous proofs of the UAT using the sigmoid function as the activation function. They demonstrated that a single hidden layer network can approximate any continuous function arbitrarily well. Barron (1993) [59] extended UAT by quantifying the approximation error and analyzing the rate of convergence. This work is crucial for understanding the practical efficiency of neural networks. Pinkus (1999) [60] provided a comprehensive survey of UAT from the perspective of approximation theory and also discussed conditions for approximation with different activation functions and the theoretical limits of neural networks. Lu et.al. (2017) [61] investigated how the width of neural networks affects their approximation capability, challenging the notion that deeper networks are always better. They also provided insights into trade-offs between depth and width. Hanin and Sellke (2018) [62] extended UAT to ReLU activation functions, showing that deep ReLU networks achieve universal approximation while maintaining minimal width constraints. Garcıa-Cervera et. al. (2024) [63] extended the universal approximation theorem to set-valued functions and its applications to Deep Operator Networks (DeepONets), which are useful in control theory and PDE modeling. Majee et.al. (2024) [64] explored the universal approximation properties of deep neural networks for solving inverse problems using Markov Chain Monte Carlo (MCMC) techniques. Toscano et. al. (2024) [65] introduced Kurkova-Kolmogorov-Arnold Networks (KKANs), an extension of UAT incorporating Kolmogorov's superposition theorem for improved approximation capabilities. Son (2025) [66] established a new framework for operator learning based on the UAT, providing a theoretical foundation for backpropagation-free deep networks.

2.1 Approximation Using Convolution Operators

Let us begin by considering the convolution operator and its role in approximating functions in the context of the Universal Approximation Theorem (UAT). Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is a continuous and bounded function. The convolution of f with a kernel function $\phi : \mathbb{R}^n \to \mathbb{R}$, denoted as $f * \phi$, is defined as

$$(f * \phi)(x) = \int_{\mathbb{R}^n} f(y)\phi(x - y)\, dy. \tag{56}$$

The kernel $\phi(x)$ is typically chosen to be smooth, compactly supported, and normalized such that

$$\int_{\mathbb{R}^n} \phi(x)\, dx = 1. \tag{57}$$

To approximate f locally, we introduce a scaling parameter $\epsilon > 0$ and define the scaled kernel $\phi_\epsilon(x)$ as

$$\phi_\epsilon(x) = \epsilon^{-n}\phi\left(\frac{x}{\epsilon}\right). \tag{58}$$

The factor ϵ^{-n} ensures that $\phi_\epsilon(x)$ remains a probability density function, satisfying

$$\int_{\mathbb{R}^n} \phi_\epsilon(x)\, dx = \int_{\mathbb{R}^n} \phi(x)\, dx = 1. \tag{59}$$

The convolution of f with the scaled kernel ϕ_ϵ is given by

$$(f * \phi_\epsilon)(x) = \int_{\mathbb{R}^n} f(y)\phi_\epsilon(x - y)\, dy. \tag{60}$$

Performing the change of variables $z = \frac{x-y}{\epsilon}$, we have $y = x - \epsilon z$ and $dy = \epsilon^n dz$. Substituting into the integral, we obtain

$$(f * \phi_\epsilon)(x) = \int_{\mathbb{R}^n} f(x - \epsilon z)\phi(z)\, dz. \tag{61}$$

This representation shows that $(f * \phi_\epsilon)(x)$ is a smoothed version of $f(x)$, where the smoothing is controlled by the parameter ϵ. As $\epsilon \to 0$, the kernel $\phi_\epsilon(x)$ becomes increasingly concentrated around x, and we recover $f(x)$ in the limit:

$$\lim_{\epsilon \to 0}(f * \phi_\epsilon)(x) = f(x), \tag{62}$$

assuming f is continuous. This result can be rigorously proven using properties of the kernel ϕ, such as its smoothness and compact support, and the dominated convergence theorem, which ensures that the integral converges uniformly to $f(x)$. Now, let us consider the role of convolution operators in the approximation of f by neural networks. A single-layer feedforward neural network is expressed as

$$\hat{f}(x) = \sum_{i=1}^{M} c_i \sigma(w_i^T x + b_i), \tag{63}$$

where $c_i \in \mathbb{R}$ are coefficients, $w_i \in \mathbb{R}^n$ are weight vectors, $b_i \in \mathbb{R}$ are biases, and $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function. The activation function $\sigma(w_i^T x + b_i)$ can be interpreted as a localized response function, analogous to the kernel $\phi(x - y)$ in convolution. By drawing an analogy between the two, we can write the neural network approximation as

$$\hat{f}(x) \approx \sum_{i=1}^{M} f(x_i)\phi_\epsilon(x - x_i)\Delta x \tag{64}$$

where $\phi_\epsilon(x)$ is interpreted as a parameterized kernel defined by w_i, b_i, and σ, and Δx represents a discretization step. The approximation error $\|f - \hat{f}\|_\infty$ can be decomposed into two components:

$$\|f - \hat{f}\|_\infty \leq \|f - f * \phi_\epsilon\|_\infty + \|f * \phi_\epsilon - \hat{f}\|_\infty. \tag{65}$$

The term $\|f - f * \phi_\epsilon\|_\infty$ represents the error introduced by smoothing f with the kernel ϕ_ϵ, and it can be made arbitrarily small by choosing ϵ sufficiently small, provided f is regular enough (e.g., Lipschitz continuous). The term $\|f * \phi_\epsilon - \hat{f}\|_\infty$ quantifies the error due to discretization, which vanishes as the number of neurons $M \to \infty$. To rigorously analyze the convergence of $\hat{f}(x)$ to $f(x)$, we rely on the density of neural network approximators in function spaces. The Universal Approximation Theorem states that, for any continuous function f on a compact domain $\Omega \subset \mathbb{R}^n$ and any $\epsilon > 0$, there exists a neural network $\hat{f}$ with finitely many neurons such that

$$\sup_{x \in \Omega} |f(x) - \hat{f}(x)| < \epsilon. \tag{66}$$

This result hinges on the ability of the activation function σ to generate a rich set of basis functions. For example, if $\sigma(x) = \max(0, x)$ (ReLU), the network approximates $f(x)$ by piecewise linear functions. If $\sigma(x) = \frac{1}{1+e^{-x}}$ (sigmoid), the network generates smooth approximations that resemble logistic regression.

In this refined proof of the UAT, convolution operators provide a unifying framework for understanding the smoothing, localization, and discretization processes that underlie neural network approximations. The interplay between $\phi_\epsilon(x)$, $f * \phi_\epsilon(x)$, and $\hat{f}(x)$ reveals the profound mathematical structure that connects classical approximation theory with modern machine learning. This connection not only enhances our theoretical understanding of neural networks but also guides the design of architectures and algorithms for practical applications.

2.1.1 Stone-Weierstrass Application

Literature Review: Rudin (1976) [67] introduced the Weierstrass approximation theorem and proves its generalization, the Stone-Weierstrass theorem. He also discussed the algebraic structure of function spaces and how the theorem ensures the uniform approximation of continuous functions by polynomials. He also presented examples and exercises related to compactness, uniform convergence, and Banach algebra structures. Stein and Shakarchi (2005) [68] extended the Stone-Weierstrass theorem into measure theory and functional analysis. He also proved the theorem in the context of Lebesgue integration. He also discussed how it applies to Hilbert spaces and orthogonal polynomials. He also connected the theorem to Fourier analysis and spectral decomposition. Conway (2019) [69] explored the Stone-Weierstrass theorem in the setting of Banach algebras and C-algebras*. He also extended the theorem to non-commutative function algebras and discussed the operator-theoretic implications of the theorem in Hilbert spaces. He also analyzed the theorem's application to spectral theory. Dieudonné (1981) [70] traced the historical development of functional analysis, including the origins of the Stone-Weierstrass theorem and discussed contributions by Karl Weierstrass and Marshall Stone. He also explored how the theorem influenced topological vector spaces and operator theory and also included perspectives on the axiomatic development of function approximation. Folland (1999) [71] discussed the Stone-Weierstrass theorem in depth with applications to probability theory and ergodic theory and used the theorem to establish the density of algebraic functions in measure spaces He also connected the Stone-Weierstrass theorem to functional approximation in Lp spaces. He also explored the interplay between the Stone-Weierstrass theorem and the Hahn-Banach theorem. Sugiura (2024) [72] extended the Stone-Weierstrass theorem to the study of reservoir computing in machine learning and proved that certain neural networks can approximate functions uniformly under the assumptions of the theorem. He bridges classical functional approximation with modern AI and deep learning. Liu et al. (2024) [73] investigated the Stone-Weierstrass theorem in normed module settings and used category theory to generalize function approximation results. He also extended the theorem beyond real-valued functions to structured mathematical objects. Martinez-Barreto (2025) [74] provided a modern formulation of the theorem with rigorous proof and reviewed applications in operator algebras and topology. He also discussed open problems related to function approximation. Chang and Wei (2024) [75] used the Stone-Weierstrass theorem to derive new operator inequalities and applied the theorem to functional analysis in quantum mechanics. Caballer et al. (2024) [76] investigated cases where the Stone-Weierstrass theorem fails and provided counterexamples and refined conditions for uniform approximation. Chen (2024) [77] extended the Stone-Weierstrass theorem to generalized function spaces and introduced a new class of uniform topological algebras. Rafiei and Akbarzadeh-T (2024) [78] used the Stone-Weierstrass theorem to analyze function approximation in fuzzy logic systems and explored the applications in control systems and AI.

The **Stone-Weierstrass Theorem** serves as a cornerstone in functional analysis, bridging the algebraic structure of continuous functions with approximation theory. This theorem, when applied to the **Universal Approximation Theorem (UAT)**, provides a rigorous foundation for asserting that neural networks can approximate any continuous function defined on a compact set. To understand this connection in its most scientifically and mathematically rigorous form, we must carefully analyze the algebra of continuous functions on a compact Hausdorff space and the role of neural networks in approximating these functions, ensuring that all mathematical nuances are explored with extreme precision. Let X be a compact Hausdorff space, and let $C(X)$ represent the space of continuous real-valued functions on X. The **supremum norm** $\|f\|_\infty$ for a function $f \in C(X)$ is defined as:

$$\|f\|_\infty = \sup_{x \in X} |f(x)| \tag{67}$$

This supremum norm is critical in defining the proximity between continuous functions, as we seek to approximate any function $f \in C(X)$ by a function g from a subalgebra $A \subset C(X)$. The **Stone-**

Weierstrass theorem guarantees that if the subalgebra A satisfies two essential properties—(1) it contains the constant functions, and (2) it separates points—then the closure of A in the supremum norm will be the entire space $C(X)$. To formalize this, we define the **point separation property** as follows: for every pair of distinct points $x_1, x_2 \in X$, there exists a function $h \in A$ such that $h(x_1) \neq h(x_2)$. This condition ensures that functions from A are sufficiently "rich" to distinguish between different points in X. Mathematically, this is expressed as:

$$\exists h \in A \text{ such that } h(x_1) \neq h(x_2) \quad \forall x_1, x_2 \in X, x_1 \neq x_2 \tag{68}$$

Given these two properties, the Stone-Weierstrass theorem asserts that for any continuous function $f \in C(X)$ and any $\epsilon > 0$, there exists an element $g \in A$ such that:

$$\|f - g\|_\infty < \epsilon \tag{69}$$

This result ensures that any continuous function on a compact Hausdorff space can be approximated arbitrarily closely by functions from a sufficiently rich subalgebra. In the context of the **Universal Approximation Theorem (UAT)**, we seek to apply the Stone-Weierstrass theorem to the approximation capabilities of neural networks. Let $K \subseteq \mathbb{R}^n$ be a compact subset, and let $f \in C(K)$ be a continuous function defined on this set. A feedforward neural network with a non-linear activation function σ has the form:

$$\hat{f}_\theta(x) = \sum_{i=1}^{N} w_i \sigma(\langle \mathbf{w}_i, x \rangle + b_i) \tag{70}$$

where $\langle \mathbf{w}_i, x \rangle$ represents the inner product between the weight vector $\mathbf{w}_i$ and the input x, and b_i represents the bias term. The activation function σ is typically non-linear (such as the sigmoid or ReLU function), and the parameters $\theta = \{w_i, b_i\}_{i=1}^N$ are the weights and biases of the network. The function $\hat{f}_\theta(x)$ is a weighted sum of the non-linear activations applied to the affine transformations of x.

We now explore the connection between neural networks and the Stone-Weierstrass theorem. A critical observation is that the set of functions defined by a neural network with non-linear activation is a subalgebra of $C(K)$ provided the activation function σ is sufficiently rich in its non-linearity. This non-linearity ensures that the network can separate points in K, meaning that for any two distinct points $x_1, x_2 \in K$, there exists a network function $\hat{f}_\theta$ that takes distinct values at these points. This satisfies the point separation condition required by the Stone-Weierstrass theorem. To formalize this, consider two distinct points $x_1, x_2 \in K$. Since σ is non-linear, the function $\hat{f}_\theta(x)$ with appropriately chosen weights and biases will satisfy:

$$\hat{f}_\theta(x_1) \neq \hat{f}_\theta(x_2) \tag{71}$$

Thus, the algebra of neural network functions satisfies the point separation property. By applying the Stone-Weierstrass theorem, we conclude that this algebra is dense in $C(K)$, meaning that for any continuous function $f \in C(K)$ and any $\epsilon > 0$, there exists a neural network function $\hat{f}_\theta$ such that:

$$\|f(x) - \hat{f}_\theta(x)\|_\infty < \epsilon \quad \forall x \in K \tag{72}$$

This rigorous result shows that neural networks with a non-linear activation function can approximate any continuous function on a compact set arbitrarily closely in the supremum norm, thereby proving the Universal Approximation Theorem. To further explore this, consider the error term:

$$\|f(x) - \hat{f}_\theta(x)\|_\infty \tag{73}$$

For a given function f and a compact set K, this error term can be made arbitrarily small by increasing the number of neurons in the hidden layer of the neural network. This increases the capacity of the network, effectively enlarging the subalgebra of functions generated by the network, thereby improving the approximation. As the number of neurons increases, the network's ability to approximate any function from $C(K)$ becomes increasingly precise, which aligns with the conclusion of the Stone-Weierstrass theorem that the network functions form a dense subalgebra in $C(K)$. Thus, the Universal Approximation Theorem, derived through the Stone-Weierstrass theorem, rigorously proves that neural networks can approximate any continuous function on a compact set to any desired degree of accuracy. The combination of the non-linearity of the activation function and the architecture of the neural network guarantees that the network can generate a dense subalgebra of continuous functions, ultimately allowing it to approximate any function from $C(K)$. This result not only formalizes the approximation power of neural networks but also provides a deep theoretical foundation for understanding their capabilities as universal approximators.

2.2 Depth vs. Width: Capacity Analysis

2.2.1 Bounding the Expressive Power

The Kolmogorov-Arnold Superposition Theorem is a foundational result in the mathematical analysis of multivariate continuous functions and their decompositions, providing a framework that underpins the expressive power of neural networks. It asserts that any continuous multivariate function can be expressed as a finite composition of continuous univariate functions and addition. It was first conjectured by Andrey Kolmogorov in 1956 and later rigorously proved by Vladimir Arnold in 1957. Formally, the theorem guarantees that any continuous multivariate function $f : [0, 1]^n \to \mathbb{R}$ can be represented as a finite composition of continuous univariate functions Φ_q and ψ_{pq}. Specifically, for $f(x_1, x_2, \ldots, x_n)$, there exist functions $\Phi_q : \mathbb{R} \to \mathbb{R}$ and $\psi_{pq} : \mathbb{R} \to \mathbb{R}$, such that

$$f(x_1, x_2, \ldots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \psi_{pq}(x_p) \right), \tag{74}$$

where the functions $\psi_{pq}(x_p)$ encode the univariate projections of the input variables x_p, and the outer functions Φ_q aggregate these projections into the final output. This decomposition highlights a fundamental property of multivariate continuous functions: their expressiveness can be captured through hierarchical compositions of simpler, univariate components.

Literature Review: There are some Classical References on the Kolmogorov-Arnold Superposition Theorem (KST). Kolmogorov (1957) [79] in his Foundational Paper on KST established that any continuous function of several variables can be represented as a superposition of continuous functions of a single variable and addition. This was groundbreaking because it provided a universal function decomposition method, independent of inner-product spaces. He proved that there exist functions ϕ_q and ψ_q such that any function $f(x_1, x_2, \ldots, x_n)$ can be expressed as:

$$f(x_1, ..., x_n) = \sum_{q=1}^{2n+1} \phi_q \left(\sum_{p=1}^{n} \psi_{qp}(x_p) \right) \tag{75}$$

where the ψ_{qp} are univariate functions. Kolmogorov provided a mathematical basis for approximation theory and neural networks, influencing modern machine learning architectures. Arnold (1963) [80] refined Kolmogorov's theorem by proving that one can restrict the superposition to functions of at most two variables instead of one. Arnold's formulation led to the **Kolmogorov-Arnold representation**:

$$f(x_1, ..., x_n) = \sum_{q=1}^{2n+1} \phi_q \left(x_q + \sum_{p=1}^{n} \psi_{qp}(x_p) \right) \tag{76}$$

making the theorem more suitable for practical computations. Arnold strengthened the expressivity of neural networks, inspiring alternative function representations in high-dimensional settings.

Lorentz (2008) [81] in his book discusses the significance of KST in approximation theory and constructive mathematics. He provided error estimates for approximating multivariate functions using Kolmogorov-type decompositions. He showed how KST fits within Bernstein approximation theory. He helped frame KST in the context of function approximation, bridging it to computational applications. Pinkus (1999) [60] analyzed the role of **KST in multilayer perceptrons (MLPs)**, showing how it influences function expressibility in neural networks. He demonstrated that feedforward neural networks can approximate arbitrary functions using Kolmogorov superposition. He also provided bounds on network depth and width required for universal approximation. He played a crucial role in understanding the theoretical power of deep learning. There are several modern Contributions in KST (2024–2025). Guilhoto and Perdikaris (2024) [82] explored how KST can be reformulated using deep learning architectures. They proposed Kolmogorov-Arnold Networks (KANs), a new type of neural network inspired by KST. They showed that KANs outperform traditional feedforward networks in function approximation tasks. They also provided empirical evidence of KAN efficiency in real-world datasets. They also introduced a new paradigm in machine learning, making function decomposition more interpretable. Alhafiz, M. R. et al. (2025) [83] applied KST-based networks to turbulence modeling in fluid mechanics. They demonstrated how KANs improve predictive accuracy for Navier-Stokes turbulence models. They showed a reduction in computational complexity compared to classical turbulence models. They also developed a data-driven turbulence modeling framework leveraging KST. They advanced machine learning applications in computational fluid dynamics (CFD). Lorencin, I. et al. (2024) [84] used KST-inspired neural networks for predicting propulsion system parameters in ships. They implemented KANs to model hybrid ship propulsion (Combined Diesel-Electric and Gas - CODLAG) and demonstrated a highly accurate prediction model for propulsion efficiency. They also provided a new benchmark dataset for ship propulsion research. They extended KST applications to naval engineering & autonomous systems.

Paper	**Main Contribution**	**Impact**
Kolmogorov (1957)	Original KST theorem	Laid foundation for function decomposition
Arnold (1963)	Refinement using 2-variable functions	Made KST more practical for computation
Lorentz (2008)	KST in approximation theory	Linked KST to function approximation erro
Pinkus (1999)	KST in neural networks	Theoretical basis for deep learning
Perdikaris (2024)	Deep learning reinterpretation	Proposed Kolmogorov-Arnold Networks
Alhafiz (2025)	KST-based turbulence modeling	Improved CFD simulations
Lorencin (2024)	KST in naval propulsion	Optimized ship energy efficiency

In the context of neural networks, this result establishes the theoretical universality of function approximation. A neural network with a single hidden layer approximates a function $f(x_1, x_2, \ldots, x_n)$ by representing it as

$$f(x_1, x_2, \ldots, x_n) \approx \sum_{i=1}^{W} a_i \sigma \left(\sum_{j=1}^{n} w_{ij} x_j + b_i \right), \tag{77}$$

where W is the width of the hidden layer, σ is a nonlinear activation function, w_{ij} are weights, b_i are biases, and a_i are output weights. The expressive power of such shallow networks depends critically on the width W, as the universal approximation theorem ensures that $W \to \infty$ suffices to approximate any continuous function arbitrarily well. However, for a fixed approximation error $\epsilon > 0$, the required width grows exponentially with the input dimension n, satisfying a lower bound of

$$W \geq C \cdot \epsilon^{-n}, \tag{78}$$

where C depends on the function's Lipschitz constant. This exponential dependence, sometimes called the "curse of dimensionality," underscores the inefficiency of shallow architectures in capturing high-dimensional dependencies.

The advantage of depth becomes apparent when we consider deep neural networks, which utilize hierarchical representations. A deep network with D layers and width W per layer constructs a function as a composition of layer-wise transformations:

$$h^{(k)} = \sigma\left(W^{(k)} h^{(k-1)} + b^{(k)}\right), \quad h^{(0)} = x, \tag{79}$$

where $h^{(k)}$ denotes the output of the k-th layer, $W^{(k)}$ is the weight matrix, $b^{(k)}$ is the bias vector, and σ is the nonlinear activation. The final output of the network is then given by

$$f(x) \approx h^{(D)} = \sigma\left(W^{(D)} h^{(D-1)} + b^{(D)}\right). \tag{80}$$

The depth D of the network allows it to approximate hierarchical compositions of functions. For example, if a target function $f(x)$ has a compositional structure

$$f(x) = g_1 \circ g_2 \circ \cdots \circ g_D(x), \tag{81}$$

where each g_i is a simple function, the depth D directly corresponds to the number of nested transformations. This compositional hierarchy enables deep networks to approximate functions efficiently, achieving a reduction in the required parameter count. The approximation error ϵ for a deep network decreases polynomially with D, satisfying

$$\epsilon \leq O\left(\frac{1}{D^2}\right), \tag{82}$$

which is exponentially more efficient than the error scaling for shallow networks. In light of the Kolmogorov-Arnold theorem, the decomposition

$$f(x_1, x_2, \ldots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \psi_{pq}(x_p)\right) \tag{83}$$

demonstrates how deep networks align naturally with the structure of multivariate functions. The inner functions ψ_{pq} capture local dependencies, while the outer functions Φ_q aggregate these into a global representation. This layered decomposition mirrors the depth-based structure of neural networks, where each layer learns a specific aspect of the function's complexity. Finally, the parameter count in a deep network with D layers and width W per layer is given by

$$P \leq O(D \cdot W^2), \tag{84}$$

whereas a shallow network requires

$$P \geq O(W^n) \tag{85}$$

parameters for the same approximation accuracy. This exponential difference in parameter count illustrates the superior efficiency of deep architectures, particularly for high-dimensional functions. By leveraging the hierarchical decomposition inherent in the Kolmogorov-Arnold theorem, deep networks achieve expressive power that scales favorably with both dimension and complexity.

2.2.2 Fourier Analysis of Expressivity

Literature Review: Juárez-Osorio et. al. (2024) [215] applied Fourier analysis to design quantum convolutional neural networks (QCNNs) for time series forecasting. The Fourier series decomposition helps analyze and optimize expressivity in quantum architectures, making QCNNs better at capturing periodic and non-periodic structures in data. Umeano and Kyriienko (2024) [216] introduced Fourier-based quantum feature maps that transform classical data into quantum states with enhanced expressivity. The Fourier transform plays a central role in mapping high-dimensional data efficiently while maintaining interpretability. Liu et. al. (2024) [217] extended Graph Convolutional Networks (GCNs) by integrating Fourier analysis and spectral wavelets to improve graph expressivity. It bridges the gap between frequency-domain analysis and graph embeddings, making GCNs more effective for complex data structures. Vlasic (2024) [218] presented a Fourier series-inspired feature mapping technique to encode classical data into quantum circuits. It demonstrates how Fourier coefficients can enhance the representational capacity of quantum models, leading to better compression and generalization. Kim et. al. (2024) [219] introduced Neural Fourier Modelling (NFM), a novel approach to representing time-series data compactly while preserving its expressivity. It outperforms traditional models like Short-Time Fourier Transform (STFT) in retaining long-term dependencies. Xie et. al. (2024) [220] explored how Fourier basis functions can be used to enhance the expressivity of tensor networks while maintaining computational efficiency. It establishes trade-offs between expressivity and model complexity in machine learning architectures. Liu et. al. (2024) [221] integrated spectral modulation and Fourier transforms into implicit neural representations for text-to-image synthesis. Fourier analysis improves global coherence while preserving local expressivity in generative models. Zhang (2024) [222] demonstrated how Fourier and Lock-in spectrum techniques can represent long-term variations in mechanical signals. The Fourier-based decomposition allows for more expressive representations of mechanical failures and degradation. Hamed and Lachiri (2024) [223] applied Fourier transformations to speech synthesis models, improving their ability to transfer expressive content from text to speech. Fourier series allows capturing prosody, rhythm, and tone variations effectively. Lehmann et. al. (2024) [224] integrated Fourier-based deep learning models for seismic activity prediction. It explores the expressivity of Fourier Neural Operators (FNOs) in capturing wave propagations in different geological environments.

The Fourier analysis of expressivity in neural networks seeks to rigorously quantify how neural architectures, characterized by their depth and width, can approximate functions through the decomposition of those functions into their Fourier spectra. Consider a square-integrable function $f : \mathbb{R}^d \to \mathbb{R}$, for which the Fourier transform is defined as

$$\hat{f}(\boldsymbol{\xi}) = \int_{\mathbb{R}^d} f(\mathbf{x}) e^{-i2\pi\boldsymbol{\xi}\cdot\mathbf{x}}\, d\mathbf{x} \tag{86}$$

where $\boldsymbol{\xi} \in \mathbb{R}^d$ represents the frequency. The inverse Fourier transform reconstructs the function as

$$f(\mathbf{x}) = \int_{\mathbb{R}^d} \hat{f}(\boldsymbol{\xi}) e^{i2\pi\boldsymbol{\xi}\cdot\mathbf{x}}\, d\boldsymbol{\xi} \tag{87}$$

The magnitude $|\hat{f}(\boldsymbol{\xi})|$ reflects the energy contribution of the frequency $\boldsymbol{\xi}$ to f. Neural networks approximate f by capturing its Fourier spectrum, but the architecture fundamentally governs how efficiently this approximation can be achieved, especially in the presence of high-frequency components.

For shallow networks with one hidden layer and a finite number of neurons, the universal approximation theorem establishes that

$$f(\mathbf{x}) \approx \sum_{i=1}^{n} a_i \phi(\mathbf{w}_i \cdot \mathbf{x} + b_i) \tag{88}$$

where ϕ is the activation function, $\mathbf{w}_i \in \mathbb{R}^d$ are weights, $b_i \in \mathbb{R}$ are biases, and $a_i \in \mathbb{R}$ are coefficients. The Fourier transform of this representation can be expressed as

$$\hat{f}(\boldsymbol{\xi}) \approx \sum_{i=1}^{n} a_i \hat{\phi}(\boldsymbol{\xi}) e^{-i2\pi\boldsymbol{\xi}\cdot\mathbf{b}_i} \tag{89}$$

where $\hat{\phi}(\boldsymbol{\xi})$ denotes the Fourier transform of the activation function. For smooth activation functions like sigmoid or tanh, $\hat{\phi}(\boldsymbol{\xi})$ decays exponentially as $\|\boldsymbol{\xi}\| \to \infty$, limiting the network's ability to approximate functions with high-frequency content unless the width n is exceedingly large. Specifically, the Fourier coefficients decay as

$$|\hat{f}(\boldsymbol{\xi})| \sim e^{-\beta\|\boldsymbol{\xi}\|} \tag{90}$$

where $\beta > 0$ depends on the smoothness of ϕ. This restriction implies that shallow networks are biased toward low-frequency functions unless their width scales exponentially with the input dimension d. Deep networks, on the other hand, leverage their hierarchical structure to overcome these limitations. A deep network with L layers recursively composes functions, producing an output of the form

$$f(\mathbf{x}) = \phi_L(\mathbf{W}^{(L)}\phi_{L-1}(\mathbf{W}^{(L-1)} \cdots \phi_1(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \cdots) + \mathbf{b}^{(L)}) \tag{91}$$

where ϕ_l is the activation function at layer l, $\mathbf{W}^{(l)}$ are weight matrices, and $\mathbf{b}^{(l)}$ are bias vectors. The Fourier transform of this composition can be analyzed iteratively. If $\mathbf{h}^{(l)} = \phi_l(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$ represents the output of the l-th layer, then

$$\widehat{\mathbf{h}^{(l)}}(\boldsymbol{\xi}) = \hat{\phi}_l(\boldsymbol{\xi}) * \widehat{\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}}(\boldsymbol{\xi}) \tag{92}$$

where $*$ denotes convolution and $\hat{\phi}_l$ is the Fourier transform of the activation function. The recursive application of this convolution amplifies high-frequency components, enabling deep networks to approximate functions whose Fourier spectra exhibit polynomial decay. Specifically, the Fourier coefficients of a deep network decay as

$$|\hat{f}(\boldsymbol{\xi})| \sim \|\boldsymbol{\xi}\|^{-\alpha L} \tag{93}$$

where α depends on the activation function. This is in stark contrast to the exponential decay observed in shallow networks.

The activation function plays a pivotal role in shaping the Fourier spectrum of neural networks. For example, the rectified linear unit (ReLU) $\phi(x) = \max(0, x)$ introduces significant high-frequency components into the network. The Fourier transform of the ReLU activation is given by

$$\hat{\phi}(\xi) = \frac{1}{2\pi i \xi} \tag{94}$$

which decays more slowly than the Fourier transforms of smooth activations. Consequently, ReLU-based networks are particularly effective at approximating functions with oscillatory behavior. To illustrate, consider the function

$$f(\mathbf{x}) = \sin(2\pi\boldsymbol{\xi} \cdot \mathbf{x}) \tag{95}$$

A shallow network requires an exponentially large number of neurons to approximate f when $\|\boldsymbol{\xi}\|$ is large, but a deep network can achieve the same approximation with polynomially fewer parameters by leveraging its hierarchical structure. The expressivity of deep networks can be further quantified by considering their ability to approximate bandlimited functions, i.e., functions f whose Fourier spectra are supported on $\|\boldsymbol{\xi}\| \leq \omega_{\max}$. For a shallow network with width n, the required number of neurons scales as

$$n \sim (\omega_{\max})^d \tag{96}$$

where d is the input dimension. In contrast, for a deep network with depth L, the width scales as

$$n \sim (\omega_{\max})^{d/L} \tag{97}$$

reflecting the exponential efficiency of depth in distributing the approximation of frequency components across layers. For example, if $f(\mathbf{x}) = \cos(2\pi\boldsymbol{\xi} \cdot \mathbf{x})$ with $\|\boldsymbol{\xi}\| = \omega_{\max}$, a deep network requires significantly fewer parameters than a shallow network to approximate f to the same accuracy.

In summary, the Fourier analysis of expressivity rigorously demonstrates the superiority of deep networks over shallow ones in approximating complex functions. Depth introduces a hierarchical compositional structure that enables the efficient representation of high-frequency components, while width provides a rich basis for approximating the function's Fourier spectrum. Together, these properties explain the remarkable capacity of deep neural networks to approximate functions with intricate spectral structures, offering a mathematically rigorous foundation for understanding their expressivity.

3 Training Dynamics and NTK Linearization

Literature Review: Trevisan et. al. [85] investigated how knowledge distillation can be analyzed using the Neural Tangent Kernel (NTK) framework and demonstrated that under certain conditions, the training dynamics of a student model in knowledge distillation closely follow NTK linearization. They explored how NTK affects generalization and feature transfer in the distillation process. They provided theoretical insight into why knowledge distillation improves performance in deep networks. Bonfanti et. al. (2024) [86] studied how NTK behaves in the nonlinear regime, particularly in Physics-Informed Neural Networks (PINNs). They showed that when PINNs operate outside the NTK regime, their performance degrades due to high sensitivity to initialization and weight updates. They established conditions under which NTK linearization is insufficient for PINNs, emphasizing the need for nonlinear adaptations. They provided practical guidelines for designing PINNs that maintain stable training dynamics. Jacot et. al. (2018) [87] introduced the Neural Tangent Kernel (NTK) as a fundamental framework for analyzing infinite-width neural networks. They proved that as width approaches infinity, neural networks evolve as linear models governed by the NTK. They derived generalization bounds for infinitely wide networks and connected training dynamics to kernel methods. They established NTK as a core tool in deep learning theory, leading to further developments in training dynamics research. Lee et. al. (2019) [88] extended NTK theory to arbitrarily deep networks, showing that even deep architectures behave as linear models under gradient descent and proved that training dynamics remain stable regardless of network depth when width is sufficiently large. They explored practical implications for initializing and optimizing deep networks. They strengthened NTK theory by confirming its validity beyond shallow networks. Yang and Hu (2022) [89] challenged the conventional NTK assumption that feature learning is negligible in infinite-width networks and showed that certain activation functions can induce nontrivial feature learning even in infinite-width regimes. They suggested that feature learning can be integrated into NTK theory, opening new directions in kernel-based deep learning research. Xiang et. al. (2023) [90] investigated how finite-width effects impact training dynamics under NTK assumptions and showed that finite-width networks deviate from NTK predictions due to higher-order corrections in weight updates. They derived corrections to NTK theory for practical networks, improving its predictive power for real-world architectures. They refined NTK approximations, making them more applicable to modern deep-learning models. Lee et. al. (2019) [91] extended NTK linearization to deep convolutional networks, analyzing their training dynamics under infinite width and showed how locality and weight sharing in CNNs impact NTK behavior. They also demonstrated practical consequences for CNN training in real-world applications. They bridged NTK theory and convolutional architectures, providing new theoretical tools for CNN analysis.

3.1 Gradient Flow and Stationary Points

The dynamics of gradient flow in neural network training are fundamentally governed by the continuous evolution of parameters $\theta(t)$ under the influence of the negative gradient of the loss function, expressed as

$$\frac{d\theta(t)}{dt} = -\nabla_\theta \mathcal{L}(\theta(t)). \tag{98}$$

The loss function, typically of the form

$$\mathcal{L}(\theta) = \frac{1}{2n} \sum_{i=1}^{n} \|f(x_i;\theta) - y_i\|^2, \tag{99}$$

measures the discrepancy between the network's predicted outputs $f(x_i;\theta)$ and the true labels y_i. At stationary points of the flow, the condition

$$\nabla_\theta \mathcal{L}(\theta^*) = 0 \tag{100}$$

holds, indicating that the gradient vanishes. To classify these stationary points, the Hessian matrix $H = \nabla_\theta^2 \mathcal{L}(\theta)$ is examined. For eigenvalues $\{\lambda_i\}$ of H, the nature of the stationary point is determined: $\lambda_i > 0$ for all i corresponds to a local minimum, $\lambda_i < 0$ for all i to a local maximum, and mixed signs indicate a saddle point.Under gradient flow $\frac{d\theta(t)}{dt} = -\nabla_\theta \mathcal{L}(\theta(t))$, the trajectory converges to critical points:

$$\lim_{t\to\infty} \|\nabla_\theta \mathcal{L}(\theta(t))\| = 0. \tag{101}$$

The gradient flow also governs the temporal evolution of the network's predictions $f(x;\theta(t))$. A Taylor series expansion of $f(x;\theta)$ about an initial parameter θ_0 gives:

$$f(x;\theta) = f(x;\theta_0) + J_f(x;\theta_0)(\theta-\theta_0) + \frac{1}{2}(\theta-\theta_0)^\top H_f(x;\theta_0)(\theta-\theta_0) + \mathcal{O}(\|\theta-\theta_0\|^3), \tag{102}$$

where $J_f(x;\theta_0) = \nabla_\theta f(x;\theta_0)$ is the Jacobian and $H_f(x;\theta_0)$ is the Hessian of $f(x;\theta)$ with respect to θ. In the NTK (neural tangent kernel) regime, higher-order terms are negligible due to the large parameterization of the network, and the linear approximation suffices:

$$f(x;\theta) \approx f(x;\theta_0) + J_f(x;\theta_0)(\theta-\theta_0). \tag{103}$$

Under gradient flow, the time derivative of the network's predictions is given by:

$$\frac{df(x;\theta(t))}{dt} = J_f(x;\theta(t))\frac{d\theta(t)}{dt}. \tag{104}$$

Substituting the parameter dynamics $\frac{d\theta(t)}{dt} = -\nabla_\theta \mathcal{L}(\theta(t)) = -\sum_{i=1}^n (f(x_i;\theta(t)) - y_i) J_f(x_i;\theta(t))$, this becomes:

$$\frac{df(x;\theta(t))}{dt} = -\sum_{i=1}^{n} J_f(x;\theta(t)) J_f(x_i;\theta(t))^\top (f(x_i;\theta(t)) - y_i). \tag{105}$$

Defining the NTK as $\mathcal{K}(x,x';\theta) = J_f(x;\theta)J_f(x';\theta)^\top$, and assuming constancy of the NTK during training ($\mathcal{K}(x,x';\theta) \approx \mathcal{K}_0(x,x')$), the evolution equation simplifies to:

$$\frac{df(x;\theta(t))}{dt} = -\sum_{i=1}^{n} \mathcal{K}_0(x,x_i)(f(x_i;\theta(t)) - y_i). \tag{106}$$

Rewriting in matrix form, let $\mathbf{f}(t) = [f(x_1;\theta(t)),\ldots,f(x_n;\theta(t))]^\top$ and $\mathbf{y} = [y_1,\ldots,y_n]^\top$. The NTK matrix $\mathcal{K}_0 \in \mathbb{R}^{n\times n}$ evaluated at initialization defines the system:

$$\frac{d\mathbf{f}(t)}{dt} = -\mathcal{K}_0(\mathbf{f}(t) - \mathbf{y}). \tag{107}$$

The solution to this linear system is:

$$\mathbf{f}(t) = e^{-\mathcal{K}_0 t}\mathbf{f}(0) + (\mathbf{I} - e^{-\mathcal{K}_0 t})\mathbf{y}. \tag{108}$$

As $t \to \infty$, the predictions converge to the labels: $\mathbf{f}(t) \to \mathbf{y}$, implying zero training error. The eigenvalues of $\mathcal{K}_0$ determine the rates of convergence. Diagonalizing $\mathcal{K}_0$ as $\mathcal{K}_0 = Q\Lambda Q^\top$, where Q is orthogonal and $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$, the dynamics in the eigenbasis are:

$$\frac{d\tilde{\mathbf{f}}(t)}{dt} = -\Lambda(\tilde{\mathbf{f}}(t) - \tilde{\mathbf{y}}), \tag{109}$$

with $\tilde{\mathbf{f}}(t) = Q^\top \mathbf{f}(t)$ and $\tilde{\mathbf{y}} = Q^\top \mathbf{y}$. Solving, we obtain:

$$\tilde{\mathbf{f}}(t) = e^{-\Lambda t}\tilde{\mathbf{f}}(0) + (\mathbf{I} - e^{-\Lambda t})\tilde{\mathbf{y}}. \tag{110}$$

Each mode decays exponentially with a rate proportional to the eigenvalue λ_i. Modes with larger λ_i converge faster, while smaller eigenvalues slow convergence.

The NTK framework thus rigorously explains the linearization of training dynamics in overparameterized neural networks. This linear behavior ensures that the optimization trajectory remains within a convex region of the parameter space, leading to both convergence and generalization. By leveraging the constancy of the NTK, the complexity of nonlinear neural networks is reduced to an analytically tractable framework that aligns closely with empirical observations.

3.1.1 Hessian Structure

The Hessian matrix, $H(\theta) = \nabla_\theta^2 \mathcal{L}(\theta)$, serves as a critical construct in the mathematical framework of optimization, capturing the second-order partial derivatives of the loss function $\mathcal{L}(\theta)$ with respect to the parameter vector $\theta \in \mathbb{R}^d$. Each element $H_{ij} = \frac{\partial^2 \mathcal{L}(\theta)}{\partial\theta_i \partial\theta_j}$ reflects the curvature of the loss surface along the (i, j)-direction. The symmetry of $H(\theta)$, guaranteed by the Schwarz theorem under the assumption of continuous second partial derivatives, implies $H_{ij} = H_{ji}$. This property ensures that the eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_d$ of $H(\theta)$ are real and the eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_d$ are orthogonal, satisfying the eigenvalue equation

$$H(\theta)\mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \text{for all } i. \tag{111}$$

The behavior of the loss function around a specific parameter value θ_0 can be rigorously analyzed using a second-order Taylor expansion. This expansion is given by:

$$\mathcal{L}(\theta) = \mathcal{L}(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta \mathcal{L}(\theta_0) + \frac{1}{2}(\theta - \theta_0)^\top H(\theta_0)(\theta - \theta_0) + \mathcal{O}(\|\theta - \theta_0\|^3). \tag{112}$$

Here, the term $(\theta - \theta_0)^\top \nabla_\theta \mathcal{L}(\theta_0)$ represents the linear variation of the loss, while the quadratic term $\frac{1}{2}(\theta - \theta_0)^\top H(\theta_0)(\theta - \theta_0)$ describes the curvature effects. The eigenvalues of $H(\theta_0)$ dictate the nature of the critical point θ_0. Specifically, if all $\lambda_i > 0$, θ_0 is a local minimum; if all $\lambda_i < 0$, it is a local maximum; and if the eigenvalues have mixed signs, θ_0 is a saddle point. The leading-order approximation to the change in the loss function, $\Delta\mathcal{L} \approx \frac{1}{2}\delta\theta^\top H(\theta_0)\delta\theta$, highlights the dependence on the eigenstructure of $H(\theta_0)$. In the context of gradient descent, parameter updates follow the iterative scheme:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}(\theta^{(t)}), \tag{113}$$

where η is the learning rate. Substituting the Taylor expansion of $\nabla_\theta \mathcal{L}(\theta^{(t)})$ around θ_0 gives:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \left[\nabla_\theta \mathcal{L}(\theta_0) + H(\theta_0)(\theta^{(t)} - \theta_0)\right]. \tag{114}$$

To analyze this update rigorously, we project $\theta^{(t)} - \theta_0$ onto the eigenbasis of $H(\theta_0)$, expressing it as:

$$\theta^{(t)} - \theta_0 = \sum_{i=1}^{d} c_i^{(t)} \mathbf{v}_i, \tag{115}$$

where $c_i^{(t)} = \mathbf{v}_i^\top(\theta^{(t)} - \theta_0)$. Substituting this expansion into the gradient descent update rule yields:

$$c_i^{(t+1)} = c_i^{(t)} - \eta \left[\mathbf{v}_i^\top \nabla_\theta \mathcal{L}(\theta_0) + \lambda_i c_i^{(t)}\right]. \tag{116}$$

The convergence of this iterative scheme is governed by the condition $|1-\eta\lambda_i| < 1$, which constrains the learning rate η relative to the spectrum of $H(\theta_0)$. For eigenvalues λ_i with large magnitudes, excessively large learning rates η can cause oscillatory or divergent updates.

In the Neural Tangent Kernel (NTK) regime, the evolution of a neural network during training can be approximated by a linearization of the network output around the initialization. Let $f_\theta(x)$ denote the output of the network for input x. Linearizing $f_\theta(x)$ around θ_0 gives:

$$f_\theta(x) \approx f_{\theta_0}(x) + \nabla_\theta f_{\theta_0}(x)^\top(\theta - \theta_0). \tag{117}$$

The NTK, defined as:

$$K(x, x') = \nabla_\theta f_{\theta_0}(x)^\top \nabla_\theta f_{\theta_0}(x'), \tag{118}$$

remains approximately constant during training for sufficiently wide networks. The training dynamics of the parameters are described by:

$$\frac{d\theta}{dt} = -\nabla_\theta \mathcal{L}(\theta), \tag{119}$$

which, under the NTK approximation, becomes:

$$\frac{d\theta}{dt} = -K\nabla_\theta \mathcal{L}(\theta), \tag{120}$$

where K is the NTK matrix evaluated at initialization. The evolution of the loss function is governed by the eigenvalues of K, which control the rate of convergence in different directions.

The spectral properties of the Hessian play a pivotal role in the generalization properties of neural networks. Empirical studies reveal that the eigenvalue spectrum of $H(\theta)$ often exhibits a "bulk-and-spike" structure, with a dense bulk of eigenvalues near zero and a few large outliers. The bulk corresponds to flat directions in the loss landscape, which contribute to the robustness and generalization of the model, while the spikes represent sharp directions associated with overfitting. This spectral structure can be analyzed using random matrix theory, where the density of eigenvalues $\rho(\lambda)$ is modeled by distributions such as the Marchenko-Pastur law:

$$\rho(\lambda) = \frac{1}{2\pi\lambda q}\sqrt{(\lambda_+ - \lambda)(\lambda - \lambda_-)}, \tag{121}$$

where $\lambda_\pm = (1 \pm \sqrt{q})^2$ are the spectral bounds and $q = \frac{d}{n}$ is the ratio of the number of parameters to the number of data points. This rigorous analysis links the Hessian structure to both the optimization dynamics and the generalization performance of neural networks, providing a comprehensive mathematical understanding of the training process. The Hessian $H(\theta)$ satisfies:

$$H(\theta) = \nabla_\theta^2 \mathcal{L}(\theta) = \mathbb{E}_{(x,y)}\left[\nabla_\theta f_\theta(x) \nabla_\theta f_\theta(x)^\top\right]. \tag{122}$$

For overparameterized networks, $H(\theta)$ is nearly degenerate, implying the existence of flat minima.

3.1.2 NTK Linearization

The dynamics of neural networks under gradient flow can be comprehensively described by beginning with the parameterized representation of the network $f_\theta(x)$, where $\theta \in \mathbb{R}^p$ denotes the set of trainable parameters, $x \in \mathbb{R}^d$ is the input, and $f_\theta(x) \in \mathbb{R}^m$ represents the output. The objective of training is to minimize a loss function $\mathcal{L}(\theta)$, defined over a dataset $\{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}^m$ represent the input-target pairs. The evolution of the parameters during training is governed by the gradient flow equation $\frac{d\theta}{dt} = -\nabla_\theta \mathcal{L}(\theta)$, where $\nabla_\theta \mathcal{L}(\theta)$ is the gradient of the loss function with respect to the parameters. To analyze the dynamics of the network outputs, we first consider the time derivative of $f_\theta(x)$. Using the chain rule, this is expressed as:

$$\frac{\partial f_\theta(x)}{\partial t} = \nabla_\theta f_\theta(x)^\top \frac{d\theta}{dt}. \tag{123}$$

Substituting $\frac{d\theta}{dt} = -\nabla_\theta \mathcal{L}(\theta)$, we have:

$$\frac{\partial f_\theta(x)}{\partial t} = -\nabla_\theta f_\theta(x)^\top \nabla_\theta \mathcal{L}(\theta). \tag{124}$$

The gradient of the loss function, $\mathcal{L}(\theta)$, can be expressed explicitly in terms of the training data. For a generic loss function over the dataset, this takes the form:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f_\theta(x_i), y_i), \tag{125}$$

where $\ell(f_\theta(x_i), y_i)$ represents the loss for the i-th data point. The gradient of the loss with respect to the parameters is therefore given by:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_\theta f_\theta(x_i) \nabla_{f_\theta(x_i)} \ell(f_\theta(x_i), y_i). \tag{126}$$

Substituting this back into the time derivative of $f_\theta(x)$, we obtain:

$$\frac{\partial f_\theta(x)}{\partial t} = -\frac{1}{n} \sum_{i=1}^n \nabla_\theta f_\theta(x)^\top \nabla_\theta f_\theta(x_i) \nabla_{f_\theta(x_i)} \ell(f_\theta(x_i), y_i). \tag{127}$$

To introduce the Neural Tangent Kernel (NTK), we define it as the Gram matrix of the Jacobians of the network output with respect to the parameters:

$$\Theta(x, x'; \theta) = \nabla_\theta f_\theta(x)^\top \nabla_\theta f_\theta(x'). \tag{128}$$

Using this definition, the time evolution of the output becomes:

$$\frac{\partial f_\theta(x)}{\partial t} = -\frac{1}{n} \sum_{i=1}^n \Theta(x, x_i; \theta) \nabla_{f_\theta(x_i)} \ell(f_\theta(x_i), y_i). \tag{129}$$

In the overparameterized regime, where the number of parameters p is significantly larger than the number of training data points n, it has been empirically and theoretically observed that the NTK $\Theta(x, x'; \theta)$ remains nearly constant during training. Specifically, $\Theta(x, x'; \theta) \approx \Theta(x, x'; \theta_0)$, where θ_0 represents the parameters at initialization. This constancy significantly simplifies the analysis of the network's training dynamics. To see this, consider the solution to the differential equation governing the output dynamics. Let $F(t) \in \mathbb{R}^{n \times m}$ represent the matrix of network outputs for all training inputs, where the i-th row corresponds to $f_\theta(x_i)$. The dynamics can be expressed in matrix form as:

$$\frac{\partial F(t)}{\partial t} = -\frac{1}{n} \Theta(\theta_0) \nabla_F \mathcal{L}(F), \tag{130}$$

where $\Theta(\theta_0) \in \mathbb{R}^{n \times n}$ is the NTK matrix evaluated at initialization, and $\nabla_F \mathcal{L}(F)$ is the gradient of the loss with respect to the output matrix F. For the special case of a mean squared error loss, $\mathcal{L}(F) = \frac{1}{2n} \|F - Y\|_F^2$, where $Y \in \mathbb{R}^{n \times m}$ is the matrix of target outputs, the gradient simplifies to:

$$\nabla_F \mathcal{L}(F) = \frac{1}{n}(F - Y). \tag{131}$$

Substituting this into the dynamics, we obtain:

$$\frac{\partial F(t)}{\partial t} = -\frac{1}{n^2}\Theta(\theta_0)(F(t) - Y). \tag{132}$$

The solution to this differential equation is:

$$F(t) = Y + e^{-\frac{\Theta(\theta_0)}{n^2}t}(F(0) - Y), \tag{133}$$

where $F(0)$ represents the initial outputs of the network. As $t \to \infty$, the exponential term vanishes, and the network outputs converge to the targets Y, provided that $\Theta(\theta_0)$ is positive definite. The rate of convergence is determined by the eigenvalues of $\Theta(\theta_0)$, with smaller eigenvalues corresponding to slower convergence along the associated eigenvectors. To understand the stationary points of this system, we note that these occur when $\frac{\partial F(t)}{\partial t} = 0$. From the dynamics, this implies:

$$\Theta(\theta_0)(F - Y) = 0. \tag{134}$$

If $\Theta(\theta_0)$ is invertible, this yields $F = Y$, indicating that the network exactly interpolates the training data at the stationary point. However, if $\Theta(\theta_0)$ is not full-rank, the stationary points form a subspace of solutions satisfying $(I - \Pi)(F - Y) = 0$, where Π is the projection operator onto the column space of $\Theta(\theta_0)$.

The NTK framework provides a mathematically rigorous lens to analyze training dynamics, elucidating the interplay between parameter evolution, kernel properties, and loss convergence in neural networks. By linearizing the training dynamics through the NTK, we achieve a deep understanding of how overparameterized networks evolve under gradient flow and how they reach stationary points, revealing their capacity to interpolate data with remarkable precision.

3.2 NTK Regime

The Neural Tangent Kernel (NTK) regime is a fundamental framework for understanding the dynamics of gradient descent in highly overparameterized neural networks. Consider a neural network $f(\mathbf{x}; \boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta} \in \mathbb{R}^P$, where P represents the total number of parameters, and $\mathbf{x} \in \mathbb{R}^d$ is the input vector. For a training dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, the loss function $L(t)$ at time t is given by

$$L(t) = \frac{1}{2N}\sum_{i=1}^{N}\left(f(\mathbf{x}_i; \boldsymbol{\theta}(t)) - y_i\right)^2. \tag{135}$$

The parameters evolve according to gradient descent as $\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \eta\nabla_{\boldsymbol{\theta}}L(t)$, where $\eta > 0$ is the learning rate. In the NTK regime, we consider the first-order Taylor expansion of the network output around the initialization $\boldsymbol{\theta}_0$:

$$f(\mathbf{x}; \boldsymbol{\theta}) \approx f(\mathbf{x}; \boldsymbol{\theta}_0) + \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_0)^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_0). \tag{136}$$

This linear approximation transforms the nonlinear dynamics of f into a simpler, linearized form. To analyze training, we introduce the Jacobian matrix $J \in \mathbb{R}^{N\times P}$, where $J_{ij} = \frac{\partial f(\mathbf{x}_i;\boldsymbol{\theta}_0)}{\partial\theta_j}$. The vector of outputs $\mathbf{f}(t) \in \mathbb{R}^N$, aggregating predictions over the dataset, evolves as

$$\mathbf{f}(t) = \mathbf{f}(0) + J(\boldsymbol{\theta}(t) - \boldsymbol{\theta}_0). \tag{137}$$

The NTK $\Theta \in \mathbb{R}^{N\times N}$ is defined as

$$\Theta_{ij} = \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_i; \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_j; \boldsymbol{\theta}_0). \tag{138}$$

As $P \to \infty$, the NTK converges to a deterministic matrix that remains nearly constant during training. Substituting the linearized form of $\mathbf{f}(t)$ into the gradient descent update equation gives

$$\mathbf{f}(t+1) = \mathbf{f}(t) - \frac{\eta}{N}\Theta(\mathbf{f}(t) - \mathbf{y}), \tag{139}$$

where $\mathbf{y} \in \mathbb{R}^N$ is the vector of true labels. Defining the residual $\mathbf{r}(t) = \mathbf{f}(t) - \mathbf{y}$, the dynamics of training reduce to

$$\mathbf{r}(t+1) = \left(I - \frac{\eta}{N}\Theta\right)\mathbf{r}(t). \tag{140}$$

The eigendecomposition $\Theta = Q\Lambda Q^\top$, with orthogonal Q and diagonal $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_N)$, allows us to analyze the decay of residuals in the eigenbasis of Θ:

$$\tilde{\mathbf{r}}(t+1) = \left(I - \frac{\eta}{N}\Lambda\right)\tilde{\mathbf{r}}(t), \tag{141}$$

where $\tilde{\mathbf{r}}(t) = Q^\top \mathbf{r}(t)$. Each component decays as

$$\tilde{r}_i(t) = \left(1 - \frac{\eta\lambda_i}{N}\right)^t \tilde{r}_i(0). \tag{142}$$

For small η, the training dynamics are approximately continuous, governed by

$$\frac{d\mathbf{r}(t)}{dt} = -\frac{1}{N}\Theta\mathbf{r}(t), \tag{143}$$

leading to the solution

$$\mathbf{r}(t) = \exp\left(-\frac{\Theta t}{N}\right)\mathbf{r}(0). \tag{144}$$

The NTK for specific architectures, such as fully connected ReLU networks, can be derived using layerwise covariance matrices. Let $\Sigma^{(l)}(\mathbf{x}, \mathbf{x}')$ denote the covariance between pre-activations at layer l. The recurrence relation for $\Sigma^{(l)}$ is

$$\Sigma^{(l)}(\mathbf{x}, \mathbf{x}') = \frac{1}{2\pi}\|\mathbf{z}^{(l-1)}(\mathbf{x})\|\|\mathbf{z}^{(l-1)}(\mathbf{x}')\|\left(\sin\theta + (\pi - \theta)\cos\theta\right), \tag{145}$$

where $\theta = \cos^{-1}\left(\frac{\Sigma^{(l-1)}(\mathbf{x},\mathbf{x}')}{\sqrt{\Sigma^{(l-1)}(\mathbf{x},\mathbf{x})\Sigma^{(l-1)}(\mathbf{x}',\mathbf{x}')}}\right)$. The NTK, a sum over contributions from all layers, quantifies how parameter updates propagate through the network.

In the infinite-width limit, the NTK framework predicts generalization properties, as the kernel matrix Θ governs both training and test-time behavior. The NTK connects neural networks to classical kernel methods, offering a bridge between deep learning and well-established theoretical tools in approximation theory. This regime's deterministic and analytical tractability enables precise characterizations of network performance, convergence rates, and robustness to initialization and learning rate variations.

4 Generalization Bounds: PAC-Bayes and Spectral Analysis

4.1 PAC-Bayes Formalism

Literature Review: McAllester (1999) [92] introduced the PAC-Bayes bound, a fundamental theorem that provides generalization guarantees for Bayesian learning models. He established a trade-off between complexity and empirical risk, serving as the theoretical foundation for modern PAC-Bayesian analysis. Catoni (2007) [93] in his book rigorously extended the PAC-Bayes framework by linking it with information-theoretic and statistical mechanics concepts and introduced exponential and Gibbs priors for learning, improving PAC-Bayesian bounds for supervised classification. Germain et. al. (2009) [94] applied PAC-Bayes theory to linear classifiers, including SVMs and logistic regression. They demonstrated that PAC-Bayesian generalization bounds are tighter than classical Vapnik-Chervonenkis (VC) dimension bounds. Seeger (2002) [95] extended PAC-Bayes bounds to Gaussian Process models, proving tight generalization guarantees for Bayesian classifiers. He laid the groundwork for probabilistic kernel methods. Alquier et. al. (2006) [96] connected variational inference and PAC-Bayes bounds, proving that variational approximations can preserve generalization guarantees of PAC-Bayesian bounds. Dziugaite and Roy (2017) [97] gave one of the first applications of PAC-Bayes to deep learning. They derived nonvacuous generalization bounds for stochastic neural networks, bridging theory and practice. Rivasplata et. al. (2020) [98] provided novel PAC-Bayes bounds that improve over existing guarantees, making PAC-Bayesian bounds more practical for modern ML applications. Lever et. al. (2013) [99] explored data-dependent priors in PAC-Bayes theory, showing that adaptive priors lead to tighter generalization bounds. Rivasplata et. al. (2018) [100] introduced instance-dependent priors, improving personalized learning and making PAC-Bayesian methods more useful for real-world machine learning problems. Lindemann et. al. (2024) [101] integrated PAC-Bayes theory with conformal prediction to improve formal verification in control systems, demonstrating PAC-Bayes' relevance to safety-critical applications.

The PAC-Bayes formalism is a foundational framework in statistical learning theory, designed to provide probabilistic guarantees on the generalization performance of learning algorithms. By combining principles from the PAC (Probably Approximately Correct) framework and Bayesian reasoning, PAC-Bayes delivers bounds that characterize the expected performance of hypotheses drawn from posterior distributions, given a finite sample of data. This document presents an extremely rigorous and mathematically precise description of the PAC-Bayes formalism, emphasizing its theoretical constructs and implications.

At the core of the PAC-Bayes formalism lies the ambition to rigorously quantify the generalization ability of hypotheses $h \in \mathcal{H}$ based on their performance on a finite dataset $S \sim \mathcal{D}^m$, where $\mathcal{D}$ represents the underlying, and typically unknown, data distribution. The PAC framework, which was originally designed to provide high-confidence guarantees on the true risk

$$R(h) = \mathbb{E}_{z\sim\mathcal{D}}[\ell(h, z)], \tag{146}$$

is enriched in PAC-Bayes by incorporating principles from Bayesian reasoning. This integration allows for bounds not just on individual hypotheses but on distributions Q over $\mathcal{H}$, yielding a sophisticated characterization of generalization that inherently accounts for the variability and uncertainty in the hypothesis space. There are some Mathematical Constructs: True and Empirical Risks. The true risk $R(h)$, as defined by the expected loss, is typically inaccessible due to the unknown nature of $\mathcal{D}$. Instead, the empirical risk

$$\hat{R}(h, S) = \frac{1}{m}\sum_{i=1}^{m} \ell(h, z_i) \tag{147}$$

serves as a computable proxy. The key question addressed by PAC-Bayes is: *How does $\hat{R}(h, S)$ relate to $R(h)$, and how can we bound the deviation probabilistically?* For a distribution Q over $\mathcal{H}$, these risks are generalized as:

$$R(Q) = \mathbb{E}_{h\sim Q}[R(h)], \quad \hat{R}(Q, S) = \mathbb{E}_{h\sim Q}[\hat{R}(h, S)]. \tag{148}$$

This generalization is pivotal because it allows the analysis to transcend individual hypotheses and consider probabilistic ensembles, where $Q(h)$ represents a posterior belief over the hypothesis space conditioned on the observed data. We now need to discuss how Prior and Posterior Distributions encode knowledge and complexity. The prior P is a fixed distribution over $\mathcal{H}$ that reflects pre-data assumptions about the plausibility of hypotheses. Crucially, P must be independent of S to avoid biasing the bounds. The posterior Q, however, is data-dependent and typically chosen to minimize a combination of empirical risk and complexity. This choice is guided by the PAC-Bayes inequality, which regularizes Q via its Kullback-Leibler (KL) divergence from P:

$$\mathrm{KL}(Q\|P) = \int_{\mathcal{H}} Q(h) \log \frac{Q(h)}{P(h)}\, dh. \tag{149}$$

The KL divergence quantifies the informational cost of updating P to Q, serving as a penalty term that discourages overly complex posteriors. This regularization is critical in preventing overfitting, ensuring that Q achieves a balance between data fidelity and model simplicity.

Let's derive the PAC-Bayes Inequality: Probabilistic and Information-Theoretic Foundations. The derivation of the PAC-Bayes inequality hinges on a combination of probabilistic tools and information-theoretic arguments. A central step involves applying a change of measure from P to Q, leveraging the identity:

$$\mathbb{E}_{h\sim Q}[f(h)] = \mathbb{E}_{h\sim P}\left[f(h)\frac{Q(h)}{P(h)}\right]. \tag{150}$$

This allows the incorporation of Q into bounds that originally apply to fixed h. By analyzing the moment-generating function of deviations between $\hat{R}(h, S)$ and $R(h)$, and applying Hoeffding's inequality to the empirical loss, we arrive at the following bound for any Q and P, with probability at least $1-\delta$:

$$R(Q) \leq \hat{R}(Q, S) + \sqrt{\frac{\mathrm{KL}(Q\|P) + \log\frac{1}{\delta}}{2m}}. \tag{151}$$

The generalization bound is therefore given by:

$$\mathcal{L}(f) - \mathcal{L}_{\text{emp}}(f) \leq \sqrt{\frac{\mathcal{KL}(Q\|P) + \log(1/\delta)}{2N}}, \tag{152}$$

where $\mathcal{KL}(Q\|P)$ quantifies the divergence between the posterior Q and prior P. This bound is remarkable because it explicitly ties the true risk $R(Q)$ to the empirical risk $\hat{R}(Q, S)$, the KL divergence, and the sample size m. The PAC-Bayes bound encapsulates three competing forces: the empirical risk $\hat{R}(Q, S)$, the complexity penalty $\mathrm{KL}(Q\|P)$, and the confidence term $\sqrt{\frac{\log\frac{1}{\delta}}{2m}}$. This interplay reflects a fundamental trade-off in learning:

1. **Empirical Risk:** $\hat{R}(Q, S)$ captures how well the posterior Q fits the training data.
2. **Complexity:** The KL divergence ensures that Q remains close to P, discouraging overfitting and promoting generalization.
3. **Confidence:** The term $\sqrt{\frac{\log\frac{1}{\delta}}{2m}}$ shrinks with increasing sample size, tightening the bound and enhancing reliability.

The KL term also introduces an inherent regularization effect, penalizing hypotheses that deviate significantly from prior knowledge. This aligns with Occam's Razor, favoring simpler explanations that are consistent with the data.

There are several extensions and Advanced Applications of Pac-Bayes Formalism. While the classical PAC-Bayes framework assumes i.i.d. data, recent advancements have generalized the theory to handle structured data, such as in time-series and graph-based learning. Furthermore, alternative divergence measures, like Rényi divergence or Wasserstein distance, have been explored to accommodate scenarios where KL divergence may be inappropriate. In practical settings, PAC-Bayes bounds have been instrumental in analyzing neural networks, Bayesian ensembles, and stochastic processes, offering theoretical guarantees even in high-dimensional, non-convex optimization landscapes.

4.2 Spectral Regularization

The concept of spectral regularization, which refers to the preferential learning of low-frequency modes by neural networks before high-frequency modes, emerges from a combination of Fourier analysis, optimization theory, and the inherent properties of deep neural networks. This phenomenon is tightly connected to the functional approximation capabilities of neural networks and can be rigorously understood through the lens of Fourier decomposition and the gradient descent optimization process.

Literature Review: Jin et. al. (2025) [102] introduced a novel confusional spectral regularization technique to improve fairness in machine learning models. The study focuses on the spectral norm of the robust confusion matrix and proposes a method to control spectral properties, ensuring more robust and unbiased learning. It provides insights into how regularization can mitigate biases in classification tasks. Ye et. al. (2025) [103] applied spectral clustering with regularization to detect small clusters in complex networks. The work enhances spectral clustering techniques by integrating regularization methods, allowing improved performance in anomaly detection and community detection tasks. The approach significantly improves robustness in highly noisy data environments. Bhattacharjee and Bharadwaj (2025) [104] explored how spectral domain representations can benefit from autoencoder-based feature extraction combined with stochastic regularization techniques. The authors propose a Symmetric Autoencoder (SymAE) that enables better generalization of spectral features, particularly useful in high-dimensional data and deep learning applications. Wu et. al. (2025) [105] applied spectral regularization to geophysical data processing, specifically for high-resolution velocity spectrum analysis. The approach enhances the resolution of velocity estimation in seismic imaging by using hyperbolic Radon transform regularization, demonstrating how spectral regularization can benefit applications beyond traditional ML. Ortega et. al. (2025) [106] applied Tikhonov regularization to atmospheric spectral analysis, optimizing gas retrieval strategies in high-resolution spectroscopic observations. The work significantly improves methane (CH4) and nitrous oxide (N2O) detection accuracy by reducing noise in spectral measurements, showcasing the impact of spectral regularization in remote sensing and environmental monitoring. Kazmi et. al. (2025) [107] proposed a spectral regularization-based federated learning model to improve robustness in cybersecurity threat detection. The model addresses the issue of non-IID data in SDN (Software Defined Networks) by utilizing spectral norm-based regularization within deep learning architectures. Zhao et. al. (2025) [108] introduced a regularized deep spectral clustering method, which enhances feature selection and clustering robustness. The authors utilize projected adaptive feature selection combined with spectral graph regularization, improving clustering accuracy and interpretability in high-dimensional datasets. Saranya and Menaka (2025) [109] integrated spectral regularization with quantum-based machine learning to analyze EEG signals for Autism Spectrum Disorder (ASD) detection. The proposed method improves spatial filtering and feature extraction using wavelet-based regularization, leading to more reliable EEG pattern recognition. Dhalbisoi et. al. (2024) [110] developed a Regularized Zero-Forcing (RZF) method for spectral efficiency optimization in beyond 5G networks. The authors demonstrate that spectral regularization techniques can significantly improve signal-to-noise ratios in wireless communication systems, optimizing data transmission in massive MIMO architectures. Wei et. al. (2025) [111] explored the use of spectral regularization in medical imaging, particularly in 3D near-infrared spectral tomography. The proposed model integrates regularized convolutional neural networks (CNNs) to improve tissue imaging resolution and accuracy, demonstrating an application of spectral regularization in biomedical engineering.

Let us define a target function $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^d$, and its Fourier transform $\hat{f}(\boldsymbol{\xi})$ as

$$\hat{f}(\boldsymbol{\xi}) = \int_{\mathbb{R}^d} f(\mathbf{x}) e^{-i2\pi\boldsymbol{\xi}\cdot\mathbf{x}}\, d\mathbf{x} \tag{153}$$

This transform breaks down $f(\mathbf{x})$ into frequency components indexed by $\boldsymbol{\xi}$. In the context of deep learning, we seek to approximate $f(\mathbf{x})$ with a neural network output $f_{\text{NN}}(\mathbf{x};\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the set of trainable parameters. The loss function to be minimized is typically the mean squared error:

$$\mathcal{L}(\boldsymbol{\theta}) = \int_{\mathbb{R}^d} \left|f(\mathbf{x}) - f_{\text{NN}}(\mathbf{x};\boldsymbol{\theta})\right|^2\, d\mathbf{x} \tag{154}$$

We can equivalently express this loss in the Fourier domain, leveraging Parseval's theorem:

$$\mathcal{L}(\boldsymbol{\theta}) = \int_{\mathbb{R}^d} \left|\hat{f}(\boldsymbol{\xi}) - \hat{f}_{\text{NN}}(\boldsymbol{\xi};\boldsymbol{\theta})\right|^2\, d\boldsymbol{\xi} \tag{155}$$

To solve for $\boldsymbol{\theta}$, we employ gradient descent:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \tag{156}$$

where η is the learning rate. The gradient of the loss function with respect to $\boldsymbol{\theta}$ is

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) = 2\int_{\mathbb{R}^d} \left(\hat{f}_{\text{NN}}(\boldsymbol{\xi};\boldsymbol{\theta}) - \hat{f}(\boldsymbol{\xi})\right) \nabla_{\boldsymbol{\theta}}\hat{f}_{\text{NN}}(\boldsymbol{\xi};\boldsymbol{\theta})\, d\boldsymbol{\xi} \tag{157}$$

At the core of this gradient descent process lies the behavior of the gradient $\nabla_{\boldsymbol{\theta}}\hat{f}_{\text{NN}}(\boldsymbol{\xi};\boldsymbol{\theta})$ with respect to the frequency components $\boldsymbol{\xi}$. For neural networks, particularly those with ReLU activations, the gradients of the output with respect to the parameters are expected to decay for high-frequency components. This can be approximated as

$$\mathcal{R}(\boldsymbol{\xi}) \sim \frac{1}{1 + \|\boldsymbol{\xi}\|^2} \tag{158}$$

which implies that the neural network is inherently more sensitive to low-frequency components of the target function during early iterations of training. This spectral decay is a direct consequence of the structure of the network's activations, which are more sensitive to low-frequency features due to their smoother, lower-order terms. To understand the role of the neural tangent kernel (NTK), which governs the linearized dynamics of the neural network, we define the NTK as

$$\Theta(\mathbf{x},\mathbf{x}';\boldsymbol{\theta}) = \sum_{i=1}^{P} \frac{\partial f_{\text{NN}}(\mathbf{x};\boldsymbol{\theta})}{\partial\theta_i}\frac{\partial f_{\text{NN}}(\mathbf{x}';\boldsymbol{\theta})}{\partial\theta_i} \tag{159}$$

The NTK essentially describes how the output of the network changes with respect to its parameters. The evolution of the network's output during training can be approximated by the solution to a linear system governed by the NTK. The output of the network at time t is given by

$$f_{\text{NN}}(\mathbf{x};t) = \sum_k c_k \left(1 - e^{-\eta\lambda_k t}\right)\phi_k(\mathbf{x}) \tag{160}$$

where $\{\lambda_k\}$ are the eigenvalues of Θ, and $\{\phi_k(\mathbf{x})\}$ are the corresponding eigenfunctions. The eigenvalues λ_k determine the speed of convergence for each frequency mode, with low-frequency modes (large λ_k) converging more quickly than high-frequency ones (small λ_k):

$$1 - e^{-\eta\lambda_k t} \to 1 \quad \text{for large } \lambda_k \quad \text{and} \quad 1 - e^{-\eta\lambda_k t} \to 0 \quad \text{for small } \lambda_k \tag{161}$$

This differential learning rate for frequency components leads to the spectral regularization phenomenon, where the network learns the low-frequency components of the function first, and the high-frequency modes only begin to adapt once the low-frequency ones have been approximated with sufficient accuracy. In a more formal setting, the spectral bias can also be understood in terms of Sobolev spaces. A neural network function f_{NN} can be seen as a function in a Sobolev space $W^{m,2}$, where the norm of a function f in this space is defined as

$$\|f\|_{W^{m,2}}^2 = \int_{\mathbb{R}^d} \left(1 + \|\boldsymbol{\xi}\|^2\right)^m \left|\hat{f}(\boldsymbol{\xi})\right|^2 \, d\boldsymbol{\xi} \tag{162}$$

When training a neural network, the optimization process implicitly regularizes the higher-order Sobolev norms, meaning that the network will initially approximate the target function in terms of lower-order derivatives (which correspond to low-frequency modes). This can be expressed by introducing a regularization term in the loss function:

$$\mathcal{L}_{\text{eff}}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \lambda \|f_{\mathrm{NN}}\|_{W^{m,2}}^2 \tag{163}$$

where λ is a regularization parameter that controls the trade-off between data fidelity and smoothness in the approximation.

Thus, spectral regularization emerges as a consequence of the network's architecture, the nature of gradient descent optimization, and the inherent smoothness of the functions that neural networks are capable of learning. The mathematical structure of the NTK and the regularization properties of the Sobolev spaces provide a rigorous framework for understanding why neural networks prioritize the learning of low-frequency modes, reinforcing the idea that neural networks are implicitly biased toward smooth, low-frequency approximations at the beginning of training. This insight has profound implications for the generalization behavior of neural networks and their capacity to approximate complex functions.

5 Neural Network Basics

Literature Review: Goodfellow et. al. (2016) [112] wrote one of the most comprehensive books on deep learning, covering the theoretical foundations of neural networks, optimization techniques, and probabilistic models. It is widely used in academic courses and research. Haykin (2009) [113] explained neural networks from a signal processing perspective, covering perceptrons, backpropagation, and recurrent networks with a strong mathematical approach. Schmidhuber (2015) [114] gave a historical and theoretical review of deep learning architectures, including recurrent neural networks (RNNs), convolutional neural networks (CNNs), and long short-term memory (LSTM). Bishop (2006) [115] gave a Bayesian perspective on neural networks and probabilistic graphical models, emphasizing the statistical underpinnings of learning. Poggio and Smale (2003) [116] established theoretical connections between neural networks, kernel methods, and function approximation. LeCun (2015) [117] discusses the principles behind modern deep learning, including backpropagation, unsupervised learning, and hierarchical feature extraction. Cybenko (1989) [58] proved the universal approximation theorem, demonstrating that a neural network with a single hidden layer can approximate any continuous function. Hornik et. al. (1989) [57] extended Cybenko's theorem, proving that multilayer perceptrons (MLPs) are universal function approximators. Pinkus (1999) [60] gave a rigorous mathematical discussion on neural networks from the perspective of approximation theory. Tishby and Zaslavsky (2015) [118] introduced the information bottleneck framework for understanding deep neural networks, explaining how networks learn to compress and encode information efficiently.

5.1 Perceptrons and Artificial Neurons

The perceptron is the simplest form of an artificial neural network, operating as a binary classifier. It computes the linear combination z of the input features $\vec{x} = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$ and a corresponding weight vector $\vec{w} = [w_1, w_2, \ldots, w_n]^T \in \mathbb{R}^n$, augmented by a bias term $b \in \mathbb{R}$. This can be expressed as

$$z = \vec{w}^T \vec{x} + b = \sum_{i=1}^{n} w_i x_i + b. \tag{164}$$

To determine the output, this value is passed through the step activation function, defined mathematically as

$$\phi(z) = \begin{cases} 1, & z \geq 0, \\ 0, & z < 0. \end{cases} \tag{165}$$

Thus, the perceptron's decision-making process can be expressed as

$$y = \phi(\vec{w}^T \vec{x} + b), \tag{166}$$

where $y \in \{0, 1\}$. The equation $\vec{w}^T \vec{x} + b = 0$ defines a hyperplane in $\mathbb{R}^n$, which acts as the decision boundary. For any input $\vec{x}$, the classification is determined by the sign of $\vec{w}^T \vec{x} + b$, specifically $y = 1$ if $\vec{w}^T \vec{x} + b \geq 0$ and $y = 0$ otherwise. Geometrically, this classification corresponds to partitioning the input space into two distinct half-spaces. To train the perceptron, a supervised learning algorithm adjusts the weights $\vec{w}$ and the bias b iteratively using labeled training data $\{(\vec{x}_i, y_i)\}_{i=1}^{m}$, where y_i represents the ground truth. When the predicted output $y_{\text{pred}} = \phi(\vec{w}^T \vec{x}_i + b)$ differs from y_i, the weight vector and bias are updated according to the rule

$$\vec{w} \leftarrow \vec{w} + \eta(y_i - y_{\text{pred}})\vec{x}_i, \tag{167}$$

and

$$b \leftarrow b + \eta(y_i - y_{\text{pred}}), \tag{168}$$

where $\eta > 0$ is the learning rate. Each individual weight w_j is updated as

$$w_j \leftarrow w_j + \eta(y_i - y_{\text{pred}})x_{ij}. \tag{169}$$

For a linearly separable dataset, the Perceptron Convergence Theorem asserts that the algorithm will converge to a solution after a finite number of updates. Specifically, the number of updates is bounded by

$$\frac{R^2}{\gamma^2}, \tag{170}$$

where $R = \max_i \|\vec{x}_i\|$ is the maximum norm of the input vectors, and γ is the minimum margin, defined as

$$\gamma = \min_i \frac{y_i(\vec{w}^T \vec{x}_i + b)}{\|\vec{w}\|}. \tag{171}$$

The limitations of the perceptron, particularly its inability to solve linearly inseparable problems such as the XOR problem, necessitate the extension to artificial neurons with non-linear activation functions. A popular choice is the sigmoid activation function

$$\phi(z) = \frac{1}{1 + e^{-z}}, \tag{172}$$

which maps $z \in \mathbb{R}$ to the continuous interval $(0, 1)$. The derivative of the sigmoid function, essential for gradient-based optimization, is

$$\phi'(z) = \phi(z)(1 - \phi(z)). \tag{173}$$

Another widely used activation function is the hyperbolic tangent $\tanh(z)$, defined as

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{174}$$

with derivative

$$\tanh'(z) = 1 - \tanh^2(z). \tag{175}$$

ReLU, or Rectified Linear Unit, is defined as

$$\phi(z) = \max(0, z), \tag{176}$$

with derivative

$$\phi'(z) = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0. \end{cases} \tag{177}$$

These non-linear activations enable the network to approximate non-linear decision boundaries, a capability absent in the perceptron. Artificial neurons form the building blocks of multi-layer perceptrons (MLPs), where neurons are organized into layers. For an L-layer network, the input $\vec{x}$ is transformed layer by layer. At layer l, the output is

$$\vec{z}^{(l)} = \phi^{(l)}(\vec{W}^{(l)}\vec{z}^{(l-1)} + \vec{b}^{(l)}), \tag{178}$$

where $\vec{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix, $\vec{b}^{(l)} \in \mathbb{R}^{n_l}$ is the bias vector, and $\phi^{(l)}$ is the activation function. The network's output is

$$\hat{\vec{y}} = \phi^{(L)}(\vec{W}^{(L)}\vec{z}^{(L-1)} + \vec{b}^{(L)}). \tag{179}$$

The Universal Approximation Theorem guarantees that MLPs with sufficient neurons and non-linear activations can approximate any continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$ to arbitrary precision. Formally, for any $\epsilon > 0$, there exists an MLP $g(\vec{x})$ such that

$$\|f(\vec{x}) - g(\vec{x})\|_\infty < \epsilon \tag{180}$$

for all $\vec{x} \in \mathbb{R}^n$. Training an MLP minimizes a loss function L that quantifies the error between predicted outputs $\hat{\vec{y}}$ and ground truth labels $\vec{y}$. For regression, the mean squared error is

$$L = \frac{1}{m}\sum_{i=1}^{m} \|\hat{\vec{y}}_i - \vec{y}_i\|^2, \tag{181}$$

and for classification, the cross-entropy loss is

$$L = -\frac{1}{m}\sum_{i=1}^{m} \left[\vec{y}_i^T \log \hat{\vec{y}}_i + (1 - \vec{y}_i)^T \log(1 - \hat{\vec{y}}_i)\right]. \tag{182}$$

Optimization uses stochastic gradient descent (SGD), updating parameters $\Theta = \{\vec{W}^{(l)}, \vec{b}^{(l)}\}_{l=1}^{L}$ as

$$\Theta \leftarrow \Theta - \eta \nabla_\Theta L. \tag{183}$$

Gradients are computed via backpropagation:

$$\frac{\partial L}{\partial \vec{W}^{(l)}} = \delta^{(l)} \vec{z}^{(l-1)T}, \tag{184}$$

where $\delta^{(l)}$ is the error signal at layer l, recursively calculated as

$$\delta^{(l)} = (\vec{W}^{(l+1)T}\delta^{(l+1)}) \circ \phi'^{(l)}(\vec{z}^{(l)}). \tag{185}$$

This recursive structure, combined with chain rule applications, efficiently propagates error signals from the output layer back to the input layer.

Artificial neurons and their extensions have thus provided the foundation for modern deep learning. Their mathematical underpinnings and computational frameworks are instrumental in solving a wide array of problems, from classification and regression to complex decision-making. The interplay of linear algebra, calculus, and optimization theory in their formulation ensures that these networks are both theoretically robust and practically powerful.

5.2 Feedforward Neural Networks

Feedforward neural networks (FNNs) are mathematical constructs designed to approximate arbitrary mappings $f : \mathbb{R}^n \to \mathbb{R}^m$ by composing affine transformations and nonlinear activation functions. At their core, these networks consist of L layers, where each layer k transforms its input $\vec{a}_{k-1} \in \mathbb{R}^{m_{k-1}}$ into an output $\vec{a}_k \in \mathbb{R}^{m_k}$ via the operation

$$\vec{a}_k = f_k(W_k\vec{a}_{k-1} + \vec{b}_k). \tag{186}$$

Here, $W_k \in \mathbb{R}^{m_k \times m_{k-1}}$ represents the weight matrix, $\vec{b}_k \in \mathbb{R}^{m_k}$ is the bias vector, and $f_k : \mathbb{R}^{m_k} \to \mathbb{R}^{m_k}$ is a component-wise activation function. Formally, if we denote the input layer as $\vec{a}_0 = \vec{x}$, the final output of the network, $\vec{y} \in \mathbb{R}^m$, is given by $\vec{a}_L = f_L(W_L\vec{a}_{L-1} + \vec{b}_L)$. Each transformation in this sequence can be described as $\vec{z}_k = W_k\vec{a}_{k-1} + \vec{b}_k$, followed by the activation $\vec{a}_k = f_k(\vec{z}_k)$. The affine transformation $\vec{z}_k = W_k\vec{a}_{k-1} + \vec{b}_k$ encapsulates the linear combination of inputs with weights W_k and the addition of biases $\vec{b}_k$. For any two layers k and $k+1$, the overall transformation can be represented by

$$\vec{z}_{k+1} = W_{k+1}(W_k\vec{a}_{k-1} + \vec{b}_k) + \vec{b}_{k+1}. \tag{187}$$

Expanding this, we have

$$\vec{z}_{k+1} = W_{k+1}W_k\vec{a}_{k-1} + W_{k+1}\vec{b}_k + \vec{b}_{k+1}. \tag{188}$$

Without the nonlinearity introduced by f_k, the network reduces to a single affine transformation

$$\vec{y} = W\vec{x} + \vec{b}, \tag{189}$$

where $W = W_L W_{L-1} \cdots W_1$ and

$$\vec{b} = W_L W_{L-1} \cdots W_2\vec{b}_1 + \cdots + \vec{b}_L. \tag{190}$$

Thus, the incorporation of nonlinear activation functions is critical, as it enables the network to approximate non-linear mappings. Activation functions f_k are applied element-wise to the pre-activation vector $\vec{z}_k$. The choice of activation significantly affects the network's behavior and training. For example, the sigmoid activation $f(x) = \frac{1}{1+e^{-x}}$ compresses inputs into the range $(0, 1)$ and has a derivative given by

$$f'(x) = f(x)(1 - f(x)). \tag{191}$$

The hyperbolic tangent activation $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ maps inputs to $(-1, 1)$ with a derivative

$$f'(x) = 1 - \tanh^2(x). \tag{192}$$

The ReLU activation $f(x) = \max(0, x)$, commonly used in modern networks, has a derivative

$$f'(x) = \begin{cases} 1 & x > 0, \\ 0 & x \leq 0. \end{cases} \tag{193}$$

These derivatives are essential for gradient-based optimization. The objective of training a feedforward neural network is to minimize a loss function $\mathcal{L}$, which measures the discrepancy between the predicted outputs $\vec{y}_i$ and the true targets $\vec{t}_i$ over a dataset $\{(\vec{x}_i, \vec{t}_i)\}_{i=1}^N$. For regression problems, the mean squared error (MSE) is often used, given by

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \|\vec{y}_i - \vec{t}_i\|^2 = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{m}(y_{i,j} - t_{i,j})^2. \tag{194}$$

In classification tasks, the cross-entropy loss is widely employed, defined as

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{m} t_{i,j}\log(y_{i,j}), \tag{195}$$

where $t_{i,j}$ represents the one-hot encoded labels. The gradient of $\mathcal{L}$ with respect to the network parameters is computed using backpropagation, which applies the chain rule iteratively to propagate errors from the output layer to the input layer. During backpropagation, the error signal at the output layer is computed as

$$\delta_L = \frac{\partial \mathcal{L}}{\partial \vec{z}_L} = \nabla_{\vec{y}} \mathcal{L} \odot f'_L(\vec{z}_L), \tag{196}$$

where $\odot$ denotes the Hadamard product. For hidden layers, the error signal propagates backward as

$$\delta_k = (W_{k+1}^T \delta_{k+1}) \odot f'_k(\vec{z}_k). \tag{197}$$

The gradients of the loss with respect to the weights and biases are then given by

$$\frac{\partial \mathcal{L}}{\partial W_k} = \delta_k \vec{a}_{k-1}^T, \quad \frac{\partial \mathcal{L}}{\partial \vec{b}_k} = \delta_k. \tag{198}$$

These gradients are used to update the parameters through optimization algorithms like stochastic gradient descent (SGD), where

$$W_k \leftarrow W_k - \eta \frac{\partial \mathcal{L}}{\partial W_k}, \quad \vec{b}_k \leftarrow \vec{b}_k - \eta \frac{\partial \mathcal{L}}{\partial \vec{b}_k}, \tag{199}$$

with $\eta > 0$ as the learning rate. The universal approximation theorem rigorously establishes that a feedforward neural network with at least one hidden layer and sufficiently many neurons can approximate any continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$ on a compact domain $D \subset \mathbb{R}^n$. Specifically, for any $\epsilon > 0$, there exists a network $\hat{f}$ such that $\|f(\vec{x}) - \hat{f}(\vec{x})\| < \epsilon$ for all $\vec{x} \in D$. This expressive capability arises because the composition of affine transformations and nonlinear activations allows the network to approximate highly complex functions by partitioning the input space into regions and assigning different functional behaviors to each.

In summary, feedforward neural networks are a powerful mathematical framework grounded in linear algebra, calculus, and optimization. Their capacity to model intricate mappings between input and output spaces has made them a cornerstone of machine learning, with rigorous mathematical principles underpinning their structure and training. The combination of affine transformations, nonlinear activations, and gradient-based optimization enables these networks to achieve unparalleled flexibility and performance in a wide range of applications.

5.3 Activation Functions

In the context of **neural networks**, activation functions serve as an essential component that enables the network to approximate complex, non-linear mappings. When a neural network processes input data, each neuron computes a weighted sum of the inputs and then applies an activation function $\sigma(z)$ to produce the output. Mathematically, let the input vector to the neuron be $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, and let the weight vector associated with these inputs be $\mathbf{w} = (w_1, w_2, \ldots, w_n)$. The corresponding bias term is denoted as b. The net input z to the activation function is then given by:

$$z = \mathbf{w}^\top \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b \tag{200}$$

where $\mathbf{w}^\top \mathbf{x}$ represents the dot product of the weight vector and the input vector. The activation function $\sigma(z)$ is then applied to this net input to obtain the output of the neuron a:

$$a = \sigma(z) = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right). \tag{201}$$

The activation function introduces a *non-linearity* into the neuron's response, which is a crucial aspect of neural networks because, without it, the network would only be able to perform linear transformations of the input data, limiting its ability to approximate complex, real-world functions. The non-linearity introduced by $\sigma(z)$ is fundamental because it enables the network to capture intricate relationships between the input and output, making neural networks capable of solving problems that require hierarchical feature extraction, such as image classification, time-series forecasting, and language modeling. The importance of non-linearity is most clearly evident when considering the mathematical formulation of a multi-layer neural network. For a feed-forward neural network with L layers, the output $\hat{y}$ of the network is given by the composition of successive affine transformations and activation functions. Let $\mathbf{x}$ denote the input vector, W_k and b_k be the weight matrix and bias vector for the k-th layer, and σ_k be the activation function for the k-th layer. The output of the network is:

$$\hat{y} = \sigma_L(W_L \sigma_{L-1}(W_{L-1} \dots \sigma_1(W_1 \mathbf{x} + b_1) + b_2) + \dots + b_L). \tag{202}$$

If $\sigma(z)$ were a linear function, say $\sigma(z) = c \cdot z$ for some constant c, the composition of such functions would still result in a linear function. Specifically, if each σ_k were linear, the overall network function would simplify to a single linear transformation:

$$\hat{y} = c_1 \cdot \mathbf{x} + c_2, \tag{203}$$

where c_1 and c_2 are constants dependent on the parameters of the network. In this case, the network would have no greater expressive power than a simple linear regression model, regardless of the number of layers. Thus, the *non-linearity* introduced by activation functions allows neural networks to approximate *any continuous function*, as guaranteed by the *universal approximation theorem*. This theorem states that a feed-forward neural network with at least one hidden layer and a sufficiently large number of neurons can approximate any continuous function $f(\mathbf{x})$, provided the activation function is non-linear and the network has enough capacity.

Next, consider the mathematical properties that the activation function $\sigma(z)$ must possess. First, it must be *differentiable* to allow the use of gradient-based optimization methods like *backpropagation* for training. Backpropagation relies on the *chain rule* of calculus to compute the gradients of the loss function $\mathcal{L}$ with respect to the parameters (weights and biases) of the network. Suppose $\mathcal{L} = \mathcal{L}(\hat{y}, \mathbf{y})$ is the loss function, where $\hat{y}$ is the predicted output of the network and $\mathbf{y}$ is the true label. During training, we compute the gradient of $\mathcal{L}$ with respect to the weights using the chain rule. Let $a_k = \sigma_k(z_k)$ represent the output of the activation function at layer k, where z_k is the input to the activation function. The gradient of the loss with respect to the weights at layer k is given by:

$$\frac{\partial \mathcal{L}}{\partial W_k} = \frac{\partial \mathcal{L}}{\partial a_k} \frac{\partial a_k}{\partial z_k} \frac{\partial z_k}{\partial W_k}. \tag{204}$$

The term $\frac{\partial a_k}{\partial z_k}$ is the *derivative* of the activation function, which must exist and be well-defined for gradient-based optimization to work effectively. If the activation function is not differentiable, the backpropagation algorithm cannot compute the gradients, preventing the training process from proceeding.

Now consider the specific forms of activation functions commonly used in practice. The **sigmoid** activation function is one of the most well-known, defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{205}$$

Its derivative is:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)), \tag{206}$$

which can be derived by applying the chain rule to the expression for $\sigma(z)$. Although sigmoid is differentiable and smooth, it suffers from the *vanishing gradient problem*, especially for large positive or negative values of z. Specifically, as $z \to \infty$, $\sigma'(z) \to 0$, and similarly as $z \to -\infty$, $\sigma'(z) \to 0$. This results in very small gradients during backpropagation, making it difficult for the network to learn when the input values become extreme. To mitigate the vanishing gradient problem, the **hyperbolic tangent (tanh)** function is often used as an alternative. It is defined as:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{207}$$

with derivative:

$$\tanh'(z) = 1 - \tanh^2(z). \tag{208}$$

The tanh function outputs values in the range $(-1, 1)$, which helps to center the data around zero. While the tanh function overcomes some of the vanishing gradient issues associated with the sigmoid function, it still suffers from the problem for large $|z|$, where the gradients approach zero. The **Rectified Linear Unit (ReLU)** is another commonly used activation function. It is defined as:

$$\text{ReLU}(z) = \max(0, z), \tag{209}$$

with derivative:

$$\text{ReLU}'(z) = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0. \end{cases} \tag{210}$$

ReLU is particularly advantageous because it is computationally efficient, as it only requires a comparison to zero. Moreover, for positive values of z, the derivative is constant and equal to 1, which helps avoid the vanishing gradient problem. However, ReLU can suffer from the *dying ReLU problem*, where neurons output zero for all inputs if the weights are initialized poorly or if the learning rate is too high, leading to inactive neurons that do not contribute to the learning process. To address the dying ReLU problem, the **Leaky ReLU** activation function is introduced, defined as:

$$\text{Leaky ReLU}(z) = \begin{cases} z, & z > 0, \\ \alpha z, & z \leq 0, \end{cases} \tag{211}$$

where α is a small constant, typically chosen to be 0.01. The derivative of the Leaky ReLU is:

$$\text{Leaky ReLU}'(z) = \begin{cases} 1, & z > 0, \\ \alpha, & z \leq 0. \end{cases} \tag{212}$$

Leaky ReLU ensures that neurons do not become entirely inactive by allowing a small, non-zero gradient for negative values of z. Finally, for classification tasks, particularly when there are multiple classes, the **Softmax** activation function is often used in the output layer of the neural network. The Softmax function is defined as:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}, \tag{213}$$

where z_i is the input to the i-th neuron in the output layer, and the denominator ensures that the outputs sum to 1, making them interpretable as probabilities. The Softmax function is typically used in multi-class classification problems, where the network must predict one class out of several possible categories.

In summary, activation functions are a vital component of neural networks, enabling them to learn intricate patterns in data, allowing for the successful application of neural networks to diverse tasks. Different activation functions—such as sigmoid, tanh, ReLU, Leaky ReLU, and Softmax—each offer distinct advantages and limitations, and their choice significantly impacts the performance and training dynamics of the neural network.

5.4 Loss Functions

In neural networks, the **loss function** is a crucial mathematical tool that quantifies the difference between the predicted output of the model and the true output or target. Let $\mathbf{x}_i$ be the input vector and $\mathbf{y}_i$ the corresponding target vector for the i-th training example. The network, parameterized by weights $\mathbf{W}$, generates a prediction denoted as $\hat{\mathbf{y}}_i = f(\mathbf{x}_i; \mathbf{W})$, where $f(\mathbf{x}_i; \mathbf{W})$ represents the model's output. The objective of training the neural network is to minimize the discrepancy between the predicted output $\hat{\mathbf{y}}_i$ and the true label $\mathbf{y}_i$ across all training examples, effectively learning the mapping function from inputs to outputs. A typical objective function is the **average loss** over a dataset of N samples:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{y}_i, \hat{\mathbf{y}}_i) \tag{214}$$

where $L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$ represents the loss function that computes the error between the true output $\mathbf{y}_i$ and the predicted output $\hat{\mathbf{y}}_i$ for each data point. To minimize this objective function, optimization algorithms such as **gradient descent** are used. The general update rule for the weights $\mathbf{W}$ is given by:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{W}) \tag{215}$$

where η is the **learning rate**, and $\nabla_{\mathbf{W}} \mathcal{L}(\mathbf{W})$ is the gradient of the loss function with respect to the weights. The gradient is computed using **backpropagation**, which applies the **chain rule** of calculus to propagate the error backward through the network, updating the parameters to minimize the loss. For this, we use the partial derivatives of the loss with respect to each layer's weights and biases, ensuring the error is distributed appropriately across all layers. For regression tasks, the **Mean Squared Error (MSE)** loss is frequently used. This loss function quantifies the error as the average squared difference between the predicted and true values. The MSE for a dataset of N examples is given by:

$$L_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{216}$$

where $\hat{\mathbf{y}}_i = f(\mathbf{x}_i; \mathbf{W})$ is the network's predicted output for the i-th input $\mathbf{x}_i$. The gradient of the MSE with respect to the network's output $\hat{\mathbf{y}}_i$ is:

$$\frac{\partial L_{\text{MSE}}}{\partial \hat{\mathbf{y}}_i} = 2\left(\hat{\mathbf{y}}_i - \mathbf{y}_i\right) \tag{217}$$

This gradient guides the weight update in the direction that minimizes the squared error, leading to a better fit of the model to the training data. For **classification tasks**, the **cross-entropy loss** is often employed, as it is particularly well-suited to tasks where the output is a probability distribution over multiple classes. In the binary classification case, where the target label $\mathbf{y}_i$ is either 0 or 1, the binary cross-entropy loss function is defined as:

$$L_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \tag{218}$$

where $\hat{y}_i = f(\mathbf{x}_i; \mathbf{W})$ is the predicted probability that the i-th sample belongs to the positive class (i.e., class 1). For multiclass classification, where the target label $\mathbf{y}_i$ is a one-hot encoded vector representing the true class, the general form of the cross-entropy loss is:

$$L_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c}) \tag{219}$$

where C is the number of classes, and $\hat{y}_{i,c} = f(\mathbf{x}_i; \mathbf{W})$ is the predicted probability that the i-th sample belongs to class c. The gradient of the cross-entropy loss with respect to the predicted probabilities $\hat{\mathbf{y}}_i$ is:

$$\frac{\partial L_{\text{CE}}}{\partial \hat{y}_{i,c}} = \hat{y}_{i,c} - y_{i,c} \tag{220}$$

This gradient facilitates the weight update by adjusting the model's parameters to reduce the difference between the predicted probabilities and the actual class labels.

In neural network training, the optimization process often involves regularization techniques to prevent **overfitting**, especially in cases with high-dimensional data or deep networks. **L2 regularization** (also known as **Ridge regression**) is one common approach, which penalizes large weights by adding a term proportional to the squared L2 norm of the weights to the loss function. The regularized loss function becomes:

$$L_{\text{reg}} = L_{\text{MSE}} + \lambda \sum_{j=1}^{n} W_j^2 \tag{221}$$

where λ is the regularization strength, and W_j represents the parameters of the network. The gradient of the regularized loss with respect to the weights is:

$$\frac{\partial L_{\text{reg}}}{\partial W_j} = \frac{\partial L_{\text{MSE}}}{\partial W_j} + 2\lambda W_j \tag{222}$$

This additional term discourages large values of the weights, reducing the complexity of the model and helping it generalize better to unseen data. Another form of regularization is **L1 regularization** (or **Lasso regression**), which promotes sparsity in the model by adding the L1 norm of the weights to the loss function. The L1 regularized loss function is:

$$L_{\text{reg}} = L_{\text{MSE}} + \lambda \sum_{j=1}^{n} |W_j| \tag{223}$$

The gradient of this regularized loss function with respect to the weights is:

$$\frac{\partial L_{\text{reg}}}{\partial W_j} = \frac{\partial L_{\text{MSE}}}{\partial W_j} + \lambda \, \text{sign}(W_j) \tag{224}$$

where $\text{sign}(W_j)$ is the sign function, which returns 1 for positive values of W_j, -1 for negative values, and 0 for $W_j = 0$. L1 regularization encourages the model to select only a small subset of features by forcing many of the weights to exactly zero, thus simplifying the model and promoting interpretability. The optimization process for neural networks can be viewed as solving a **non-convex optimization problem**, given the highly non-linear activation functions and the deep architectures typically used. In this context, **stochastic gradient descent (SGD)** is commonly employed to perform the optimization by updating the weights based on the gradient computed from a random mini-batch of the data. The update rule for SGD can be expressed as:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} L_{\text{batch}} \tag{225}$$

where $\nabla_{\mathbf{W}} L_{\text{batch}}$ is the gradient of the loss function computed over the mini-batch, and η is the learning rate. Due to the non-convexity of the objective function, SGD tends to converge to a **local minimum** or a **saddle point**, rather than the global minimum, especially in deep neural networks with many layers.

In summary, the loss function plays a central role in guiding the optimization process in neural network training by quantifying the error between the predicted and true outputs. Different loss functions are employed depending on the nature of the problem, with MSE being common for regression and cross-entropy used for classification. Regularization techniques such as L2 and L1 regularization are incorporated to prevent overfitting and ensure better generalization. Through optimization algorithms like gradient descent, the neural network parameters are iteratively updated based on the gradients of the loss function, with the ultimate goal of minimizing the loss across all training examples.

6 Training Neural Networks

Literature Review: Sorrenson (2025) [119] introduced a framework enabling exact maximum likelihood training of unrestricted neural networks. It presents new training methodologies based on probabilistic models and applies them to scientific applications. Liu and Shi (2015) [120] applied advanced neural network theory to meteorological predictions. It uses sensitivity analysis and new training techniques to mitigate sample size limitations. Das et. al. (2025) [121] integrated Finite Integral Transform (FIT) with gradient-enhanced physics-informed neural networks (g-PINN), optimizing training in engineering applications. Zhang et. al. (2025) [122] in his thesis explores neural tangent kernel (NTK) theory to model the gradient descent training process of deep networks and its implications for structural identification. Ali and Hussein (2025) [123] developed a hybrid approach combining fuzzy set theory and artificial neural networks, enhancing training robustness through heuristic optimization. Li (2025) [124] introduced a deep learning-based strategy to train neural networks for imperfect-information extensive-form games, emphasizing offline training techniques. Hu et. al. (2025) [125] explored the convergence properties of deep learning-based PDE solvers, analyzing training loss and function space properties. Chen et. al. (2025) [126] developed a Transformer-based neural network training framework for risk analysis, incorporating feature maps and game-theoretic interpretation. Sun et. al. (2025) [127] established a new benchmarking suite for optimizing neural architecture search (NAS) techniques in training spiking neural networks. Zhang et. al. (2025) [128] proposed a novel iterative training approach for neural networks, enhancing convergence guarantees in theory and practice.

6.1 Backpropagation Algorithm

Consider a neural network with L layers, where each layer l (with $l = 1, 2, \dots, L$) consists of a weight matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$, a bias vector $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$, and an activation function $\sigma^{(l)}$ which is applied element-wise. The network takes as input a vector $x^{(i)} \in \mathbb{R}^{n_0}$ for the i-th training sample, where n_0 is the number of input features, and propagates it through the layers to produce an output $\hat{y}^{(i)} \in \mathbb{R}^{n_L}$, where n_L is the number of output units. The network parameters (weights and biases) $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ are to be optimized to minimize a loss function that captures the error between the predicted output $\hat{y}^{(i)}$ and the true target $y^{(i)}$ for all training examples. For each training sample, we define the loss function $\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$ as the squared error:

$$\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2}\|\hat{y}^{(i)} - y^{(i)}\|_2^2, \tag{226}$$

where $\|\cdot\|_2$ represents the Euclidean norm. The total loss $J(\theta)$ for the entire dataset is the average of the individual losses:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}), \tag{227}$$

where N is the number of training samples. For squared error loss, we can write:

$$J(\theta) = \frac{1}{2N}\sum_{i=1}^{N} \|\hat{y}^{(i)} - y^{(i)}\|_2^2. \tag{228}$$

The forward pass through the network consists of computing the activations at each layer. For the l-th layer, the pre-activation $z^{(l)}$ is calculated as:

$$z^{(l)} = \mathbf{W}^{(l)} a^{(l-1)} + \mathbf{b}^{(l)}, \tag{229}$$

where $a^{(l-1)}$ is the activation from the previous layer and $\mathbf{W}^{(l)}$ is the weight matrix connecting the $(l-1)$-th layer to the l-th layer. The output of the layer, i.e., the activation $a^{(l)}$, is computed by applying the activation function $\sigma^{(l)}$ element-wise to $z^{(l)}$:

$$a^{(l)} = \sigma^{(l)}(z^{(l)}). \tag{230}$$

The final output of the network is given by the activation $a^{(L)}$ at the last layer, which is the predicted output $\hat{y}^{(i)}$:

$$\hat{y}^{(i)} = a^{(L)}. \tag{231}$$

The backpropagation algorithm computes the gradient of the loss function $J(\theta)$ with respect to each parameter (weights and biases). First, we compute the error at the output layer. Let $\delta^{(L)}$ represent the error at layer L. This is computed by taking the derivative of the loss function with respect to the activations at the output layer:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \odot \sigma^{(L)'}(z^{(L)}), \tag{232}$$

where $\odot$ denotes element-wise multiplication, and $\sigma^{(L)'}(z^{(L)})$ is the derivative of the activation function applied element-wise to $z^{(L)}$. For squared error loss, the derivative with respect to the activations is:

$$\frac{\partial \mathcal{L}}{\partial a^{(L)}} = \hat{y}^{(i)} - y^{(i)} \tag{233}$$

so the error term at the output layer is:

$$\delta^{(L)} = (\hat{y}^{(i)} - y^{(i)}) \odot \sigma^{(L)'}(z^{(L)}) \tag{234}$$

To propagate the error backward through the network, we compute the errors at the hidden layers. For each hidden layer $l = L-1, L-2, \ldots, 1$, the error $\delta^{(l)}$ is calculated by the chain rule:

$$\delta^{(l)} = \left(\mathbf{W}^{(l+1)T} \delta^{(l+1)}\right) \odot \sigma^{(l)'}(z^{(l)}) \tag{235}$$

where $\mathbf{W}^{(l+1)T} \in \mathbb{R}^{n_{l+1} \times n_l}$ is the transpose of the weight matrix connecting layer l to layer $l+1$. This equation uses the fact that the error at layer l depends on the error at the next layer, modulated by the weights, and the derivative of the activation function at layer l. Once the errors $\delta^{(l)}$ are computed for all layers, we can compute the gradients of the loss function with respect to the parameters (weights and biases). The gradient of the loss with respect to the weights $\mathbf{W}^{(l)}$ is:

$$\frac{\partial J(\theta)}{\partial \mathbf{W}^{(l)}} = \frac{1}{N} \sum_{i=1}^{N} \delta^{(l)} (a^{(l-1)})^T \tag{236}$$

The gradient of the loss with respect to the biases $\mathbf{b}^{(l)}$ is:

$$\frac{\partial J(\theta)}{\partial \mathbf{b}^{(l)}} = \frac{1}{N} \sum_{i=1}^{N} \delta^{(l)} \tag{237}$$

After computing these gradients, we update the parameters using an optimization algorithm such as gradient descent. The weight update rule is:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial J(\theta)}{\partial \mathbf{W}^{(l)}}, \tag{238}$$

and the bias update rule is:

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \frac{\partial J(\theta)}{\partial \mathbf{b}^{(l)}} \tag{239}$$

where η is the learning rate controlling the step size in the gradient descent update. This process of forward pass, backpropagation, and parameter update is repeated over multiple epochs, with each epoch consisting of a forward pass, a backward pass, and a parameter update, until the network converges to a local minimum of the loss function.

At each step of backpropagation, the chain rule is applied recursively to propagate the error backward through the network, adjusting each weight and bias to minimize the total loss. The derivative of the activation function $\sigma^{(l)'}(z^{(l)})$ is critical, as it dictates how the error is modulated at each layer. Depending on the choice of activation function (e.g., ReLU, sigmoid, or tanh), the derivative will take different forms, and this choice has a direct impact on the learning dynamics and convergence rate of the network. Thus, backpropagation serves as the computational backbone of neural network training. By calculating the gradients of the loss function with respect to the network parameters through efficient error propagation, backpropagation allows the network to adjust its parameters iteratively, gradually minimizing the error and improving its performance across tasks. This process is mathematically rigorous, utilizing fundamental principles of calculus and optimization, ensuring that the neural network learns effectively from its training data.

6.2 Gradient Descent Variants

The training of neural networks using gradient descent and its variants is a mathematically intensive process that aims to minimize a differentiable scalar loss function $\mathcal{L}(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the parameter vector of the neural network. The loss function is often expressed as

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i), \tag{240}$$

where $(\boldsymbol{x}_i, y_i)$ are the input-output pairs in the training dataset of size N, and $\ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)$ is the sample-specific loss. The minimization problem is solved iteratively, starting from an initial guess $\boldsymbol{\theta}^{(0)}$ and updating according to the rule

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}), \tag{241}$$

where $\eta > 0$ is the learning rate, and $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ is the gradient of the loss with respect to $\boldsymbol{\theta}$. The gradient, computed via backpropagation, follows the chain rule and propagates through the network's layers to adjust weights and biases optimally. In a feedforward neural network with L layers, the computations proceed as follows. The input to layer l is

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}, \tag{242}$$

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ and $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_l}$ are the weight matrix and bias vector for the layer, respectively, and $\boldsymbol{a}^{(l-1)}$ is the activation vector from the previous layer. The output is then

$$\boldsymbol{a}^{(l)} = f^{(l)}(\boldsymbol{z}^{(l)}), \tag{243}$$

where $f^{(l)}$ is the activation function. Backpropagation begins with the computation of the error at the output layer,

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \ell}{\partial \boldsymbol{a}^{(L)}} \odot f'^{(L)}(\boldsymbol{z}^{(L)}), \tag{244}$$

where $f'^{(L)}(\cdot)$ is the derivative of the activation function. For hidden layers, the error propagates recursively as

$$\boldsymbol{\delta}^{(l)} = (\boldsymbol{W}^{(l+1)})^{\top} \boldsymbol{\delta}^{(l+1)} \odot f'^{(l)}(\boldsymbol{z}^{(l)}). \tag{245}$$

The gradients for weight and bias updates are then computed as

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{a}^{(l-1)})^\top \tag{246}$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)}, \tag{247}$$

respectively. The dynamics of gradient descent are deeply influenced by the curvature of the loss surface, encapsulated by the Hessian matrix

$$\boldsymbol{H}(\boldsymbol{\theta}) = \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}). \tag{248}$$

For a small step size η, the change in the loss function can be approximated as

$$\Delta \mathcal{L} \approx -\eta \|\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})\|^2 + \frac{\eta^2}{2} (\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}))^\top \boldsymbol{H}(\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}). \tag{249}$$

This reveals that convergence is determined not only by the gradient magnitude but also by the curvature of the loss surface along the gradient direction. The eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_d$ of $\boldsymbol{H}(\boldsymbol{\theta})$ dictate the local geometry, with large condition numbers $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ slowing convergence due to ill-conditioning. Stochastic gradient descent (SGD) modifies the standard gradient descent by computing updates based on a single data sample $(\boldsymbol{x}_i, y_i)$, leading to

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i). \tag{250}$$

While SGD introduces variance into the updates, this stochasticity helps escape saddle points characterized by zero gradient but mixed curvature. To balance computational efficiency and stability, mini-batch SGD computes gradients over a randomly selected subset $\mathcal{B} \subset \{1, \ldots, N\}$ of size $|\mathcal{B}|$, yielding

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i). \tag{251}$$

Momentum methods enhance convergence by incorporating a memory of past gradients. The velocity term

$$\boldsymbol{v}^{(k+1)} = \gamma \boldsymbol{v}^{(k)} + \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \tag{252}$$

accumulates gradient information, and the parameter update is

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \boldsymbol{v}^{(k+1)}. \tag{253}$$

Analyzing momentum in the eigenspace of $\boldsymbol{H}(\boldsymbol{\theta})$, with $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$, reveals that the effective step size in each eigendirection is

$$\eta_{\text{eff},i} = \frac{\eta}{1 - \gamma \lambda_i}, \tag{254}$$

showing that momentum accelerates convergence in low-curvature directions while damping oscillations in high-curvature directions. Adaptive gradient methods, such as AdaGrad, RMSProp, and Adam, refine learning rates for individual parameters. In AdaGrad, the adaptive learning rate is

$$\eta_i^{(k+1)} = \frac{\eta}{\sqrt{\boldsymbol{G}_{ii}^{(k+1)} + \epsilon}}, \tag{255}$$

where

$$\boldsymbol{G}_{ii}^{(k+1)} = \boldsymbol{G}_{ii}^{(k)} + \left(\nabla_{\boldsymbol{\theta}_i} \mathcal{L}(\boldsymbol{\theta})\right)^2. \tag{256}$$

RMSProp modifies this with an exponentially weighted average

$$\boldsymbol{G}_{ii}^{(k+1)} = \beta \boldsymbol{G}_{ii}^{(k)} + (1 - \beta) \left(\nabla_{\boldsymbol{\theta}_i} \mathcal{L}(\boldsymbol{\theta})\right)^2. \tag{257}$$

Adam combines RMSProp with momentum, where the first and second moments are

$$\boldsymbol{m}^{(k+1)} = \beta_1 \boldsymbol{m}^{(k)} + (1 - \beta_1)\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \tag{258}$$

and

$$\boldsymbol{v}^{(k+1)} = \beta_2 \boldsymbol{v}^{(k)} + (1 - \beta_2)\left(\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})\right)^2. \tag{259}$$

Bias corrections yield

$$\hat{\boldsymbol{m}}^{(k+1)} = \frac{\boldsymbol{m}^{(k+1)}}{1 - \beta_1^k}, \quad \hat{\boldsymbol{v}}^{(k+1)} = \frac{\boldsymbol{v}^{(k+1)}}{1 - \beta_2^k}. \tag{260}$$

The final parameter update is

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \frac{\hat{\boldsymbol{m}}^{(k+1)}}{\sqrt{\hat{\boldsymbol{v}}^{(k+1)}} + \epsilon}. \tag{261}$$

In conclusion, gradient descent and its variants provide a rich framework for optimizing neural network parameters. While standard gradient descent offers a basic approach, advanced methods like momentum and adaptive gradients significantly enhance convergence by tailoring updates to the landscape of the loss surface and the dynamics of training.

6.2.1 SGD (Stochastic Gradient Descent) Optimizer

Literature Review: Lauand and Meyn (2025) [175] established a theoretical framework for SGD using Markovian dynamics to improve convergence properties. It integrates quasi-periodic linear systems into SGD, enhancing its robustness in non-stationary environments. Maranjyan et al. (2025) [176] developed an asynchronous SGD algorithm that meets the theoretical lower bounds for time complexity. It introduces ring-based communication to optimize parallel execution without degrading convergence rates. Gao and Gündüz (2025) [177] proposed a stochastic gradient descent-based approach to optimize graph neural networks in wireless networks. It rigorously analyzes the stochastic optimization problem and proves its convergence guarantees. Yoon et. al. (2025) [178] investigated federated SGD in multi-agent learning and derives theoretical guarantees on its communication efficiency while achieving equilibrium. Verma and Maiti (2025) [179] proposed a periodic learning rate (using sine and cosine functions) for SGD-based optimizers, theoretically proving its benefits in stability and computational efficiency. Borowski and Miasojedow (2025) [180] extended the Robbins-Monro theorem to analyze convergence guarantees of SGD, refining the theoretical understanding of projected stochastic approximation algorithms. Dong et al. (2025) [181] applied stochastic gradient descent to brain network modeling, providing a theoretical framework for optimizing neural control strategies. Jiang et. al. (2025) [182] analyzed the bias-variance tradeoff in decentralized SGD, proving convergence rates and proposing an error-correction mechanism for biased gradients. Sonobe et. al. (2025) [183] connected SGD with Bayesian inference, presenting a theoretical analysis of how stochastic optimization methods approximate posterior distributions. Zhang and Jia (2025) [184] examined the theoretical properties of policy gradients in reinforcement learning, proving convergence guarantees for stochastic optimal control problems.

The **Stochastic Gradient Descent (SGD) optimizer** is an iterative method designed to minimize an objective function $f(\mathbf{w})$ by updating a parameter vector $\mathbf{w}$ in the direction of the negative gradient. The fundamental optimization problem can be expressed as

$$\min_{\mathbf{w}} f(\mathbf{w}), \tag{262}$$

where

$$f(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\mathbf{w}; \mathbf{x}_i, y_i) \tag{263}$$

represents the empirical risk, constructed from a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$. Here, $\ell(\mathbf{w}; \mathbf{x}_i, y_i)$ denotes the loss function, $\mathbf{w} \in \mathbb{R}^d$ is the parameter vector, N is the dataset size, and $f(\mathbf{w})$ approximates the true population risk

$$\mathbb{E}_{\mathbf{x},y}[\ell(\mathbf{w}; \mathbf{x}, y)]. \tag{264}$$

Standard gradient descent involves the update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla f(\mathbf{w}^{(t)}), \tag{265}$$

where $\eta > 0$ is the learning rate and

$$\nabla f(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \nabla \ell(\mathbf{w}; \mathbf{x}_i, y_i) \tag{266}$$

is the full gradient. However, for large-scale datasets, the computation of $\nabla f(\mathbf{w})$ becomes computationally prohibitive, motivating the adoption of stochastic approximations. The stochastic approximation relies on the idea of estimating the gradient $\nabla f(\mathbf{w})$ using a single data point or a small batch of data points. Denoting the random index sampled at iteration t as i_t, the stochastic gradient can be written as

$$\widehat{\nabla} f(\mathbf{w}^{(t)}) = \nabla \ell(\mathbf{w}^{(t)}; \mathbf{x}_{i_t}, y_{i_t}). \tag{267}$$

Consequently, the update rule becomes

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \widehat{\nabla} f(\mathbf{w}^{(t)}). \tag{268}$$

For a mini-batch $\mathcal{B}_t$ of size m, the stochastic gradient generalizes to

$$\widehat{\nabla} f(\mathbf{w}^{(t)}) = \frac{1}{m} \sum_{i \in \mathcal{B}_t} \nabla \ell(\mathbf{w}^{(t)}; \mathbf{x}_i, y_i). \tag{269}$$

An important property of $\widehat{\nabla} f(\mathbf{w})$ is its unbiasedness:

$$\mathbb{E}[\widehat{\nabla} f(\mathbf{w})] = \nabla f(\mathbf{w}). \tag{270}$$

However, the variance of $\widehat{\nabla} f(\mathbf{w})$, defined as

$$\mathrm{Var}[\widehat{\nabla} f(\mathbf{w})] = \mathbb{E}[\|\widehat{\nabla} f(\mathbf{w}) - \nabla f(\mathbf{w})\|^2], \tag{271}$$

introduces stochastic noise into the updates, where $\mathrm{Var}[\widehat{\nabla} f(\mathbf{w})] \approx \frac{\sigma^2}{m}$ and

$$\sigma^2 = \mathbb{E}[\|\nabla \ell(\mathbf{w}; \mathbf{x}) - \nabla f(\mathbf{w})\|^2] \tag{272}$$

is the variance of the gradients. To analyze the convergence properties of SGD, we assume $f(\mathbf{w})$ to be L-smooth, meaning

$$\|\nabla f(\mathbf{w}_1) - \nabla f(\mathbf{w}_2)\| \leq L \|\mathbf{w}_1 - \mathbf{w}_2\|, \tag{273}$$

and $f(\mathbf{w})$ to be bounded below by $f^* = \inf_{\mathbf{w}} f(\mathbf{w})$. Using Taylor expansion, we can write

$$f(\mathbf{w}^{(t+1)}) \leq f(\mathbf{w}^{(t)}) - \eta \|\nabla f(\mathbf{w}^{(t)})\|^2 + \frac{\eta^2 L}{2} \|\widehat{\nabla} f(\mathbf{w}^{(t)})\|^2. \tag{274}$$

Taking expectations yields

$$\mathbb{E}[f(\mathbf{w}^{(t+1)})] \leq \mathbb{E}[f(\mathbf{w}^{(t)})] - \frac{\eta}{2} \mathbb{E}[\|\nabla f(\mathbf{w}^{(t)})\|^2] + \frac{\eta^2 L}{2} \sigma^2, \tag{275}$$

showing that the convergence rate depends on the interplay between the learning rate η, the smoothness constant L, and the gradient variance σ^2. For η small enough, the dominant term in convergence is $-\frac{\eta}{2}\mathbb{E}[\|\nabla f(\mathbf{w}^{(t)})\|^2]$, leading to monotonic decrease in $f(\mathbf{w}^{(t)})$. In the strongly convex case, where $f(\mathbf{w})$ satisfies

$$f(\mathbf{w}_1) \geq f(\mathbf{w}_2) + \nabla f(\mathbf{w}_2)^\top(\mathbf{w}_1 - \mathbf{w}_2) + \frac{\mu}{2}\|\mathbf{w}_1 - \mathbf{w}_2\|^2 \tag{276}$$

for $\mu > 0$, SGD achieves linear convergence. Specifically,

$$\mathbb{E}[\|\mathbf{w}^{(t)} - \mathbf{w}^*\|^2] \leq (1 - \eta\mu)^t\|\mathbf{w}^{(0)} - \mathbf{w}^*\|^2 + \frac{\eta\sigma^2}{2\mu}. \tag{277}$$

For non-convex functions, where $\nabla^2 f(\mathbf{w})$ can have both positive and negative eigenvalues, SGD may converge to a local minimizer or saddle point. Stochasticity plays a pivotal role in escaping strict saddle points $\mathbf{w}_s$ where $\nabla f(\mathbf{w}_s) = 0$ but $\lambda_{\min}(\nabla^2 f(\mathbf{w}_s)) < 0$.

6.2.2 Adam (Adaptive Moment Estimation) Optimizer

Literature Review: Kingma and Ba (2014) [165] introduced the Adam optimizer. It presents Adam as an adaptive gradient-based optimization method that combines momentum and adaptive learning rate techniques. The authors rigorously prove its advantages over traditional optimizers such as SGD and RMSProp. Reddy et. al. (2019) [166] analyzed the convergence properties of Adam and identified cases where it may fail to converge. The authors propose AMSGrad, an improved variant of Adam that guarantees better theoretical convergence behavior. Jin et. al. (2024) [167] introduced MIAdam (Multiple Integral Adam), which modified Adam's update rules to enhance generalization. The authors theoretically and empirically demonstrate its effectiveness in avoiding sharp minima. Adly et. al. (2024) [168] proposed EXAdam, an improvement over Adam that uses cross-moments in parameter updates. This leads to faster convergence while maintaining the adaptability of Adam. Theoretical derivations show improved variance reduction in updates. Liu et. al. (2024) [169] provided a rigorous mathematical proof of convergence for Adam when applied to linear inverse problems. The authors compare Adam's convergence rate with standard gradient descent and prove its efficiency in noisy settings. Yang (2025) [170] generalized Adam by introducing a biased stochastic optimization framework. The authors show that under specific conditions, Adam's bias correction step is insufficient, leading to poor convergence on strongly convex functions. Park and Lee (2024) [171] developed SMMF, a novel variant of Adam that factorizes momentum tensors, reducing memory usage. Theoretical bounds show that SMMF preserves Adam's adaptability while improving efficiency. Mahjoubi et al. (2025) [172] provided a comparative analysis of Adam, SGD, and RMSProp in deep learning models. It demonstrates scenarios where Adam outperforms other methods, particularly in high-dimensional optimization problems. Seini and Adam (2024) [173] examined how Adam's optimization framework can be adapted to human-AI collaborative learning models. The paper provides a theoretical foundation for integrating Adam into AI-driven education platforms. Teessar (2024) [174] discussed Adam's application in survey and social science research, where adaptive optimization is used to fine-tune questionnaire analysis models. This highlights Adam's versatility outside deep learning.

The Adaptive Moment Estimation (Adam) optimizer can be considered a sophisticated, hybrid optimization algorithm combining elements of momentum-based methods and adaptive learning rate techniques, which is why it has become a cornerstone in the optimization of complex machine learning models, particularly those used in deep learning. Adam's formulation is centered on computing and using both the first and second moments (i.e., the mean and the variance) of the gradient with respect to the loss function at each parameter update. This process effectively adapts the learning rate for each parameter, based on its respective gradient's statistical properties. The moment-based adjustments provide robustness against issues such as poor conditioning of the objective function and gradient noise, which are prevalent in large-scale optimization problems.

Mathematically, at each iteration t, the Adam optimizer updates the parameter vector $\theta_t \in \mathbb{R}^n$, where n is the number of parameters of the model, based on the gradient g_t, which is the gradient of the objective function with respect to θ_t, i.e., $g_t = \nabla_\theta f(\theta_t)$. In its essence, Adam computes two distinct quantities: the first moment estimate m_t and the second moment estimate v_t, which are recursive moving averages of the gradients and the squared gradients, respectively. The first moment estimate m_t is given by

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{278}$$

where $\beta_1 \in [0, 1)$ is the decay rate for the first moment. This recurrence equation represents a weighted moving average of the gradients, which is intended to capture the *directional momentum* in the optimization process. By incorporating the first moment, Adam accumulates information about the historical gradients, which helps mitigate oscillations and stabilizes the convergence direction. The term $(1 - \beta_1)$ ensures that the most recent gradient g_t receives a more significant weight in the computation of m_t. Similarly, the second moment estimate v_t, which represents the exponentially decaying average of the squared gradients, is updated as

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{279}$$

where $\beta_2 \in [0, 1)$ is the decay rate for the second moment. This moving average of squared gradients captures the *variance* of the gradient at each iteration. The second moment v_t thus acts as an estimate of the *curvature* of the objective function, which allows the optimizer to adjust the step size for each parameter accordingly. Specifically, large values of v_t correspond to parameters that experience high gradient variance, signaling a need for smaller updates to prevent overshooting, while smaller values of v_t correspond to parameters with low gradient variance, where larger updates are appropriate. This mechanism is akin to automatically tuning the learning rate for each parameter based on the local geometry of the loss function. At initialization, both m_t and v_t are typically set to zero. This initialization introduces a bias toward zero, particularly at the initial time steps, causing the estimates of the moments to be somewhat underrepresented in the early iterations. To correct for this bias, *bias correction* terms are introduced. The bias-corrected first moment $\hat{m}_t$ is given by

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \tag{280}$$

and the bias-corrected second moment $\hat{v}_t$ is given by

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \tag{281}$$

The purpose of these corrections is to offset the initial tendency of m_t and v_t to underestimate the true values due to their initialization at zero. As the iteration progresses, the bias correction terms become less significant, and the estimates of the moments converge to their true values, allowing for more accurate parameter updates. The actual update rule for the parameters θ_t is determined by using the bias-corrected first and second moment estimates $\hat{m}_t$ and $\hat{v}_t$, respectively. The update equation is given by

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{282}$$

where η is the global learning rate, and ϵ is a small constant (typically 10^{-8}) added to the denominator for numerical stability. This update rule incorporates both the momentum (through $\hat{m}_t$) and the adaptive learning rate (through $\hat{v}_t$). The factor $\sqrt{\hat{v}_t} + \epsilon$ is particularly crucial as it ensures that parameters with large gradient variance (i.e., those with large values in v_t) receive smaller updates, whereas parameters with smaller gradient variance (i.e., those with small values in v_t) receive larger updates, thus preventing divergence in high-variance regions.

The learning rate adjustment in Adam is dynamic in nature, as it is controlled by the second moment estimate $\hat{v}_t$, which means that Adam has a per-parameter learning rate for each parameter. For each parameter, the learning rate is inversely proportional to the square root of its corresponding second moment estimate $\hat{v}_t$, leading to *adaptive learning rates*. This is what enables Adam to operate effectively in highly non-convex optimization landscapes, as it reduces the learning rate in directions where the gradient exhibits high variance, thus stabilizing the updates, and increases the learning rate where the gradient variance is low, speeding up convergence. In the case where Adam is applied to convex objective functions, convergence can be analyzed mathematically. Under standard assumptions, such as bounded gradients and a decreasing learning rate, the convergence of Adam can be shown by proving that

$$\sum_{t=1}^{\infty} \eta_t^2 < \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t = \infty, \tag{283}$$

where η_t is the learning rate at time step t. The first condition ensures that the learning rate decays sufficiently rapidly to guarantee convergence, while the second ensures that the learning rate does not decay too quickly, allowing for continual updates as the algorithm progresses. However, Adam is not without its limitations. One notable issue arises from the fact that the second moment estimate v_t may decay too quickly, causing overly aggressive updates in regions where the gradient variance is relatively low. To address this, the AMSGrad variant was introduced. AMSGrad modifies the second moment update rule by replacing v_t with

$$\hat{v}_t^{\text{new}} = \max(\hat{v}_{t-1}, \hat{v}_t), \tag{284}$$

thereby ensuring that $\hat{v}_t$ never decreases, which helps prevent the optimizer from making overly large updates in situations where the second moment estimate may be miscalculated. By forcing $\hat{v}_t$ to increase or remain constant, AMSGrad reduces the chance of large, destabilizing parameter updates, thereby improving the stability and convergence of the optimizer, particularly in difficult or ill-conditioned optimization problems. Additionally, further extensions of Adam, such as AdaBelief, introduce additional modifications to the second moment estimate by introducing a belief-based mechanism to correct the moment estimates. Specifically, AdaBelief estimates the second moment $\hat{v}_t$ in a way that adjusts based on the *belief* in the direction of the gradient, offering further stability in cases where gradients may be sparse or noisy. These innovations underscore the flexibility of Adam and its variants in optimizing complex loss functions across a range of machine learning tasks.

Ultimately, the Adam optimizer stands as a highly sophisticated, mathematically rigorous optimization algorithm, effectively combining momentum and adaptive learning rates. By using both the first and second moments of the gradient, Adam dynamically adjusts the parameter updates, providing a robust and efficient optimization framework for non-convex, high-dimensional objective functions. The use of bias correction, coupled with the adaptive nature of the optimizer, allows it to operate effectively across a wide range of problem settings, making it a go-to method for many machine learning and deep learning applications. The mathematical rigor behind Adam ensures that it remains a highly stable and efficient optimization technique, capable of overcoming many of the challenges posed by large-scale and noisy gradient information in machine learning models.

6.2.3 RMSProp (Root Mean Squared Propagation) Optimizer

Literature Review: Bensaid et. al. (2024) [155] provides a rigorous analysis of the convergence properties of RMSProp under non-convex settings. It utilizes stability theory to examine how RMSProp adapts to different loss landscapes and demonstrates how adaptivity plays a crucial role in ensuring convergence. The study offers theoretical insights into the efficiency of RMSProp in smoothing out noisy gradients. Liu and Ma (2024) [156] investigated loss oscillations observed in adaptive optimizers, including RMSProp. It explains how RMSProp's exponential moving average mechanism contributes to this phenomenon and proposes a novel perspective on tuning hyperparameters to mitigate oscillations. Li (2024) [157] explored the fundamental theoretical properties of adaptive optimizers, with a special focus on RMSProp. It rigorously examines the interplay between smoothness conditions and the adaptive nature of RMSProp, showing how it balances stability and convergence speed. Heredia (2024) [158] presented a new mathematical framework for analyzing RMSProp using integro-differential equations. The model provides deeper theoretical insights into how RMSProp updates gradients differently from AdaGrad and Adam, particularly in terms of gradient smoothing. Ye (2024) [159] discussed how preconditioning methods, including RMSProp, enhance gradient descent optimization. It explains why RMSProp's adaptive learning rate is beneficial in high-dimensional settings and provides a theoretical justification for its effectiveness in regularized optimization problems. Compagnoni et. al. (2024) [160] employed stochastic differential equations (SDEs) to model the behavior of RMSProp and other adaptive optimizers. It provides new theoretical insights into how noise affects the optimization process and how RMSProp adapts to different gradient landscapes. Yao et. al. (2024) [161] presented a system response curve analysis of first-order optimization methods, including RMSProp. The authors develop a dynamic equation for RMSProp that explains its stability and effectiveness in deep learning tasks. Wen and Lei (2024) [162] explored an alternative optimization framework that integrates RMSProp-style updates with an ADMM approach. It provides theoretical guarantees for the convergence of RMSProp in non-convex optimization problems. Hannibal et. al. (2024) [163] critiques the convergence properties of popular optimizers, including RMSProp. It rigorously proves that in certain settings, RMSProp may not lead to a global minimum, emphasizing the importance of hyperparameter tuning. Yang (2025) [164] extended the theoretical understanding of adaptive optimizers like RMSProp by analyzing the impact of bias in stochastic gradient updates. It provides a rigorous mathematical treatment of how bias affects convergence.

The Root Mean Squared Propagation (RMSProp) optimizer is a sophisticated variant of the gradient descent algorithm that adapts the learning rate for each parameter in a non-linear, non-convex optimization problem. The fundamental issue with standard gradient descent lies in the constant learning rate η, which fails to account for the varying magnitudes of the gradients in different directions of the parameter space. This lack of adaptation can cause inefficient optimization, where large gradients may lead to overshooting and small gradients lead to slow convergence. RMSProp addresses this problem by dynamically adjusting the learning rate based on the historical gradient magnitudes, offering a more tailored and efficient approach. Consider the objective function $f(\theta)$, where $\theta \in \mathbb{R}^n$ is the vector of parameters that we aim to optimize. Let $\nabla f(\theta)$ denote the gradient of $f(\theta)$ with respect to θ, which is a vector of partial derivatives:

$$\nabla f(\theta) = \left[\frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, \dots, \frac{\partial f(\theta)}{\partial \theta_n}\right]^T. \tag{285}$$

In traditional gradient descent, the update rule for θ is:

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t), \tag{286}$$

where η is the learning rate, a scalar constant. However, this approach does not account for the fact that the gradient magnitudes may differ significantly along different directions in the parameter space, especially in high-dimensional, non-convex functions. The RMSProp optimizer introduces a solution by adapting the learning rate for each parameter in proportion to the magnitude of the historical gradients. The key modification in RMSProp is the introduction of a running average of the squared gradients for each parameter θ_i, denoted as $E[g^2]_{i,t}$, which captures the cumulative magnitude of the gradients over time. The update rule for $E[g^2]_{i,t}$ is given by the exponential moving average formula:

$$E[g^2]_{i,t} = \beta E[g^2]_{i,t-1} + (1-\beta) g_{i,t}^2, \tag{287}$$

where $g_{i,t} = \frac{\partial f(\theta_t)}{\partial \theta_i}$ is the gradient of the objective function with respect to the parameter θ_i at time step t, and β is the decay factor, typically set close to 1 (e.g., $\beta = 0.9$). This recurrence relation allows the gradient history to influence the current update while exponentially forgetting older gradient information. The value of β determines the memory of the squared gradients, where higher values of β give more weight to past gradients. The update for θ_i in RMSProp is then given by:

$$\theta_{i,t+1} = \theta_{i,t} - \frac{\eta}{\sqrt{E[g^2]_{i,t} + \epsilon}} g_{i,t}, \tag{288}$$

where ϵ is a small positive constant (typically $\epsilon = 10^{-8}$) introduced to avoid division by zero and ensure numerical stability. The term $\frac{1}{\sqrt{E[g^2]_{i,t}+\epsilon}}$ dynamically adjusts the learning rate for each parameter based on the magnitude of the squared gradient history. This adjustment allows RMSProp to take larger steps in directions where gradients have historically been small, and smaller steps in directions where gradients have been large, leading to a more stable and efficient optimization process.

Mathematically, the key advantage of RMSProp over traditional gradient descent lies in its ability to adapt the learning rate according to the local geometry of the objective function. In regions where the objective function is steep (large gradients), RMSProp reduces the effective learning rate by dividing by $\sqrt{E[g^2]_{i,t}}$, mitigating the risk of overshooting. Conversely, in flatter regions with smaller gradients, RMSProp increases the learning rate, allowing for faster convergence. This self-adjusting mechanism is crucial in high-dimensional optimization tasks, where the gradients along different directions can vary greatly in magnitude, as is often the case in deep learning tasks involving neural networks. The exponential moving average of squared gradients used in RMSProp is analogous to a form of local normalization, where each parameter is scaled by the inverse of the running average of its gradient squared. This normalization ensures that the optimizer does not become overly sensitive to gradients in any particular direction, thus stabilizing the optimization process. In more formal terms, if the objective function $f(\theta)$ exhibits sharp curvatures along certain directions, RMSProp mitigates the effects of such curvatures by scaling down the step size along those directions. This scaling behavior can be interpreted as a form of gradient re-weighting, where the influence of each parameter's gradient is modulated by its historical behavior, making the optimizer more robust to ill-conditioned optimization problems. The introduction of ϵ ensures that the denominator never becomes zero, even in the case where the squared gradient history for a parameter θ_i becomes extremely small. This is crucial for maintaining the numerical stability of the algorithm, particularly in scenarios where gradients may vanish or grow exceedingly small over many iterations, as seen in certain deep learning applications, such as training very deep neural networks. By providing a small non-zero lower bound to the learning rate, ϵ ensures that the updates remain smooth and predictable.

RMSProp's performance is heavily influenced by the choice of β, which controls the trade-off between long-term history and recent gradient information. When β is close to 1, the optimizer relies more heavily on the historical gradients, which is useful for capturing long-term trends in the optimization landscape. On the other hand, smaller values of β allow the optimizer to be more responsive to recent gradient changes, which can be beneficial in highly non-stationary environments or rapidly changing optimization landscapes. In the context of deep learning, RMSProp is particularly effective for optimizing objective functions with complex, high-dimensional parameter spaces, such as those encountered in training deep neural networks. The non-convexity of such objective functions often leads to a gradient that can vary significantly in magnitude across different layers of the network. RMSProp helps to balance the updates across these layers by adjusting the learning rate based on the historical gradients, ensuring that all layers receive appropriate updates without being dominated by large gradients from any single layer. This adaptability helps in preventing gradient explosions or vanishing gradients, which are common issues in deep learning optimization. In summary, RMSProp provides a robust and efficient optimization technique by adapting the learning rate based on the historical squared gradients of each parameter. The exponential decay of the squared gradient history allows RMSProp to strike a balance between stability and adaptability, preventing overshooting and promoting faster convergence in non-convex optimization problems. The introduction of ϵ ensures numerical stability, and the parameter β offers flexibility in controlling the influence of past gradients. This makes RMSProp particularly well-suited for high-dimensional optimization tasks, especially in deep learning applications, where the parameter space is vast, and gradient magnitudes can differ significantly across dimensions. By effectively normalizing the gradients and dynamically adjusting the learning rates, RMSProp significantly enhances the efficiency and stability of gradient-based optimization methods.

6.3 Overfitting and Regularization Techniques

Literature Review: Goodfellow (2016) et. al. [112] provides a comprehensive introduction to deep learning, including a thorough discussion on overfitting and regularization techniques. It explains methods such as L1/L2 regularization, dropout, batch normalization, and data augmentation, which help improve generalization. The authors explore the bias-variance tradeoff and practical solutions to reduce overfitting in neural networks. Hastie et. al. (2009) [129] discusses overfitting in statistical learning models, particularly in regression and classification. The book covers regularization techniques like Ridge Regression (L2) and Lasso (L1), as well as cross-validation techniques for preventing overfitting. It is fundamental for understanding model complexity control in machine learning. Bishop (2006) [115] in his book provided an in-depth mathematical foundation of machine learning models, with particular attention to regularization methods such as Bayesian inference, early stopping, and weight decay. It emphasized probabilistic interpretations of regularization, demonstrating how overfitting can be mitigated through prior distributions in Bayesian models. Murphy (2012) [130] in his book presents a Bayesian approach to machine learning, covering regularization techniques from a probabilistic viewpoint. It discusses penalization methods, Bayesian regression, and variational inference as tools to control model complexity and prevent overfitting. The book is useful for those looking to understand uncertainty estimation in ML models. Srivastava et. al. (2014) [131] introduced Dropout, a widely used regularization technique in deep learning. The authors show how randomly dropping units during training reduces co-adaptation of neurons, thereby enhancing model generalization. This technique remains a key part of modern neural network training pipelines. Zou and Hastie (2005) [132] introduced Elastic Net, a combination of L1 (Lasso) and L2 (Ridge) regularization, which addresses the limitations of Lasso in handling correlated features. It is particularly useful for high-dimensional data, where feature selection and regularization are crucial. Vapnik (1995) [133] in his introduced Statistical Learning Theory and the VC-dimension, which quantifies model complexity. It provides the mathematical framework explaining why overfitting occurs and how regularization constraints reduce generalization error. It forms the theoretical basis of Support Vector Machines (SVMs) and Structural Risk Minimization. Ng (2004) [134] compares L1 (Lasso) and L2 (Ridge) regularization, demonstrating their impact on feature selection and model stability. It shows that L1 regularization is more effective for sparse models, whereas L2 preserves information better in highly correlated feature spaces. This work is essential for choosing the right regularization technique for specific datasets. Li (2025) [135] explored regularization techniques in high-dimensional clinical trial data using ensemble methods, Bayesian optimization, and deep learning regularization techniques. It highlights the practical application of regularization to prevent overfitting in medical AI. Yasuda (2025) [136] focused on regularization in hybrid machine learning models, specifically Gaussian–Discrete RBMs. It extends L1/L2 penalties and dropout strategies to improve the generalization of deep generative models. It's valuable for those working on deep learning architectures and unsupervised learning.

Overfitting in neural networks is a critical issue where the model learns to excessively fit the training data, capturing not just the true underlying patterns but also the noise and anomalies present in the data. This leads to poor generalization to unseen data, resulting in a model that has a low training error but a high test error. Mathematically, consider a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i \in \mathbb{R}^d$ represents the input feature vector for each data point, and $y_i \in \mathbb{R}$ represents the corresponding target value. The goal is to fit a neural network model $f(\mathbf{x}; \mathbf{w})$ parameterized by weights $\mathbf{w} \in \mathbb{R}^M$, where M denotes the number of parameters in the model. The model's objective is to minimize the empirical risk, given by the mean squared error between the predicted values and the true target values:

$$\hat{R}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} L(f(\mathbf{x}_i; \mathbf{w}), y_i) \tag{289}$$

where L denotes the loss function, typically the squared error $L(\hat{y}_i, y_i) = (\hat{y}_i - y_i)^2$. In this framework, the neural network tries to minimize the empirical risk on the training set. However, the true goal is to minimize the expected risk $R(\mathbf{w})$, which reflects the model's performance on the true distribution $P(\mathbf{x}, y)$ of the data. This expected risk is given by:

$$R(\mathbf{w}) = \mathbb{E}_{\mathbf{x},y}[L(f(\mathbf{x}; \mathbf{w}), y)] \tag{290}$$

Overfitting occurs when the model minimizes $\hat{R}(\mathbf{w})$ to an excessively small value, but $R(\mathbf{w})$ remains large, indicating that the model has fit the noise in the training data, rather than capturing the true data distribution. This discrepancy arises from an overly complex model that learns to memorize the training data rather than generalizing across different inputs. A fundamental insight into the overfitting phenomenon comes from the bias-variance decomposition of the generalization error. The total error in a model's prediction $\hat{f}(\mathbf{x})$ of the true target function $g(\mathbf{x})$ can be decomposed as:

$$\mathcal{E} = \mathbb{E}[(g(\mathbf{x}) - \hat{f}(\mathbf{x}))^2] = \text{Bias}^2(\hat{f}(\mathbf{x})) + \text{Var}(\hat{f}(\mathbf{x})) + \sigma^2 \tag{291}$$

where $\text{Bias}^2(\hat{f}(\mathbf{x}))$ represents the squared difference between the expected model prediction and the true function, $\text{Var}(\hat{f}(\mathbf{x}))$ is the variance of the model's predictions across different training sets, and σ^2 is the irreducible error due to the intrinsic noise in the data. In the context of overfitting, the model typically exhibits low bias (as it fits the training data very well) but high variance (as it is highly sensitive to the fluctuations in the training data). Therefore, regularization techniques aim to reduce the variance of the model while maintaining its ability to capture the true underlying relationships in the data, thereby improving generalization. One of the most popular methods to mitigate overfitting is **L2 regularization** (also known as weight decay), which adds a penalty term to the loss function based on the squared magnitude of the weights. The regularized loss function is given by:

$$\hat{R}_{\text{reg}}(\mathbf{w}) = \hat{R}(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2 = \hat{R}(\mathbf{w}) + \lambda \sum_{j=1}^{M} w_j^2 \tag{292}$$

where λ is a positive constant controlling the strength of the regularization. The gradient of the regularized loss function with respect to the weights is:

$$\nabla_{\mathbf{w}} \hat{R}_{\text{reg}}(\mathbf{w}) = \nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) + 2\lambda \mathbf{w} \tag{293}$$

The term $2\lambda\mathbf{w}$ introduces weight shrinkage, which discourages the model from fitting excessively large weights, thus preventing overfitting by reducing the model's complexity. This regularization approach is a direct way to control the model's capacity by penalizing large weight values, leading to a simpler model that generalizes better. In contrast, **L1 regularization** adds a penalty based on the absolute values of the weights:

$$\hat{R}_{\text{reg}}(\mathbf{w}) = \hat{R}(\mathbf{w}) + \lambda \|\mathbf{w}\|_1 = \hat{R}(\mathbf{w}) + \lambda \sum_{j=1}^{M} |w_j| \tag{294}$$

The gradient of the L1 regularized loss function is:

$$\nabla_{\mathbf{w}} \hat{R}_{\text{reg}}(\mathbf{w}) = \nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) + \lambda \,\text{sgn}(\mathbf{w}) \tag{295}$$

where $\text{sgn}(\mathbf{w})$ denotes the element-wise sign function. L1 regularization has a unique property of inducing sparsity in the weights, meaning it drives many of the weights to exactly zero, effectively selecting a subset of the most important features. This feature selection mechanism is particularly useful in high-dimensional settings, where many input features may be irrelevant. A more advanced regularization technique is **dropout**, which randomly deactivates a fraction of neurons during training. Let $\mathbf{h}_i$ represent the activation of the i-th neuron in a given layer. During training, dropout produces a binary mask $\mathbf{m}_i$ sampled from a Bernoulli distribution with success probability p, i.e., $m_i \sim \text{Bernoulli}(p)$, such that:

$$\mathbf{h}_i^{\text{drop}} = \frac{1}{p} \mathbf{m}_i \odot \mathbf{h}_i \tag{296}$$

where $\odot$ denotes element-wise multiplication. The factor $1/p$ ensures that the expected value of the activations remains unchanged during training. Dropout effectively forces the network to learn redundant representations, reducing its reliance on specific neurons and promoting better generalization. By training an ensemble of subnetworks with shared weights, dropout helps to prevent the network from memorizing the training data, thus reducing overfitting. **Early stopping** is another technique to prevent overfitting, which involves halting the training process when the validation error starts to increase. The model is trained on the training set, but its performance is evaluated on a separate validation set. If the validation error $R_{\text{val}}(t)$ increases after several epochs, training is stopped to prevent further overfitting. Mathematically, the stopping criterion is:

$$t^* = \arg\min_t R_{\text{val}}(t) \tag{297}$$

where t^* represents the epoch at which the validation error reaches its minimum. This technique avoids the risk of continuing to fit the training data beyond the point where the model starts to lose its ability to generalize. **Data augmentation** artificially enlarges the training dataset by applying transformations to the original data. Let $T = \{T_1, T_2, \ldots, T_K\}$ represent a set of transformations (such as rotations, scaling, and translations). For each training example $(\mathbf{x}_i, y_i)$, the augmented dataset $\mathcal{D}'$ consists of K new examples:

$$\mathcal{D}' = \{(T_k(\mathbf{x}_i), y_i) \mid i = 1, 2, \ldots, N, k = 1, 2, \ldots, K\} \tag{298}$$

These transformations create new, varied examples, which help the model generalize better by preventing it from fitting too closely to the original, potentially noisy data. Data augmentation is particularly beneficial in domains like image processing, where transformations like rotations and flips do not change the underlying label but provide additional examples to learn from. **Batch normalization** normalizes the activations of each mini-batch to reduce internal covariate shift and stabilize the learning process. Given a mini-batch $\mathcal{B} = \{\mathbf{h}_i\}_{i=1}^m$ with activations $\mathbf{h}_i$, the mean and variance of the activations across the mini-batch are computed as:

$$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^m \mathbf{h}_i, \quad \sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^m (\mathbf{h}_i - \mu_{\mathcal{B}})^2 \tag{299}$$

The normalized activations are then given by:

$$\hat{\mathbf{h}}_i = \frac{\mathbf{h}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{300}$$

where ϵ is a small constant for numerical stability. Batch normalization helps to smooth the optimization landscape, allowing for faster convergence and mitigating the risk of overfitting by preventing the model from getting stuck in sharp, narrow minima in the loss landscape.

In conclusion, overfitting is a significant challenge in training neural networks, and its prevention requires a combination of techniques aimed at controlling model complexity, improving generalization, and reducing sensitivity to noise in the training data. Regularization methods such as L2 and L1 regularization, dropout, and early stopping, combined with strategies like data augmentation and batch normalization, are fundamental to improving the performance of neural networks on unseen data and ensuring that they do not overfit the training set. The mathematical formulations and optimization strategies outlined here provide a detailed and rigorous framework for understanding and mitigating overfitting in machine learning models.

6.3.1 Dropout

Dropout, a regularization technique in neural networks, is designed to address overfitting, a situation where a model performs well on training data but fails to generalize to unseen data. The general problem of overfitting in machine learning arises when a model becomes excessively complex, with a high number of parameters, and learns to model noise in the data rather than the true underlying patterns. This can result in poor generalization performance on new, unseen data. In the context of neural networks, the solution often involves *regularization techniques* to penalize complexity and prevent the model from memorizing the data. Dropout, introduced by Geoffrey Hinton et al., represents a unique and powerful method to regularize neural networks by introducing stochasticity during the training process, which forces the model to generalize better and prevents overfitting. To understand the mathematics behind dropout, let $f_\theta(x)$ represent the output of a neural network for input x with parameters θ. The goal during training is to minimize a loss function that measures the discrepancy between the predicted output and the true target y. Without any regularization, the objective is to minimize the empirical loss:

$$L_{\text{empirical}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i) \tag{301}$$

where $\mathcal{L}(f_\theta(x_i), y_i)$ is the loss function (e.g., cross-entropy or mean squared error), and N is the number of data samples. A model trained to minimize this loss function without regularization will likely overfit to the training data, capturing the noise rather than the underlying distribution of the data. Dropout addresses this by randomly "dropping out" a fraction of the network's neurons during each training iteration, which is mathematically represented by modifying the activations of neurons.

Let us consider a feedforward neural network with a set of activations a_i for the neurons in the i-th layer, which is computed as $a_i = f(Wx_i + b_i)$, where W represents the weight matrix, x_i the input to the neuron, and b_i the bias. During training with dropout, for each neuron, a random Bernoulli variable r_i is introduced, where:

$$r_i \sim \text{Bernoulli}(p) \tag{302}$$

with probability p representing the retention probability (i.e., the probability that a neuron is kept active), and $1 - p$ representing the probability that a neuron is "dropped" (set to zero). The activation of the i-th neuron is then modified as follows:

$$a_i' = r_i \cdot a_i = r_i \cdot f(Wx_i + b_i) \tag{303}$$

where r_i is a random binary mask for each neuron. During each forward pass, different neurons are randomly dropped out, and the network is effectively training on a different subnetwork, forcing the network to learn a more robust set of features that do not depend on any particular neuron. In this way, dropout acts as a form of *ensemble learning*, as each forward pass corresponds to a different realization of the network.

The mathematical expectation of the loss function with respect to the dropout mask r can be written as:

$$\mathbb{E}_r[L_{\text{dropout}}(\theta, r)] = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i, r), y_i) \tag{304}$$

where $f_\theta(x_i, r)$ is the output of the network with the dropout mask r. Since the dropout mask is random, the loss is an expectation over all possible configurations of dropout masks. This randomness induces an implicit *ensemble effect*, where the model is trained not just on a single set of parameters θ, but effectively on a *distribution* of models, each corresponding to a different dropout configuration. The model is, therefore, regularized because the network is forced to generalize across these different subnetworks, and overfitting to the training data is prevented. One way to gain deeper insight into dropout is to consider its connection with *Bayesian inference*. In the context of deep learning, dropout can be viewed as an approximation to *Bayesian posterior inference*. In Bayesian terms, we seek the posterior distribution of the network's parameters θ, given the data D, which can be written as:

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)} \tag{305}$$

where $p(D|\theta)$ is the likelihood of the data given the parameters, $p(\theta)$ is the prior distribution over the parameters, and $p(D)$ is the marginal likelihood of the data. Dropout approximates this posterior by averaging over the outputs of many different subnetworks, each corresponding to a different dropout configuration. This interpretation is formalized by observing that each forward pass with a different dropout mask corresponds to a different realization of the model, and averaging over all dropout masks gives an approximation to the Bayesian posterior. Thus, the expected output of the network, given the data x, under dropout is:

$$\mathbb{E}_r[f_\theta(x)] = \frac{1}{M}\sum_{i=1}^{M} f_\theta(x, r_i) \tag{306}$$

where r_i is a dropout mask drawn from the Bernoulli distribution and M is the number of Monte Carlo samples of dropout configurations. This expectation can be interpreted as a form of *ensemble averaging*, where each individual forward pass corresponds to a different model sampled from the posterior.

Dropout is also highly effective because it controls the *bias-variance tradeoff*. The bias-variance tradeoff is a fundamental concept in statistical learning, where increasing model complexity reduces bias but increases variance, and vice versa. A highly complex model tends to have low bias but high variance, meaning it fits the training data very well but fails to generalize to new data. Regularization techniques, such as dropout, seek to reduce variance without increasing bias excessively. Dropout achieves this by introducing stochasticity in the learning process. By randomly deactivating neurons during training, the model is forced to learn *robust features* that do not depend on the presence of specific neurons. In mathematical terms, the variance of the model's output can be expressed as:

$$\text{Var}(f_\theta(x)) = \mathbb{E}_r[(f_\theta(x))^2] - (\mathbb{E}_r[f_\theta(x)])^2 \tag{307}$$

By averaging over multiple dropout configurations, the variance is reduced, leading to better generalization performance. Although dropout introduces some bias by reducing the network's capacity (since fewer neurons are available at each step), the variance reduction outweighs the bias increase, resulting in improved generalization. Another key mathematical aspect of dropout is its relationship with *stochastic gradient descent (SGD)*. In the standard SGD framework, the parameters θ are updated using the gradient of the loss with respect to the parameters. In the case of dropout, the gradient is computed based on a *stochastic subnetwork* at each training iteration, which introduces an element of randomness into the optimization process. The parameter update rule with dropout can be written as:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathbb{E}_r[L_{\text{dropout}}(\theta, r)] \tag{308}$$

where η is the learning rate, and ∇_θ is the gradient of the loss with respect to the model parameters. The expectation is taken over all possible dropout configurations, which means that at each step, the gradient update is based on a different realization of the model. This stochasticity helps the optimization process by preventing the model from getting stuck in local minima, improving convergence towards global minima, and enhancing generalization. Finally, it is important to note that dropout has a close connection with *low-rank approximations*. During each forward pass with dropout, certain neurons are effectively removed, which reduces the rank of the weight matrix, as some rows or columns of the matrix are set to zero. This stochastic reduction in rank forces the network to learn lower-dimensional representations of the data, effectively performing *low-rank regularization*. This aspect of dropout can be formalized by observing that each dropout mask corresponds to a sparse matrix, and the network is effectively learning a low-rank approximation of the data distribution. By doing so, dropout prevents the network from learning overly complex representations that could overfit the data, leading to improved generalization.

In summary, dropout is a powerful and mathematically sophisticated regularization technique that introduces randomness into the training process. By randomly deactivating neurons during each forward pass, dropout forces the model to generalize better and prevents overfitting. Dropout can be understood as approximating Bayesian posterior inference over the model parameters and acts as a form of ensemble learning. It controls the bias-variance tradeoff, reduces variance, and improves generalization. The stochastic nature of dropout also introduces a form of noise injection during training, which aids in avoiding local minima and ensures convergence to global minima. Additionally, dropout induces low-rank regularization, which further improves generalization by preventing overly complex representations. Through these mathematical and statistical insights, dropout has become a cornerstone technique in deep learning, enhancing the performance of neural networks on unseen data.

6.3.2 L1/L2 Regularization and Overfitting

L1 and L2 regularization plays a critical role in mitigating **overfitting**. Overfitting occurs when a model fits not only the underlying data distribution but also the noise in the data, leading to poor generalization to unseen examples. Overfitting is especially prevalent in models with a large number of features, where the model becomes overly flexible and may capture spurious correlations between the features and the target variable. This often results in a model with high variance, where small fluctuations in the data cause significant changes in the model predictions. To combat this, **regularization techniques** are employed, which introduce a penalty term into the objective function, discouraging overly complex models that fit noise.

Given a set of n observations $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where each $\mathbf{x}_i \in \mathbb{R}^p$ is a feature vector and $y_i \in \mathbb{R}$ is the corresponding target value, the task is to find a parameter vector $\theta \in \mathbb{R}^p$ that minimizes the **loss function**. In standard linear regression, the objective is to minimize the **mean squared error (MSE)**, defined as:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \mathbf{x}_i^T \theta \right)^2 = \frac{1}{n} \|\mathbf{X}\theta - \mathbf{y}\|^2 \tag{309}$$

where $\mathbf{X} \in \mathbb{R}^{n \times p}$ is the design matrix, with rows $\mathbf{x}_i^T$, and $\mathbf{y} \in \mathbb{R}^n$ is the vector of target values. The solution to this problem, without any regularization, is given by the **ordinary least squares**

(OLS) solution:

$$\hat{\theta}_{\text{OLS}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} \tag{310}$$

This formulation, however, can lead to overfitting when p is large or when $\mathbf{X}^T\mathbf{X}$ is nearly singular. In such cases, **regularization** is used to modify the loss function, adding a penalty term $\mathcal{R}(\theta)$ to the objective function that discourages large values for the parameters θ_i. The **regularized loss function** is given by:

$$\mathcal{L}_{\text{regularized}}(\theta) = \mathcal{L}(\theta) + \lambda\mathcal{R}(\theta) \tag{311}$$

where λ is a **regularization parameter** that controls the strength of the penalty. The term $\mathcal{R}(\theta)$ penalizes the complexity of the model by imposing constraints on the magnitude of the coefficients. Let us explore two widely used forms of regularization: **L1 regularization** (Lasso) and **L2 regularization** (Ridge). L1 regularization involves adding the ℓ_1**-norm** of the parameter vector θ as the penalty term:

$$\mathcal{R}_{L1}(\theta) = \sum_{i=1}^{p} |\theta_i| \tag{312}$$

The corresponding **L1 regularized loss function** is:

$$\mathcal{L}_{L1}(\theta) = \frac{1}{n}\|\mathbf{X}\theta - \mathbf{y}\|^2 + \lambda\sum_{i=1}^{p} |\theta_i| \tag{313}$$

This formulation promotes sparsity in the parameter vector θ, causing many coefficients to become exactly zero, effectively performing **feature selection**. In high-dimensional settings where many features are irrelevant, L1 regularization helps reduce the model complexity by forcing irrelevant features to be excluded from the model. The effect of the L1 penalty can be understood geometrically by noting that the constraint region defined by the ℓ_1-norm is a **diamond-shaped** region in p-dimensional space. When solving this optimization problem, the coefficients often lie on the boundary of this diamond, leading to a sparse solution with many coefficients being exactly zero. Mathematically, the **soft-thresholding** solution that arises from solving the L1 regularized optimization problem is given by:

$$\hat{\theta}_i = \text{sign}(\theta_i)\max\left(0, |\theta_i| - \lambda\right) \tag{314}$$

This soft-thresholding property drives coefficients to zero when their magnitude is less than λ, resulting in a sparse solution. L2 regularization, on the other hand, uses the ℓ_2**-norm** of the parameter vector θ as the penalty term:

$$\mathcal{R}_{L2}(\theta) = \sum_{i=1}^{p} \theta_i^2 \tag{315}$$

The corresponding **L2 regularized loss function** is:

$$\mathcal{L}_{L2}(\theta) = \frac{1}{n}\|\mathbf{X}\theta - \mathbf{y}\|^2 + \lambda\sum_{i=1}^{p} \theta_i^2 \tag{316}$$

This penalty term does not force any coefficients to be exactly zero but rather **shrinks** the coefficients towards zero, effectively reducing their magnitudes. The L2 regularization helps stabilize the solution when there is multicollinearity in the features by reducing the impact of highly correlated features. The optimization problem with L2 regularization leads to a **ridge regression** solution, which is given by the following expression:

$$\hat{\theta}_{\text{ridge}} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y} \tag{317}$$

where $\mathbf{I}$ is the identity matrix. The L2 penalty introduces a **circular or spherical** constraint in the parameter space, resulting in a solution where all coefficients are reduced in magnitude, but none are eliminated. The **Elastic Net** regularization is a hybrid technique that combines both L1 and L2 regularization. The regularized loss function for Elastic Net is given by:

$$\mathcal{L}_{\text{ElasticNet}}(\theta) = \frac{1}{n}\|\mathbf{X}\theta - \mathbf{y}\|^2 + \lambda_1 \sum_{i=1}^{p} |\theta_i| + \lambda_2 \sum_{i=1}^{p} \theta_i^2 \tag{318}$$

In this case, λ_1 and λ_2 control the strength of the L1 and L2 penalties, respectively. The Elastic Net regularization is particularly useful when dealing with datasets where many features are correlated, as it combines the **sparsity-inducing** property of L1 regularization with the **stability-enhancing** property of L2 regularization. The Elastic Net has been shown to outperform L1 and L2 regularization in some cases, particularly when there are groups of correlated features. The optimization problem can be solved using **coordinate descent** or **proximal gradient methods**, which efficiently handle the mixed penalties. The choice of regularization parameter λ is critical in controlling the **bias-variance tradeoff**. A small value of λ leads to a low-penalty model that is more prone to overfitting, while a large value of λ forces the coefficients to shrink towards zero, potentially leading to underfitting. Thus, it is important to select an optimal value for λ to strike a balance between bias and variance. This can be achieved by using **cross-validation** techniques, where the model is trained on a subset of the data, and the performance is evaluated on the remaining data.

In conclusion, both L1 and L2 regularization techniques play an important role in addressing overfitting by controlling the complexity of the model. L1 regularization encourages sparsity and feature selection, while L2 regularization reduces the magnitude of the coefficients without eliminating any features. By incorporating these regularization terms into the objective function, we can achieve a more balanced bias-variance tradeoff, enhancing the model's ability to generalize to new, unseen data.

6.4 Hyperparameter Tuning

Literature Review: Luo et. al. (2003) [137] provided a deep dive into Bayesian Optimization, a widely used method for hyperparameter tuning. It covers theoretical foundations, practical applications, and advanced strategies for establishing an appropriate range for hyperparameters. This resource is essential for researchers interested in probabilistic approaches to tuning machine learning models. Alrayes et. al. (2025) [138] explored the use of statistical learning and optimization algorithms to fine-tune hyperparameters in machine learning models applied to IoT networks. The paper emphasizes privacy-preserving approaches, making it valuable for practitioners working with secure data environments. Cho et. al. (2020) [139] discussed basic enhancement strategies when using Bayesian Optimization for Hyperparameter Tuning of Deep Neural Networks. Ibrahim et. al. (2025) [140] focused on hyperparameter tuning for XGBoost, a widely used machine learning model, in the context of medical diagnosis. It showcases a comparative analysis of tuning techniques to optimize model performance in real-world healthcare applications. Abdel-Salam et. al. (2025) [141] introduced an evolved framework for tuning deep learning models using multiple optimization algorithms. It presented a novel approach that outperforms traditional techniques in training deep networks. Vali (2025) [142] in his Doctoral thesis covers how vector quantization techniques aid in reducing hyperparameter search space for deep learning models. It emphasizes computational efficiency in speech and image processing applications. Vincent and Jidesh (2023) [143] in their paper explored various hyperparameter optimization techniques, comparing their performance on image classification datasets using AutoML models. It focuses on Bayesian optimization and introduces genetic algorithms, differential evolution, and covariance matrix adaptation—evolutionary strategy (CMA-ES) for acquisition function optimization. Results show that CMA-ES and differential evolution enhance Bayesian optimization, while genetic algorithms degrade its performance.

Razavi-Termeh et. al. (2025) [144] explored the role of geospatial artificial intelligence (GeoAI) in mapping flood-prone areas, leveraging metaheuristic algorithms for hyperparameter tuning. It offers insights into machine learning applications in environmental science. Kiran and Ozyildirim (2022) [145] proposed a distributed variable-length genetic algorithm to optimize hyperparameters in reinforcement learning (RL), improving training efficiency and robustness. Unlike traditional deep RL, which lacks extensive tuning due to complexity, our approach systematically enhances performance across various RL tasks, outperforming Bayesian methods. Results show that more generations yield optimal, computationally efficient solutions, advancing RL for real-world applications.

Hyperparameter tuning in neural networks represents an intricate, highly mathematical optimization challenge that is fundamental to achieving optimal performance on a given task. This process can be framed as a bi-level optimization problem, where the outer optimization concerns the selection of hyperparameters $h \in \mathcal{H}$ to minimize a validation loss function $\mathcal{L}_{\text{val}}(\theta^*(h); h)$, while the inner optimization determines the optimal model parameters θ^* by minimizing the training loss $\mathcal{L}_{\text{train}}(\theta; h)$. This can be expressed rigorously as follows:

$$h^* = \arg\min_{h \in \mathcal{H}} \mathcal{L}_{\text{val}}(\theta^*(h); h), \quad \text{where} \quad \theta^*(h) = \arg\min_{\theta} \mathcal{L}_{\text{train}}(\theta; h). \tag{319}$$

Here, $\mathcal{H}$ denotes the hyperparameter space, which is often high-dimensional, non-convex, and computationally expensive to traverse. The training loss function $\mathcal{L}_{\text{train}}(\theta; h)$ is typically represented as an empirical risk computed over the training dataset $\{(x_i, y_i)\}_{i=1}^N$:

$$\mathcal{L}_{\text{train}}(\theta; h) = \frac{1}{N} \sum_{i=1}^{N} \ell(f(x_i; \theta, h), y_i), \tag{320}$$

where $f(x_i; \theta, h)$ is the neural network output given the input x_i, parameters θ, and hyperparameters h, and $\ell(a, b)$ is the loss function quantifying the discrepancy between prediction a and ground truth b. For classification tasks, ℓ often takes the form of cross-entropy loss:

$$\ell(a, b) = -\sum_{k=1}^{C} b_k \log a_k, \tag{321}$$

where C is the number of classes, and a_k and b_k are the predicted and true probabilities for the k-th class, respectively. Central to the training process is the optimization of θ via gradient-based methods such as stochastic gradient descent (SGD). The parameter updates are governed by:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}_{\text{train}}(\theta^{(t)}; h), \tag{322}$$

where $\eta > 0$ is the learning rate, a critical hyperparameter controlling the step size. The stability and convergence of SGD depend on η, which must satisfy:

$$0 < \eta < \frac{2}{\lambda_{\max}(H)}, \tag{323}$$

where $\lambda_{\max}(H)$ is the largest eigenvalue of the Hessian matrix $H = \nabla_\theta^2 \mathcal{L}_{\text{train}}(\theta; h)$. This condition ensures that the gradient descent steps do not overshoot the minimum. To analyze convergence behavior, the loss function $\mathcal{L}_{\text{train}}(\theta; h)$ near a critical point θ^* can be approximated via a second-order Taylor expansion:

$$\mathcal{L}_{\text{train}}(\theta; h) \approx \mathcal{L}_{\text{train}}(\theta^*; h) + \frac{1}{2}(\theta - \theta^*)^\top H (\theta - \theta^*), \tag{324}$$

where H is the Hessian matrix of second derivatives. The eigenvalues of H reveal the local curvature of the loss surface, with positive eigenvalues indicating directions of convexity and negative eigenvalues corresponding to saddle points. Regularization is often introduced to improve generalization by penalizing large parameter values. For L_2 regularization, the modified training loss is:

$$\mathcal{L}_{\text{train}}^{\text{reg}}(\theta; h) = \mathcal{L}_{\text{train}}(\theta; h) + \frac{\lambda}{2}\|\theta\|_2^2, \tag{325}$$

where $\lambda > 0$ is the regularization coefficient. The gradient of the regularized loss becomes:

$$\nabla_\theta \mathcal{L}_{\text{train}}^{\text{reg}}(\theta; h) = \nabla_\theta \mathcal{L}_{\text{train}}(\theta; h) + \lambda\theta. \tag{326}$$

Another key hyperparameter is the weight initialization strategy, which affects the scale of activations and gradients throughout the network. For a layer with n_{in} inputs, He initialization samples weights from:

$$w_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right), \tag{327}$$

to ensure that the variance of activations remains stable as data propagate through layers. The activation function $g(z)$ also plays a crucial role. The Rectified Linear Unit (ReLU), defined as $g(z) = \max(0, z)$, introduces sparsity and mitigates vanishing gradients. However, it suffers from the "dying neuron" problem, as its derivative $g'(z)$ is zero for $z \leq 0$. The search for optimal hyperparameters can be approached using grid search, random search, or more advanced methods like Bayesian optimization. In Bayesian optimization, a surrogate model $p(\mathcal{L}_{\text{val}}(h))$, often a Gaussian Process (GP), is constructed to approximate the validation loss. The acquisition function $a(h)$, such as Expected Improvement (EI), guides the exploration of $\mathcal{H}$ by balancing exploitation of regions with low predicted loss and exploration of uncertain regions:

$$a(h) = \mathbb{E}\left[\max(0, \mathcal{L}_{\text{val,min}} - \mathcal{L}_{\text{val}}(h))\right], \tag{328}$$

where $\mathcal{L}_{\text{val,min}}$ is the best observed validation loss. Hyperparameter tuning is computationally intensive due to the high dimensionality of $\mathcal{H}$ and the nested nature of the optimization problem. Early stopping, a widely used strategy, halts training when the improvement in validation loss falls below a threshold:

$$\frac{\mathcal{L}_{\text{val}}^{(t+1)} - \mathcal{L}_{\text{val}}^{(t)}}{\mathcal{L}_{\text{val}}^{(t)}} < \epsilon, \tag{329}$$

where $\epsilon > 0$ is a small constant. Advanced techniques like Hyperband leverage multi-fidelity optimization, allocating resources dynamically to promising hyperparameter configurations based on partial training evaluations.

In conclusion, hyperparameter tuning for training neural networks is an exceptionally mathematically rigorous process, grounded in nested optimization, gradient-based methods, probabilistic modeling, and computational heuristics. Each component, from learning rates and regularization to initialization and optimization strategies, contributes to the complex interplay that defines neural network performance.

6.4.1 Grid Search

Literature Review: Rohman and Farikhin (2025) [397] explored the impact of Grid Search and Random Search in hyperparameter tuning for Random Forest classifiers in the context of diabetes prediction. The study provides a comparative analysis of different hyperparameter tuning strategies and demonstrates that Grid Search improves classification accuracy by selecting optimal hyperparameter combinations systematically. Rohman (2025) [398] applied Grid Search-based hyperparameter tuning to optimize machine learning models for early brain tumor detection. The study emphasizes the importance of systematic hyperparameter selection and provides insights into how Grid Search affects diagnostic accuracy and computational efficiency in medical applications.

Nandi et al. (2025) [399] examined the use of Grid Search for deep learning hyperparameter tuning in baby cry sound recognition systems. The authors present a novel pipeline that systematically selects the best hyperparameters for neural networks, improving both precision and recall in sound classification. Sianga et. al. (2025) [400] applied Grid Search and Randomized Search to optimize machine learning models predicting cardiovascular disease risk. The study finds that Grid Search consistently outperforms randomized methods in accuracy, highlighting its effectiveness in medical diagnostic models. Li et. al. (2025) [401] applied Stratified 5-fold cross-validation combined with Grid Search to fine-tune Extreme Gradient Boosting (XGBoost) models in predicting post-surgical complications. The results suggest that hyperparameter tuning significantly improves predictive performance, with Grid Search leading to the best model stability and interpretability. Lázaro et. al. (2025) [402] implemented Grid Search and Bayesian Optimization to optimize K-Nearest Neighbors (KNN) and Decision Trees for incident classification in aviation safety. The research underscores how different hyperparameter tuning methods affect the generalization of machine learning models in NLP-based accident reports. Li et. al. (2025) [403] proposed RAINER, an ensemble learning model that integrates Grid Search for optimal hyperparameter tuning. The study demonstrates how parameter optimization enhances the predictive capabilities of rainfall models, making Grid Search an essential step in climate modeling. Khurshid et. al. (2025) [404] compared Bayesian Optimization with Grid Search for hyperparameter tuning in diabetes prediction models. The study finds that while Bayesian methods are computationally faster, Grid Search delivers more precise hyperparameter selection, especially for models with structured medical data. Kanwar et. al. (2025) [405] applied Grid Search for tuning Random Forest classifiers in landslide susceptibility mapping. The study demonstrates that fine-tuned models improve the identification of high-risk zones, reducing false positives in predictive landslide models. Fadil et. al. (2025) [406] evaluated the role of Grid Search and Random Search in hyperparameter tuning for XGBoost regression models in corrosion prediction. The authors find that Grid Search-based models achieve higher R^2 scores, making them ideal for complex chemical modeling applications.

Grid search is a highly structured and exhaustive method for hyperparameter tuning in machine learning, where a predetermined grid of hyperparameter values is systematically explored. The goal is to identify the set of hyperparameters $\vec{h} = (h_1, h_2, \dots, h_p)$ that yields the optimal performance metric for a given machine learning model. Let p represent the total number of hyperparameters to be tuned, and for each hyperparameter h_i, let the candidate set be $\mathcal{H}_i = \{h_{i1}, h_{i2}, \dots, h_{im_i}\}$, where m_i is the number of candidate values for h_i. The hyperparameter search space is then the Cartesian product of all candidate sets:

$$\mathcal{S} = \mathcal{H}_1 \times \mathcal{H}_2 \times \cdots \times \mathcal{H}_p. \tag{330}$$

Thus, the total number of configurations to be evaluated is:

$$|\mathcal{S}| = \prod_{i=1}^{p} m_i. \tag{331}$$

For example, if we have two hyperparameters h_1 and h_2 with 3 possible values each, the total number of combinations to explore is 9. This search space grows exponentially as the number of hyperparameters increases, posing a significant computational challenge. Grid search involves iterating over all configurations in $\mathcal{S}$, evaluating the model's performance for each configuration.

Let us define the performance metric $M(\vec{h}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}})$, which quantifies the model's performance for a given hyperparameter configuration $\vec{h}$, where $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{val}}$ are the training and validation datasets, respectively. This metric might represent accuracy, error rate, F1-score, or any other relevant criterion, depending on the problem at hand. The hyperparameters are then tuned by maximizing or minimizing M across the search space:

$$\vec{h}^* = \arg\max_{\vec{h} \in \mathcal{S}} M(\vec{h}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}), \tag{332}$$

or in the case of a minimization problem:

$$\vec{h}^* = \arg\min_{\vec{h}\in\mathcal{S}} M(\vec{h}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}). \tag{333}$$

For each hyperparameter combination, the model is trained on $\mathcal{D}_{\text{train}}$ and evaluated on $\mathcal{D}_{\text{val}}$. The process requires the repeated evaluation of the model over all $|\mathcal{S}|$ configurations, each yielding a performance metric. To mitigate overfitting and ensure the reliability of the performance metric, cross-validation is frequently used. In k-fold cross-validation, the dataset $\mathcal{D}_{\text{train}}$ is partitioned into k disjoint subsets $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_k$. The model is trained on $\mathcal{D}_{\text{train}}^{(j)} = \bigcup_{i\neq j} \mathcal{D}_i$ and validated on $\mathcal{D}_j$. For each fold j, we compute the performance metric:

$$M_j(\vec{h}) = M(\vec{h}, \mathcal{D}_{\text{train}}^{(j)}, \mathcal{D}_j). \tag{334}$$

The overall cross-validation performance for a hyperparameter configuration $\vec{h}$ is the average of the k individual fold performances:

$$\overline{M}(\vec{h}) = \frac{1}{k}\sum_{j=1}^{k} M_j(\vec{h}). \tag{335}$$

Thus, the grid search with cross-validation aims to find the optimal hyperparameters by maximizing or minimizing the average performance across all folds. The computational complexity of grid search is a key consideration. If we denote C as the cost of training and evaluating the model for a single configuration, the total cost for grid search is:

$$\mathcal{O}\left(\prod_{i=1}^{p} m_i \cdot k \cdot C\right), \tag{336}$$

where k represents the number of folds in cross-validation. This results in an exponential increase in the total computation time as the number of hyperparameters p and the number of candidate values m_i increase. For large search spaces, grid search can become computationally expensive, making it infeasible for high-dimensional hyperparameter optimization problems. To illustrate with a specific example, consider two hyperparameters h_1 and h_2 with the following sets of candidate values:

$$\mathcal{H}_1 = \{0.01, 0.1, 1.0\}, \quad \mathcal{H}_2 = \{0.1, 1.0, 10.0\}. \tag{337}$$

The search space is:

$$\mathcal{S} = \mathcal{H}_1 \times \mathcal{H}_2 = \{(0.01, 0.1), (0.01, 1.0), (0.01, 10.0), (0.1, 0.1), \dots, (1.0, 10.0)\}. \tag{338}$$

There are 9 configurations to evaluate. For each configuration, assume we perform 3-fold cross-validation, where the performance metrics for the first fold are:

$$M_1(0.1, 1.0) = 0.85, \quad M_2(0.1, 1.0) = 0.87, \quad M_3(0.1, 1.0) = 0.86, \tag{339}$$

giving the cross-validation performance:

$$\overline{M}(0.1, 1.0) = \frac{1}{3}\sum_{j=1}^{3} M_j(0.1, 1.0) = \frac{1}{3}(0.85 + 0.87 + 0.86) = 0.86. \tag{340}$$

This process is repeated for all 9 combinations of h_1 and h_2. Grid search, while exhaustive and deterministic, can fail to efficiently explore the hyperparameter space, especially when the number of hyperparameters is large. The search is confined to a discrete grid and cannot interpolate between points to capture optimal configurations that may lie between grid values. Furthermore, because grid search evaluates each configuration independently, it can be computationally expensive for high-dimensional spaces, as the number of configurations grows exponentially with p and m_i.

In conclusion, grid search is a methodologically rigorous and systematic approach to hyperparameter optimization, ensuring that all predefined configurations are evaluated exhaustively. However, its computational cost increases exponentially with the number of hyperparameters and their respective candidate values, which can limit its applicability for large-scale problems. As a result, more advanced techniques such as random search, Bayesian optimization, or evolutionary algorithms are often used for hyperparameter tuning when the computational budget is limited. Despite these challenges, grid search remains a powerful tool for demonstrating the principles of hyperparameter tuning and is well-suited for problems with relatively small search spaces.

6.4.2 Random Search

Literature Review: Sianga et. al. (2025) [400] explored Random Search vs. Grid Search for tuning machine learning models in cardiovascular disease risk prediction. It finds that Random Search significantly reduces computation time while maintaining high accuracy, making it preferable for high-dimensional datasets in medical applications. Lázaro et. al. (2025) [402] applied Random Search and Grid Search to optimize models for accident classification using NLP. The study highlights Random Search's efficiency in tuning K-Nearest Neighbors (KNN) and Decision Trees, leading to faster convergence with minimal loss in accuracy. Emmanuel et. al. (2025) [407] introduced a hybrid approach combining Random Search with Differential Evolution optimization to enhance deep-learning-based protein interaction models. The study demonstrates how Random Search improves generalization and reduces overfitting. Gaurav et. al. (2025) [408] evaluated Random Search optimization in Random Forest classifiers for driver identification. They compare Random Search, Bayesian Optimization, and Genetic Algorithms, concluding that Random Search provides a balance between efficiency and performance. Kanwar et. al. (2025) [405] applied Random Search hyperparameter tuning to Random Forest models for landslide risk assessment. It finds that Random Search significantly reduces computation time without compromising model accuracy, making it ideal for large-scale geospatial analyses. Ning et al. (2025) [409] evaluated Random Search for optimizing mortality prediction models in infected pancreatic necrosis patients. The authors conclude that Random Search outperforms exhaustive Grid Search in finding optimal hyperparameters with significant speed improvements. Muñoz et. al. (2025) [410] presented a novel optimization strategy that combines Random Search with a secretary algorithm to improve hyperparameter tuning efficiency. It demonstrates how Random Search can be adapted to dynamic optimization problems in real-time AI applications. Balcan et. al. (2025) [411] explored the theoretical underpinnings of Random Search in deep learning optimization. They provide a rigorous analysis of the sample complexity required for effective tuning, establishing mathematical guarantees for Random Search efficiency. Azimi et. al. (2025) [412] compared Random Search with metaheuristic algorithms (e.g., Genetic Algorithms and Particle Swarm Optimization) in supercapacitor modeling. The results indicate that Random Search provides a robust baseline for hyperparameter optimization in deep learning models. Shibina and Thasleema (2025) [413] applied Random Search for optimizing ensemble learning classifiers in medical diagnosis. The results show Random Search's advantage in finding optimal hyperparameters for detecting Parkinson's disease using voice features, making it a practical alternative to Bayesian Optimization.

In machine learning, hyperparameter tuning is the process of selecting the best configuration of hyperparameters $\mathbf{h} = (h_1, h_2, \dots, h_d)$, where each h_i represents the i-th hyperparameter. The hyperparameters $\mathbf{h}$ control key aspects of model learning, such as the learning rate, regularization strength, or the architecture of the neural network. These hyperparameters are not directly optimized through the learning process itself but are instead set before training begins. Given a set of hyperparameters, the model performance is evaluated by computing a loss function $L(\mathbf{h})$, which typically represents the error on a validation set, and possibly regularization terms to mitigate overfitting. The objective is to minimize this loss function to find the optimal set of hyperparameters:

$$\mathbf{h}^* = \arg\min_{\mathbf{h}} L(\mathbf{h}), \tag{341}$$

where $L(\mathbf{h})$ is the loss function that quantifies how well the model generalizes to unseen data. The minimization of this function is often subject to constraints on the range or type of values that each h_i can take, forming a constrained optimization problem:

$$\mathbf{h}^* = \arg\min_{\mathbf{h}\in\mathcal{H}} L(\mathbf{h}), \tag{342}$$

where $\mathcal{H}$ represents the feasible hyperparameter space. Hyperparameter tuning is typically carried out by selecting a search method that explores this space efficiently, with the goal of finding the global or local optimum of the loss function.

One such search method is **random search**, which is a straightforward yet effective approach to exploring the hyperparameter space. Instead of exhaustively searching over a grid of values for each hyperparameter (as in grid search), random search samples hyperparameters $\mathbf{h}_t = (h_{t,1}, h_{t,2}, \ldots, h_{t,d})$ from a predefined distribution for each hyperparameter h_i. For each iteration t, the hyperparameters are independently sampled from probability distributions $\mathcal{D}_i$ associated with each hyperparameter h_i, where the probability distribution might be continuous or discrete. Specifically, for continuous hyperparameters, $h_{t,i}$ is drawn from a uniform or normal distribution over an interval $H_i = [a_i, b_i]$:

$$h_{t,i} \sim \mathcal{U}(a_i, b_i), \quad h_{t,i} \in H_i, \tag{343}$$

where $\mathcal{U}(a_i, b_i)$ denotes the uniform distribution between a_i and b_i. For discrete hyperparameters, $h_{t,i}$ is sampled from a discrete set of values $H_i = \{h_{i1}, h_{i2}, \ldots, h_{iN_i}\}$ with each value equally probable:

$$h_{t,i} \sim \mathcal{D}_i, \quad h_{t,i} \in \{h_{i1}, h_{i2}, \ldots, h_{iN_i}\}, \tag{344}$$

where $\mathcal{D}_i$ denotes the discrete distribution over the set $\{h_{i1}, h_{i2}, \ldots, h_{iN_i}\}$. Thus, each hyperparameter is selected independently from its corresponding distribution. After selecting a new set of hyperparameters $\mathbf{h}_t$, the model is trained with this configuration, and its performance is evaluated by computing the loss function $L(\mathbf{h}_t)$. The process is repeated for T iterations, generating a sequence of hyperparameter configurations $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T$, and for each configuration, the associated loss function values $L(\mathbf{h}_1), L(\mathbf{h}_2), \ldots, L(\mathbf{h}_T)$ are computed. The optimal set of hyperparameters $\mathbf{h}^*$ is then selected as the one that minimizes the loss:

$$\mathbf{h}^* = \arg\min_{t\in\{1,2,\ldots,T\}} L(\mathbf{h}_t). \tag{345}$$

Thus, random search performs an approximate optimization of the hyperparameter space, where the computational cost per iteration is C (the time to evaluate the model's performance for a given set of hyperparameters), and the total computational cost is $O(T \cdot C)$. This makes random search a computationally feasible approach, especially when T is moderate. The computational efficiency of random search can be compared to that of grid search, which exhaustively searches the hyperparameter space by discretizing each hyperparameter h_i into a set of values $h_{i1}, h_{i2}, \ldots, h_{in_i}$, where n_i is the number of values for the i-th hyperparameter. The total number of grid search configurations is given by:

$$N_{\text{grid}} = \prod_{i=1}^{d} n_i, \tag{346}$$

and the computational cost of grid search is $O(N_{\text{grid}} \cdot C)$, which grows exponentially with the number of hyperparameters d. In this sense, grid search can become prohibitively expensive when the dimensionality d of the hyperparameter space is large. Random search, on the other hand, requires only T evaluations, and since each evaluation is independent of the others, the computational cost grows linearly with T, making it more efficient when d is large. The probabilistic nature of random search further enhances its efficiency. Suppose that only a subset of hyperparameters, say k, significantly influences the model's performance. Let S be the subspace of $\mathcal{H}$ consisting of hyperparameter configurations that produce low loss values, and let the complementary space $\mathcal{H} \backslash S$ correspond to configurations that are unlikely to achieve low loss. In this case, the task becomes one of searching within the subspace S, rather than the entire space $\mathcal{H}$. The random search method is well-suited to such problems, as it can probabilistically focus on the relevant subspace by drawing hyperparameter values from distributions $\mathcal{D}_i$ that prioritize areas of the hyperparameter space with low loss. More formally, the probability of selecting a hyperparameter set $\mathbf{h}_t$ from the relevant subspace S is given by:

$$P(\mathbf{h}_t \in S) = \prod_{i=1}^{d} P(h_{t,i} \in S_i), \tag{347}$$

where S_i is the relevant region for the i-th hyperparameter, and $P(h_{t,i} \in S_i)$ is the probability that the i-th hyperparameter lies within the relevant region. As the number of iterations T increases, the probability that random search selects a hyperparameter set $\mathbf{h}_t \in S$ increases as well, approaching 1 as $T \to \infty$:

$$P(\mathbf{h}_t \in S) = 1 - (1 - P_0)^T, \tag{348}$$

where P_0 is the probability of sampling a hyperparameter set from the relevant subspace in one iteration. Thus, random search tends to explore the subspace of low-loss configurations, improving the chances of finding an optimal or near-optimal configuration as T increases.

The exploration behavior of random search contrasts with that of grid search, which, despite its systematic nature, may fail to efficiently explore sparsely populated regions of the hyperparameter space. When the hyperparameter space is high-dimensional, the grid search must evaluate exponentially many configurations, regardless of the relevance of the hyperparameters. This leads to inefficiencies when only a small fraction of hyperparameters significantly contribute to the loss function. Random search, by sampling independently and uniformly across the entire space, is not subject to this curse of dimensionality and can more effectively locate regions that matter for model performance. Mathematically, random search has an additional advantage when the hyperparameters exhibit smooth or continuous relationships with the loss function. In this case, random search can probe the space probabilistically, discovering gradients of loss that grid search, due to its fixed grid structure, may miss. Furthermore, random search is capable of finding the optimum even when the loss function is non-convex, provided that the space is explored adequately. This becomes particularly relevant in the presence of highly irregular loss surfaces, as random search has the potential to escape local minima more effectively than grid search, which is constrained by its fixed sampling grid.

In conclusion, random search is a highly efficient and scalable approach for hyperparameter optimization in machine learning. By sampling hyperparameters from predefined probability distributions and evaluating the associated loss function, random search provides a computationally feasible method for high-dimensional hyperparameter spaces, outperforming grid search in many cases. Its probabilistic nature allows it to focus on relevant regions of the hyperparameter space, making it particularly advantageous when only a subset of hyperparameters significantly impacts the model's performance. As the number of iterations T increases, random search becomes more likely to converge to the optimal configuration, making it a powerful tool for hyperparameter tuning in complex models.

6.4.3 Bayesian Optimization

Literature Review: Chang et. al. (2025) [414] applied Bayesian Optimization (BO) for hyperparameter tuning in machine learning models used for predicting landslide displacement. It explores the impact of BO in optimizing Support Vector Machines (SVM), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRU), demonstrating how Bayesian techniques improve model accuracy and convergence rates. Cihan (2025) [415] used Bayesian Optimization to fine-tune XGBoost, LightGBM, Elastic Net, and Adaptive Boosting models for predicting biomass gasification output. The study finds that Bayesian Optimization outperforms Grid and Random Search in reducing computational overhead while improving predictive accuracy. Makomere et. al. (2025) [416] integrated Bayesian Optimization for hyperparameter tuning in deep learning-based industrial process modeling. The study provides insights into how BO improves model generalization and reduces prediction errors in chemical process monitoring. Bakır (2025) [417] introduced TuneDroid, an automated Bayesian Optimization-based framework for hyperparameter tuning of Convolutional Neural Networks (CNNs) used in cybersecurity. The results suggest that Bayesian Optimization accelerates model training while improving malware detection accuracy. Khurshid et. al. (2025) [404] compared Bayesian Optimization and Random Search for tuning hyperparameters in XGBoost-based diabetes prediction models. It concludes that Bayesian Optimization provides a superior trade-off between speed and accuracy compared to traditional search methods. Liu et. al. (2025) [418] explored Bayesian Optimization's ability to fine-tune deep learning models for predicting acoustic performance in engineering systems. The authors demonstrate how Bayesian methods improve prediction accuracy while reducing computational costs. Balcan et. al. (2025) [411] provided a rigorous analysis of the sample complexity required for Bayesian Optimization in deep learning. The findings show that Bayesian Optimization requires fewer samples to converge to optimal solutions compared to other hyperparameter tuning techniques. Ma et. al. (2025) [419] integrated Bayesian Optimization with Support Vector Machines (SVMs) for anomaly detection in high-speed machining. They find that Bayesian Optimization allows more effective exploration of hyperparameter spaces, leading to improved model reliability. Bouzaidi et. al. (2025) [420] explored the impact of Bayesian Optimization on CNN-based models for image classification. It demonstrates how Bayesian techniques outperform traditional methods like Grid Search in transfer learning scenarios. Mustapha et. al. (2025) [421] integrated Bayesian Optimization for tuning a hybrid deep learning framework combining Convolutional Neural Networks (CNNs) and Vision Transformers (ViTs) for pneumonia detection. The results confirm that Bayesian Optimization enhances the efficiency of multi-model architectures in medical imaging.

Bayesian Optimization (BO) is a powerful, mathematically sophisticated method for optimizing complex, black-box objective functions, which is particularly useful in the context of hyperparameter tuning in machine learning models. These objective functions, denoted as $f : \mathcal{X} \to \mathbb{R}$, are often expensive to evaluate due to factors such as time-consuming training of models or noisy observations. In hyperparameter tuning, the objective function typically represents some performance metric of a machine learning model (e.g., accuracy, error, or loss) evaluated at specific hyperparameter configurations. The goal of Bayesian Optimization is to find the hyperparameter setting $\mathbf{x}^* \in \mathcal{X}$ that minimizes (or maximizes) the objective function, such that:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}) \tag{349}$$

Given that exhaustive search is computationally prohibitive, BO uses a probabilistic approach to efficiently explore the hyperparameter space. This is achieved by treating the objective function $f(\mathbf{x})$ as a random function and utilizing a **surrogate model** to approximate it, which allows for strategic decisions about which points in the space $\mathcal{X}$ to evaluate. The surrogate model is typically represented by a **Gaussian Process (GP)**, which provides both a prediction and an uncertainty estimate at any point in $\mathcal{X}$. The GP is a non-parametric, probabilistic model that assumes that function values at any finite set of points follow a joint Gaussian distribution. Specifically, for a set of observed points $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, the corresponding function values $\{f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_n)\}$ are assumed to be jointly distributed as:

$$\begin{bmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix} \sim \mathcal{N}(\mathbf{m}, \mathbf{K}) \tag{350}$$

where $\mathbf{m} = [m(\mathbf{x}_1), m(\mathbf{x}_2), \ldots, m(\mathbf{x}_n)]^\top$ is the mean vector and $\mathbf{K}$ is the covariance matrix whose entries are defined by a covariance (or kernel) function $k(\mathbf{x}, \mathbf{x}')$, which encodes assumptions about the smoothness and periodicity of the objective function. The kernel function plays a crucial role in determining the properties of the Gaussian Process. A commonly used kernel is the **Squared Exponential (SE)** kernel, which is defined as:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right) \tag{351}$$

where σ_f^2 is the variance, which scales the function values, and ℓ is the length scale, which controls the smoothness of the function by dictating how quickly the function values can change with respect to the inputs. Once the Gaussian Process has been specified, Bayesian Optimization proceeds by updating the posterior distribution over the objective function after each new evaluation. Given a set of n observed pairs $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $y_i = f(\mathbf{x}_i) + \epsilon_i$ and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ represents observational noise, we update the posterior of the GP to reflect the observed data. The posterior mean $\mu(\mathbf{x}_*)$ and variance $\sigma^2(\mathbf{x}_*)$ at a new point $\mathbf{x}_*$ are given by the following equations:

$$\mu(\mathbf{x}_*) = \mathbf{k}_*^\top \mathbf{K}^{-1} \mathbf{y} \tag{352}$$

$$\sigma^2(\mathbf{x}_*) = k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{k}_*^\top \mathbf{K}^{-1} \mathbf{k}_* \tag{353}$$

where $\mathbf{k}_*$ is the vector of covariances between the test point $\mathbf{x}_*$ and the observed points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, and $\mathbf{K}$ is the covariance matrix of the observed points. The updated mean $\mu(\mathbf{x}_*)$ provides the model's best guess for the value of the function at $\mathbf{x}_*$, and $\sigma^2(\mathbf{x}_*)$ quantifies the uncertainty associated with this estimate.

In Bayesian Optimization, the central objective is to select the next hyperparameter setting $\mathbf{x}_*$ to evaluate in such a way that the number of function evaluations is minimized while still making progress toward the global optimum. This is achieved by optimizing an **acquisition function**. The acquisition function $\alpha(\mathbf{x})$ represents a trade-off between exploiting regions of the input space where the objective function is expected to be low and exploring regions where the model's uncertainty is high. Several acquisition functions have been proposed, including **Expected Improvement (EI)**, **Probability of Improvement (PI)**, and **Upper Confidence Bound (UCB)**. The **Expected Improvement (EI)** acquisition function is one of the most widely used and is defined as:

$$\text{EI}(\mathbf{x}) = (f_{\text{best}} - \mu(\mathbf{x}))\Phi\left(\frac{f_{\text{best}} - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right) + \sigma(\mathbf{x})\phi\left(\frac{f_{\text{best}} - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right) \tag{354}$$

where f_{best} is the best observed value of the objective function, $\Phi(\cdot)$ and $\phi(\cdot)$ are the cumulative distribution and probability density functions of the standard normal distribution, respectively, and $\sigma(\mathbf{x})$ is the standard deviation at $\mathbf{x}$. The first term measures the potential for improvement, weighted by the probability of achieving that improvement, and the second term reflects the uncertainty at $\mathbf{x}$, encouraging exploration in uncertain regions. The acquisition function is maximized at each iteration to select the next point $\mathbf{x}_*$:

$$\mathbf{x}_* = \arg\max_{\mathbf{x} \in \mathcal{X}} \text{EI}(\mathbf{x}) \tag{355}$$

An alternative acquisition function is the **Probability of Improvement (PI)**, which is simpler and directly measures the probability that the objective function at $\mathbf{x}$ will exceed the current best value:

$$\mathrm{PI}(\mathbf{x}) = \Phi\left(\frac{f_{\text{best}} - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right) \tag{356}$$

Another common acquisition function is the **Upper Confidence Bound (UCB)**, which balances exploration and exploitation by selecting the point with the highest upper confidence bound:

$$\mathrm{UCB}(\mathbf{x}) = \mu(\mathbf{x}) + \kappa\sigma(\mathbf{x}) \tag{357}$$

where κ is a hyperparameter that controls the trade-off between exploration (κ large) and exploitation (κ small). After selecting $\mathbf{x}_*$, the function is evaluated at this point, and the observed value $y_* = f(\mathbf{x}_*)$ is used to update the posterior distribution of the Gaussian Process. This process is repeated iteratively, and each new observation refines the model's understanding of the objective function, guiding the search for the optimal $\mathbf{x}^*$. One of the primary advantages of Bayesian Optimization is its ability to efficiently optimize expensive-to-evaluate functions by focusing the search on the most promising regions of the input space. However, as the number of observations increases, the computational complexity of maintaining the Gaussian Process model grows cubically with respect to the number of points, due to the need to invert the covariance matrix $\mathbf{K}$. This cubic complexity, $\mathcal{O}(n^3)$, can be prohibitive for large datasets. To mitigate this, techniques such as **sparse Gaussian Processes** have been developed, which approximate the full covariance matrix by using a smaller set of inducing points, thus reducing the computational cost while maintaining the flexibility of the Gaussian Process model.

In conclusion, Bayesian Optimization represents a mathematically rigorous and efficient method for hyperparameter tuning, where a Gaussian Process surrogate model is used to approximate the unknown objective function, and an acquisition function guides the search for the optimal solution by balancing exploration and exploitation. Despite its computational challenges, especially in high-dimensional problems, the method is widely applicable in contexts where evaluating the objective function is expensive, and it has been shown to outperform traditional optimization techniques in many real-world scenarios.

7 Convolution Neural Networks

Literature Review: Goodfellow et. al. (2016) [112] wrote one of the most foundational textbooks on deep learning, covering CNNs in depth. It introduces theoretical principles, including convolutions, backpropagation, and optimization methods. The book also discusses applications of CNNs in image processing and beyond. LeCun et. al. (2015) [117] provides a historical overview of CNNs and deep learning. LeCun, one of the inventors of CNNs, explains why convolutions help in image recognition and discusses their applications in vision, speech, and reinforcement learning. Krizhevsky et. al. (2012) [146] and Krizhevsky et. al. (2017) [147] introduced AlexNet, the first modern deep CNN, which won the 2012 ImageNet Challenge. It demonstrated that deep CNNs can achieve unprecedented accuracy in image classification tasks, paving the way for deep learning's dominance. Simonyan and Zisserman (2015) [148] introduced VGGNet, which demonstrated that increasing network depth using small 3x3 convolutions can improve performance. It also provided insights into layer design choices and their effects on accuracy. He et. al. (2016) [149] introduced ResNet, which solved the vanishing gradient problem in deep networks by using skip connections. This revolutionized CNN design by allowing models as deep as 1000 layers to be trained efficiently. Cohen and Welling (2016) [150] extended CNNs using group theory, enabling equivariant feature learning. This improved CNN robustness to rotations and translations, making them more efficient in symmetry-based tasks. Zeiler and Fergus (2014) [151] introduced deconvolution techniques to visualize CNN feature maps, making it easier to interpret and debug CNNs. It showed how different layers detect patterns, textures, and objects. Liu et.al. (2021) [152] introduced Vision Transformers (ViTs) that outperform CNNs in some vision tasks. This paper discusses the limitations of CNNs and how transformers can be hybridized with CNN architectures. Lin et.al. (2013) [153] introduced the 1x1 convolution, which improved feature learning efficiency. This concept became a key component of modern CNN architectures such as ResNet and MobileNet. Rumelhart et. al. (1986) [154] formalized backpropagation, the training method used for CNNs. Without this discovery, CNNs and deep learning would not exist today.

7.1 Key Concepts

A **Convolutional Neural Network (CNN)** is a deep learning model primarily used for analyzing grid-like data, such as images, video, and time-series data with spatial or temporal dependencies. The fundamental operation of CNNs is the **convolution** operation, which is employed to extract local patterns from the input data. The input to a CNN is generally represented as a tensor $\mathbf{I} \in \mathbb{R}^{H \times W \times C}$, where H is the height, W is the width, and C is the number of channels (for RGB images, $C = 3$).

At the core of a CNN is the convolutional layer, where the input image $\mathbf{I}$ is convolved with a set of filters or kernels $\mathbf{K} \in \mathbb{R}^{f_h \times f_w \times C}$, where f_h and f_w are the height and width of the filter, respectively. The filter $\mathbf{K}$ slides across the input image $\mathbf{I}$, and the result of this convolution is a set of feature maps that are indicative of certain local patterns in the image. The element-wise convolution at location (i, j) of the feature map is given by:

$$\mathbf{I} * \mathbf{K} = \sum_{p=1}^{f_h} \sum_{q=1}^{f_w} \sum_{r=1}^{C} \mathbf{I}_{i+p-1,j+q-1,r} \cdot \mathbf{K}_{p,q,r} \tag{358}$$

where $\mathbf{I}_{i+p-1,j+q-1,r}$ denotes the value of the r-th channel of the input image at position $(i+p-1, j+q-1)$, and $\mathbf{K}_{p,q,r}$ is the corresponding filter value at (p, q, r). This operation is done for each location (i, j) of the output feature map. The resulting feature map $\mathbf{F}$ has spatial dimensions $H' \times W'$, where:

$$H' = \left\lfloor \frac{H + 2p - f_h}{s} \right\rfloor + 1, \quad W' = \left\lfloor \frac{W + 2p - f_w}{s} \right\rfloor + 1 \tag{359}$$

where p is the padding, and s is the stride of the filter during its sliding motion. The convolution operation provides a translation-invariant representation of the input image, as each filter detects patterns across the entire image. After this convolution, a non-linear activation function, typically the **Rectified Linear Unit (ReLU)**, is applied to introduce non-linearity into the network and ensure it can model complex patterns. The ReLU activation function operates element-wise and is given by:

$$\text{ReLU}(x) = \max(0, x) \tag{360}$$

Thus, for each feature map $\mathbf{F}$, the output after ReLU is:

$$\mathbf{F}'_{i,j,k} = \max(0, \mathbf{F}_{i,j,k}) \tag{361}$$

This ensures that negative values in the feature map are discarded, which helps with the sparse representation of activations, mitigating the vanishing gradient problem in deeper layers. In CNNs, pooling operations follow the convolution and activation layers. Pooling serves to reduce the spatial dimensions of the feature maps, thus decreasing computational complexity and making the representation more invariant to translations. **Max pooling**, which is the most common form, selects the maximum value within a specified window size $p_h \times p_w$. Given an input feature map $\mathbf{F} \in \mathbb{R}^{H' \times W' \times K}$, max pooling operates as follows:

$$\mathbf{P}_{i,j,k} = \max\left(\mathbf{F}_{i,j,k}, \mathbf{F}_{i+1,j,k}, \mathbf{F}_{i,j+1,k}, \mathbf{F}_{i+1,j+1,k}\right) \tag{362}$$

where $\mathbf{P}$ is the pooled feature map. This pooling operation effectively reduces the spatial dimensions of each feature map, resulting in an output $\mathbf{P} \in \mathbb{R}^{H'' \times W'' \times K}$, where:

$$H'' = \left\lfloor \frac{H'}{p_h} \right\rfloor, \quad W'' = \left\lfloor \frac{W'}{p_w} \right\rfloor \tag{363}$$

Max pooling introduces an element of robustness by capturing only the strongest features within the local regions, discarding irrelevant information, and ensuring that the network is invariant to small translations. The CNN architecture typically contains multiple convolutional layers followed by pooling layers. After these operations, the feature maps are flattened into a one-dimensional vector and passed into one or more **fully connected (dense) layers**. A fully connected layer computes a linear transformation of the form:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{364}$$

where $\mathbf{a}^{(l-1)}$ is the input to the layer, $\mathbf{W}^{(l)}$ is the weight matrix, and $\mathbf{b}^{(l)}$ is the bias vector. The output of this linear transformation is then passed through a non-linear activation function, such as ReLU or softmax for classification tasks. For classification, the **softmax** function is often applied to convert the output into a probability distribution:

$$\mathbf{y}_i = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)} \tag{365}$$

where C is the number of output classes, and $\mathbf{y}_i$ is the probability of the i-th class. The softmax function ensures that the output probabilities sum to 1, providing a valid classification output. The CNN is trained using **backpropagation**, which computes the gradients of the loss function $\mathcal{L}$ with respect to the network's parameters (i.e., weights and biases). Backpropagation uses the **chain rule** to propagate the error gradients through each layer. The gradients with respect to the convolutional filters $\mathbf{K}$ are computed by:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{K}} = \frac{\partial \mathcal{L}}{\partial \mathbf{F}} * \mathbf{I} \tag{366}$$

where $*$ denotes the convolution operation. Similarly, the gradients for the fully connected layers are computed by:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \mathbf{a}^{(l-1)} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \tag{367}$$

Once the gradients are computed, the weights are updated using an optimization algorithm like **gradient descent**:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} \tag{368}$$

where η is the learning rate. This optimization ensures that the network's parameters are adjusted in the direction of the negative gradient, minimizing the loss function and thereby improving the performance of the CNN. Regularization techniques are commonly applied to avoid overfitting. **Dropout**, for instance, randomly deactivates a subset of neurons during training, preventing the network from becoming too reliant on any specific feature and promoting better generalization. The dropout operation at a given layer l with dropout rate p is defined as:

$$\mathbf{a}^{(l)} \sim \text{Dropout}(\mathbf{a}^{(l)}, p) \tag{369}$$

where the activations $\mathbf{a}^{(l)}$ are randomly set to zero with probability p, and the remaining activations are scaled by $\frac{1}{1-p}$. Another regularization technique is **batch normalization**, which normalizes the inputs of each layer to have zero mean and unit variance, thus improving training speed and stability. Mathematically, batch normalization is defined as:

$$\hat{x} = \frac{x - \mu_B}{\sigma_B}, \quad \mathbf{y} = \gamma \hat{x} + \beta \tag{370}$$

where μ_B and σ_B are the mean and standard deviation of the batch, and γ and β are learned scaling and shifting parameters.

In conclusion, the mathematical backbone of a **Convolutional Neural Network (CNN)** relies heavily on the convolution operation, non-linear activations, pooling, and fully connected transformations. The convolutional layers extract hierarchical features by applying filters to the input data, while pooling reduces the spatial dimensions and introduces invariance to translations. The fully connected layers aggregate these features for classification or regression tasks. The network is trained using backpropagation and optimization techniques such as gradient descent. Regularization methods like dropout and batch normalization are used to improve generalization and training efficiency. The mathematical formalism behind CNNs is essential for understanding their power in various machine learning tasks, particularly in computer vision.

7.2 Applications in Image Processing

7.2.1 Image Classification

Literature Review: Thiriveedhi et. al. (2025) [185] presented a novel CNN-based architecture for diagnosing Acute Lymphoblastic Leukemia (ALL), integrating explainable AI (XAI) techniques. The proposed model outperforms traditional CNNs by providing human-interpretable insights into medical image classification. The research highlights how CNNs can be effectively applied to medical imaging with enhanced transparency. Ramos-Briceño et. al. (2025) [186] demonstrated the superior classification accuracy of CNNs in malaria parasite detection. The research uses deep CNNs to classify malaria species in blood samples and achieves state-of-the-art performance. The paper provides valuable insights into CNN-based image classification for biomedical applications. Espino-Salinas et. al. (2025) [187] applied CNNs to mental health diagnostics by classifying motion activity patterns as images. The paper explores the novel application of CNNs beyond traditional image classification by transforming time-series data into visual representations and utilizing CNNs to detect psychiatric disorders. Ran et. al. (2025) [188] introduced a CNN-based hyperspectral imaging method for early diagnosis of pancreatic neuroendocrine tumors. The paper highlights CNNs' ability to process multispectral data for complex medical imaging tasks, further expanding their utility in pathology and cancer detection. Araujo et. al. (2025) [189] demonstrated how CNNs can be employed in industrial monitoring and predictive maintenance. The research introduces an innovative CNN-based approach for detecting faults in ZnO surge arresters using thermal imaging, proving CNNs' robustness in non-destructive testing applications. Sari et. al. (2025) [190] applied CNNs to cultural heritage preservation, specifically Batik pattern classification. The study showcases CNNs' adaptability in fine-grained image classification and highlights the importance of deep learning in automated textile pattern recognition. Wang et. al. (2025) [191] proposed CF-WIAD, a novel semi-supervised learning method that leverages CNNs for skin lesion classification. The research demonstrates how CNNs can be used to effectively classify dermatological images, particularly in low-data environments, which is a key challenge in medical AI. Cai et. al. (2025) [192] introduced DFNet, a CNN-based residual network that improves feature extraction by incorporating differential features. The study highlights CNNs' role in advanced feature engineering, which is crucial for applications such as facial recognition and object classification. Vishwakarma and Deshmukh (2025) [193] presented CNNM-FDI, a CNN-based fire detection model that enhances real-time safety monitoring. The study explores CNNs' application in environmental monitoring, emphasizing fast-response classification models for early disaster prevention. Ranjan et. al. (2025) [194] merged CNNs, Autoencoders, GANs, and Zero-Shot Learning to improve hyperspectral image classification. The research underscores how CNNs can be augmented with generative models to enhance classification in limited-label datasets, a crucial area in remote sensing applications.

The process of image classification in Convolutional Neural Networks (CNNs) involves a sophisticated interplay of linear algebra, calculus, probability theory, and optimization. The primary goal is to map a high-dimensional input image to a specific class label. Let $\mathbf{I} \in \mathbb{R}^{H \times W \times C}$ represent the input image, where H, W, and C are the height, width, and number of channels (usually 3 for RGB images) of the image, respectively. Each pixel of the image can be represented as $\mathbf{I}(i, j, c)$, which denotes the intensity of the c-th channel at pixel position (i, j). The objective of the CNN is to transform this raw input image into a label, typically one of M classes, using a hierarchical feature extraction process that includes convolutions, nonlinearities, pooling, and fully connected layers.

The convolution operation is central to CNNs and forms the basis for the feature extraction process. Let $\mathbf{K} \in \mathbb{R}^{k \times k \times C}$ be a filter (or kernel) with spatial dimensions $k \times k$ and C channels, where k is typically a small odd integer, such as 3 or 5. The filter $\mathbf{K}$ is convolved with the input image $\mathbf{I}$ to produce a feature map $\mathbf{S} \in \mathbb{R}^{(H-k+1) \times (W-k+1) \times F}$, where F is the number of filters used in the convolution. For a given spatial position (i, j) in the feature map, the convolution operation is defined as:

$$\mathbf{S}_{i,j,f} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \sum_{c=0}^{C-1} \mathbf{I}(i+m, j+n, c) \cdot \mathbf{K}_{m,n,c,f} \tag{371}$$

where $\mathbf{S}_{i,j,f}$ represents the value at position (i, j) in the feature map corresponding to the f-th filter. This operation computes a weighted sum of pixel values in the receptive field of size $k \times k \times C$ around pixel (i, j), where the weights are given by the filter values. The result is a new feature map that captures local patterns such as edges or textures in the image. This local feature extraction is performed for each position (i, j) across the entire image, producing a set of feature maps for each filter. To introduce non-linearity into the network and allow it to model complex functions, the feature map $\mathbf{S}$ is passed through a non-linear activation function, typically the Rectified Linear Unit (ReLU), which is defined element-wise as:

$$\sigma(x) = \max(0, x) \tag{372}$$

This activation function outputs 0 for negative values and passes positive values unchanged, ensuring that the network can learn complex, non-linear relationships. The output of the activation function for the feature map is denoted as $\mathbf{S}^+$, where each element of $\mathbf{S}^+$ is computed as:

$$\mathbf{S}^+_{i,j,f} = \max(0, \mathbf{S}_{i,j,f}) \tag{373}$$

This element-wise operation enhances the network's ability to capture and represent complex patterns, thereby aiding in the learning process. After the convolution and activation, the feature map is downsampled using a pooling operation. The most common form of pooling is max pooling, which selects the maximum value in a local region of the feature map. Given a pooling window of size $p \times p$ and stride s, the max pooling operation for the feature map $\mathbf{S}^+$ is given by:

$$\mathbf{P}_{i,j,f} = \max_{(u,v) \in p \times p} \mathbf{S}^+_{i+u,j+v,f} \tag{374}$$

where $\mathbf{P}$ represents the pooled feature map. This operation reduces the spatial dimensions of the feature map by a factor of p, while preserving the most important features in each region. Pooling serves several purposes, including dimensionality reduction, translation invariance, and noise reduction. It also helps prevent overfitting by limiting the number of parameters and computations in the network.

Once the feature maps are obtained through convolution, activation, and pooling, they are flattened into a one-dimensional vector $\mathbf{F} \in \mathbb{R}^N$, where N is the total number of elements in the pooled feature map. The flattened vector $\mathbf{F}$ is then fed into one or more fully connected layers. These layers perform linear transformations of the input, which are essentially weighted sums of the inputs, followed by the addition of a bias term. The output of a fully connected layer can be expressed as:

$$\mathbf{O} = \mathbf{W} \cdot \mathbf{F} + \mathbf{b} \tag{375}$$

where $\mathbf{W} \in \mathbb{R}^{M \times N}$ is the weight matrix, $\mathbf{b} \in \mathbb{R}^{M}$ is the bias vector, and $\mathbf{O} \in \mathbb{R}^{M}$ is the raw output or logit for each of the M classes. The fully connected layer computes a set of logits for the classes based on the learned features from the convolutional and pooling layers. To convert the logits into class probabilities, a **softmax** function is applied. The softmax function is a generalization of the logistic function to multiple classes and transforms the logits into a probability distribution. The probability of class k is given by:

$$P(y = k \mid \mathbf{O}) = \frac{e^{O_k}}{\sum_{k=1}^{M} e^{O_k}} \tag{376}$$

where O_k is the logit corresponding to class k, and the denominator ensures that the sum of probabilities across all classes equals 1. The class label with the highest probability is selected as the final prediction:

$$y = \arg\max_k P(y = k \mid \mathbf{O}) \tag{377}$$

The prediction is made based on the computed class probabilities, and the network aims to minimize the discrepancy between the predicted probabilities and the true labels during training. To optimize the network's parameters, we minimize a **loss function** that measures the difference between the predicted probabilities and the actual labels. The **cross-entropy loss** is commonly used in classification tasks and is defined as:

$$\mathcal{L} = -\sum_{k=1}^{M} y_k \log P(y = k \mid \mathbf{O}) \tag{378}$$

where y_k is the true label in one-hot encoding, and $P(y = k \mid \mathbf{O})$ is the predicted probability for class k. The goal of training is to minimize this loss function, which corresponds to maximizing the likelihood of the correct class under the predicted probability distribution.

The optimization of the network parameters is performed using **gradient descent** and its variants, such as stochastic gradient descent (SGD), which iteratively updates the parameters based on the gradients of the loss function. The gradients are computed using **backpropagation**, a method that applies the chain rule of calculus to compute the partial derivatives of the loss with respect to each parameter. For a fully connected layer, the gradient of the loss with respect to the weights $\mathbf{W}$ is given by:

$$\nabla_{\mathbf{W}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{O}} \cdot \frac{\partial \mathbf{O}}{\partial \mathbf{W}} = \delta \cdot \mathbf{F}^T \tag{379}$$

where $\delta = \frac{\partial \mathcal{L}}{\partial \mathbf{O}}$ is the error term (also known as the delta) for the logits, and $\mathbf{F}^T$ is the transpose of the flattened feature vector. The parameters are updated using the following rule:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L} \tag{380}$$

where η is the learning rate, controlling the step size of the updates. This process is repeated for each batch of training data until the network converges to a set of parameters that minimize the loss function. Through this complex and iterative process, CNNs are able to learn to classify images by automatically extracting hierarchical features from raw input data. The combination of convolution, activation, pooling, and fully connected layers enables the network to learn increasingly abstract and high-level representations of the input image, ultimately achieving high accuracy in image classification tasks.

7.2.2 Object Detection

Literature Review: Naseer and Jalal (2025) [195] presented a multimodal deep learning framework that integrates RGB-D images for enhanced semantic scene classification. The study leverages a Convolutional Neural Network (CNN)-based object detection model to extract and process features from RGB and depth images, aiming to improve scene recognition accuracy in cluttered and complex environments. By incorporating multimodal inputs, the model effectively addresses the challenges associated with occlusions and background noise, which are common issues in traditional object detection frameworks. The researchers demonstrate how CNNs, when combined with depth-aware semantic information, can significantly enhance object localization and classification performance. Through extensive evaluations, they validate that their framework outperforms conventional single-stream CNNs in various real-world scenarios, making a compelling case for RGB-D integration in deep learning-based object detection systems. Wang and Wang (2025) [196] builds upon the Faster R-CNN object detection framework, introducing a novel improvement that significantly enhances detection accuracy in highly dynamic and complex environments. The study proposes an optimized anchor box generation mechanism, which allows the network to efficiently detect objects of varying scales and aspect ratios, particularly those that are small or heavily occluded. By incorporating a refined region proposal network (RPN), the authors mitigate localization errors and reduce false-positive detections. The paper also explores the impact of feature pyramid networks (FPNs) in hierarchical feature extraction, demonstrating their effectiveness in improving the detection of fine-grained details. The authors conduct an extensive empirical evaluation, comparing their improved Faster R-CNN model against existing object detection architectures, proving its superior performance in terms of precision and recall, particularly for applications involving customized icon generation and user interface design. Ramana et. al. (2025) [197] introduced a Deep Convolutional Graph Neural Network (DCGNN) that integrates Spectral Pyramid Pooling (SPP) and fused keypoint generation to significantly improve 3D object detection performance. The study employs ResNet-50 as the backbone CNN architecture and enhances its feature extraction capability by introducing multi-scale spectral feature aggregation. Through the integration of graph neural networks (GNNs), the model can effectively capture spatial relationships between object components, leading to highly accurate 3D bounding box predictions. The proposed methodology is rigorously evaluated on multiple benchmark datasets, demonstrating its superior ability to handle occlusion, scale variation, and viewpoint changes. Additionally, the paper presents a novel fusion strategy that combines keypoint-based object representation with spectral domain feature embeddings, allowing the model to achieve unparalleled robustness in automated 3D object detection tasks. Shin et. al. (2025) [198] explores the application of deep learning-based object detection in the field of microfluidics and droplet-based bioengineering. The authors utilize YOLOv10n, an advanced CNN-based object detection framework, to develop an automated system for tracking and categorizing double emulsion droplets in high-throughput experimental setups. By fine-tuning the YOLO architecture, the study achieves remarkable improvements in detection sensitivity and classification accuracy, enabling real-time identification of droplet morphology, phase separation dynamics, and stability characteristics. The researchers further introduce an adaptive feature refinement strategy, wherein the CNN model continuously learns from real-time experimental variations, allowing for automated calibration and correction of droplet misclassification. The paper also demonstrates the practical implications of this AI-driven approach in drug delivery systems, encapsulation technologies, and synthetic biology applications. Taca et. al. (2025) [199] provided a comprehensive comparative analysis of multiple CNN-based object detection architectures applied to aphid classification in large-scale agricultural datasets. The researchers evaluate the performance of YOLO, SSD, Faster R-CNN, and EfficientDet, analyzing their trade-offs in terms of accuracy, inference speed, and computational efficiency. Through an extensive experimental setup involving 48,000 annotated images, the authors demonstrate that certain CNN models excel in specific detection scenarios, such as YOLO for real-time aphid localization and Faster R-CNN for high-precision classification. Furthermore, the paper introduces an innovative hybrid ensemble strategy, combining the strengths of multiple CNN architectures to achieve optimal detection performance. The authors validate their findings on real-world agricultural environments, emphasizing the importance of deep learning-driven pest detection in sustainable farming practices. Ulaş et. al. (2025) [200] explored the application of CNN-based object detection in the domain of astronomical time-series analysis, specifically targeting oscillation-like patterns in eclipsing binary light curves. The study systematically evaluates multiple state-of-the-art object detection models, including YOLO, Faster R-CNN, and SSD, to determine their effectiveness in identifying transient light fluctuations that indicate oscillatory behavior in celestial bodies. One of the key contributions of this paper is the introduction of a customized pre-processing pipeline that optimizes raw observational data by removing noise and enhancing feature visibility using wavelet-based signal decomposition techniques. The researchers further implement a hybrid detection mechanism, integrating CNN-based spatial feature extraction with recurrent neural networks (RNNs) to capture both spatial and temporal dependencies within light curve datasets. Extensive validation on large-scale astronomical datasets demonstrates that this approach significantly outperforms traditional statistical methods in detecting oscillatory behavior, paving the way for AI-driven automation in astrophysical event classification. Valensi et. al. (2025) [201] presents an advanced semi-supervised deep learning framework for pleural line detection and segmentation in lung ultrasound (LUS) imaging, leveraging the power of foundation models and CNN-based object detection architectures. The study highlights the shortcomings of conventional fully supervised learning in medical imaging, where annotated datasets are limited and labor-intensive to create. To overcome this challenge, the researchers incorporate a semi-supervised learning strategy, utilizing self-training techniques combined with pseudo-labeling to improve model generalization. The framework employs YOLOv8-based object detection, specifically optimized for medical feature localization, which significantly enhances detection accuracy in cases of low-contrast and high-noise ultrasound images. Furthermore, the study integrates a multi-scale feature extraction strategy, combining convolutional layers with attention mechanisms to ensure precise identification of pleural lines across different imaging conditions. Experimental results demonstrate that this hybrid approach achieves a substantial increase in segmentation accuracy, particularly in detecting subtle abnormalities linked to pneumothorax and pleural effusion, making it a highly valuable tool in clinical diagnostic applications. Arulalan et. al. (2025) [202] proposed an optimized object detection pipeline that integrates a novel convolutional neural network (CNN) architecture, BS2ResNet, with bidirectional LSTM (LTK-Bi-LSTM) for improved spatiotemporal object recognition. Unlike conventional CNN-based object detectors, which focus solely on static spatial features, this study introduces a hybrid deep learning framework that captures both spatial and temporal dependencies. The proposed BS2ResNet model enhances feature extraction by utilizing bottleneck squeeze-and-excitation blocks, which selectively emphasize important spatial information while suppressing redundant feature maps. Additionally, the integration of LTK-Bi-LSTM layers allows the model to effectively capture temporal correlations, making it highly robust for detecting moving objects in dynamic environments. This approach is validated on multiple benchmark datasets, including autonomous driving and video surveillance datasets, where it demonstrates superior performance in handling occlusions, rapid motion, and low-light conditions. The findings indicate that combining deep convolutional networks with sequence-based modeling significantly improves object detection accuracy in complex real-world scenarios, offering critical advancements for applications in intelligent transportation, security, and real-time monitoring. Zhu et. al. (2025) [203] investigated a novel adversarial attack strategy targeting CNN-based object detection models, with a specific focus on binary image segmentation tasks such as salient object detection and camouflage object detection. The paper introduces a high-transferability adversarial attack framework, which generates adversarial perturbations capable of fooling a wide range of deep learning models, including YOLO, Mask R-CNN, and U-Net-based segmentation networks. The researchers employ adversarial example augmentation, where synthetic adversarial patterns are iteratively refined through gradient-based optimization techniques, ensuring that the adversarial attacks remain effective across different architectures and datasets. A particularly important contribution is the introduction of a dual-stage attack pipeline, wherein the model first learns to generate localized, high-impact adversarial noise and then optimizes for cross-model generalization. Extensive experiments demonstrate that this approach significantly degrades detection performance across multiple state-of-the-art models, revealing critical vulnerabilities in current CNN-based object detectors. This research provides valuable insights into deep learning security and underscores the urgent need for robust adversarial defense mechanisms in high-stakes applications such as autonomous systems, medical imaging, and biometric security. Guo et. al. (2025) [204] introduced a deep learning-based agricultural monitoring system, utilizing CNNs for agronomic entity detection and attribute extraction. The research highlights the limitations of traditional rule-based and manual annotation systems in agricultural monitoring, which are prone to errors and inefficiencies. By leveraging CNN-based object detection models, the proposed system enables real-time crop analysis, accurately identifying key agronomic attributes such as plant height, leaf structure, and disease symptoms. A significant innovation in this study is the incorporation of inter-layer feature fusion, wherein multi-scale convolutional features are integrated across different network depths to improve detection robustness under varying lighting and environmental conditions. Additionally, the authors employ a hybrid feature selection mechanism, combining spatial attention networks with spectral domain feature extraction, which enhances the model's ability to distinguish between healthy and diseased crops with high precision. The research is validated through rigorous field trials, demonstrating that CNN-based agronomic monitoring can significantly enhance crop yield predictions, reduce human labor in precision agriculture, and optimize resource allocation in farming operations.

Object detection in Convolutional Neural Networks (CNNs) is a multifaceted computational process that intertwines both classification and localization. It involves detecting objects within an image and predicting their positions via bounding boxes. This task can be mathematically decomposed into the combined problems of classification and regression, both of which are intricately handled by the convolutional layers of a deep neural network. These layers extract hierarchical features at different levels of abstraction, starting from low-level features like edges and corners to high-level semantic concepts such as textures and object parts. These feature maps are then processed by fully connected layers for classification and bounding box regression tasks.

In the mathematical framework, let the input image be represented by a matrix $I \in \mathbb{R}^{H \times W \times C}$, where H, W, and C are the height, width, and number of channels (typically 3 for RGB images). Convolution operations in a CNN serve as the fundamental building blocks to extract spatial hierarchies of features. The convolution operation involves the application of a kernel $K \in \mathbb{R}^{m \times n \times C}$ to the input image, where m and n are the spatial dimensions of the kernel, and C is the number of input channels. The convolution operation is performed by sliding the kernel over the image and computing the element-wise multiplication between the kernel and the image patch, yielding the following equation for the feature map $O(x, y)$:

$$O(x,y) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{c=0}^{C-1} I(x+i, y+j, c) \cdot K(i,j,c) \tag{381}$$

Here, $O(x, y)$ represents the feature map at the location (x, y), which is generated by applying the kernel K. The sum is taken over the spatial extent of the kernel as it slides over the image. This convolutional operation helps the network capture local patterns in the input image, such as edges, corners, and textures, which are crucial for identifying objects. Once the convolution is performed, a non-linear activation function such as the Rectified Linear Unit (ReLU) is applied to introduce non-linearity into the system. The ReLU activation function is given by:

$$f(x) = \max(0, x) \tag{382}$$

This activation function helps the network model complex non-linear relationships between features and is computationally efficient. The application of ReLU ensures that the network can learn complex decision boundaries that are necessary for tasks like object detection.

In CNN-based object detection, the goal is to predict the class of an object and localize its position via a bounding box. The bounding box is parametrized by four coordinates: (x, y) for the center of the box, and w, h for the width and height. The task can be viewed as a twofold problem: (1) classify the object and (2) predict the bounding box that best encodes the object's spatial position. Mathematically, this requires the network to output both class probabilities and bounding box coordinates for each object within the image. The classification task is typically performed using a softmax function, which converts the network's raw output logits z_i for each class i into probabilities $P(y_i|r)$. The softmax function is defined as:

$$P(y_i|r) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \tag{383}$$

where k is the number of possible classes, z_i is the raw score for class i, and $P(y_i|r)$ is the probability that the region r belongs to class y_i. This function ensures that the predicted scores are valid probabilities that sum to one, which allows the network to make a probabilistic decision regarding the class of the object in each region. Simultaneously, the network must also predict the four parameters of the bounding box for each object. The network's predicted bounding box parameters are typically denoted as $\hat{B} = (\hat{x}, \hat{y}, \hat{w}, \hat{h})$, while the ground truth bounding box is denoted by $B = (x, y, w, h)$. The error between the predicted and true bounding boxes is quantified using a loss function, with the smooth L_1 loss being a commonly used metric for bounding box regression. The smooth L_1 loss for each parameter of the bounding box is defined as:

$$\mathcal{L}_{\text{bbox}} = \sum_{i=1}^{4} \text{SmoothL1}(B_i - \hat{B}_i) \tag{384}$$

The smooth L_1 function is defined as:

$$\text{SmoothL1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{if } |x| \geq 1 \end{cases} \tag{385}$$

This loss function is used to reduce the impact of large errors, thereby making the training process more robust. The goal is to minimize this loss during the training phase to improve the network's ability to predict both the class and the bounding box of objects.

For training, a combined loss function is used that combines both the classification loss and the bounding box regression loss. The total loss function can be written as:

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{bbox}} \tag{386}$$

where $\mathcal{L}_{\text{cls}}$ is the classification loss, typically computed using the cross-entropy between the predicted probabilities and the ground truth labels. The cross-entropy loss for classification is given by:

$$\mathcal{L}_{\text{cls}} = -\sum_{i=1}^{k} y_i \log(\hat{y}_i) \tag{387}$$

where y_i is the true label, and $\hat{y}_i$ is the predicted probability for class i. The total objective function for training is therefore a weighted sum of the classification and bounding box regression losses, and the network is optimized to minimize this combined loss function. Object detection architectures like Region-based CNNs (R-CNNs) take a two-stage approach where the task is broken into generating region proposals and classifying these regions. Region Proposal Networks (RPNs) are employed to generate candidate regions $r_1, r_2, \ldots, r_n$, which are then passed through the network to obtain their feature representations. The bounding box refinement and classification for each proposal are then performed by a fully connected layer. The loss function for R-CNNs combines both classification and bounding box regression losses for each proposal, and the objective is to minimize:

$$\mathcal{L}_{\text{R-CNN}} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{bbox}} \tag{388}$$

Another popular architecture, YOLO (You Only Look Once), frames object detection as a single regression task. The image is divided into a grid of $S \times S$ cells, where each cell predicts the class probabilities and bounding box parameters. The output vector for each cell consists of:

$$\hat{y}_i = (x, y, w, h, c, P_1, P_2, \ldots, P_k) \tag{389}$$

where (x, y) are the coordinates of the bounding box center, w and h are the dimensions of the box, c is the confidence score, and $P_1, P_2, \ldots, P_k$ are the class probabilities. The total loss for YOLO combines the classification loss, bounding box regression loss, and confidence loss, which can be written as:

$$\mathcal{L}_{\text{YOLO}} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{bbox}} + \mathcal{L}_{\text{conf}} \tag{390}$$

where $\mathcal{L}_{\text{cls}}$ is the classification loss, $\mathcal{L}_{\text{bbox}}$ is the bounding box regression loss, and $\mathcal{L}_{\text{conf}}$ is the confidence loss, which penalizes predictions with low confidence. This approach allows YOLO to make object detection predictions in a single pass through the network, enabling faster inference. The Single Shot Multibox Detector (SSD) improves on YOLO by generating bounding boxes at multiple feature scales, which allows for detecting objects of varying sizes. The loss function for SSD is similar to that of YOLO, comprising the classification loss and bounding box localization loss, given by:

$$\mathcal{L}_{\text{SSD}} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{loc}} \tag{391}$$

where $\mathcal{L}_{\text{cls}}$ is the classification loss, and $\mathcal{L}_{\text{loc}}$ is the smooth L_1 loss for bounding box regression. This multi-scale approach enhances the network's ability to detect objects at different levels of resolution, improving its robustness to objects of different sizes.

Thus, object detection in CNNs involves a sophisticated architecture of convolution, activation, pooling, and multi-stage loss functions that guide the network in accurately detecting and localizing objects in an image. The choice of architecture and loss function plays a critical role in the performance and efficiency of the detection system, with modern architectures like R-CNN, YOLO, and SSD each offering distinct advantages depending on the application requirements.

7.3 Real-World Applications

7.3.1 Medical Imaging

Literature Review: Yousif et. al. (2024) [205] applied CNNs for melanoma skin cancer detection, integrating a Binary Grey Wolf Optimization (GWO) algorithm to enhance feature selection. It demonstrates the effectiveness of deep learning in classifying dermatoscopic images and highlights feature extraction techniques for accurate classification. Rahman et. al. (2025) [206] gave a systematic review that covers CNN-based leukemia detection using medical imaging. The study compares different deep learning architectures such as ResNet, VGG, and EfficientNet, providing a benchmark for future studies. Joshi and Gowda (2025) [207] introduced an attention-guided Graph CNN (VSA-GCNN) for brain tumor segmentation and classification. It leverages spatial relationships within MRI scans to improve diagnostic accuracy. The use of graph neural networks (GNNs) combined with CNNs is a novel approach in medical imaging. Ng et al. (2025) [208] developed a CNN-based cardiac MRI analysis model to predict ischemic cardiomyopathy without contrast agents. It highlights the ability of deep learning models to extract diagnostic information from non-contrast images, reducing the need for invasive procedures. Nguyen et al. (2025) [209] presented a multi-view tumor region-adapted synthesis model for mammograms using CNNs. The approach enhances breast cancer detection by using 3D spatial feature extraction techniques, improving tumor localization and classification. Chen et. al. (2025) [210] explored CNN-based denoising for medical images using a penalized least squares (PLS) approach. The study applies deep learning for noise reduction in MRI scans, leading to improved clarity in low-signal-to-noise ratio (SNR) images. Pradhan et al. (2025) [211] discussed CNN-based diabetic retinopathy detection. It introduces an Atrous Residual U-Net architecture, enhancing image segmentation performance for early-stage diagnosis of retinal diseases. Örenç et al. (2025) [212] evaluated ensemble CNN models for adenoid hypertrophy detection in X-ray images. It demonstrates transfer learning and feature fusion techniques, which improve CNN-based medical diagnostics. Jiang et al. (2025) [213] introduced a cross-modal attention network for MRI image denoising, particularly effective when some imaging modalities are missing. It highlights cross-domain knowledge transfer using CNNs. Al-Haidri et. al. (2025) [214] developed a CNN-based framework for automatic myocardial fibrosis segmentation in cardiac MRI scans. It emphasizes quantitative feature extraction techniques that enhance precision in cardiac diagnostics.

Convolutional Neural Networks (CNNs) have become an indispensable tool in the field of medical imaging, driven by their ability to automatically learn spatial hierarchies of features directly from image data without the need for handcrafted feature extraction. The convolutional layers in CNNs are designed to exploit the spatial structure of the input data, making them particularly well-suited for tasks in medical imaging, where spatial relationships in images often encode critical diagnostic information. The fundamental building block of CNNs, the convolution operation, is mathematically expressed as

$$S(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} I(i+m, j+n) \cdot K(m,n), \tag{392}$$

where $S(i,j)$ represents the value of the output feature map at position (i,j), $I(i,j)$ is the input image, $K(m,n)$ is the convolutional kernel (a learnable weight matrix), and k denotes the kernel radius (for example, $k = 1$ for a 3×3 kernel). This equation fundamentally captures how local patterns, such as edges, textures, and more complex features, are extracted by sliding the kernel across the image. The convolution operation is performed for each channel of a multi-channel input (e.g., RGB images or multi-modal medical images), and the results are summed across channels, leading to multi-dimensional feature maps. For a 3D input tensor, the convolution extends to include depth:

$$S(i,j,d') = \sum_{d=1}^{D} \sum_{m=-k}^{k} \sum_{n=-k}^{k} I(i+m, j+n, d) \cdot K(m,n,d), \tag{393}$$

where D is the depth of the input tensor, and d' is the depth index of the output feature map. CNNs incorporate nonlinear activation functions after convolutional layers to introduce nonlinearity into the model, allowing it to learn complex mappings. A commonly used activation function is the Rectified Linear Unit (ReLU), mathematically defined as

$$f(x) = \max(0,x). \tag{394}$$

This function ensures sparsity in the activations, which is advantageous for computational efficiency and generalization. More advanced activation functions, such as parametric ReLU (PReLU), extend this concept by allowing learnable parameters for the negative slope:

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ ax & \text{if } x \le 0, \end{cases} \tag{395}$$

where a is a learnable parameter. Pooling layers are employed in CNNs to downsample the spatial dimensions of feature maps, thereby reducing computational complexity and the risk of overfitting. Max pooling is defined mathematically as

$$P(i,j) = \max_{(m,n)\in\mathcal{R}} S(i+m, j+n), \tag{396}$$

where $\mathcal{R}$ is the pooling region (e.g., 2×2). Average pooling computes the mean value instead:

$$P(i,j) = \frac{1}{|\mathcal{R}|} \sum_{(m,n)\in\mathcal{R}} S(i+m, j+n). \tag{397}$$

In medical imaging, CNNs are widely used for image classification tasks such as detecting abnormalities (e.g., tumors, fractures, or lesions). Consider a classification problem where the input is a mammogram image, and the output is a binary label $y \in \{0,1\}$, indicating benign or malignant. The CNN model outputs a probability score $\hat{y}$, computed as

$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}, \tag{398}$$

where z is the output of the final layer before the sigmoid activation. The binary cross-entropy loss function is then used to train the model:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i) \right]. \tag{399}$$

For image segmentation tasks, where the goal is to assign a label to each pixel, architectures such as U-Net are commonly used. U-Net employs an encoder-decoder structure, where the encoder extracts features through a series of convolutional and pooling layers, and the decoder reconstructs the image through upsampling and concatenation operations. The objective function for segmentation is often the Dice coefficient loss, defined as

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2\sum_i p_i g_i}{\sum_i p_i + \sum_i g_i}, \tag{400}$$

where p_i and g_i are the predicted and ground truth values for pixel i, respectively. In the context of image reconstruction, such as in magnetic resonance imaging (MRI), CNNs are used to reconstruct high-quality images from undersampled k-space data. The reconstruction problem is formulated as minimizing the difference between the reconstructed image I_{pred} and the ground truth I_{true}, often using the ℓ_2-norm:

$$\mathcal{L}_{\text{reconstruction}} = \|I_{\text{pred}} - I_{\text{true}}\|_2^2. \tag{401}$$

Generative adversarial networks (GANs) have also been applied to medical imaging, particularly for enhancing image resolution or synthesizing realistic images from noisy inputs. A GAN consists of a generator G and a discriminator D, where G learns to generate images $G(z)$ from latent noise z, and D distinguishes between real and fake images. The loss functions for G and D are given by

$$\mathcal{L}_D = -\mathbb{E}[\log D(x)] - \mathbb{E}[\log(1 - D(G(z)))], \tag{402}$$

$$\mathcal{L}_G = -\mathbb{E}[\log D(G(z))]. \tag{403}$$

Multi-modal imaging, where data from different modalities (e.g., MRI and PET) are combined, further highlights the utility of CNNs. For instance, feature maps from MRI and PET images are concatenated at intermediate layers to exploit complementary information, improving diagnostic accuracy. Attention mechanisms are often incorporated to focus on the most relevant regions of the image. For example, a spatial attention map A_s can be computed as

$$A_s = \sigma(W_2 \cdot \text{ReLU}(W_1 \cdot F + b_1) + b_2), \tag{404}$$

where F is the input feature map, W_1 and W_2 are learnable weight matrices, and b_1 and b_2 are biases. Despite their success, CNNs in medical imaging face challenges, including data scarcity and interpretability. Transfer learning addresses data scarcity by fine-tuning pre-trained models on small medical datasets. Techniques such as Grad-CAM provide interpretability by visualizing regions that influence the network's predictions. Mathematically, Grad-CAM computes the importance of a feature map A^k for a class c as

$$\alpha_k^c = \frac{1}{Z} \sum_{i,j} \frac{\partial y^c}{\partial A_{i,j}^k}, \tag{405}$$

where y^c is the score for class c and Z is a normalization constant. The class activation map is then obtained as

$$L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_k \alpha_k^c A^k \right). \tag{406}$$

In summary, CNNs have transformed medical imaging by enabling automated and highly accurate analysis of complex medical images. Their applications span disease detection, segmentation, reconstruction, and multi-modal imaging, with continued advancements addressing challenges in data efficiency and interpretability. Their mathematical foundations and computational frameworks provide a robust basis for future innovations in this critical field.

7.3.2 Autonomous Vehicles

Literature Review: Ojala and Zhou (2024) [323] proposed a CNN-based approach for detecting and estimating object distances from thermal images in autonomous driving. They developed a deep convolutional model for distance estimation using a single thermal camera and introduced theoretical formulations for thermal imaging data preprocessing within CNN pipelines. Popordanoska and Blaschko (2025) [324] investigated the mathematical underpinnings of CNN calibration in high-risk domains, including autonomous vehicles. They analyzed the confidence calibration problem in CNNs used for self-driving perception and developed a Bayesian-inspired regularization approach to improve CNN decision reliability in autonomous driving. Alfieri et. al. (2024) [325] explored deep reinforcement learning (DRL) methods with CNNs for optimizing route planning in autonomous vehicles. They bridged CNN-based vision models with Deep Q-Learning, enabling adaptive path optimization in real-world driving conditions and established a novel theoretical connection between Q-learning and CNN-based object detection for autonomous navigation. Zanardelli (2025) [326] examined decision-making frameworks using CNNs in autonomous vehicle systems. He developed a statistical model integrating CNNs with reinforcement learning to improve self-driving car decision-making and provided a rigorous probabilistic analysis of how CNNs handle uncertainty in real-world driving environments. Norouzi et. al. (2025) [327] analyzed the role of transfer learning in CNN models for autonomous vehicle perception. They introduced pre-trained CNNs for vehicle object detection using multi-sensor data fusion and provided a rigorous theoretical justification for integrating Kalman filtering and Dempster-Shafer theory with CNNs. Wang et. al. (2024) [328] investigated the mathematics of uncertainty quantification in CNN-based perception models for self-driving cars. They used Bayesian CNNs to model uncertainty in semantic segmentation for autonomous driving and proposed a Dempster-Shafer theory-based fusion mechanism for combining multiple CNN outputs. Xia et. al. [329] integrated CNN-based perception models with reinforcement learning (RL) to improve autonomous vehicle trajectory tracking. They uses CNNs for lane detection and integrated them into a RL-based path planner. They also established a theoretical framework linking CNN-based scene recognition to control theory. Liu et. al. (2024) [330] introduced a CNN-based multi-view feature extraction framework for spatial-temporal analysis in self-driving cars. They developed a hybrid CNN-graph attention model to extract temporal driving patterns. They also made theoretical advancements in multi-view learning and feature fusion for CNNs in autonomous vehicle decision-making. Chakraborty and Deka (2025)

[331] applied CNN-based multimodal sensor fusion to autonomous vehicles and UAVs for real-time navigation. They did theoretical analysis of CNN feature fusion mechanisms for real-time perception and developed mask region-based CNNs (Mask-RCNNs) for enhanced object recognition in autonomous navigation. Mirindi et. al. (2025) [332] investigated the role of CNNs and AI in smart autonomous transportation. They did theoretical discussion on the Unified Theory of AI Adoption in autonomous driving and introduced hybrid Recurrent Neural Networks (RNNs) and CNN architectures for vehicle trajectory prediction.

Convolutional Neural Networks (CNNs) are fundamental in the implementation of autonomous vehicles, forming the backbone of the perception and decision-making systems that allow these vehicles to interpret and respond to their environment. At the core of a CNN is the convolution operation, which mathematically transforms an input image or signal into a feature map, allowing the extraction of spatial hierarchies of information. The convolution operation in its continuous form is defined as:

$$s(t) = \int_{-\infty}^{\infty} x(\tau) w(t-\tau)\, d\tau, \tag{407}$$

where $x(\tau)$ represents the input, $w(t-\tau)$ is the filter or kernel, and $s(t)$ is the output feature. In the discrete domain, especially for image processing, this operation can be written as:

$$S(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} X(i+m, j+n) \cdot W(m,n), \tag{408}$$

where $X(i,j)$ denotes the pixel intensity at coordinate (i,j) of the input image, and $W(m,n)$ represents the convolutional kernel values. This operation enables the detection of local patterns such as edges, corners, or textures, which are then aggregated across layers to recognize complex features like shapes and objects. In the context of autonomous vehicles, CNNs process sensor data from cameras, LiDAR, and radar to identify critical features such as other vehicles, pedestrians, road signs, and lane boundaries. For object detection, CNN-based architectures such as YOLO (You Only Look Once) and Faster R-CNN employ a backbone network like ResNet, which uses successive convolutional layers to extract hierarchical features from the input image. The object detection task involves two primary outputs: bounding box coordinates and object class probabilities. Mathematically, bounding box regression is modeled as a multi-task learning problem. The loss function for bounding box regression is often formulated as:

$$L_{\text{reg}} = \sum_{i=1}^{N} \sum_{j \in \{x,y,w,h\}} \text{SmoothL1}(t_{ij} - \hat{t}_{ij}), \tag{409}$$

where t_{ij} and $\hat{t}_{ij}$ are the ground-truth and predicted bounding box parameters (e.g., center coordinates x, y and dimensions w, h). Simultaneously, the classification loss, typically cross-entropy, is computed as:

$$L_{\text{cls}} = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(p_{i,c}), \tag{410}$$

where $y_{i,c}$ is a binary indicator for whether the object at index i belongs to class c, and $p_{i,c}$ is the predicted probability. The total loss function is a weighted combination:

$$L_{\text{total}} = \alpha L_{\text{reg}} + \beta L_{\text{cls}}. \tag{411}$$

Semantic segmentation, another critical task, requires pixel-level classification to assign a label (e.g., road, vehicle, pedestrian) to each pixel in an image. Fully Convolutional Networks (FCNs) or U-Net architectures are commonly used for this purpose. These architectures utilize an encoder-decoder structure where the encoder extracts spatial features, and the decoder reconstructs the spatial resolution to generate pixel-wise predictions. The loss function for semantic segmentation is a sum over all pixels and classes, given as:

$$L = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} \log(p_{i,c}), \tag{412}$$

where $y_{i,c}$ is the ground-truth binary label for pixel i and class c, and $p_{i,c}$ is the predicted probability. Advanced architectures also employ skip connections to preserve high-resolution spatial information, enabling sharper segmentation boundaries.

Depth estimation is essential for autonomous vehicles to understand the 3D structure of their surroundings. CNNs are used to predict depth maps from monocular images or stereo pairs. The depth estimation process is modeled as a regression problem, where the loss function is designed to minimize the difference between the predicted depth $\hat{d}_i$ and the ground-truth depth d_i. A commonly used loss function for this task is the scale-invariant loss:

$$L_{\text{scale-inv}} = \frac{1}{n}\sum_{i=1}^{n}\left(\log d_i - \log \hat{d}_i\right)^2 - \frac{1}{n^2}\left(\sum_{i=1}^{n}\left(\log d_i - \log \hat{d}_i\right)\right)^2. \tag{413}$$

This loss ensures that the relative depth differences are minimized, which is critical for accurate 3D reconstruction. Lane detection, another critical application, uses CNNs to detect road lanes and boundaries. The task often involves predicting the lane markings as polynomial curves. CNNs process the input image to extract lane features, and post-processing involves fitting a curve, such as:

$$y = ax^2 + bx + c, \tag{414}$$

where a, b, c are the coefficients predicted by the network. The fitting process minimizes an error function, typically the sum of squared differences between the detected lane points and the curve:

$$E = \sum_{i=1}^{N}(y_i - (ax_i^2 + bx_i + c))^2. \tag{415}$$

In autonomous vehicles, these CNN tasks are integrated into an end-to-end pipeline. The input data from cameras, LiDAR, and radar is first processed using CNNs to extract features relevant to the vehicle's perception. The outputs, including object detections, semantic maps, depth maps, and lane boundaries, are then passed to the planning module, which computes the vehicle's trajectory. For instance, detected objects provide information about obstacles, while lane boundaries guide path planning algorithms. The planning process involves solving optimization problems where the objective function incorporates constraints from the CNN outputs. For example, a trajectory optimization problem may minimize a cost function:

$$J = \int_0^T \left(w_1\dot{x}^2 + w_2\dot{y}^2 + w_3 c(t)\right) dt, \tag{416}$$

where $\dot{x}$ and $\dot{y}$ are the lateral and longitudinal velocities, and $c(t)$ is a collision penalty based on object detections.

In conclusion, CNNs provide the computational framework for perception tasks in autonomous vehicles, enabling real-time interpretation of complex sensory data. By leveraging mathematical principles of convolution, loss optimization, and hierarchical feature extraction, CNNs transform raw sensor data into actionable insights, paving the way for safe and efficient autonomous navigation.

7.4 Popular CNN Architectures

Literature Review: Choudhury et. al. (2024) [333] presented a comparative theoretical study of CNN architectures, including AlexNet, VGG, and ResNet, for satellite-based aircraft identification. They analyzed the architectural differences and learning strategies used in VGG, AlexNet, and ResNet and theoretically explained how VGG's depth, AlexNet's feature extraction, and ResNet's residual learning contribute to CNN advancements. Almubarok and Rosiani (2024) [334] discussed the computational efficiency of CNN architectures, particularly focusing on AlexNet, VGG, and ResNet in comparison to MobileNetV2. They established theoretical efficiency trade-offs between depth, parameter count, and accuracy in AlexNet, VGG, and ResNet and highlighted ResNet's advantage in optimization due to skip connections, compared to AlexNet and VGG's traditional deep structures. Ding (2024) [335] explored CNN architectures (AlexNet, VGG, and ResNet) for medical image classification, particularly in Traditional Chinese Medicine (TCM). He introduced ResNet-101 with Squeeze-and-Excitation (SE) blocks, expanding theoretical understanding of deep feature representations in CNNs and discussed VGG's weight-sharing strategy and AlexNet's layered feature extraction, improving classification accuracy. He et. al. (2015) [336] introduced Residual Learning, demonstrating how deep CNNs benefit from identity mappings to tackle vanishing gradients. They formulated the mathematical justification of residual blocks in deep networks and Established the theoretical backbone of ResNet's identity mapping for deep optimization. Simonyan and Zisserman (2014) [148] presented the VGG architecture, which demonstrates how depth improvement enhances feature extraction. They developed the theoretical formulation of increasing CNN depth and its impact on feature hierarchies and provided an analytical framework for receptive field expansion in deep CNNs. Krizhevsky et. al. (2012) [337] introduced AlexNet, the first CNN model to achieve state-of-the-art performance in ImageNet classification. They introduced ReLU activation as a breakthrough in CNN training and established dropout regularization theory, preventing overfitting in deep networks. Sultana et. al. (2019) [338] compared the feature extraction strategies of AlexNet, VGG, and ResNet for object recognition. They gave theoretical explanation of hierarchical feature learning in CNN architectures and examined VGG's use of small convolutional filters and how it impacts feature map depth. Sattler et. al. (2019) [339] investigated the fundamental limitations of CNN architectures such as AlexNet, VGG, and ResNet. They established formal constraints on convolutional filters in CNNs and developed a theoretical model for CNN generalization error in classification tasks.

7.4.1 AlexNet

The **AlexNet Convolutional Neural Network (CNN)** is a deep learning model that operates on raw pixel values to perform image classification. Given an input image, represented as a 3D tensor $\mathbf{I}_0 \in \mathbb{R}^{H\times W\times C}$, where H is the height, W is the width, and C represents the number of input channels (typically $C = 3$ for RGB images), the network performs a series of operations, such as convolutions, activation functions, pooling, and fully connected layers, to transform this input into a final output vector $\mathbf{y} \in \mathbb{R}^K$, where K is the number of output classes. The objective of AlexNet is to minimize a loss function that measures the discrepancy between the predicted output and the true label, typically using the **cross-entropy loss** function.

At the heart of AlexNet's architecture are the **convolutional layers**, which are designed to learn local patterns in the image by convolving a set of filters over the input image. Specifically, the first convolutional layer performs a convolution of the input image $\mathbf{I}_0$ with a set of filters $\mathbf{W}_1^{(k)} \in \mathbb{R}^{F_1\times F_1\times C}$, where F_1 is the size of the filter and C is the number of channels in the input. The convolution operation for a given filter $\mathbf{W}_1^{(k)}$ and input image $\mathbf{I}_0$ at position (i, j) is defined as:

$$Y_1^{(k)}(i,j) = \sum_{u=1}^{F_1}\sum_{v=1}^{F_1}\sum_{c=1}^{C} W_1^{(k)}(u,v,c)\cdot I_0(i+u-1, j+v-1, c) + b_1^{(k)} \tag{417}$$

where $b_1^{(k)}$ is the bias term for the k-th filter, and the output of this convolution is a feature map $Y_1^{(k)}(i,j)$ that captures the response of the filter at each spatial location (i,j). The result of this convolution operation is a set of feature maps $Y_1^{(k)} \in \mathbb{R}^{H' \times W'}$, where the dimensions of the output are $H' = H - F_1 + 1$ and $W' = W - F_1 + 1$ if no padding is applied. Subsequent to the convolutional operation, the output feature maps $Y_1^{(k)}$ are passed through a **ReLU (Rectified Linear Unit)** activation function, which introduces non-linearity into the network. The ReLU function is defined as:

$$\text{ReLU}(z) = \max(0, z) \tag{418}$$

This function transforms negative values in the feature map $Y_1^{(k)}$ into zero, while leaving positive values unchanged, thus allowing the network to model complex, non-linear patterns in the data. The output of the ReLU activation function is denoted by $A_1^{(k)}(i,j) = \text{ReLU}(Y_1^{(k)}(i,j))$. Following the activation function, a **max-pooling** operation is performed to downsample the feature maps and reduce their spatial dimensions. Given a pooling window of size $P \times P$, the max-pooling operation computes the maximum value in each window, which is mathematically expressed as:

$$Y_1^{\text{pool}}(i,j) = \max\left(A_1^{(k)}(i',j') : (i',j') \in \text{pooling window}\right) \tag{419}$$

where $A_1^{(k)}$ is the feature map after ReLU, and the resulting pooled output $Y_1^{\text{pool}}(i,j)$ has reduced spatial dimensions, typically $H'' = \frac{H'}{P}$ and $W'' = \frac{W'}{P}$. This operation helps retain the most important features while discarding irrelevant spatial details, which makes the network more robust to small translations in the input image. The convolutional and pooling operations are repeated across multiple layers, with each layer learning progressively more complex patterns from the input data. In the second convolutional layer, for example, we convolve the feature maps from the first layer $A_1^{(k)}$ with a new set of filters $\mathbf{W}_2^{(k)} \in \mathbb{R}^{F_2 \times F_2 \times K_1}$, where K_1 is the number of feature maps produced by the first convolutional layer. The convolution for the second layer is expressed as:

$$Y_2^{(k)}(i,j) = \sum_{u=1}^{F_2} \sum_{v=1}^{F_2} \sum_{c=1}^{K_1} W_2^{(k)}(u,v,c) \cdot A_1^{(c)}(i+u-1, j+v-1) + b_2^{(k)} \tag{420}$$

This process is iterated for each subsequent convolutional layer, where each new set of filters learns higher-level features, such as edges, textures, and object parts. The activation maps produced by each convolutional layer are passed through the ReLU activation function, and max-pooling is applied after each convolutional layer to reduce the spatial dimensions.

After the last convolutional layer, the feature maps are **flattened** into a 1D vector $\mathbf{a}_f \in \mathbb{R}^N$, where N is the total number of activations across all channels and spatial dimensions. This flattened vector is then passed to **fully connected (FC) layers** for classification. Each fully connected layer performs a linear transformation, followed by a non-linear activation. The output of the i-th neuron in the fully connected layer is given by:

$$z_i = \sum_{j=1}^{N} W_{ij} \cdot a_f(j) + b_i \tag{421}$$

where W_{ij} is the weight connecting neuron j in the previous layer to neuron i in the current layer, and b_i is the bias term. The output of the fully connected layer is a vector of class scores $\mathbf{z} \in \mathbb{R}^K$, which represents the unnormalized log-probabilities of the input image belonging to each class. To convert these scores into a valid probability distribution, the **softmax** function is applied:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{422}$$

The softmax function ensures that the output values are in the range $[0, 1]$ and sum to 1, thus representing a probability distribution over the K classes. The final output of the network is a probability vector $\hat{\mathbf{y}} \in \mathbb{R}^K$, where each element $\hat{y}_i$ corresponds to the predicted probability that the input image belongs to class i. To train the AlexNet model, the network minimizes the **cross-entropy loss** function between the predicted probabilities $\hat{\mathbf{y}}$ and the true labels $\mathbf{y}$, which is given by:

$$L = -\sum_{i=1}^{K} y_i \log(\hat{y}_i) \tag{423}$$

where y_i is the true label (1 if the image belongs to class i, 0 otherwise), and $\hat{y}_i$ is the predicted probability for class i. The goal of training is to adjust the weights W and biases b in the network to minimize this loss. The parameters of the network are updated using **gradient descent**. To compute the gradients, the **backpropagation** algorithm is used. The gradient of the loss with respect to the weights W in a fully connected layer is given by:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial W} \tag{424}$$

where $\frac{\partial L}{\partial z}$ is the gradient of the loss with respect to the output of the layer, and $\frac{\partial z}{\partial W}$ is the gradient of the output with respect to the weights. These gradients are then used to update the weights using the gradient descent update rule:

$$W \leftarrow W - \eta \cdot \frac{\partial L}{\partial W} \tag{425}$$

where η is the learning rate. This process is repeated iteratively for each layer of the network.

Regularization techniques such as **dropout** are often applied to prevent overfitting during training. Dropout involves randomly setting a fraction of the activations to zero during each training step, which helps prevent the network from relying too heavily on any one feature and encourages the model to learn more robust features. Once trained, the AlexNet model can be used to classify new images by passing them through the network and selecting the class with the highest probability. The combination of convolutional layers, ReLU activations, pooling, fully connected layers, and softmax activation makes AlexNet a powerful and efficient architecture for image classification tasks.

7.4.2 ResNet

At the heart of the ResNet architecture lies the notion of *residual learning*, where instead of learning the direct transformation $\mathbf{y} = f(\mathbf{x}; \mathbf{W})$, the network learns the residual function $\mathcal{F}(\mathbf{x}, \mathbf{W})$, i.e., the difference between the input and output. The network output $\mathbf{y}$ can therefore be expressed as:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}; \mathbf{W}) + \mathbf{x} \tag{426}$$

This formulation represents the core difference from traditional neural networks where the model learns a mapping directly from the input $\mathbf{x}$ to the output $\mathbf{y}$. The introduction of the identity shortcut connection $\mathbf{x}$ introduces a powerful mechanism by which the network can learn the residual, and if the optimal residual transformation is the identity function, the network can essentially learn $\mathbf{y} = \mathbf{x}$, improving optimization. This reduces the challenge of training deeper networks, where deep layers often lead to vanishing gradients, because the gradient can propagate directly through these shortcuts, bypassing intermediate layers.

Let's formalize this residual learning. Let the input to the residual block be $\mathbf{x}_l$ and the output

$\mathbf{y}_l$. In a conventional neural network, the transformation from input to output at the l-th layer would be:

$$\mathbf{y}_l = \mathcal{F}(\mathbf{x}_l; \mathbf{W}_l) \tag{427}$$

where $\mathcal{F}$ represents the function learned by the layer, parameterized by $\mathbf{W}_l$. In contrast, for ResNet, the output is the sum of the learned residual function $\mathcal{F}(\mathbf{x}_l; \mathbf{W}_l)$ and the input $\mathbf{x}_l$ itself, yielding:

$$\mathbf{y}_l = \mathcal{F}(\mathbf{x}_l; \mathbf{W}_l) + \mathbf{x}_l \tag{428}$$

This addition of the identity shortcut connection enables the network to bypass layers if needed, facilitating the learning process and addressing the vanishing gradient issue. To formalize the optimization problem, we define the residual learning objective as the minimization of the loss function $\mathcal{L}$ with respect to the parameters $\mathbf{W}_l$:

$$\mathcal{L} = \sum_{i=1}^{N} \mathcal{L}_i(\mathbf{y}_i, \mathbf{t}_i) \tag{429}$$

where N is the number of training samples, $\mathbf{t}_i$ are the target outputs, and $\mathcal{L}_i$ is the loss for the i-th sample. The training process involves adjusting the parameters $\mathbf{W}_l$ via gradient descent, which in turn requires the gradients of the loss function with respect to the network parameters. The gradient of $\mathcal{L}$ with respect to $\mathbf{W}_l$ can be expressed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}_i}{\partial \mathbf{y}_i} \cdot \frac{\partial \mathbf{y}_i}{\partial \mathbf{W}_l} \tag{430}$$

Since the residual block adds the input directly to the output, the derivative of the output with respect to the weights $\mathbf{W}_l$ is given by:

$$\frac{\partial \mathbf{y}_l}{\partial \mathbf{W}_l} = \frac{\partial \mathcal{F}(\mathbf{x}_l; \mathbf{W}_l)}{\partial \mathbf{W}_l} \tag{431}$$

Now, let's explore how this addition of the residual connection directly influences the backpropagation process. In a traditional feedforward network, the backpropagated gradients for each layer depend solely on the output of the preceding layer. However, in a residual network, the gradient flow is enhanced because the identity mapping $\mathbf{x}_l$ is directly passed to the subsequent layer. This ensures that the gradients will not be lost as the network deepens, a phenomenon that becomes critical in very deep networks. The gradient with respect to the loss $\mathcal{L}$ at layer l is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_l} \cdot \frac{\partial \mathbf{y}_l}{\partial \mathbf{x}_l} \tag{432}$$

Since $\mathbf{y}_l = \mathcal{F}(\mathbf{x}_l; \mathbf{W}_l) + \mathbf{x}_l$, the derivative of $\mathbf{y}_l$ with respect to $\mathbf{x}_l$ is:

$$\frac{\partial \mathbf{y}_l}{\partial \mathbf{x}_l} = \mathbf{I} + \frac{\partial \mathcal{F}(\mathbf{x}_l; \mathbf{W}_l)}{\partial \mathbf{x}_l} \tag{433}$$

where $\mathbf{I}$ is the identity matrix. This ensures that the gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{x}_l}$ can propagate more easily through the network, as it is now augmented by the identity matrix term. Thus, this term helps preserve the gradient's magnitude during backpropagation, solving the vanishing gradient problem that typically arises in deep networks. Furthermore, to ensure that the dimensions of the input and output of a residual block match, especially when the number of channels changes, ResNet introduces *projection shortcuts*. These are used when the dimensionality of $\mathbf{x}_l$ and $\mathbf{y}_l$ do not align, typically through a 1×1 convolution. The projection shortcut modifies the residual block's output to be:

$$\mathbf{y}_l = \mathcal{F}(\mathbf{x}_l; \mathbf{W}_l) + \mathbf{W}_x \cdot \mathbf{x}_l \tag{434}$$

where $\mathbf{W}_x$ is a convolutional filter, and $\mathcal{F}(\mathbf{x}_l; \mathbf{W}_l)$ is the residual transformation. The introduction of the 1×1 convolution ensures that the input $\mathbf{x}_l$ is mapped to the appropriate dimensionality, while still benefiting from the residual learning framework. The ResNet architecture can be extended by stacking multiple residual blocks. For a network with L layers, the output after passing through the entire network can be written recursively as:

$$\mathbf{y}^{(L)} = \mathbf{x} + \mathcal{F}(\mathbf{y}^{(L-1)}; \mathbf{W}_L) \tag{435}$$

where $\mathbf{y}^{(L-1)}$ is the output after $L-1$ layers. The recursive nature of this formula ensures that the network's output is built layer by layer, with each layer contributing a transformation relative to the input passed to it. Mathematically, the gradient of the loss function with respect to the parameters in deep residual networks can be expressed recursively, where each layer's gradient involves contributions from the identity shortcut connection. This facilitates the training of very deep networks by maintaining a stable and consistent flow of gradients during the backpropagation process.

Thus, the *Residual Neural Network (ResNet)* significantly improves the trainability of deep neural networks by introducing residual learning, allowing the network to focus on learning the difference between the input and output rather than the entire transformation. This approach, combined with identity shortcut connections and projection shortcuts for dimensionality matching, ensures that gradients flow effectively through the network, even in very deep architectures. The resulting ResNet architecture has been proven to enable the training of networks with hundreds of layers, yielding impressive performance on a wide range of tasks, from image classification to semantic segmentation, while mitigating issues such as vanishing gradients. Through its recursive structure and rigorous mathematical formulation, ResNet has become a foundational architecture in modern deep learning.

7.4.3 VGG

The **Visual Geometry Group (VGG) Convolutional Neural Network (CNN)**, introduced by Simonyan and Zisserman in 2014, presents a detailed exploration of the effect of depth on the performance of deep neural networks, specifically within the context of computer vision tasks such as image classification. The VGG architecture is grounded in the hypothesis that deeper networks, when constructed with small, consistent convolutional kernels, are more capable of capturing hierarchical patterns in data, particularly in the domain of visual recognition. In contrast to other CNN architectures, VGG prioritizes the usage of small 3×3 convolution filters (with a stride of 1) stacked in increasing depth, rather than relying on larger filters (e.g., 5×5 or 7×7), thus offering computational benefits without sacrificing representational power. This design choice inherently encourages sparse local receptive fields, which ensures a richer learning capacity when extended across deeper layers.

Let $I \in \mathbb{R}^{H \times W \times C}$ represent an input image of height H, width W, and C channels, where the channels correspond to different color representations (e.g., RGB for $C = 3$). For the convolution operation applied at a particular layer k, the output feature map $O^{(k)}$ can be computed by convolving the input I with a set of kernels $K^{(k)}$ corresponding to the k-th layer. The convolution for each spatial location i, j can be described as:

$$O_{i,j}^{(k)} = \sum_{u=1}^{k_h} \sum_{v=1}^{k_w} \sum_{c'=1}^{C_{\text{in}}} K_{u,v,c',c}^{(k)} I_{i+u,j+v,c'} + b_c^{(k)} \tag{436}$$

where $O_{i,j}^{(k)}$ is the output value at location (i, j) of the feature map for the k-th filter, $K_{u,v,c',c}^{(k)}$ is the u, v-th spatial element of the c'-to-c filter in layer k, and $b_c^{(k)}$ represents the bias term for the output channel c. The convolutional layer's kernel $K^{(k)}$ is typically initialized with small values and learned during training, while the bias $b^{(k)}$ is added to shift the activation of the neuron. A key aspect of the VGG architecture is that these convolution layers are consistently followed by non-linear **ReLU (Rectified Linear Unit)** activation functions, which introduce local non-linearity to the model. The ReLU function is mathematically defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{437}$$

This transformation is applied element-wise, ensuring that negative values are mapped to zero, which, as an effect, activates only positive feature responses. The non-linearity introduced by ReLU aids the network in learning complex patterns and overcoming issues such as vanishing gradients that often arise in deeper networks. In VGG, the network is constructed by stacking these convolutional layers with ReLU activations. Each convolution layer is followed by **max-pooling** operations, typically with 2 × 2 filters and a stride of 2. Max-pooling reduces the spatial dimensions of the feature maps and extracts the most significant features from each region of the image. The max-pooling operation is mathematically expressed as:

$$O_{i,j} = \max_{(u,v)\in P} I_{i+u,j+v} \tag{438}$$

where P is the pooling window, and $O_{i,j}$ is the pooled value at position (i, j). The pooling operation performs downsampling, ensuring translation invariance while retaining the most prominent features. The effect of this pooling operation is to reduce computational complexity, lower the number of parameters, and make the network invariant to small translations and distortions in the input image. The architecture of VGG typically culminates in a series of **fully connected (FC) layers** after several convolutional and pooling layers have extracted relevant features from the input image. Let the output of the final convolutional layer, after flattening, be denoted as $X \in \mathbb{R}^d$, where d represents the dimensionality of the feature vector obtained by flattening the last convolutional feature map. The fully connected layers then transform this vector into the output, as expressed by:

$$\mathbf{O} = \mathbf{W}\mathbf{X} + \mathbf{b} \tag{439}$$

where $\mathbf{W} \in \mathbb{R}^{d' \times d}$ is the weight matrix of the fully connected layer, $\mathbf{b} \in \mathbb{R}^{d'}$ is the bias vector, and $\mathbf{O} \in \mathbb{R}^{d'}$ is the output vector. The output vector $\mathbf{O}$ represents the unnormalized scores for each of the d' possible classes in a classification task. This is typically followed by the application of a **softmax** function to convert these raw scores into a probability distribution:

$$\sigma(o_i) = \frac{e^{o_i}}{\sum_{j=1}^{d'} e^{o_j}} \tag{440}$$

where o_i is the score for class i, and the softmax function ensures that the outputs are positive and sum to one, facilitating their interpretation as class probabilities. This softmax function is a crucial step in multi-class classification tasks as it normalizes the output into a probabilistic format. During the training phase, the model minimizes the **cross-entropy loss** between the predicted probabilities and the actual class labels, often represented as one-hot encoded vectors. The cross-entropy loss is given by:

$$L = -\sum_{i=1}^{d'} y_i \log(p_i) \tag{441}$$

where y_i is the true label for class i in one-hot encoded form, and p_i is the predicted probability for class i. This loss function is the appropriate objective for classification tasks, as it measures the difference between the true and predicted probability distributions. The optimization of the parameters in the VGG network is carried out using **stochastic gradient descent (SGD)** or its variants. The weight update rule in gradient descent is:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} L \tag{442}$$

where η is the learning rate, and $\nabla_{\mathbf{W}} L$ is the gradient of the loss with respect to the weights. The gradient is computed through **backpropagation**, applying the chain rule of derivatives to propagate errors backward through the network, updating the weights at each layer based on the contribution of each parameter to the final output error.

A key advantage of the VGG architecture lies in its use of smaller, deeper layers compared to previous networks like AlexNet, which used larger convolution filters. By using multiple small kernels (such as 3×3), the VGG network can create richer representations without exponentially increasing the number of parameters. The depth of the network, achieved by stacking these small convolution filters, enables the model to extract increasingly abstract and hierarchical features from the raw pixel data. Despite its success, VGG's computational demands are relatively high due to the large number of parameters, especially in the fully connected layers. The fully connected layers, which connect every neuron in one layer to every neuron in the next, account for a significant portion of the model's total parameters. To mitigate this limitation, later architectures, such as **ResNet**, introduced **skip connections**, which allow gradients to flow more efficiently through the network, thus enabling even deeper architectures without incurring the same computational costs. Nevertheless, the VGG network set an important precedent in the design of deep convolutional networks, demonstrating the power of deep architectures and the effectiveness of small convolutional filters. The model's simplicity and straightforward design have influenced subsequent architectures, reinforcing the notion that deeper models, when carefully constructed, can achieve exceptional performance on complex tasks like image classification, despite the challenges posed by computational cost and model complexity.

8 Recurrent Neural Networks (RNNs)

Literature Review: Schmidhuber (2015) [114] provided an extensive historical perspective on neural networks, including RNNs. Schmidhuber describes key architectures such as Long Short-Term Memory (LSTM) and their importance in solving the vanishing gradient problem. He also explains fundamental learning algorithms for training RNNs and provides insights into applications like sequence prediction and speech recognition. Lipton et. al. (2015) [264] offers a rigorous critique of RNNs and their various implementations. The authors discuss the fundamental challenges of training RNNs, including long-range dependencies and computational inefficiencies. The paper also presents benchmarks comparing different architectures like vanilla RNNs, LSTMs, and GRUs. offers a rigorous critique of RNNs and their various implementations. The authors discuss the fundamental challenges of training RNNs, including long-range dependencies and computational inefficiencies. The paper also presents benchmarks comparing different architectures like vanilla RNNs, LSTMs, and GRUs. Pascanu et. al. (2013) [265] formally analyzes why training RNNs is difficult, particularly focusing on the vanishing and exploding gradient problem. The authors propose gradient clipping as a practical solution and discuss ways to improve training efficiency for RNNs. Goodfellow et. al. (2016) [112] in their book book dedicates an entire chapter to recurrent neural networks, discussing their theoretical foundations, backpropagation through time (BPTT), and key architectures such as LSTMs and GRUs. It also provides mathematical derivations of optimization techniques used in training deep RNNs. Jaeger (2001) [266] introduced the Echo State Network (ESN), an alternative recurrent architecture that requires only the output weights to be trained. The ESN approach has become highly influential in RNN research, particularly for solving stability and efficiency problems. Hochreiter and Schmidhuber (1997) [267] introduced the LSTM architecture, which solves the vanishing gradient problem in RNNs by incorporating memory cells with gating mechanisms. LSTMs are now a standard in sequence modeling tasks, such as speech recognition and natural language processing. Kawakami (2008) [268] provided a deep dive into supervised learning techniques for RNNs, particularly for sequence labeling problems. Graves discusses Connectionist Temporal Classification (CTC), a popular loss function for RNN-based speech and handwriting recognition. Bengio et. al. (1994) [269] mathematically proved why RNNs struggle with learning long-term dependencies. It identifies the root causes of the vanishing and exploding gradient problems, setting the stage for future architectures like LSTMs. Bhattamishra et. al. (2020) [270] rigorously compared the theoretical capabilities of RNNs and Transformers. The authors analyze expressiveness, memory retention, and training efficiency, providing insights into why Transformers are increasingly replacing RNNs in NLP. Siegelmann (1993) [271] provided a rigorous theoretical treatment of RNNs, analyzing their convergence properties, stability conditions, and computational complexity. It discusses mathematical frameworks for understanding RNN generalization and optimization challenges.

8.1 Key Concepts

Recurrent Neural Networks (RNNs) are a class of neural architectures specifically designed for processing sequential data, leveraging their recursive structure to model temporal dependencies. At the core of an RNN lies the concept of a hidden state $\mathbf{h}_t \in \mathbb{R}^m$, which evolves over time as a function of the current input $\mathbf{x}_t \in \mathbb{R}^n$ and the previous hidden state $\mathbf{h}_{t-1}$. This evolution is governed by the recurrence relation:

$$\mathbf{h}_t = f_h \left(\mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{h}_{t-1} + \mathbf{b}_h \right), \tag{443}$$

where $\mathbf{W}_{xh} \in \mathbb{R}^{m \times n}$ is the input-to-hidden weight matrix, $\mathbf{W}_{hh} \in \mathbb{R}^{m \times m}$ is the hidden-to-hidden weight matrix, $\mathbf{b}_h \in \mathbb{R}^m$ is the bias vector, and f_h is a non-linear activation function, typically

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{444}$$

or the rectified linear unit $\mathrm{ReLU}(x) = \max(0, x)$. The recursive nature of this update equation ensures that $\mathbf{h}_t$ inherently encodes information about the sequence $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_t\}$, allowing the network to maintain a dynamic representation of past inputs. The output $\mathbf{y}_t \in \mathbb{R}^o$ at time t is computed as:

$$\mathbf{y}_t = f_y \left(\mathbf{W}_{hy} \mathbf{h}_t + \mathbf{b}_y \right), \tag{445}$$

where $\mathbf{W}_{hy} \in \mathbb{R}^{o \times m}$ is the hidden-to-output weight matrix, $\mathbf{b}_y \in \mathbb{R}^o$ is the output bias, and f_y is an activation function such as the softmax function:

$$f_y(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{o} e^{z_j}} \tag{446}$$

for classification tasks. Expanding the recurrence relation iteratively, the hidden state at time t can be expressed as:

$$\mathbf{h}_t = f_h \left(\mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} f_h \left(\mathbf{W}_{xh} \mathbf{x}_{t-1} + \mathbf{W}_{hh} f_h \left(\dots f_h \left(\mathbf{W}_{xh} \mathbf{x}_1 + \mathbf{W}_{hh} \mathbf{h}_0 + \mathbf{b}_h \right) + \mathbf{b}_h \right) + \mathbf{b}_h \right) + \mathbf{b}_h \right). \tag{447}$$

This expansion illustrates the depth of temporal dependency captured by the network and highlights the computational challenges of maintaining long-term memory. Specifically, the gradient of the loss function L, given by:

$$L = \sum_{t=1}^{T} \ell \left(\mathbf{y}_t, \mathbf{y}_t^{\text{true}} \right), \tag{448}$$

with $\ell \left(\mathbf{y}_t, \mathbf{y}_t^{\text{true}} \right)$ representing a task-specific loss such as cross-entropy:

$$\ell \left(\mathbf{y}_t, \mathbf{y}_t^{\text{true}} \right) = - \sum_{i=1}^{o} \mathbf{y}_t^{\text{true}}(i) \log \mathbf{y}_t(i), \tag{449}$$

is computed through backpropagation through time (BPTT). The gradient of L with respect to $\mathbf{W}_{hh}$, for instance, is given by:

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \ell_t}{\partial \mathbf{h}_t} \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}}, \tag{450}$$

where $\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$ represents the chain of derivatives from time step k to t. Unlike feedforward neural networks, where each input is processed independently, RNNs maintain a hidden state h_t that acts as a dynamic memory, evolving recursively as the input sequence progresses. Formally, given an input sequence $\{x_1, x_2, \dots, x_T\}$, where $x_t \in \mathbb{R}^n$ represents the input vector at time t, the hidden state $h_t \in \mathbb{R}^m$ is updated via the recurrence relation:

$$h_t = f_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \tag{451}$$

where $W_{xh} \in \mathbb{R}^{m\times n}$, $W_{hh} \in \mathbb{R}^{m\times m}$, and $b_h \in \mathbb{R}^m$ are learnable parameters, and f_h is a nonlinear activation function such as tanh or ReLU. The recursive structure inherently allows the hidden state h_t to encode the entire history of the sequence up to time t. The output $y_t \in \mathbb{R}^o$ at each time step is computed as:

$$y_t = f_y(W_{hy}h_t + b_y), \tag{452}$$

where $W_{hy} \in \mathbb{R}^{o\times m}$ and $b_y \in \mathbb{R}^o$ are additional learnable parameters, and f_y is an optional output activation function, such as the softmax function for classification. To elucidate the recursive dynamics, we can expand h_t explicitly in terms of the initial hidden state h_0 and all previous inputs $\{x_1, \dots, x_t\}$:

$$h_t = f_h(W_{xh}x_t + W_{hh}f_h(W_{xh}x_{t-1} + W_{hh}f_h(\dots f_h(W_{xh}x_1 + W_{hh}h_0 + b_h) + b_h) + b_h) + b_h). \tag{453}$$

This nested structure highlights the temporal dependencies and the potential challenges in training, such as the vanishing and exploding gradient problems. During training, the loss function L, which aggregates the discrepancies between the predicted outputs y_t and the ground truth y_t^{true}, is typically defined as:

$$L = \sum_{t=1}^{T} \ell(y_t, y_t^{\text{true}}), \tag{454}$$

where ℓ is a task-specific loss function, such as the mean squared error (MSE)

$$\ell(y, y^{\text{true}}) = \frac{1}{2}\|y - y^{\text{true}}\|^2 \tag{455}$$

for regression or the cross-entropy loss for classification. To optimize L, gradient-based methods are employed, requiring the computation of derivatives of L with respect to all parameters, such as W_{xh}, W_{hh}, and b_h. Using backpropagation through time (BPTT), the gradient of L with respect to W_{hh} is expressed as:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \ell_t}{\partial h_t} \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \frac{\partial h_k}{\partial W_{hh}}. \tag{456}$$

Here,

$$\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{457}$$

is the product of Jacobian matrices that encode the influence of h_k on h_t. The Jacobian $\frac{\partial h_j}{\partial h_{j-1}}$ itself is given by:

$$\frac{\partial h_j}{\partial h_{j-1}} = W_{hh} \odot f_h'(a_j), \tag{458}$$

where

$$a_j = W_{xh}x_j + W_{hh}h_{j-1} + b_h, \tag{459}$$

and $f'_h(a_j)$ denotes the elementwise derivative of the activation function. The repeated multiplication of these Jacobians can lead to exponential growth or decay of the gradients, depending on the spectral radius $\rho(W_{hh})$. Specifically, if $\rho(W_{hh}) > 1$, gradients explode, whereas if $\rho(W_{hh}) < 1$, gradients vanish, severely hampering the training process for long sequences. To address these issues, modifications such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) introduce gating mechanisms that explicitly regulate the flow of information. In LSTMs, the cell state c_t, governed by additive dynamics, prevents vanishing gradients. The cell state is updated as:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c), \tag{460}$$

where f_t is the forget gate, i_t is the input gate, and U_c, W_c, and b_c are learnable parameters.

8.2 Sequence Modeling and Long Short-Term Memory (LSTM) and GRUs

Literature Review: Potter and Egon (2024) [387] provided an extensive study of RNNs and their enhancements (LSTM and GRU) for time-series forecasting. The authors conduct an empirical comparison between these architectures and analyze their effectiveness in capturing long-term dependencies in sequential data. The study concludes that GRUs are computationally efficient but slightly less expressive than LSTMs, whereas standard RNNs suffer from vanishing gradients. Yatkin et. al. (2025) [388] introduced a topological perspective to RNNs, including LSTM and GRU, to address inconsistencies in real-world applications. The authors propose stability-enhancing mechanisms to improve RNN performance in finance and climate modeling. Their results show that topologically-optimized GRUs outperform traditional LSTMs in maintaining memory over long sequences. Saifullah (2024) [389] applied LSTM and GRU networks to biomedical image classification (chicken egg fertility detection). The paper demonstrates that GRU's simpler architecture leads to faster convergence while LSTMs achieve slightly higher accuracy due to better memory retention. The results highlight domain-specific strengths of LSTM vs. GRU, particularly in handling sparse feature representations. Alonso (2024) [390] rigorously explored the mathematical foundations of RNNs, LSTMs, and GRUs. The author provides a deep analysis of gating mechanisms, vanishing gradient solutions, and optimization techniques that improve sequence modeling. A theoretical comparison is drawn between hidden state dynamics in GRUs vs. LSTMs, supporting their application in NLP and time-series forecasting. Tu et. al. (2024) [391] in a medical AI study evaluates LSTMs and GRUs for predicting patient physiological metrics during sedation. The authors find that LSTMs retain more long-term dependencies in time-series medical data, making them suitable for patient monitoring, while GRUs are preferable for real-time predictions due to their lower computational overhead. Zuo et. al. (2025) [392] applied hybrid GRUs for predicting customer movements in stores using real-time location tracking. The authors propose a modified GRU-LSTM hybrid model that achieves state-of-the-art accuracy in trajectory prediction. The study demonstrates that GRUs alone may lack fine-grained memory retention, but a hybrid approach improves forecasting ability. Lima et. al. (2025) [393] developed an industrial AI application that demonstrated the efficiency of GRUs in process optimization. The study finds that GRUs outperform LSTMs in real-time predictive control of steel slab heating, showcasing their efficiency in applications where faster computations are required. Khan et. al. (2025) [394] integrated LSTMs with statistical ARIMA models to improve wind power forecasting. They demonstrate that hybrid LSTM-ARIMA models outperform standalone RNNs in handling weather-related sequential data, which is highly volatile. Guo and Feng (2024) [395] in an environmental AI study proposed a temporal attention-enhanced LSTM model to predict greenhouse climate variables. The research introduces a novel position-aware LSTM architecture that improves multi-step forecasting, which is critical for precision agriculture. Abdelhamid (2024) [396] explored IoT-based energy forecasting using deep RNN architectures, including LSTM and GRU. The study concludes that GRUs provide faster inference speeds but LSTMs capture more accurate long-range dependencies, making them more reliable for complex forecasting.

Sequence modeling in Recurrent Neural Networks (RNNs) represents a powerful framework for capturing temporal dependencies in sequential data, enabling the learning of both short-term and long-term patterns. The primary characteristic of RNNs lies in their recurrent architecture, where the hidden state h_t at time step t is updated as a function of both the current input x_t and the hidden state at the previous time step h_{t-1}. Mathematically, this recurrent relationship can be expressed as:

$$h_t = f(W_h h_{t-1} + W_x x_t + b_h) \tag{461}$$

Here, W_h and W_x are weight matrices corresponding to the previous hidden state h_{t-1} and the current input x_t, respectively, while b_h is a bias term. The function $f(\cdot)$ is a non-linear activation function, typically chosen as the hyperbolic tangent tanh or rectified linear unit (ReLU). The output y_t at each time step is derived from the hidden state h_t through a linear transformation, followed by a non-linear activation, yielding:

$$y_t = g(W_y h_t + b_y) \tag{462}$$

where W_y is the weight matrix connecting the hidden state to the output space, and b_y is the associated bias term. The function $g(\cdot)$ is generally a softmax activation for classification tasks or a linear activation for regression problems. The key feature of this structure is the interdependence of the hidden state across time steps, allowing the model to capture the history of past inputs and produce predictions that incorporate temporal context. Training an RNN involves minimizing a loss function L, which quantifies the discrepancy between the predicted outputs y_t and the true outputs y_t^{true} across all time steps. A common loss function used in classification tasks is the cross-entropy loss, while regression tasks often utilize mean squared error. To optimize the parameters of the network, the model employs **Backpropagation Through Time (BPTT)**, a variant of the standard backpropagation algorithm adapted for sequential data. The primary challenge in BPTT arises from the recurrent nature of the network, where the hidden state at each time step depends on the previous hidden state. The gradient of the loss function with respect to the hidden state at time step t is computed recursively, reflecting the temporal structure of the model. The chain rule is applied to compute the gradient of the loss with respect to the hidden state:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} + \sum_{t'=t+1}^{T} \frac{\partial L}{\partial h_{t'}} \cdot \frac{\partial h_{t'}}{\partial h_t} \tag{463}$$

Here, $\frac{\partial L}{\partial y_t}$ is the gradient of the loss with respect to the output, and $\frac{\partial y_t}{\partial h_t}$ represents the Jacobian of the output with respect to the hidden state. The second term in this expression corresponds to the accumulated gradients propagated from future time steps, incorporating the temporal dependencies across the entire sequence. This recursive gradient calculation allows for updating the weights and biases of the RNN, adjusting them to minimize the total error across the sequence. The gradients of the loss function with respect to the parameters of the network, such as W_h, W_x, and W_y, are computed using the chain rule. For example, the gradient of the loss with respect to W_x is:

$$\frac{\partial L}{\partial W_x} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \cdot x_t^\top \tag{464}$$

This captures the contribution of each input to the overall error at all time steps, ensuring that the model learns the correct relationships between inputs and hidden states. Similarly, the gradients with respect to W_h and b_h account for the recurrence in the hidden state, enabling the model to adjust its internal parameters in response to the sequential nature of the data. Despite their theoretical elegance, RNNs face significant practical challenges during training, primarily due to the **vanishing gradients problem**. This issue arises when the gradients propagate through many time steps, causing them to decay exponentially, especially when using activation functions like tanh. As a result, the influence of distant time steps diminishes, making it difficult for the network to learn long-term dependencies. The mathematical manifestation of this problem is seen in the norm of the Jacobian matrices associated with the hidden state updates. If the spectral radius of the weight matrices W_h is close to or greater than 1, the gradients can either vanish or explode, leading to unstable training dynamics. To mitigate this issue, several solutions have been proposed, including the use of Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), which introduce gating mechanisms to better control the flow of information through the network. LSTMs, for example, incorporate a memory cell C_t, which allows the network to store information over long periods of time. The update rules for the LSTM are governed by three gates: the forget gate f_t, the input gate i_t, and the output gate o_t, which control how much of the previous memory and new information to retain. The equations governing the LSTM are:

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f) \tag{465}$$

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i) \tag{466}$$

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o) \tag{467}$$

$$\tilde{C}_t = \tanh(W_C h_{t-1} + U_C x_t + b_C) \tag{468}$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{469}$$

$$h_t = o_t \cdot \tanh(C_t) \tag{470}$$

In these equations, the forget gate f_t determines how much of the previous memory cell C_{t-1} to retain, the input gate i_t governs how much new information to store in the candidate memory cell $\tilde{C}_t$, and the output gate o_t controls how much of the memory cell should influence the current output. The LSTM's architecture allows for the maintenance of long-term dependencies by selectively forgetting or retaining information, effectively alleviating the vanishing gradient problem and enabling the network to learn from longer sequences. The GRU, an alternative to the LSTM, simplifies this architecture by combining the forget and input gates into a single update gate z_t, and introduces a reset gate r_t to control the influence of the previous hidden state. The GRU's update rules are:

$$z_t = \sigma(W_z h_{t-1} + U_z x_t + b_z) \tag{471}$$

$$r_t = \sigma(W_r h_{t-1} + U_r x_t + b_r) \tag{472}$$

$$\tilde{h}_t = \tanh(W h_{t-1} + U x_t + b) \tag{473}$$

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \tag{474}$$

Here, z_t controls the amount of the previous hidden state h_{t-1} to retain, and r_t determines how much of the previous hidden state should influence the candidate hidden state $\tilde{h}_t$. The GRU's simplified structure still allows it to effectively capture long-range dependencies while being computationally more efficient than the LSTM.

In summary, sequence modeling in RNNs involves a series of recurrent updates to the hidden state, driven by both the current input and the previous hidden state, and is trained via backpropagation through time. The introduction of specialized gating mechanisms in LSTMs and GRUs alleviates issues such as vanishing gradients, enabling the networks to learn and maintain long-term dependencies. Through these advanced architectures, RNNs can effectively model complex temporal relationships, making them powerful tools for tasks such as time-series prediction, natural language processing, and sequence generation.

8.3 Applications in Natural Language Processing

Literature Review: Yang et. al. (2020) [377] explored the effectiveness of deep learning models, including RNNs, for sentiment analysis in e-commerce platforms. It emphasizes how RNN architectures, including LSTMs and GRUs, outperform traditional NLP techniques by capturing sequential dependencies in customer reviews. The study provides empirical evidence demonstrating the superior accuracy of RNNs in analyzing consumer sentiment. Manikandan et. al. (2025) [378] investigated how RNNs can improve spam detection in email filtering. By leveraging recurrent structures, the study demonstrates how RNNs effectively identify patterns in email text that indicate spam or phishing attempts. It also compares RNN-based models with other ML approaches, highlighting the robustness of RNNs in handling contextual word sequences. Isiaka et. al. (2025) [379] examined AI technologies, particularly deep learning models, for predictive healthcare applications. It highlights how RNNs can analyze patient records and medical reports using NLP techniques. The study shows that RNN-based NLP models enhance medical diagnostics and decision-making by extracting meaningful insights from unstructured text data. Petrov et. al. (2025) [380] discussed the role of RNNs in emotion classification from textual data, an essential NLP task. The paper evaluates various RNN-based architectures, including BiLSTMs, to enhance the accuracy of emotion recognition in social media texts and chatbot responses. Liang (2025) [381] focused on the application of RNNs in educational settings, specifically for automated grading and feedback generation. The study presents an RNN-based NLP system capable of analyzing student responses, providing real-time assessments, and generating contextual feedback. Jin (2025) [382] explored how RNNs optimize text generation tasks related to pharmaceutical education. It demonstrates how NLP-powered RNN models generate high-quality textual summaries from medical literature, ensuring accurate knowledge dissemination in the pharmaceutical industry. McNicholas et. al. (2025) [383] investigated how RNNs facilitate clinical decision-making in critical care by extracting insights from unstructured medical text. The research highlights how RNN-based NLP models enhance patient care by predicting outcomes based on clinical notes and physician reports. Abbas and Khammas (2024) [384] introduced an RNN-based NLP technique for detecting malware in IoT networks. The study illustrates how RNN classifiers process logs and textual patterns to identify malicious software, making RNNs crucial in cybersecurity applications. Kalonia and Upadhyay (2025) [385] applied RNNs to software fault prediction using NLP techniques. It shows how recurrent networks analyze bug reports and software documentation to predict potential failures in software applications, aiding developers in proactive debugging. Han et. al. (2025) [386] discussed RNN applications in conversational AI, focusing on chatbots and virtual assistants. The study presents an RNN-driven NLP model for improving dialogue management and user interactions, significantly enhancing the responsiveness of AI-powered chat systems.

Recurrent Neural Networks (RNNs) are deep learning architectures that are explicitly designed to handle sequential data, a key feature that makes them indispensable for applications in Natural Language Processing (NLP). The mathematical foundation of RNNs lies in their ability to process sequences of inputs, $x_1, x_2, \ldots, x_T$, where T denotes the length of the sequence. At each time step t, the network updates its hidden state, h_t, using both the current input x_t and the previous hidden state h_{t-1}. This recursive relationship is represented mathematically by the following equation:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \tag{475}$$

Here, σ is a nonlinear activation function such as the hyperbolic tangent (tanh) or the rectified linear unit (ReLU), W_h is the weight matrix associated with the previous hidden state h_{t-1}, W_x is the weight matrix associated with the current input x_t, and b is a bias term. The nonlinearity introduced by σ allows the network to learn complex relationships between the input and the output. The output y_t at each time step is computed as:

$$y_t = W_y h_t + c \tag{476}$$

where W_y is the weight matrix corresponding to the output and c is the bias term for the output. The output y_t is then used to compute the predicted probability distribution over possible outputs at each time step, typically through a softmax function for classification tasks:

$$P(y_t|h_t) = \text{softmax}(W_y h_t + c) \tag{477}$$

In NLP tasks such as language modeling, the objective is to predict the next word in a sequence given all previous words. The RNN is trained to estimate the conditional probability distribution $P(w_t|w_1, w_2, \dots, w_{t-1})$ of the next word w_t based on the previous words. The full likelihood of the sequence $w_1, w_2, \dots, w_T$ can be written as:

$$P(w_1, w_2, \dots, w_T) = \prod_{t=1}^{T} P(w_t|w_1, w_2, \dots, w_{t-1}) \tag{478}$$

For an RNN, this conditional probability is modeled by recursively updating the hidden state and generating a probability distribution for each word. At each time step, the probability of the next word is computed as:

$$P(w_t|h_{t-1}) = \text{softmax}(W_y h_t + c) \tag{479}$$

The network is trained by minimizing the negative log-likelihood of the true word sequence:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(w_t|h_{t-1}) \tag{480}$$

This loss function guides the optimization of the weight matrices W_h, W_x, and W_y to maximize the likelihood of the correct word sequences. As the network learns from large datasets, it develops the ability to predict words based on the context provided by previous words in the sequence. A key extension of RNNs in NLP is machine translation, where one sequence of words in one language is mapped to another sequence in a target language. This is typically modeled using sequence-to-sequence (Seq2Seq) architectures, which consist of two RNNs: the encoder and the decoder. The encoder RNN processes the input sequence $x_1, x_2, \dots, x_T$, updating its hidden state at each time step:

$$h_t^{\text{enc}} = \sigma(W_h^{\text{enc}} h_{t-1}^{\text{enc}} + W_x^{\text{enc}} x_t + b^{\text{enc}}) \tag{481}$$

The final hidden state h_T^{enc} of the encoder is passed to the decoder as its initial hidden state. The decoder RNN generates the target sequence $y_1, y_2, \dots, y_T$ by updating its hidden state at each time step, using both the previous hidden state h_{t-1}^{dec} and the previous output y_{t-1}:

$$h_t^{\text{dec}} = \sigma(W_h^{\text{dec}} h_{t-1}^{\text{dec}} + W_x^{\text{dec}} y_{t-1} + b^{\text{dec}}) \tag{482}$$

The decoder produces a probability distribution over the target vocabulary at each time step:

$$P(y_t|h_t^{\text{dec}}) = \text{softmax}(W_y^{\text{dec}} h_t^{\text{dec}} + c^{\text{dec}}) \tag{483}$$

The training of the Seq2Seq model is based on minimizing the cross-entropy loss function:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(y_t|h_t^{\text{dec}}) \tag{484}$$

This ensures that the network learns to map input sequences to output sequences. By training on a large corpus of paired sentences, the Seq2Seq model learns to translate sentences from one language to another, with the encoder capturing the context of the input sentence and the decoder generating the translated sentence.

RNNs are also effective in sentiment analysis, a task where the goal is to classify the sentiment of a sentence (positive, negative, or neutral). Given a sequence of words $x_1, x_2, \ldots, x_T$, the RNN processes each word sequentially, updating its hidden state:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \tag{485}$$

After processing the entire sentence, the final hidden state h_T is used to classify the sentiment. The output is obtained by applying a softmax function to the final hidden state:

$$y = \text{softmax}(W_y h_T + c) \tag{486}$$

where W_y is the weight matrix associated with the output layer. The network is trained to minimize the cross-entropy loss:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(y|h_T) \tag{487}$$

This allows the RNN to classify the overall sentiment of the sentence by learning the relationships between words and sentiment labels. Sentiment analysis is useful for applications such as customer feedback analysis, social media monitoring, and opinion mining. In Named Entity Recognition (NER), RNNs are used to identify and classify named entities, such as people, locations, and organizations, in a text. The RNN processes each word x_t in the sequence, updating its hidden state at each time step:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \tag{488}$$

The output at each time step is a probability distribution over possible entity labels:

$$P(y_t|h_t) = \text{softmax}(W_y h_t + c) \tag{489}$$

The network is trained to minimize the cross-entropy loss:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(y_t|h_t) \tag{490}$$

By learning to classify each word with the appropriate entity label, the RNN can perform information extraction tasks, such as identifying people, organizations, and locations in text. This is crucial for applications such as document categorization, knowledge graph construction, and question answering. In speech recognition, RNNs are used to transcribe spoken language into text. The input to the RNN consists of a sequence of acoustic features, such as Mel-frequency cepstral coefficients (MFCCs), which are extracted from the audio signal. At each time step t, the RNN updates its hidden state:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \tag{491}$$

The output at each time step is a probability distribution over phonemes or words:

$$P(w_t|h_t) = \text{softmax}(W_y h_t + c) \tag{492}$$

The network is trained by minimizing the negative log-likelihood:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(w_t|h_t) \tag{493}$$

By learning the mapping between acoustic features and corresponding words or phonemes, the RNN can transcribe speech into text, which is fundamental for applications such as voice assistants, transcription services, and speech-to-text systems.

In summary, RNNs are powerful tools for processing sequential data in NLP tasks such as machine translation, sentiment analysis, named entity recognition, and speech recognition. Their ability to process input sequences in a time-dependent manner allows them to capture long-range dependencies, making them well-suited for complex tasks in NLP and beyond. However, challenges such as the vanishing and exploding gradient problems necessitate the use of more advanced architectures, like LSTMs and GRUs, to enhance their performance in real-world applications.

9 Advanced Architectures

9.1 Transformers and Attention Mechanisms

Literature Review: Vaswani et. al. [340] introduced the Transformer architecture, replacing recurrent models with a fully attention-based framework for sequence processing. They formulated the self-attention mechanism, mathematically defining query-key-value (QKV) transformations. They proved scalability advantages over RNNs, showing O(1) parallelization benefits and introduced multi-head attention, enabling contextualized embeddings. Nannepagu et. al. [341] explored hybrid AI architectures integrating Transformers with deep reinforcement learning (DQN). They developed a theoretical framework for transformer-augmented reinforcement learning and discussed how self-attention refines feature representations for financial time-series prediction. Rose et. al. [342] investigated Vision Transformers (ViTs) for cybersecurity applications, examining attention-based anomaly detection. They theoretically compared self-attention with CNN feature extraction and proposed a new loss function for attention weight refinement in cybersecurity detection models. Buehler [343] explored the theoretical interplay between Graph Neural Networks (GNNs) and Transformer architectures. They developed isomorphic self-attention, which preserves graph topological information and introduced graph-structured positional embeddings within Transformer attention. Tabibpour and Madanizadeh [344] investigated Set Transformers as a theoretical extension of Transformers for high-dimensional dynamic systems and introduced permutation-invariant self-attention mechanisms to replace standard Transformers in decision-making tasks and theoretically formalized attention mechanisms for non-sequential data. Kim et. al. (2024) [310] developed a Transformer-based anomaly detection framework for video surveillance. They formalized a new spatio-temporal self-attention mechanism to detect anomalies in videos and extended standard Transformer architectures to handle high-dimensional video data. Li and Dong [345] examined Transformer-based attention mechanisms for wireless communication networks. They introduced hybrid spatial and temporal attention layers for large-scale MIMO channel estimation and provided a rigorous mathematical proof of attention-based signal recovery. Asefa and Assabie [346] investigated language-specific adaptations of Transformer-based translation models. They introduced attention mechanism regularization for low-resource language translation and analyzed the impact of different positional encoding strategies on translation quality. Liao and Chen [347] applied transformer architectures to deepfake detection, analyzing self-attention mechanisms for facial feature analysis. They theoretically compared CNNs and ViTs for forgery detection and introduced attention-head dropout to improve robustness against adversarial attacks. Jiang et. al. [348] proposed a novel Transformer-based approach for medical imaging reconstruction. They introduced Spatial and Channel-wise Transformer (SCFormer) for enhanced attention-based feature aggregation and theoretically extended contrastive learning to Transformer encoders.

The Transformer model is an advanced neural network architecture fundamentally defined by the self-attention mechanism, which enables global context-aware computations on sequential data. The model processes an input sequence represented by

$$\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}, \tag{494}$$

where n denotes the sequence length and d_{model} the embedding dimensionality. Each token in this sequence is projected into three learned spaces—queries $\mathbf{Q}$, keys $\mathbf{K}$, and values $\mathbf{V}$—using the trainable matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, and $\mathbf{W}^V$, such that

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^V, \tag{495}$$

where $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$, with d_k being the dimensionality of queries and keys. The pairwise similarity between tokens is determined by the dot product $\mathbf{Q}\mathbf{K}^\top$, scaled by the factor

$\frac{1}{\sqrt{d_k}}$ to ensure numerical stability, yielding the raw attention scores:

$$\mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}, \tag{496}$$

where $\mathbf{S} \in \mathbb{R}^{n \times n}$. These scores are normalized using the softmax function, producing the attention weights $\mathbf{A}$, where

$$A_{ij} = \frac{\exp(S_{ij})}{\sum_{k=1}^{n} \exp(S_{ik})}, \tag{497}$$

ensuring $\sum_{j=1}^{n} A_{ij} = 1$. The output of the attention mechanism is computed as a weighted sum of the values:

$$\mathbf{Z} = \mathbf{A}\mathbf{V}, \tag{498}$$

where $\mathbf{Z} \in \mathbb{R}^{n \times d_v}$, with d_v being the dimensionality of the value vectors. This process can be expressed compactly as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}. \tag{499}$$

Multi-head attention extends this mechanism by splitting $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into h distinct heads, each operating in its subspace. For the i-th head:

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) \tag{500}$$

where $\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q$, $\mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K$, $\mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V$. The outputs of all heads are concatenated and linearly transformed:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O, \tag{501}$$

where $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d_\text{model}}$. This architecture enables the model to capture multiple types of relationships simultaneously. Positional encodings are added to the input embeddings $\mathbf{X}$ to preserve sequence order. These encodings $\mathbf{P} \in \mathbb{R}^{n \times d_\text{model}}$ are defined as:

$$P_{(pos,2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d_\text{model}}}\right), \quad P_{(pos,2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d_\text{model}}}\right), \tag{502}$$

ensuring unique representations for each position pos and dimension index i. The feedforward network (FFN) applies two dense layers with an intermediate ReLU activation:

$$\text{FFN}(\mathbf{z}) = \max(0, \mathbf{z}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2, \tag{503}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_\text{model} \times d_\text{ff}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_\text{ff} \times d_\text{model}}$, and $d_\text{ff} > d_\text{model}$. Residual connections and layer normalization are applied throughout to stabilize training, with the output given by

$$\mathbf{H}_\text{out} = \text{LayerNorm}(\mathbf{H}_\text{in} + \text{FFN}(\mathbf{H}_\text{in})). \tag{504}$$

Training optimizes the cross-entropy loss over the output distribution:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(y_t \mid y_{<t}, \mathbf{x}), \tag{505}$$

where $P(y_t \mid y_{<t}, \mathbf{x})$ is modeled using the softmax over the logits $\mathbf{z}_t\mathbf{W}_\text{out} + \mathbf{b}_\text{out}$, with parameters $\mathbf{W}_\text{out}, \mathbf{b}_\text{out}$. The Transformer achieves a complexity of $O(n^2 d_k)$ per attention layer due to the computation of $\mathbf{Q}\mathbf{K}^\top$, yet its parallelization capabilities render it more efficient than recurrent networks. This mathematical formalism, coupled with innovations like sparse attention and dynamic programming, has solidified the Transformer as the cornerstone of modern sequence modeling tasks.

While this quadratic complexity poses challenges for very long sequences, it allows for greater parallelization compared to RNNs, which require $O(n)$ sequential steps. Furthermore, the memory complexity of $O(n^2)$ for storing attention weights can be mitigated using sparse approximations or hierarchical attention structures. The Transformer architecture's flexibility and effectiveness stem from its ability to handle diverse tasks by appropriately modifying its components. For example, in Vision Transformers (ViTs), the input sequence is formed by flattening image patches, and the positional encodings capture spatial relationships. In contrast, in sequence-to-sequence tasks like translation, the cross-attention mechanism enables the decoder to focus on relevant parts of the encoder's output.

In conclusion, the Transformer represents a paradigm shift in neural network design, replacing recurrence with attention and enabling unprecedented scalability and performance. The rigorous mathematical foundation of attention mechanisms, combined with the architectural innovations of multi-head attention, positional encoding, and feedforward layers, underpins its success across domains.

9.2 Generative Adversarial Networks (GANs)

Literature Review: Goodfellow et. al. [349] in their landmark paper introduced Generative Adversarial Networks (GANs), where a generator and a discriminator compete in a minimax game. They established the theoretical foundation of adversarial learning and developed the mathematical formulation of GANs using game theory. They also introduced non-cooperative minimax optimization in deep learning. Chappidi and Sundaram [350] extended GANs with graph neural networks (GNNs) for complex real-world perception tasks. They theoretically integrated GANs with reinforcement learning for self-improving models and developed dual Q-learning mechanisms that enhance GAN convergence stability. Joni [351] provided a comprehensive theoretical overview of GAN-based generative models for advanced image synthesis. They formalized GAN loss functions and their optimization challenges and introduced progressive growing GANs as a solution for high-resolution image generation. Li et. al. (2024) [305] extended GANs to materials science, optimizing the crystal structure prediction process. They developed a GAN framework for molecular modeling and demonstrates GANs in scientific simulations beyond computer vision tasks. Sekhavat (2024) [299] analyzed the philosophy and theoretical basis of GANs in artistic image generation. He discussed GANs from a cognitive science perspective and established a link between GAN training and computational aesthetics. Kalaiarasi and Sudharani (2024) [352] examined GAN-based image steganography, optimizing data hiding techniques using adversarial training. They extended the theoretical properties of adversarial training to security applications and demonstrated how GANs can minimize perceptual distortion in data hiding. Arjmandi-Tash and Mansourian (2024) [353] explored GANs in scientific computing, generating realistic photodetector datasets. They demonstrated GANs as a theoretical tool for synthetic data augmentation and formulated a probabilistic approach to GAN loss functions for sensor modeling. Gao (2024) [354] bridged the gap between GANs and Partial Differential Equations (PDEs) in physics-informed learning. He established a theoretical framework for solving PDEs using GAN-based architectures and developed a new loss function combining adversarial and variational methods. Hisama et. al. [355] applied GANs to computational chemistry, generating new alloy catalyst structures. They introduced Wasserstein GANs (WGANs) for molecular design and used GAN-generated latent spaces to predict catalyst activity. Wang and Zhang (2024) [356] proposed an improved GAN framework for medical image segmentation. They developed a novel attention-enhanced GAN for robust segmentation and provided a mathematical formulation for adversarial segmentation loss functions.

Generative Adversarial Networks (GANs) are an intricate mathematical framework designed to model complex probability distributions by leveraging a competitive dynamic between two neural networks, the generator G and the discriminator D. These networks are parametrized by weights $\theta_G \in \Theta_G$ and $\theta_D \in \Theta_D$, and their interaction is mathematically formulated as a two-player zero-sum game. The generator $G : \mathbb{R}^d \to \mathbb{R}^n$ maps latent variables $\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})$, where $p_{\mathbf{z}}$ is a prior probability distribution (commonly uniform or Gaussian), to a synthetic data sample $\hat{\mathbf{x}} = G(\mathbf{z})$. The discriminator $D : \mathbb{R}^n \to [0, 1]$ assigns a probability score $D(\mathbf{x})$ indicating whether $\mathbf{x}$ originates from the true data distribution $p_{\text{data}}(\mathbf{x})$ or the generated distribution $p_g(\mathbf{x})$, implicitly defined as the pushforward measure of $p_{\mathbf{z}}$ under G, i.e., $p_g = G_{\#}p_{\mathbf{z}}$. The optimization problem governing GANs is expressed as

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))], \tag{506}$$

where $\mathbb{E}$ denotes the expectation operator. This objective seeks to maximize the discriminator's ability to distinguish between real and generated samples while simultaneously minimizing the generator's ability to produce samples distinguishable from real data. For a fixed generator G, the optimal discriminator D^* is obtained by maximizing $V(D, G)$, yielding

$$D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}. \tag{507}$$

Substituting D^* back into the value function simplifies it to

$$V(D^*, G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}\right] + \mathbb{E}_{\mathbf{x}\sim p_g}\left[\log \frac{p_g(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}\right]. \tag{508}$$

This expression is equivalent to minimizing the Jensen-Shannon (JS) divergence between p_{data} and p_g, defined as

$$\text{JS}(p_{\text{data}} \| p_g) = \frac{1}{2}\text{KL}(p_{\text{data}} \| M) + \frac{1}{2}\text{KL}(p_g \| M), \tag{509}$$

where $M = \frac{1}{2}(p_{\text{data}} + p_g)$ and $\text{KL}(p\|q) = \int p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x}$ is the Kullback-Leibler divergence. At the Nash equilibrium, $p_g = p_{\text{data}}$, the JS divergence vanishes, and $D^*(\mathbf{x}) = \frac{1}{2}$ for all $\mathbf{x}$. The gradient updates during training are derived using stochastic gradient descent. For the discriminator, the gradients are given by

$$\nabla_{\theta_D} V(D, G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\nabla_{\theta_D} \log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[\nabla_{\theta_D} \log(1 - D(G(\mathbf{z})))\right]. \tag{510}$$

Training Generative Adversarial Networks (GANs) involves iterative updates to the parameters θ_D of the discriminator and θ_G of the generator. The discriminator's parameters are updated via gradient ascent to maximize the value function $V(D, G)$, while the generator's parameters are updated via gradient descent to minimize the same value function. Denoting the gradients of D and G with respect to their parameters as ∇_{θ_D} and ∇_{θ_G}, the updates are given by:

$$\theta_D \leftarrow \theta_D + \eta_D \nabla_{\theta_D}\left[\mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z}\left[\log\left(1 - D(G(z))\right)\right]\right], \tag{511}$$

and

$$\theta_G \leftarrow \theta_G - \eta_G \nabla_{\theta_G}\left[\mathbb{E}_{z\sim p_z}\left[\log\left(1 - D(G(z))\right)\right]\right]. \tag{512}$$

In practice, to address issues of vanishing gradients, an alternative loss function for the generator is often used, defined as:

$$-\mathbb{E}_{z\sim p_z}\left[\log D(G(z))\right]. \tag{513}$$

This modification ensures stronger gradient signals when the discriminator is performing well, effectively improving the generator's training dynamics. For the generator, the gradients in the original formulation are expressed as

$$\nabla_{\theta_G} V(D, G) = -\mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[\nabla_{\theta_G} \log(1 - D(G(\mathbf{z})))\right], \tag{514}$$

but due to vanishing gradients when $D(G(\mathbf{z}))$ is near 0, the non-saturating generator loss is preferred:

$$\mathcal{L}_G = -\mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log D(G(\mathbf{z}))]. \tag{515}$$

The convergence of GANs is inherently linked to the properties of $D^*(\mathbf{x})$ and the alignment of p_g with p_{data}. However, mode collapse and training instability are frequently observed due to the non-convex nature of the objective functions. Wasserstein GANs (WGANs) address these issues by replacing the JS divergence with the Wasserstein-1 distance, defined as

$$W(p_{\text{data}}, p_g) = \inf_{\gamma\in\Pi(p_{\text{data}},p_g)} \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\gamma}[\|\mathbf{x}-\mathbf{y}\|], \tag{516}$$

where $\Pi(p_{\text{data}}, p_g)$ is the set of all couplings of p_{data} and p_g. Using Kantorovich-Rubinstein duality, the Wasserstein distance is reformulated as

$$W(p_{\text{data}}, p_g) = \sup_{\|f\|_L\le 1} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[f(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_g}[f(\mathbf{x})], \tag{517}$$

where f is a 1-Lipschitz function. To enforce the Lipschitz constraint, gradient penalties are applied, ensuring that $\|\nabla f(\mathbf{x})\| \le 1$.

The mathematical framework of GANs integrates elements from game theory, optimization, and information geometry. Their training involves solving a high-dimensional non-convex game, where theoretical guarantees for convergence are challenging due to saddle points and complex interactions between G and D. Nevertheless, GANs represent a mathematically elegant paradigm for generative modeling, with ongoing research extending their theoretical and practical capabilities.

9.3 Autoencoders and Variational Autoencoders

Literature Review: Zhang et. al. (2024) [303] explored a theoretical connection between VAEs and rate-distortion theory in image compression. They established a mathematical framework linking probabilistic autoencoding to lossy image compression and introduced hierarchical variational inference for improving generative modeling capacity. Wang and Huang (2025) [304] developed a formal mathematical proof of convergence in over-parameterized VAEs. They established rigorous mathematical limits for training VAEs and introduced Neural Tangent Kernel (NTK) theory to study how VAEs behave under over-parameterization. Li et. al. (2024) [305] extended VAEs to materials science, optimizing crystal structure prediction. They developed a VAE-based molecular modeling framework and demonstrates the role of generative models beyond image-based applications. Huang (2024) [306] reviewed key techniques in VAEs, GANs, and Diffusion Models for image generation. They analyzed probabilistic modeling in VAEs compared to diffusion-based methods and also established a theoretical hierarchy of generative models. Chenebuah (2024) [307] investigated Autoencoders for energy materials simulation and molecular property prediction. They introduced a novel AE-VAE hybrid model for physical simulations and established a theoretical link between Density Functional Theory (DFT) and Autoencoders. Furth et. al. (2024) [308] explored Graph Neural Networks (GNNs) and VAEs for predicting chemical properties. They established theoretical properties of VAEs for graph-based learning and extended Autoencoders to chemical reaction prediction. Gong et. al. [309] investigated Conditional Variational Autoencoders (CVAEs) for material design. They introduced new loss functions for conditional generative modeling and theoretically proved how VAEs can optimize material selection. Kim et. al. [310] uses Transformer-based Autoencoders (AEs) for video anomaly detection. They established theoretical improvements of AEs for time-series anomaly detection and used spatio-temporal Autoencoder embeddings to capture anomalies in videos. Albert et. al. (2024) [311] compared Kernel Learning Embeddings (KLE) and Variational Autoencoders for dimensionality reduction. They introduced VAE-based models for atmospheric modeling and established a mathematical comparison between

VAEs and kernel-based models. Sharma et. al. (2024) [312] explored practical applications of Autoencoders in network intrusion detection. They established Autoencoders as robust feature extractors for anomaly detection and provided a formal study of Autoencoder latent space representations.

An **Autoencoder (AE)** is an unsupervised learning model that attempts to learn a compact representation of the input data $\mathbf{x} \in \mathbb{R}^d$ in a lower-dimensional latent space. This model consists of two primary components: an encoder function f_{θ_e} and a decoder function f_{θ_d}. The encoder $f_{\theta_e} : \mathbb{R}^d \to \mathbb{R}^l$ maps the input data $\mathbf{x}$ to a latent code $\mathbf{z}$, where $l \ll d$, representing the compressed information. The decoder $f_{\theta_d} : \mathbb{R}^l \to \mathbb{R}^d$ then reconstructs the input from this latent code, producing an approximation $\hat{\mathbf{x}}$. The loss function typically used to train the autoencoder is the reconstruction loss, often formulated as the Mean Squared Error (MSE):

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2. \tag{518}$$

The optimization procedure seeks to minimize the reconstruction error over the dataset D, assuming a distribution $p(\mathbf{x})$ over the input data $\mathbf{x}$. The objective is to learn the optimal parameters θ_e and θ_d, by solving the following optimization problem:

$$\min_{\theta_e, \theta_d} \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \left[\|\mathbf{x} - f_{\theta_d}(f_{\theta_e}(\mathbf{x}))\|_2^2 \right]. \tag{519}$$

This formulation drives the encoder-decoder architecture towards learning a latent representation that preserves key features of the input data, allowing it to be efficiently reconstructed. The solution to this problem is typically pursued via **stochastic gradient descent (SGD)**, where gradients of the loss with respect to the model parameters are computed and backpropagated through the network. In contrast to the deterministic autoencoder, the **Variational Autoencoder (VAE)** introduces a probabilistic model to better capture the distribution of the latent variables. A VAE models the data generation process using a latent variable $\mathbf{z} \in \mathbb{R}^l$, and aims to maximize the likelihood of observing the data $\mathbf{x}$ by integrating over all possible latent variables. Specifically, we have the joint distribution:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z}), \tag{520}$$

where $p(\mathbf{x}|\mathbf{z})$ is the likelihood of the data given the latent variables, and $p(\mathbf{z})$ is the prior distribution of the latent variables, typically chosen to be a standard Gaussian $\mathcal{N}(\mathbf{z}; 0, I)$. The prior assumption that $p(\mathbf{z}) = \mathcal{N}(0, I)$ simplifies the modeling, as it imposes no particular structure on the latent space, which allows for flexible modeling of the data distribution. The encoder in a VAE outputs a distribution $q_{\theta_e}(\mathbf{z}|\mathbf{x})$ over the latent variables, typically modeled as a multivariate Gaussian with mean $\mu_{\theta_e}(\mathbf{x})$ and variance $\sigma_{\theta_e}(\mathbf{x})$, i.e. $q_{\theta_e}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \mu_{\theta_e}(\mathbf{x}), \sigma^2_{\theta_e}(\mathbf{x})I)$. The decoder generates the likelihood of the data $\mathbf{x}$ given the latent variable $\mathbf{z}$, expressed as $p_{\theta_d}(\mathbf{x}|\mathbf{z})$, which typically takes the form of a Gaussian distribution for continuous data. A central challenge in VAE training is the **marginal likelihood** $p(\mathbf{x})$, which represents the probability of the observed data. This marginal likelihood is intractable due to the integral over the latent variables:

$$p(\mathbf{x}) = \int p_{\theta_d}(\mathbf{x}|\mathbf{z})p(\mathbf{z})\, d\mathbf{z}. \tag{521}$$

To address this, VAE training employs **variational inference**, which approximates the true posterior $p(\mathbf{z}|\mathbf{x})$ with a variational distribution $q_{\theta_e}(\mathbf{z}|\mathbf{x})$. The goal is to optimize the **Evidence Lower Bound (ELBO)**, which is a lower bound on the log-likelihood $\log p(\mathbf{x})$. The ELBO is derived using **Jensen's inequality**:

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q_{\theta_e}(\mathbf{z}|\mathbf{x})} \left[\log p_{\theta_d}(\mathbf{x}|\mathbf{z})\right] - \mathrm{KL}\left(q_{\theta_e}(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z})\right), \tag{522}$$

where the first term is the expected log-likelihood of the data given the latent variables, and the second term is the **Kullback-Leibler (KL) divergence** between the approximate posterior

$q_{\theta_e}(\mathbf{z}|\mathbf{x})$ and the prior $p(\mathbf{z})$. The KL divergence acts as a regularizer, penalizing deviations from the prior distribution. The ELBO can then be written as:

$$\mathcal{L}_{\text{VAE}}(\mathbf{x}) = \mathbb{E}_{q_{\theta_e}(\mathbf{z}|\mathbf{x})}\left[\log p_{\theta_d}(\mathbf{x}|\mathbf{z})\right] - \text{KL}\left(q_{\theta_e}(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z})\right). \tag{523}$$

This formulation balances two competing objectives: maximizing the likelihood of reconstructing $\mathbf{x}$ from $\mathbf{z}$, and minimizing the divergence between the posterior $q_{\theta_e}(\mathbf{z}|\mathbf{x})$ and the prior $p(\mathbf{z})$. In order to perform optimization, we need to compute the gradient of the ELBO with respect to the parameters θ_e and θ_d. However, since sampling from the distribution $q_{\theta_e}(\mathbf{z}|\mathbf{x})$ is non-differentiable, the **reparameterization trick** is applied. This trick allows us to reparameterize the latent variable $\mathbf{z}$ as:

$$\mathbf{z} = \mu_{\theta_e}(\mathbf{x}) + \sigma_{\theta_e}(\mathbf{x}) \cdot \epsilon, \tag{524}$$

where $\epsilon \sim \mathcal{N}(0, I)$ is a standard Gaussian noise vector. This enables the backpropagation of gradients through the latent space and allows the optimization process to proceed via stochastic gradient descent. In practice, the **Monte Carlo method** is used to estimate the expectation in the ELBO. This involves drawing K samples $\mathbf{z}_k$ from the variational posterior $q_{\theta_e}(\mathbf{z}|\mathbf{x})$ and approximating the expectation as:

$$\hat{\mathcal{L}}_{\text{VAE}}(\mathbf{x}) = \frac{1}{K}\sum_{k=1}^{K} \log p_{\theta_d}(\mathbf{x}|\mathbf{z}_k) - \frac{1}{K}\sum_{k=1}^{K} \log q_{\theta_e}(\mathbf{z}_k|\mathbf{x}). \tag{525}$$

This approximation allows for efficient optimization, even when the latent space is high-dimensional and the exact expectation is computationally prohibitive. Thus, the **training process** of a VAE involves the following steps: first, the encoder produces a distribution $q_{\theta_e}(\mathbf{z}|\mathbf{x})$ for each input $\mathbf{x}$; then, latent variables $\mathbf{z}$ are sampled from this distribution; finally, the decoder reconstructs the data $\hat{\mathbf{x}}$ from the latent variable $\mathbf{z}$. The network is trained to maximize the ELBO, which effectively balances the reconstruction loss and the KL divergence term.

In this rigorous exploration, we have presented the mathematical foundations of both autoencoders and variational autoencoders. The core distinction between the two lies in the introduction of a probabilistic framework in the VAE, which leverages variational inference to optimize a tractable lower bound on the marginal likelihood. Through this process, the VAE learns to generate data by sampling from the latent space and reconstructing the input, while maintaining a well-structured latent distribution through regularization by the KL divergence term. The optimization framework for VAEs is grounded in variational inference and the reparameterization trick, enabling gradient-based optimization techniques to efficiently train deep generative models.

10 Reinforcement Learning

Literature Review: Sutton and Barto (2018) [272] (2021) [273] wrote a definitive textbook on reinforcement learning. It covers the fundamental concepts, including Markov decision processes (MDPs), temporal difference learning, policy gradient methods, and function approximation. The second edition expands on deep reinforcement learning, covering advanced algorithms like DDPG, A3C, and PPO. Bertsekas and Tsitsiklis (1996) [274] laid the theoretical foundation for reinforcement learning by introducing neuro-dynamic programming, an extension of dynamic programming methods for decision-making under uncertainty. It rigorously covers approximate dynamic programming, policy iteration, and value function approximation. Kakade (2003) [275] in his thesis formalized the sample complexity of RL, providing theoretical guarantees for how much data is required for an agent to learn optimal policies. It introduces the PAC-RL (Probably Approximately Correct RL) framework, which has significantly influenced how RL algorithms are evaluated. Szepesvári (2010) [276] presented a rigorous yet concise overview of reinforcement learning algorithms, including value iteration, Q-learning, SARSA, function approximation, and policy gradient methods. It provides deep theoretical insights into convergence proofs and performance bounds. Haarnoja et. al. (2018) [277] introduced Soft Actor-Critic (SAC), an off-policy deep reinforcement learning algorithm that maximizes expected reward and entropy simultaneously. It provides a strong theoretical framework for handling exploration-exploitation trade-offs in high-dimensional continuous action spaces. Mnih et al. (2015) [278] introduced Deep Q-Networks (DQN), demonstrating how deep learning can be combined with Q-learning to achieve human-level performance in Atari games. The authors address key challenges in reinforcement learning, including function approximation and stability improvements. Konda and Tsitsiklis (2003) [279]. provided a rigorous theoretical analysis of Actor-Critic methods, which combine policy-based and value-based learning. It formally establishes convergence proofs for actor-critic algorithms and introduces the natural gradient method for policy improvement. Levine (2018) [280] introduced a probabilistic inference framework for reinforcement learning, linking RL to Bayesian inference. It provides a theoretical foundation for maximum entropy reinforcement learning, explaining why entropy-regularized objectives lead to better exploration and stability. Mannor et. al. (2022) [281] gave one of the most rigorous mathematical treatments of reinforcement learning theory. It covers several topics: PAC guarantees for RL algorithms, Complexity bounds for exploration, Connections between RL and control theory, Convergence rates of popular RL methods. Borkar (2008) [282] rigorously analyzed stochastic approximation methods, which form the theoretical backbone of RL algorithms like TD-learning, Q-learning, and policy gradient methods. Borkar provides a dynamical systems perspective to convergence analysis, offering deep mathematical insights.

10.1 Key Concepts

Reinforcement Learning (RL) is a branch of machine learning that deals with agents making decisions in an environment to maximize cumulative rewards over time. This formalized decision-making process can be described using concepts such as agents, states, actions, and rewards, all of which are mathematically formulated within the framework of a *Markov Decision Process (MDP)*. The following provides an extremely mathematically rigorous discussion of these key concepts. An *agent* interacts with the environment by taking actions based on the current state of the environment. The goal of the agent is to maximize the expected cumulative reward over time. A policy π is a mapping from states to probability distributions over actions. Formally, the policy π can be written as:

$$\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A}), \tag{526}$$

where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, and $\mathcal{P}(\mathcal{A})$ is the set of probability distributions over the actions. The policy can be either *deterministic*:

$$\pi(a_t|s_t) = \begin{cases} 1 & \text{if } a_t = \pi(s_t), \\ 0 & \text{otherwise}, \end{cases} \tag{527}$$

where $\pi(s_t)$ is the action chosen in state s_t, or *stochastic*, in which case the policy assigns a probability distribution over actions for each state s_t. The goal of reinforcement learning is to find an *optimal policy* $\pi^*(s_t)$, which maximizes the expected return (cumulative reward) from any initial state. The optimal policy is defined as:

$$\pi^*(s_t) = \arg\max_{\pi} \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t\right], \tag{528}$$

where γ is the *discount factor* that determines the weight of future rewards, and $\mathbb{E}[\cdot]$ represents the expectation under the policy π. The optimal policy can be derived from the *optimal action-value function* $Q^*(s_t, a_t)$, which we define in the next section. The *state* $s_t \in \mathcal{S}$ describes the current situation of the agent at time t, encapsulating all relevant information that influences the agent's decision-making process. The state space $\mathcal{S}$ may be either *discrete* or *continuous*. The state transitions are governed by a probability distribution $P(s_{t+1}|s_t, a_t)$, which represents the probability of moving from state s_t to state s_{t+1} given action a_t. These transitions satisfy the *Markov property*, meaning the future state depends only on the current state and action, not the history of previous states or actions:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t) \quad \forall s_t, s_{t+1} \in \mathcal{S}, a_t \in \mathcal{A}. \tag{529}$$

Additionally, the transition probabilities satisfy the *normalization condition*:

$$\sum_{s_{t+1} \in \mathcal{S}} P(s_{t+1}|s_t, a_t) = 1 \quad \forall s_t, a_t. \tag{530}$$

The *state distribution* $\rho_t(s_t)$ represents the probability of the agent being in state s_t at time t. The state distribution evolves over time according to the transition probabilities:

$$\rho_{t+k}(s_{t+k}) = \sum_{s_t \in \mathcal{S}} P(s_{t+k}|s_t, a_t)\rho_t(s_t), \tag{531}$$

where $\rho_t(s_t)$ is the initial distribution at time t, and $\rho_{t+k}(s_{t+k})$ is the distribution at time $t+k$. An *action* a_t taken at time t by the agent in state s_t leads to a transition to state s_{t+1} and results in a reward r_t. The agent aims to select actions that maximize its long-term reward. The *action-value function* $Q(s_t, a_t)$ quantifies the expected cumulative reward from taking action a_t in state s_t and following the optimal policy thereafter. It is defined as:

$$Q(s_t, a_t) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t, a_t\right]. \tag{532}$$

The optimal action-value function $Q^*(s_t, a_t)$ satisfies the *Bellman Optimality Equation*:

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \sum_{s_{t+1} \in \mathcal{S}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}). \tag{533}$$

This recursive equation provides the foundation for dynamic programming methods such as *value iteration* and *policy iteration*. The optimal policy $\pi^*(s_t)$ is derived by choosing the action that maximizes the action-value function:

$$\pi^*(s_t) = \arg\max_{a_t \in \mathcal{A}} Q^*(s_t, a_t). \tag{534}$$

The *optimal value function* $V^*(s_t)$, representing the expected return from state s_t under the optimal policy, is given by:

$$V^*(s_t) = \max_{a_t \in \mathcal{A}} Q^*(s_t, a_t). \tag{535}$$

The optimal value function satisfies the Bellman equation:

$$V^*(s_t) = \max_{a_t \in \mathcal{A}} \left[R(s_t, a_t) + \gamma \sum_{s_{t+1} \in \mathcal{S}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1})\right]. \tag{536}$$

The *reward* r_t at time t is a scalar value that represents the immediate benefit (or cost) the agent receives after taking action a_t in state s_t. It is a function $R(s_t, a_t)$ mapping state-action pairs to real numbers:

$$r_t = R(s_t, a_t). \tag{537}$$

The agent's objective is to maximize the cumulative reward, which is given by the *total return* from time t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{538}$$

The agent seeks to find a policy π that maximizes the expected return. The Bellman equation for the expected return is:

$$V^{\pi}(s_t) = R(s_t, \pi(s_t)) + \gamma \sum_{s_{t+1} \in \mathcal{S}} P(s_{t+1}|s_t, \pi(s_t)) V^{\pi}(s_{t+1}). \tag{539}$$

This recursive relation helps in solving for the optimal value function. An RL problem is typically modeled as a *Markov Decision Process (MDP)*, which is defined as the tuple:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma), \tag{540}$$

where:

- $\mathcal{S}$ is the state space,
- $\mathcal{A}$ is the action space,
- $P(s_{t+1}|s_t, a_t)$ is the state transition probability,
- $R(s_t, a_t)$ is the reward function,
- γ is the discount factor.

The agent's goal is to solve the MDP by finding the optimal policy $\pi^*(s_t)$ that maximizes the cumulative expected reward. Reinforcement Learning provides a powerful framework for decision-making in uncertain environments, where the agent seeks to maximize cumulative rewards over time. The core concepts—agents, states, actions, rewards—are formalized mathematically within the structure of a Markov Decision Process, enabling the application of optimization techniques such as dynamic programming, Q-learning, and policy gradient methods to solve complex decision-making problems.

10.2 Deep Q-Learning

Literature Review: Alonso and Arias (2025) [357] rigorously explored the mathematical foundations of Q-learning and its convergence properties. The authors analyze viscosity solutions and the Hamilton-Jacobi-Bellman (HJB) equation, demonstrating how Q-learning approximations align with these principles. The work provides new theoretical guarantees for Q-learning under different function approximation settings. Lu et. al. (2024) [358] proposed a factored empirical Bellman operator to mitigate the curse of dimensionality in Deep Q-learning. The authors provide rigorous theoretical analysis on how factorization reduces complexity while preserving optimality. The study improves the scalability of deep reinforcement learning models. Humayoo (2024) [359] extended Temporal Difference (TD) Learning to deep Q-learning using time-scale separation techniques. It introduces a Q(Δ)-learning approach that improves stability and convergence speed in complex environments. Empirical results validate its performance in Atari benchmarks. Jia et. al. (2024) [360] integrated Deep Q-learning (DQL) with Game Theory for anti-jamming strategies in wireless networks. It provides a rigorous theoretical framework on how multi-agent Q-learning can improve resilience against adversarial attacks. The study introduces multi-armed bandit algorithms and their convergence properties. Chai et. al. (2025) [361] provided a mathematical analysis of transfer learning in non-stationary Markov Decision Processes (MDPs). It extends Deep Q-learning to settings where the environment changes over time, establishing error bounds for Q-learning in these domains. Yao and Gong (2024) [362] developed a resilient Deep Q-network (DQN) model for multi-agent systems (MASs) under Byzantine attacks. The work introduces a novel distributed Q-learning approach with provable robustness against adversarial perturbations. Liu et. al. (2025) [363] introduced SGD-TripleQNet, a multi-Q-learning framework that integrates three Deep Q-networks. The authors provide a mathematical foundation and proof of convergence for their model. The paper bridges reinforcement learning with stochastic gradient descent (SGD) optimization. Masood et. al. (2025) [364] merged Deep Q-learning with Game Theory (GT) to optimize energy efficiency in smart agriculture. It proposes a mathematical model for dynamic energy allocation, proving the existence of Nash equilibria in Q-learning-based decision-making environments. Patrick (2024) [365] bridged economic modeling with Deep Q-learning. It formulates dynamic pricing strategies using deep reinforcement learning (DRL) and provides mathematical proofs on how RL adapts to economic shocks. Mimouni and Avrachenkov (2025) [366] introduced a novel Deep Q-learning algorithm that incorporates the Whittle index, a key concept in optimal stopping problems. It proves convergence bounds and applies the model to email recommender systems, demonstrating improved performance over traditional Q-learning methods.

Deep Q-Learning (DQL) is an advanced reinforcement learning (RL) technique where the goal is to approximate the optimal action-value function $Q^*(s,a)$ through the use of deep neural networks. In traditional Q-learning, the action-value function $Q(s,a)$ maps a state-action pair to the expected return or cumulative discounted reward from that state-action pair, under the assumption of following an optimal policy. Formally, the Q-function is defined as:

$$Q(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right] \tag{541}$$

where $\gamma \in [0,1]$ is the discount factor, which determines the weight of future rewards relative to immediate rewards, and r_t is the reward received at time step t. The optimal Q-function $Q^*(s,a)$ satisfies the Bellman optimality equation:

$$Q^*(s,a) = \mathbb{E}\left[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_0 = s, a_0 = a\right] \tag{542}$$

where s_{t+1} is the next state after taking action a in state s, and the maximization term represents the optimal future expected reward. This equation represents the recursive structure of the optimal action-value function, where each Q-value is updated based on the reward obtained in the current step and the maximum future reward expected from the next state. The goal is to learn the optimal Q-function through iterative updates, typically using the Temporal Difference (TD) method. In Deep Q-Learning, the Q-function is approximated by a deep neural network, as directly storing Q-values for every state-action pair is computationally infeasible for large state and action spaces. Let the approximated Q-function be $Q_\theta(s,a)$, where θ denotes the parameters (weights and biases) of the neural network that approximates the action-value function. The deep Q-network (DQN) aims to learn $Q_\theta(s,a)$ such that it closely approximates $Q^*(s,a)$ over time. The update of the Q-function follows the TD error principle, where the goal is to minimize the difference between the current Q-values and the target Q-values derived from the Bellman equation. The loss function for training the DQN is given by:

$$L(\theta) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\left[(y_t - Q_\theta(s_t,a_t))^2\right] \tag{543}$$

where $\mathcal{D}$ denotes the experience replay buffer containing previous transitions (s_t, a_t, r_t, s_{t+1}). The target y_t for the Q-values is defined as:

$$y_t = r_t + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a') \tag{544}$$

Here, θ^- represents the parameters of the target network, which is a slowly updated copy of the online network parameters θ. The target network Q_{θ^-} is used to generate stable targets for the Q-value updates, and its parameters are updated periodically by copying the parameters from the online network θ after every T steps. The idea behind this is to stabilize the training by preventing rapid changes in the Q-values due to feedback loops from the Q-network's predictions. The update rule for the network parameters θ follows the gradient descent method and is expressed as:

$$\nabla_\theta L(\theta) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\left[(y_t - Q_\theta(s_t,a_t))\,\nabla_\theta Q_\theta(s_t,a_t)\right] \tag{545}$$

where $\nabla_\theta Q_\theta(s_t, a_t)$ is the gradient of the Q-function with respect to the parameters θ, which is computed using backpropagation through the neural network. This gradient is used to update the parameters of the Q-network to minimize the loss function. In reinforcement learning, the agent must balance exploration (trying new actions) and exploitation (selecting actions that maximize the reward). This is often handled by using an epsilon-greedy policy, where the agent selects a random action with probability ϵ and the action with the highest Q-value with probability $1 - \epsilon$. The epsilon value is decayed over time to ensure that, as the agent learns, it shifts from exploration to more exploitation. The epsilon-greedy action selection rule is given by:

$$a_t = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q_\theta(s_t, a), & \text{with probability } 1 - \epsilon \end{cases} \tag{546}$$

This policy encourages the agent to explore different actions at the beginning of training and gradually exploit the learned Q-values as training progresses. The decay of ϵ typically follows an annealing schedule to balance exploration and exploitation effectively. A critical component in stabilizing training in Deep Q-Learning is the use of experience replay. In standard Q-learning, the updates are based on consecutive transitions, which can lead to high correlations between consecutive data points. This correlation can slow down learning or even lead to instability. Experience replay addresses this issue by storing a buffer of past experiences and sampling random mini-batches from this buffer during training. This breaks the correlation between consecutive samples and results in more stable and efficient updates. Mathematically, the loss function for training the network involves random sampling of transitions (s_t, a_t, r_t, s_{t+1}) from the experience replay buffer $\mathcal{D}$, and the update to the Q-values is computed using the Bellman error based on the sampled experiences:

$$L(\theta) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}} \left[\left(r_t + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a') - Q_\theta(s_t, a_t) \right)^2 \right] \tag{547}$$

This method ensures that the Q-values are updated in a way that is less sensitive to the order in which experiences are observed, promoting more stable learning dynamics.

Despite its success, the DQL algorithm can still suffer from certain issues such as overestimation bias and instability due to the maximization step in the Bellman equation. Overestimation bias occurs because the maximization operation $\max_{a'} Q_{\theta^-}(s_{t+1}, a')$ tends to overestimate the true value, as the Q-values are updated based on the same Q-network. To address this, Double Q-learning was introduced, which uses two separate Q-networks for action selection and value estimation, reducing overestimation bias. In Double Q-learning, the target Q-value is computed using the following equation:

$$y_t = r_t + \gamma Q_{\theta^-} \left(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a') \right) \tag{548}$$

This approach helps to mitigate the overestimation problem by decoupling the action selection from the Q-value estimation process. The value of $\arg\max$ is taken from the online network Q_θ, while the Q-value for the next state is estimated using the target network Q_{θ^-}. Another extension to improve the DQL framework is Dueling Q-Learning, which decomposes the Q-function into two separate components: the state value function $V_\theta(s)$ and the advantage function $A_\theta(s, a)$. The Q-function is then expressed as:

$$Q_\theta(s, a) = V_\theta(s) + A_\theta(s, a) \tag{549}$$

This decomposition allows the agent to learn the value of a state $V_\theta(s)$ independently of the specific actions, thus reducing the number of parameters needed for learning. This is particularly beneficial in environments where many actions have similar expected rewards, as it enables the agent to focus on identifying the value of states rather than overfitting to individual actions.

In conclusion, Deep Q-Learning is an advanced reinforcement learning method that utilizes deep neural networks to approximate the optimal Q-function, enabling agents to handle large state and action spaces. The mathematical formulation of DQL involves minimizing the loss function based on the temporal difference error, utilizing experience replay to stabilize learning, and using target networks to prevent instability. Extensions such as Double Q-learning and Dueling Q-learning further improve the performance and stability of the algorithm. Despite its remarkable successes, Deep Q-Learning still faces challenges such as overestimation bias and instability, which have been addressed with innovative modifications to the original algorithm.

10.3 Applications in Games and Robotics

Literature Review: Khlifi (2025) [368] applied Double Deep Q-Networks (DDQN) to autonomous driving. The paper discusses the transfer of RL techniques from gaming into self-driving cars, showing how deep RL can handle complex decision-making in dynamic environments. A novel reward function is introduced to improve path efficiency and safety. Kuczkowski (2024) [369] extended multi-objective RL (MORL) to traffic and robotic systems and introduced energy-efficient reinforcement learning for robotics in smart city applications. He also evaluated how RL-based traffic control systems optimize travel time and reduce energy consumption. Krauss et. al. (2025) [370] explored evolutionary algorithms for training RL-based neural networks. The approach integrates mutation-based evolution with reinforcement learning, optimizing RL policies for robot control and gaming AI. This approach shows improvements in learning speed and adaptability in multi-agent robotic environments. Ahamed et. al. (2025) [371] developed RL strategies for robotic soccer, implementing adaptive ball-kicking mechanics and used game engines to train robots, bridging simulated learning and real-world robotics. They also proposed modular robot formations, demonstrating how RL can optimize team play. Elmquist et. al. (2024) [372] focused on sim-to-real transfer in RL for robotics and developed an RL model that can adapt to real-world imperfections (e.g., lighting, texture variations). They used deep learning and image-based metrics to measure differences between simulated and real-world training environments. Kobanda et. al. (2024) [373] introduced a hierarchical approach to offline reinforcement learning (ORL) for robotic control and gaming AI. The study proposes policy subspaces that allow RL models to transfer knowledge across different tasks and demonstrated its effectiveness in goal-conditioned RL for adaptive video game AI. Shefin et. al. (2024) [367] focused on safety-critical RL applications in games and robotic manipulation. They introduced a framework for explainable reinforcement learning (XRL), making AI decisions more interpretable and applied to robotic grasping tasks, ensuring safe and reliable interactions. Xu et. al. (2025) [374] developed UPEGSim, a Gazebo-based simulation framework for underwater robotic games. They used reinforcement learning to optimize evasion strategies in underwater drone combat and highlighted RL applications in military and search-and-rescue robotics. Patadiya et. al. (2024) [375] used Deep RL to create autonomous players in racing games (Forza Horizon 5). They combined AlexNet with DRL for vision-based self-driving agents in gaming. The model learns optimal driving strategies through self-play. Janjua et. al. (2024) [376] explored RL scalability challenges in robotics and open-world games. They studied RL's adaptability in dynamic, open-ended environments (e.g., procedural game worlds) and discussed generalization techniques for RL agents, improving their performance in unpredictable scenarios.

Reinforcement Learning (RL) is a subfield of machine learning where an agent learns to make decisions by interacting with an environment. The goal of the agent is to maximize a cumulative reward signal over time by taking actions that affect its environment. The RL framework is formally represented by a *Markov Decision Process (MDP)*, which is defined by a 5-tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$, where:

- $\mathcal{S}$ is the state space, which represents all possible states the agent can be in.

- $\mathcal{A}$ is the action space, which represents all possible actions the agent can take.
- $P(s'|s, a)$ is the state transition probability, which defines the probability of transitioning from state s to state s' under action a.
- $r(s, a)$ is the reward function, which defines the immediate reward received after taking action a in state s.
- $\gamma \in [0, 1)$ is the discount factor, which determines the importance of future rewards.

The objective in RL is for the agent to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes its expected *return* (the cumulative discounted reward), which is mathematically expressed as:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right], \tag{550}$$

where s_t denotes the state at time t, and $a_t = \pi(s_t)$ is the action taken according to the policy π. The expectation is taken over the agent's interaction with the environment, under the policy π. The agent seeks to maximize this expected return by choosing actions that yield the most reward over time. The optimal *value function* $V^*(s)$ is defined as the maximum expected return that can be obtained starting from state s, and is governed by the *Bellman optimality equation*:

$$V^*(s) = \max_a \mathbb{E} \left[r(s, a) + \gamma V^*(s') \right], \tag{551}$$

where s' is the next state, and the expectation is taken with respect to the transition dynamics $P(s'|s, a)$. The *action-value function* $Q^*(s, a)$ represents the maximum expected return from taking action a in state s, and then following the optimal policy. It satisfies the Bellman optimality equation for $Q^*(s, a)$:

$$Q^*(s, a) = \mathbb{E} \left[r(s, a) + \gamma \max_{a'} Q^*(s', a') \right], \tag{552}$$

where a' is the next action to be taken, and the expectation is again over the state transition probabilities. These Bellman equations form the basis of many RL algorithms, which iteratively approximate the value functions to learn an optimal policy. To solve these equations, one of the most widely used methods is *Q-learning*, an off-policy, model-free RL algorithm. Q-learning iteratively updates the action-value function $Q(s, a)$ according to the following rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right], \tag{553}$$

where α is the learning rate that controls the step size of updates, and γ is the discount factor. The key idea behind Q-learning is that the agent learns the optimal action-value function $Q^*(s, a)$ without needing a model of the environment. The agent improves its action-value estimates over time by interacting with the environment and receiving feedback (rewards). The iterative nature of this update ensures convergence to the optimal Q^*, under the condition that all state-action pairs are visited infinitely often and α is decayed appropriately. *Policy Gradient* methods, in contrast, directly optimize the policy π_θ, which is parameterized by a vector θ. These methods are useful in high-dimensional or continuous action spaces where action-value methods may struggle. The objective in policy gradient methods is to maximize the expected return, $J(\pi_\theta)$, which is given by:

$$J(\pi_\theta) = \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right]. \tag{554}$$

The policy is updated using the *gradient ascent* method, and the gradient of the expected return with respect to θ is computed as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q(s_t, a_t) \right], \tag{555}$$

where $Q(s_t, a_t)$ is the action-value function, and $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is the *score function*, representing the sensitivity of the policy's likelihood to the policy parameters. By following this gradient, the policy parameters θ are updated to improve the agent's performance. This method, known as *REINFORCE*, is particularly effective when the action space is large or continuous, and the policy needs to be parameterized with complex models, such as deep neural networks. In both Q-learning and policy gradient methods, *exploration* and *exploitation* are essential concepts. *Exploration* refers to trying new actions that have not been sufficiently tested, whereas *exploitation* involves choosing actions that are known to yield high rewards. The *epsilon-greedy* strategy is a common way to balance exploration and exploitation, where with probability ϵ, the agent chooses a random action, and with probability $1 - \epsilon$, it chooses the action with the highest expected reward. As the agent learns, ϵ is typically decayed over time to reduce exploration and focus more on exploiting the learned policy. In more complex environments, *Boltzmann exploration* or *entropy regularization* techniques are used to maintain a controlled amount of randomness in the policy to encourage exploration. In *multi-agent games*, RL takes on additional complexity. When multiple agents interact, the environment is no longer static, as each agent's actions affect the others. In this context, RL can be used to find optimal strategies through *game theory*. A fundamental concept here is the *Nash equilibrium*, where no agent can improve its payoff by changing its strategy, assuming all other agents' strategies remain fixed. In mathematical terms, for two agents i and j, a Nash equilibrium (π_i^*, π_j^*) satisfies:

$$r_i(\pi_i^*, \pi_j^*) \geq r_i(\pi_i, \pi_j^*) \quad \text{and} \quad r_j(\pi_i^*, \pi_j^*) \geq r_j(\pi_i^*, \pi_j), \tag{556}$$

where $r_i(\pi_i, \pi_j)$ is the payoff function of agent i when playing policy π_i against agent j's policy π_j. Finding Nash equilibria in multi-agent RL is a complex and computationally challenging task, requiring the agents to learn in a non-stationary environment where the other agents' strategies are also changing over time. In the context of *robotics*, RL is used to solve high-dimensional control tasks, such as *motion planning* and *trajectory optimization*. The robot's state space is often represented by vectors of its position, velocity, and other physical parameters, while the action space consists of control inputs, such as joint torques or linear velocities. In this setting, RL algorithms learn to map states to actions that optimize the robot's performance in a task-specific way, such as minimizing energy consumption or completing a task in the least time. The dynamics of the robot are often modeled by differential equations:

$$\dot{x}(t) = f(x(t), u(t)), \tag{557}$$

where $x(t)$ is the state vector at time t, and $u(t)$ is the control input. Through RL, the robot learns to optimize the control policy $u(t)$ to maximize a reward function, typically involving a combination of task success and efficiency. Deep RL, specifically, allows for the representation of highly complex control policies using neural networks, enabling robots to tackle tasks that require high-dimensional sensory input and decision-making, such as object manipulation or autonomous navigation.

In games, RL has revolutionized the field by enabling agents to learn complex strategies in environments where hand-crafted features or simple tabular representations are insufficient. A key challenge in Deep Reinforcement Learning (DRL) is stabilizing the training process, as neural networks are prone to issues such as *overfitting*, *exploding gradients*, and *vanishing gradients*. Techniques such as *experience replay* and *target networks* are used to mitigate these challenges, ensuring stable and efficient learning. Thus, Reinforcement Learning, with its theoretical underpinnings in MDPs, Bellman equations, and policy optimization methods, provides a mathematically rich and deeply rigorous approach to solving sequential decision-making problems. Its application to fields such as games and robotics not only illustrates its versatility but also pushes the boundaries of machine learning into real-world, high-complexity scenarios.

11 Natural Language Processing (NLP)

Literature Review: Jurafsky and Martin 2023 [225] wrote book that is a cornerstone of NLP theory, covering fundamental concepts like syntax, semantics, and discourse analysis, alongside deep learning approaches to NLP. The book integrates linguistic theory with probabilistic and neural methodologies, making it an essential resource for students and researchers alike. It thoroughly explains sequence labeling, parsing, transformers, and BERT models. Manning and Schütze 1999 [226] wrote a foundational text in NLP, particularly for probabilistic models. It covers hidden Markov models (HMMs), n-gram language models, and expectation-maximization (EM), concepts that still underpin modern transformer-based NLP models. It also introduces latent semantic analysis (LSA), a precursor to modern word embeddings. Liu and Zhang (2018) [227] presented a detailed exploration of deep learning-based NLP, including word embeddings, recurrent neural networks (RNNs), LSTMs, GRUs, and transformers. It introduces the mathematical foundations of neural networks, making it a bridge between classical NLP and deep learning. Allen (1994) [228] wrote a seminal book in NLP, focusing on symbolic and rule-based approaches. It provides detailed coverage of semantic parsing, discourse modeling, and knowledge representation. While it predates deep learning, it forms a strong theoretical foundation for logical and linguistic approaches to NLP. wrote Koehn (2009) [231] wrote a definitive work on statistical NLP, particularly machine translation techniques like phrase-based translation, alignment models, and decoder algorithms. It remains relevant even as neural translation models (e.g., Transformer-based systems) dominate. We now mention some of the recent works in Natural Language Processing (NLP). Hempelmann [230] explored how linguistic theories of humor can be incorporated into Large Language Models (LLMs). It discusses the integration of formal humor theories into neural models and whether LLMs can be used to test linguistic hypotheses. Eisenstein (2020) [232] wrote a modern NLP textbook that bridges theory and practice. It covers both probabilistic and deep learning approaches, including dependency parsing, sequence-to-sequence models, and attention mechanisms. Unlike many texts, it also discusses ethics and bias in NLP models. Otter et. al. (2018) [233] provides a comprehensive review of neural architectures in NLP, covering CNNs, RNNs, attention mechanisms, and reinforcement learning for NLP. It discusses both theoretical implications and empirical advancements, making it an essential reference for deep learning in language tasks. The Oxford Handbook of Computational Linguistics (2022) [234] provides a comprehensive collection of essays covering the entire field of NLP and computational linguistics, including morphology, syntax, semantics, discourse processing, and deep learning applications. It presents theoretical debates and practical applications across different NLP domains. Li et. al. (2025) [229] introduced an advanced multi-head attention mechanism that combines explorative factor analysis with NLP models. It enhances our understanding of how transformers encode syntactic and semantic relationships.

11.1 Text Classification

Literature Review: Liu et. al. (2024) [235] provided a systematic review of text classification techniques, covering traditional machine learning methods (e.g., SVM, Naïve Bayes, Decision Trees) and deep learning approaches (CNNs, RNNs, LSTMs, and transformers). It also discusses feature extraction techniques such as TF-IDF, word embeddings, and BERT-based representations. Çekik (2025) [236] introduced a rough set-based approach for text classification, highlighting how term weighting strategies impact classification accuracy. It explores feature reduction and entropy-based selection methods to enhance text classifiers. Zhu et. al. (2025) [237] presented a novel entropy-based prefix tuning method for hierarchical text classification. It demonstrates how entropy regularization can enhance transformer-based classifiers like BERT and GPT for multi-label and hierarchical categorization. Matrane et. al. (2024) [238] investigated dialectal text classification challenges in Arabic NLP. It proposes preprocessing optimizations for low-resource dialects and demonstrates how transfer learning improves classification accuracy. Moqbel and Jain (2025)

[239] applies text classification to detect deception in online product reviews. It integrates cognitive appraisal theory and NLP-based text mining to distinguish fake vs. genuine reviews. Kumar et. al. (2025) [240] focused on medical text classification, demonstrating how NLP techniques can be applied to diagnose diseases using electronic health records (EHRs) and patient symptoms extracted from text data. Yin (2024) [241] provided a deep dive into aspect-based sentiment analysis (ABSA), discussing challenges in fine-grained text classification. It introduces new BERT-based techniques to improve aspect-level sentiment classification accuracy. Raghavan (2024) [242] examines personality classification using text data. It evaluates the performance of NLP-based personality prediction models and compares lexicon-based, deep learning, and transformer-based approaches. Semeraro et. al. (2025) [243] introduced EmoAtlas, a tool that merges psychological lexicons, artificial intelligence, and network science to perform emotion classification in textual data. It compares its accuracy with BERT and ChatGPT. Cai and Liu (2024) [244] provides a practical approach to text classification in discourse analysis. It explores Python-based techniques for analyzing therapy talk and sentiment classification in conversational texts.

Text classification is a fundamental problem in machine learning and natural language processing (NLP), where the goal is to assign predefined categories to a given text based on its content. This process involves several steps, including text preprocessing, feature extraction, model training, and evaluation. In this answer, we will explore these steps with a focus on the underlying mathematical principles and models used in text classification. The first step in text classification is preprocessing the raw text data. This typically involves the following operations:

- **Tokenization**: Breaking the text into words or tokens.
- **Stopword Removal**: Removing common words (such as ”and”, ”the”, etc.) that do not carry significant meaning.
- **Stemming and Lemmatization**: Reducing words to their base or root form, e.g., ”running” becomes ”run”.
- **Lowercasing**: Converting all words to lowercase to ensure consistency.
- **Punctuation Removal**: Removing punctuation marks.

These operations result in a cleaned and standardized text, ready for feature extraction. Once the text is preprocessed, the next step is to convert the text into numerical representations that can be fed into machine learning models. The most common methods for feature extraction include:

1. Bag-of-Words (BoW) model
2. Term Frequency-Inverse Document Frequency (TF-IDF)

In the first method (**Bag-of-Words (BoW) model**), each document is represented as a vector where each dimension corresponds to a unique word in the corpus. The value of each dimension is the frequency of the word in the document. If we have a corpus of N documents and a vocabulary of M words, the document i can be represented as a vector $\mathbf{x}_i \in \mathbb{R}^M$, where:

$$\mathbf{x}_i = [f(w_1, d_i), f(w_2, d_i), \ldots, f(w_M, d_i)] \tag{558}$$

where $f(w_j, d_i)$ is the frequency of the word w_j in the document d_i. The BoW model captures only the frequency of terms within the document and disregards their order. While simple and computationally efficient, this model does not capture the syntactic or semantic relationships between words in the document.

A more sophisticated and improved representation can be obtained through Term **Frequency-Inverse Document Frequency (TF-IDF)**, which scales the raw frequency of words by their relative importance in the corpus. TF-IDF is a more advanced technique that aims to weight words based on their importance. It considers both the frequency of a word in a document and the rarity of the word across all documents. The term frequency (TF) of a word w in document d is defined as:

$$\mathrm{TF}(w, d) = \frac{\mathrm{count}(w, d)}{\text{total number of words in } d} \tag{559}$$

The inverse document frequency (IDF) is given by:

$$\mathrm{IDF}(w) = \log\left(\frac{N}{\mathrm{DF}(w)}\right) \tag{560}$$

where N is the total number of documents and $\mathrm{DF}(w)$ is the number of documents containing the word w. The TF-IDF score is the product of these two:

$$\text{TF-IDF}(w, d) = \mathrm{TF}(w, d) \cdot \mathrm{IDF}(w) \tag{561}$$

There are several machine learning models that can be used for text classification, ranging from simpler models to more complex ones. A common approach to text classification is to use a linear model such as logistic regression or linear support vector machines (SVM). Given a feature vector $\mathbf{x}_i$ for document i, the prediction of the class label y_i can be made as:

$$\hat{y}_i = \sigma(\mathbf{w}^T \mathbf{x}_i + b) \tag{562}$$

where σ is the sigmoid function for binary classification, and $\mathbf{w}$ and b are the weight vector and bias term, respectively. The model parameters $\mathbf{w}$ and b are learned by minimizing a loss function, such as the binary cross-entropy loss. More complex models, such as *Neural Networks (NN)*, involve deeper mathematical formulations. In a typical feedforward neural network, the goal is to learn a set of parameters that map an input vector $\mathbf{x}_i$ to an output label y_i. The network consists of multiple layers of interconnected neurons, each of which applies a non-linear transformation to the input. Given an input vector $\mathbf{x}_i$, the output of the network is computed as:

$$\mathbf{h}_i^{(l)} = \sigma(\mathbf{W}^{(l)} \mathbf{h}_i^{(l-1)} + \mathbf{b}^{(l)}) \tag{563}$$

where $\mathbf{h}_i^{(l)}$ is the activation of layer l, σ is the activation function (e.g., ReLU, sigmoid, or tanh), $\mathbf{W}^{(l)}$ is the weight matrix, and $\mathbf{b}^{(l)}$ is the bias term for layer l. The input to the network is passed through several hidden layers before producing the final classification output. The output layer typically applies a *softmax* function to obtain a probability distribution over the possible classes:

$$P(y_c|\mathbf{x}_i) = \frac{\exp(\mathbf{W}_c^T \mathbf{h}_i + b_c)}{\sum_{c'} \exp(\mathbf{W}_{c'}^T \mathbf{h}_i + b_{c'})} \tag{564}$$

where $\mathbf{W}_c$ and b_c are the weights and bias for class c, and $\mathbf{h}_i$ is the output of the last hidden layer. The network is trained by minimizing a *cross-entropy loss function*:

$$\mathcal{L}(\mathbf{W}, \mathbf{b}) = -\sum_{c=1}^{C} y_{i,c} \log P(y_c|\mathbf{x}_i) \tag{565}$$

where $y_{i,c}$ is the one-hot encoded label for class c, and the goal is to minimize the difference between the predicted probability distribution and the true class distribution. Throughout the entire process, *optimization* plays a crucial role in fine-tuning model parameters to minimize classification errors. Common optimization techniques include *stochastic gradient descent (SGD)* and its variants, such as *Adam* and *RMSProp*, which update model parameters iteratively based on the gradient of the loss function with respect to the parameters. Given the loss function $\mathcal{L}(\theta)$ parameterized by θ, the gradient of the loss with respect to a parameter θ_i is computed as:

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta_i} \tag{566}$$

The parameter update rule for gradient descent is then:

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial \mathcal{L}(\theta)}{\partial \theta_i} \tag{567}$$

where η is the learning rate. For each iteration, this update rule adjusts the model parameters in the direction of the negative gradient, ultimately converging to a set of parameters that minimizes the classification error.

In summary, text classification is an advanced and multifaceted problem that requires a deep understanding of various mathematical principles, including linear algebra, probability theory, optimization, and functional analysis. The entire process, from text preprocessing to feature extraction, model training, and evaluation, involves the application of rigorous mathematical techniques that enable the effective classification of text into meaningful categories. Each of these steps, whether simple or complex, plays an integral role in transforming raw text data into actionable insights using mathematically sophisticated models and algorithms.

11.2 Machine Translation

Literature Review: Wu et. al. (2020) [245] introduced end-to-end neural machine translation (NMT), focusing on sequence-to-sequence models, attention mechanisms, and transformer architectures. It explains encoder-decoder frameworks, self-attention, and positional encoding, laying the groundwork for modern NMT. Hettiarachchi et. al. (2024) [246] presented Amharic-to-English machine translation using transformers. It introduces character embeddings and regularization techniques for handling low-resource languages, a critical challenge in multilingual NLP. Das and Sahoo (2024) [247] discussed word alignment models, a fundamental concept in SMT. It explains IBM Model 1-5, HMM alignments, and the role of alignment in phrase-based models. It also explores challenges in handling syntactic divergence across languages. Oluwatoki et. al. (2024) [248] presented one of the first transformer-based Yoruba-to-English MT systems. It highlights how multilingual NLP models struggle with resource-scarce languages and proposes Rouge-based evaluation for MT systems. Uçkan and Kurt [249] discusses the role of word embeddings (Word2Vec, GloVe, FastText) in MT. It covers semantic representation in vector spaces, crucial for context-aware translation in NMT. discussed multiword expressions (MWEs) in MT, a major challenge in NLP. It covers idiomatic expressions, collocations, and phrasal verbs, showing how neural models struggle with multiword disambiguation. Pastor et. al. (2024) [250] discussed multiword expressions (MWEs) in MT, a major challenge in NLP. It covers idiomatic expressions, collocations, and phrasal verbs, showing how neural models struggle with multiword disambiguation. Fernandes (2024) [251] compared open-source large language models (LLMs) and NMT systems in translating spatial semantics in EN-PT-BR (English-Portuguese-Brazilian Portuguese) subtitles. It highlights the limitations of both traditional and neural MT in capturing contextual spatial meanings. Jozić (2024) [252] evaluated ChatGPT's translation capabilities against specialized MT systems like eTranslation (EU Commission MT model). It shows how general-purpose LLMs can rival dedicated NMT systems but struggle with domain-specific translations. Yang (2025) [253] introduced error-detection models for NMT output, using transformer-based classifiers to detect syntactic and semantic errors in machine-generated translations.

Machine Translation (MT) in Natural Language Processing (NLP) is a highly intricate computational task that requires converting text from one language (source language) to another (target language) by using statistical, rule-based, and deep learning models, often underpinned by probabilistic and neural network-based frameworks. The goal is to determine the most probable target sequence $T = \{t_1, t_2, \ldots, t_N\}$ from the given source sequence $S = \{s_1, s_2, \ldots, s_T\}$, by modeling the conditional probability $P(T \mid S)$. The optimal translation is typically defined by:

$$T^* = \arg\max_T P(T \mid S) \tag{568}$$

This involves estimating the probability of T given S, with the assumption that the translation can be described probabilistically. In the most fundamental form of statistical machine translation (SMT), this probability is often modeled through a series of translation models that decompose the translation process into manageable components. The conditional probability $P(T \mid S)$ in SMT can be factorized using Bayes' theorem:

$$P(T \mid S) = \frac{P(S,T)}{P(S)} = \frac{P(T \mid S)P(S)}{P(S)} \tag{569}$$

Given this decomposition, the core of early SMT models, such as IBM models, sought to model the joint probability $P(S,T)$ over source and target language pairs. Specifically, in word-based models like IBM Model 1, the task reduces to estimating the probability of translating each word in the source language S to its corresponding word in the target language T. The joint probability can be written as:

$$P(S,T) = \prod_{i=1}^{T} \prod_{j=1}^{N} t(s_i \mid t_j) \tag{570}$$

where $t(s_i \mid t_j)$ is the probability of translating word s_i in the source sentence to word t_j in the target sentence. The estimation of these probabilities, $t(s_i \mid t_j)$, is typically achieved by analyzing parallel corpora through various techniques such as Expectation-Maximization (EM), which allows the unsupervised learning of these translation probabilities from large amounts of bilingual text data. The EM algorithm iterates between computing the expected alignments of words in the source and target languages and refining the model parameters accordingly. The word-based translation models, however, do not take into account the structure of the language, which often leads to suboptimal translations, especially in languages with significantly different syntactic structures. The challenges stem from the word order differences and idiomatic expressions that cannot be captured through a simple word-to-word mapping. To overcome these limitations, IBM Model 2 introduced the concept of word alignments, where an additional hidden variable A is introduced, representing a possible alignment between words in the source and target sentences. This can be expressed as:

$$P(S,T,A) = \prod_{i=1}^{T} \prod_{j=1}^{N} t(s_i \mid t_j) a(s_i \mid t_j) \tag{571}$$

where $a(s_i \mid t_j)$ denotes the alignment probability between word s_i in the source language and word t_j in the target language. By optimizing these alignment probabilities, SMT systems improve the quality of translations by better modeling the relationship between the source and target sentences. Estimating $a(s_i \mid t_j)$, however, requires computationally expensive algorithms, which can be handled by methods like EM for iterative refinement.

A more sophisticated approach was introduced with sequence-to-sequence (Seq2Seq) models, which significantly improved the translation process by leveraging deep learning techniques. The core of Seq2Seq is the encoder-decoder framework, where an encoder processes the entire source sentence and encodes it into a context vector, and a decoder generates the target sequence. In this approach, the translation probability is formulated as:

$$P(T \mid S) = P(t_1 \mid S) \prod_{i=2}^{N} P(t_i \mid t_{<i}, S) \tag{572}$$

where $t_{<i}$ denotes the previously generated target words, capturing the sequential nature of translation. The key advantage of the Seq2Seq model is its ability to model entire sentences at once, providing a richer, more flexible representation of both the source and target sequences compared to word-based models. The encoder, typically implemented using Recurrent Neural Networks (RNNs) or more advanced variants such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit

(GRU) networks, encodes the source sequence S into hidden states. The hidden state at time step t is computed recursively, based on the input x_t (the source word representation at time step t) and the previous hidden state h_{t-1}:

$$h_t = f(h_{t-1}, x_t) \tag{573}$$

where f represents the update function, which is often parameterized as a non-linear function, such as a sigmoid or tanh. This recursion generates a sequence of hidden states $\{h_1, h_2, \ldots, h_T\}$, each encoding the relevant information of the source sentence. In this model, the decoder generates the target sequence one token at a time by conditioning on the previous tokens $t_{<i}$ and the context vector c, which is typically the last hidden state from the encoder. The conditional probability of generating the next target word is given by:

$$P(t_i \mid t_{<i}, S) = \text{softmax}(Wh_t) \tag{574}$$

where W is a learned weight matrix, and h_t is the hidden state of the decoder at time step t. The softmax function converts the output of the network into a probability distribution over the vocabulary, and the word with the highest probability is chosen as the next target word.

A significant improvement to Seq2Seq was introduced through the attention mechanism. This allows the decoder to dynamically focus on different parts of the source sentence during translation, instead of relying on a single fixed-length context vector. The attention mechanism computes a set of attention weights $\alpha_{t,i}$ for each source word, which are used to compute a weighted sum of the encoder's hidden states to form a dynamic context vector c_t. The attention weight $\alpha_{t,i}$ for time step t in the decoder and source word i is calculated as:

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T} \exp(e_{t,k})} \tag{575}$$

where $e_{t,i} = \text{score}(h_t, h_i)$ is a learned scoring function, which can be modeled as:

$$e_{t,i} = \vec{v}^\top \tanh(W_1 h_t + W_2 h_i) \tag{576}$$

This attention mechanism allows the model to adaptively focus on relevant parts of the source sentence while generating each word in the target sentence, thus overcoming the limitations of fixed-length context vectors in long sentences. Training a machine translation model typically involves optimizing a loss function that quantifies the difference between the predicted target sequence and the true target sequence. The most common loss function is the negative log-likelihood:

$$L(\theta) = -\sum_{i=1}^{N} \log P(t_i \mid t_{<i}, S; \theta) \tag{577}$$

where θ represents the parameters of the model. The parameters of the neural network are updated using gradient-based optimization techniques, such as stochastic gradient descent (SGD) or Adam, with the gradient of the loss function with respect to each parameter being computed via backpropagation. In backpropagation, the gradient is computed by recursively applying the chain rule through the layers of the network. For a parameter θ, the gradient is given by:

$$\frac{\partial L(\theta)}{\partial \theta} = \frac{\partial L(\theta)}{\partial y} \frac{\partial y}{\partial \theta} \tag{578}$$

where y represents the output of the network, and $\frac{\partial L(\theta)}{\partial y}$ is the gradient of the loss with respect to the output. These gradients are then propagated backward through the network to update the parameters, thereby minimizing the loss function. The quality of a translation is often evaluated using automatic metrics such as BLEU (Bilingual Evaluation Understudy), which measures the n-gram overlap between the machine-generated translation and human references. The BLEU score for an n-gram of length n is computed as:

$$\text{BLEU}(T, R) = \exp\left(\sum_{n=1}^{N} w_n \log p_n(T, R)\right) \tag{579}$$

where $p_n(T, R)$ is the precision of n-grams between the target translation T and reference R, and w_n is the weight assigned to each n-gram length. Despite advancements, machine translation still faces challenges, such as handling rare or out-of-vocabulary words, idiomatic expressions, and the alignment of complex syntactic structures across languages. Approaches such as transfer learning, unsupervised learning, and domain adaptation are being explored to address these issues and improve the robustness and accuracy of MT systems.

11.3 Chatbots and Conversational AI

Literature Review: Linnemann and Reimann (2024) [254] explored how conversational AI, particularly chatbots, affects human interactions and social psychology. It discusses the role of Large Language Models (LLMs) and their applications in dialogue systems, providing a theoretical perspective on chatbot integration into human communication. Merkel and Schorr (2024) [255] categorizes different types of conversational agents and their NLP capabilities. It discusses the evolution from rule-based chatbots to transformer-based models, emphasizing how natural language processing has enhanced chatbot usability. Kushwaha and Singh (2022) [256] provided a technical analysis of chatbot architectures, covering intent recognition, entity extraction, and dialogue management. It compares traditional ML-based chatbot models with deep learning approaches. Macedo et. al. (2024) [257] presented a healthcare-oriented chatbot that leverages conversational AI to assist Parkinson's patients. It details speech-to-text and NLP techniques used for interactive healthcare applications. Gupta et. al. (2024) [258] outlines the theoretical foundations of generative AI-based chatbots, explaining how LLMs like ChatGPT influence conversational AI. It also introduces a framework for evaluating chatbot effectiveness. Foroughi and Iranmanesh (2025) [259] examined how AI-powered chatbots influence consumer behavior in e-commerce. It introduces a theoretical framework to understand chatbot adoption and trust. Jandhyala (2024) [260] provided a deep dive into chatbot development, covering NLP techniques, intent recognition, and multi-turn dialogue management. It also discusses best practices for chatbot deployment. Pavlović and Savić (2024) [261] explored the use of conversational AI in digital marketing, analyzing how LLM-based chatbots improve customer experience. It also evaluates sentiment analysis and feedback loops in chatbot interactions. Mannava et. al. (2024) [262] examined the ethical and functional aspects of chatbots in child education, focusing on how NLP models must be adjusted for child-appropriate interactions. Sherstinova and Mikhaylovskiy (2024) [263] focused on language-specific challenges in chatbot NLP, discussing how conversational AI models struggle with morphologically rich languages like Russian.

Chatbots and Conversational AI have evolved as some of the most sophisticated applications of Natural Language Processing (NLP), a subfield of artificial intelligence that strives to enable machines to understand, generate, and interact in human language. At the core of conversational AI is the ability to generate meaningful, contextually appropriate responses in a coherent and fluent manner. This challenge is deeply rooted in both the complexities of natural language itself and the mathematical models that attempt to approximate human understanding. This intricate task involves processing language at different levels: syntactic (structure), semantic (meaning), and pragmatic (context). These systems employ probabilistic and algebraic techniques to handle language complexities and employ *statistical models*, *deep neural networks*, and *optimization algorithms* to generate, understand, and respond to language.

In mathematical terms, conversational AI can be seen as a sequence of transformations from one set of words or symbols (the input) to another (the output). The first mathematical aspect is *language modeling*, which is crucial for predicting the likelihood of word sequences. The probability distribution of a sequence of words $w_1, w_2, \ldots, w_n$ is generally computed using the chain rule of probability:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, w_2, \ldots, w_{i-1}) \tag{580}$$

where $P(w_i | w_1, w_2, \ldots, w_{i-1})$ models the conditional probability of the word w_i given all the preceding words. This is a central concept in language generation tasks. In traditional *n-gram models*, this conditional probability is estimated by considering only a fixed number of previous words. The *bigram* model, for instance, assumes that the probability of a word depends only on the previous word, leading to:

$$P(w_i | w_1, w_2, \ldots, w_{i-1}) \approx P(w_i | w_{i-1}) \tag{581}$$

However, more advanced conversational AI systems, such as those based on *recurrent neural networks (RNNs)*, attempt to model dependencies over much longer sequences. RNNs, in particular, process the input sequence $w_1, w_2, \ldots, w_n$ recursively by maintaining a hidden state h_t that captures the context up to time t. The hidden state is computed by:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \tag{582}$$

where σ is a non-linear activation function (e.g., *tanh* or *sigmoid*), W_h, W_x are weight matrices, and b is a bias term. While RNNs provide a mechanism to capture sequential dependencies, they suffer from the *vanishing gradient* problem, particularly for long sequences. To address this issue, *Long Short-Term Memory (LSTM)* units and *Gated Recurrent Units (GRUs)* were introduced, with special gating mechanisms that help mitigate the loss of information over long time horizons. These networks introduce memory cells and gates, which regulate the flow of information in the network. For instance, the LSTM memory cell is governed by the following equations:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f), \quad i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i), \quad o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{583}$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tanh(W_c x_t + U_c h_{t-1} + b_c), \quad h_t = o_t \cdot \tanh(c_t) \tag{584}$$

where f_t, i_t, o_t are the forget, input, and output gates, respectively, and c_t represents the cell state, which carries information across time steps. The LSTM thus enables better capture of long-range dependencies by controlling the flow of information in a more structured way. In more recent times, *transformer models* have revolutionized conversational AI by replacing the sequential nature of RNNs with parallelized self-attention mechanisms. The transformer model uses *multi-head self-attention* to weigh the importance of each word in a sequence relative to all other words. The self-attention mechanism computes a weighted sum of values V based on queries Q and keys K, with the attention being computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{585}$$

where d_k is the dimension of the key vectors. This operation allows the model to attend to all parts of the input sequence simultaneously, enabling better handling of long-range dependencies and improving computational efficiency by processing sequences in parallel. Unlike RNNs, transformers do not process tokens in a fixed order but instead utilize positional encoding to inject sequence order information. The positional encoding for position i and dimension $2k$ is given by:

$$PE(i, 2k) = \sin\left(\frac{i}{10000^{2k/d}}\right), \quad PE(i, 2k+1) = \cos\left(\frac{i}{10000^{2k/d}}\right) \tag{586}$$

where d is the embedding dimension and k is the index for the dimension of the positional encoding. This approach allows transformers to handle longer sequences more efficiently than RNNs and LSTMs, and is the basis for models like *BERT*, *GPT*, and other state-of-the-art conversational models. Semantic understanding in conversational AI involves translating sentences into formal representations that can be manipulated by the system. A well-known approach for capturing meaning is *compositional semantics*, which treats the meaning of a sentence as a function of the meanings of its parts. For this, *lambda calculus* is often employed to represent the meaning of sentences as functions that operate on their arguments. For example, the sentence "John saw the car" can be represented as a lambda expression:

$$\lambda x.\text{see}(x, \text{car}) \tag{587}$$

where $\text{see}(x, y)$ is a predicate representing the action of seeing, and λx quantifies over the subject of the action. This allows for the compositional building of complex meanings from simpler components. Dialogue management is another critical aspect of conversational AI systems. This is the process of maintaining coherence and context over the course of a conversation. It involves understanding the user's input in light of prior dialogue history and generating a response that is contextually relevant. To model the dialogue state, *Markov Decision Processes (MDPs)* are commonly employed. In this context, the dialogue state is represented as a set of possible states, with actions being transitions between these states. The goal is to select actions (responses) that maximize cumulative rewards, which, in this case, corresponds to maintaining a coherent and engaging conversation. The value function $V(s)$ at state s can be computed using the Bellman equation:

$$V(s) = \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V(s') \right] \tag{588}$$

where $R(s, a)$ is the immediate reward for taking action a from state s, γ is the discount factor, and $P(s'|s, a)$ represents the transition probability to the next state s' given action a. By solving this equation, the system can determine the optimal policy for responding to user inputs in a way that maximizes long-term conversational quality. Once the dialogue state is updated, the next step in conversational AI is to generate a response. This is typically achieved using *sequence-to-sequence models*, in which the input sequence (e.g., the user's query) is processed by an encoder to produce a fixed-size context vector, and a decoder generates the output sequence (e.g., the chatbot's response). The basic structure of these models can be expressed as:

$$y_t = \text{Decoder}(y_{t-1}, h_t) \tag{589}$$

where y_t represents the token generated at time t, and h_t is the hidden state passed from the encoder. Attention mechanisms are incorporated into this framework to allow the decoder to focus on different parts of the input sequence at each step, improving the quality of the generated response. Training conversational models requires optimizing parameters through *backpropagation* and *gradient descent*. The loss function, typically *cross-entropy loss*, is minimized to update the model's parameters:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) \tag{590}$$

where $\hat{y}_i$ is the predicted probability for the correct token y_i, and N is the length of the sequence. The parameters θ are updated iteratively through gradient descent, adjusting the weights to minimize the error.

In summary, chatbots and conversational AI systems are grounded in a rich mathematical framework involving statistics, linear algebra, optimization, and neural networks. Each step, from language modeling to dialogue management, relies on carefully constructed mathematical foundations that drive the ability of machines to interact intelligently and meaningfully with humans. Through advancements in deep learning and optimization techniques, conversational AI continues to push the boundaries of what machines can understand and generate in natural language, leading to more sophisticated, human-like interactions.

12 Deep Learning Frameworks

12.1 TensorFlow

Literature Review: Takhsha et. al. (2025) [283] introduced a TensorFlow-based framework for medical deep learning applications. The authors propose a novel deep learning diagnostic system that integrates Choquet integral theory with TensorFlow-based models, improving the explainability of deep learning decisions in medical imaging. Singh and Raman (2025) [284] extended TensorFlow to Graph Neural Networks (GNNs), discussing how TensorFlow's computational graph structure aligns with graph theory. It provides a rigorous mathematical foundation for applying deep learning to non-Euclidean data structures. Yao et. al. (2024) [285] critically analyzed TensorFlow's vulnerabilities to adversarial attacks and introduces a robust deep learning ensemble framework. The authors explore autoencoder-based anomaly detection using TensorFlow to enhance cybersecurity defenses. Chen et. al. (2024) [286] provided an extensive comparison of TensorFlow pretrained models for various big data applications. It discusses techniques like transfer learning, fine-tuning, and self-supervised learning, emphasizing how TensorFlow automates hyperparameter tuning. Dumić (2024) [287] wrote as a rigorous educational resource, guiding learners through neural network construction using TensorFlow. It bridges the gap between deep learning theory and TensorFlow's practical implementation, emphasizing gradient descent, backpropagation, and weight initialization. Bajaj et. al. (2024) [288] implemented CNNs for handwritten digit recognition using TensorFlow and provides a rigorous mathematical breakdown of convolution operations, activation functions, and optimization techniques. It highlights TensorFlow's computational efficiency in large-scale character recognition tasks. Abbass and Fyath (2024) [289] introduced a TensorFlow-based framework for optical fiber communication modeling. It explores how deep learning can optimize fiber optic transmission efficiency by using TensorFlow for predictive analytics and channel equalization. Prabha et. al. (2024) [290] rigorously analyzed TensorFlow's role in precision agriculture, focusing on time-series analysis, computer vision, and reinforcement learning for crop monitoring. It delves into TensorFlow's API optimizations for handling sensor data and remote sensing images. Abdelmadjid and Abdeldjallil (2024) [291] examined TensorFlow Lite for edge computing, rigorously testing optimized CNN architectures on low-power devices. It provides a theoretical comparison of computational efficiency, energy consumption, and model accuracy in resource-constrained environments. Mlambo (2024) [292] bridged Bayesian inference and deep learning, providing a rigorous derivation of Bayesian Neural Networks (BNNs) implemented in TensorFlow. It explores how TensorFlow integrates probabilistic models with deep learning frameworks.

TensorFlow operates primarily on *tensors*, which are multi-dimensional arrays generalizing scalars, vectors, and matrices. For instance, a scalar is a rank-0 tensor, a vector is a rank-1 tensor, a matrix is a rank-2 tensor, and tensors of higher ranks represent multi-dimensional arrays. These tensors can be written mathematically as:

$$\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n} \tag{591}$$

where $d_1, d_2, \ldots, d_n$ represent the dimensions of the tensor. TensorFlow leverages efficient *tensor operations* that allow the manipulation of large-scale data in a computationally optimized manner. These operations are the foundation of all the transformations and calculations within TensorFlow models. For example, the *dot product* of two vectors $\vec{a}$ and $\vec{b}$ is a scalar:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i \tag{592}$$

Similarly, for matrices, operations like matrix multiplication $A \cdot B$ are highly optimized, taking advantage of *batch processing* and *parallelism* on devices such as GPUs and TPUs. TensorFlow's underlying libraries, such as Eigen, employ these parallel strategies to optimize memory usage and reduce computation time. The heart of TensorFlow's efficiency lies in its *computation graph*, which represents the relationships between different operations. The computation graph is a directed acyclic graph (DAG) where nodes represent computational operations, and the edges represent the flow of data (tensors). Each operation in the graph is a function, f, that maps a set of inputs to an output tensor:

$$y = f(x_1, x_2, \ldots, x_n) \tag{593}$$

The graph is built by users or automatically by TensorFlow, where the nodes represent operations such as addition, multiplication, or more complex transformations. Once the computation graph is defined, TensorFlow optimizes the graph by reordering computations, applying algebraic transformations, or parallelizing independent subgraphs. The graph is executed either in a dynamic manner (eager execution) or after optimization (static graph execution), depending on the user's preference. *Automatic differentiation* is another key feature of TensorFlow, and it relies on the *chain rule* of differentiation to compute gradients. The gradient of a scalar-valued function $f(x_1, x_2, \ldots, x_n)$ with respect to an input tensor x_i is computed as:

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial f}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{594}$$

where y_j represents intermediate variables computed during the forward pass of the network. In the context of a neural network, this chain rule is used to propagate errors backward from the output to the input layers during the *backpropagation* process, where the objective is to update the network's weights to minimize the loss function L. Consider a neural network with a simple architecture, consisting of an input layer, one hidden layer, and an output layer. Let X represent the input tensor, W_1 and b_1 the weights and biases of the hidden layer, and W_2 and b_2 the weights and biases of the output layer. The forward pass can be written as:

$$h = \sigma(W_1 X + b_1) \tag{595}$$

$$\hat{y} = W_2 h + b_2 \tag{596}$$

where σ is the activation function, such as the ReLU function $\sigma(x) = \max(0, x)$, and $\hat{y}$ is the predicted output. The objective in training a model is to minimize a loss function $L(\hat{y}, y)$, where y represents the true labels. The loss function can take different forms, such as the *mean squared error* for regression tasks:

$$L(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{597}$$

or the *cross-entropy loss* for classification tasks:

$$L(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{598}$$

where C is the number of classes, and $\hat{y}_i$ is the predicted probability of class i under the softmax function. The optimization of this loss function requires the computation of the *gradients* of L with respect to the model parameters W_1, b_1, W_2, b_2. This is achieved through *backpropagation*, which applies the chain rule iteratively through the layers of the network. To perform optimization, TensorFlow employs algorithms like *Gradient Descent (GD).* The basic gradient descent update rule for parameters θ is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta) \tag{599}$$

where η is the learning rate, and $\nabla_\theta L(\theta)$ represents the gradient of the loss function with respect to the model parameters θ. Variants of gradient descent, such as *Stochastic Gradient Descent (SGD)*, update the parameters using a subset (mini-batch) of the training data rather than the entire dataset:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \frac{1}{m} \sum_{i=1}^{m} L(\theta, x_i, y_i) \tag{600}$$

where m is the batch size, and (x_i, y_i) are the data points in the mini-batch. More sophisticated optimizers like *Adam* (Adaptive Moment Estimation) use both momentum (first moment) and scaling (second moment) to adapt the learning rate for each parameter. The update rule for Adam is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta L(\theta) \tag{601}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta L(\theta))^2 \tag{602}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{603}$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{604}$$

where β_1 and β_2 are the exponential decay rates, and ϵ is a small constant to prevent division by zero. The inclusion of both the first and second moments allows Adam to adaptively adjust the learning rate, speeding up convergence. In addition to standard optimization methods, TensorFlow supports *distributed computing*, enabling model training across multiple devices, such as GPUs and TPUs. In a distributed setting, the model's parameters are split across different workers, each handling a portion of the data. The gradients computed by each worker are averaged, and the global parameters are updated:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta L_i(\theta) \tag{605}$$

where $L_i(\theta)$ is the loss computed on the i-th device, and N is the total number of devices. TensorFlow's efficient parallelism ensures that large-scale data processing tasks can be carried out with high computational throughput, thus speeding up model training on large datasets.

TensorFlow also facilitates *model deployment* on different platforms. *TensorFlow Lite* enables model inference on mobile devices by converting trained models into optimized, smaller formats. This process involves *quantization*, which reduces the precision of the weights and activations, thereby reducing memory consumption and computation time. The conversion process aims to balance model accuracy and performance, ensuring that deep learning models can run efficiently on resource-constrained devices like smartphones and IoT devices. For *web applications*, TensorFlow offers *TensorFlow.js*, which allows users to run machine learning models directly in the browser, leveraging the computational power of the client-side GPU or CPU. This is particularly useful for real-time interactions where low-latency predictions are required without sending data to a server. Moreover, TensorFlow provides an ecosystem that extends beyond basic machine learning tasks. For instance, *TensorFlow Extended (TFX)* supports the deployment of machine learning models in production environments, automating the steps from model training to deployment. *TensorFlow Probability* supports probabilistic modeling and uncertainty estimation, which are critical in domains such as reinforcement learning and Bayesian inference.

12.2 PyTorch

Literature Review: Galaxy Yanshi Team of Beihang University [293] examined the use of PyTorch as a deep learning framework for real-time astronaut facial recognition in space stations. It explores the Bayesian coding theory within PyTorch models and its significance in optimizing neural network architectures. It provides a theoretical exploration of probability distributions in PyTorch models, demonstrating how deep learning can be used in constrained computational environments. Tabel (2024) [294] extended PyTorch to Spiking Neural Networks (SNNs), a biologically inspired neural network type. It details a new theoretical approach for learning spike timings using PyTorch's computational graph. The paper bridges neuromorphic computing and PyTorch's automatic differentiation, expanding the theory behind temporal deep learning. Naderi et. al. (2024) [295] introduced a hybrid physics-based deep learning framework that integrates discrete element modeling (DEM) with PyTorch-based networks. It demonstrates how physical simulation problems can be formulated as deep learning models in PyTorch, providing new insights into neural solvers for scientific computing. Polaka (2024) [296] evaluated reinforcement learning (RL) theories within PyTorch, exploring the mathematical rigor of RL frameworks in safe AI applications. The author provided a strong theoretical foundation for understanding deep reinforcement learning (DeepRL) in PyTorch, emphasizing how state-of-the-art RL theories are embedded in the framework. Erdogan et. al. (2024) [297] explored the theoretical framework for reducing stochastic communication overheads in large-scale recommendation systems built using PyTorch. It introduced an optimized gradient synchronization method that can enhance PyTorch-based deep learning models for distributed computing. Liao et. al. (2024) [298] extended the Iterative Partial Diffusion Model (IPDM) framework, implemented in PyTorch, for medical image processing and advanced the theory of deep generative models in PyTorch, specifically in diffusion-based learning techniques. Sekhavat et. al. (2024) [299] examined the theoretical intersection between deep learning in PyTorch and artificial intelligence creativity, referencing Nietzschean philosophical concepts. The author also explored how PyTorch enables neural creativity and provides a rigorous theoretical model for computational aesthetics. Cai et. al. (2025) [300] developed a new theoretical framework for explainability in neural networks using Shapley values, implemented in PyTorch and enhanced the mathematical rigor of explainable AI (XAI) using PyTorch's autograd system to analyze feature importance. Na (2024) [301] proposed a novel ensemble learning theory using PyTorch, specifically in weakly supervised learning (WSL). The paper extends Bayesian learning models in PyTorch for handling sparse labeled data, addressing critical gaps in WSL. Khajah (2024) [302] combined item response theory (IRT) and Bayesian knowledge tracing (BKT) using PyTorch to model generalizable skill discovery. This study presents a rigorous statistical theory for adaptive learning systems using PyTorch's probabilistic programming capabilities.

The **dynamic computation graph** in PyTorch forms the core of its ability to perform efficient and flexible machine learning tasks, especially deep learning models. To understand the underlying mathematical and computational principles, we must explore how the graph operates, what it represents, and how it changes during the execution of a machine learning program. Unlike the static computation graphs employed in frameworks like TensorFlow (pre-Eager execution mode), PyTorch constructs the computation graph dynamically, as the operations are performed in the forward pass. This allows PyTorch to adapt to various input sizes, model structures, and control flows that can change during execution. This adaptability is essential in enabling PyTorch to handle models like recurrent neural networks (RNNs), which operate on sequences of varying lengths, or models that incorporate conditionals in their computation steps.

The **computation graph** itself can be mathematically represented as a **directed acyclic graph (DAG)**, where the nodes represent operations and intermediate results, while the edges represent the flow of data between these nodes. Each operation (e.g., addition, multiplication, or non-linear activation) is applied to tensors, and the outputs of these operations are used as inputs for subsequent operations. The central feature of PyTorch's dynamic computation graph is its **construction at runtime**. For instance, when a tensor $\mathbf{A}$ is created, it might be involved in a series of operations that eventually lead to the calculation of a loss function $\mathcal{L}$. As each operation is executed, PyTorch constructs an edge from the node representing the input tensor $\mathbf{A}$ to the node representing the output tensor $\mathbf{B}$. Mathematically, the transformation between these tensors can be described by:

$$\mathbf{B} = f(\mathbf{A}; \theta) \tag{606}$$

where f represents the transformation function (which could be a linear or nonlinear operation), and θ represents the parameters involved in this transformation (e.g., weights or biases in the case of neural networks). The construction of the dynamic graph allows PyTorch to deal with **variable-length sequences**, which are common in tasks such as **time-series prediction**, **natural language processing (NLP)**, and **speech recognition**. The length of the sequence can change depending on the input data, and thus, the number of iterations or layers required in the computation will also vary. In a **recurrent neural network (RNN)**, for example, the hidden state $\mathbf{h}_t$ at each time step t is a function of the previous hidden state $\mathbf{h}_{t-1}$ and the input at the current time step $\mathbf{x}_t$. This can be described mathematically as:

$$\mathbf{h}_t = f(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}) \tag{607}$$

where f is typically a non-linear activation function (e.g., a hyperbolic tangent or a sigmoid), and $\mathbf{W}_h, \mathbf{W}_x, \mathbf{b}$ represent the weight matrices and bias vector, respectively. This equation encapsulates the recursive nature of RNNs, where each output depends on the previous output and the current input. In a static computation graph, the number of operations for each sequence would need to be predefined, leading to inefficiency when sequences of different lengths are processed. However, in PyTorch, the computation graph is created dynamically for each sequence, which allows for the efficient handling of varying-length sequences and avoids redundant computation.

The key to PyTorch's efficiency lies in **automatic differentiation**, which is managed by its **autograd** system. When a tensor $\mathbf{A}$ has the property `requires_grad=True`, PyTorch starts tracking all operations performed on it. Suppose that the tensor $\mathbf{A}$ is involved in a sequence of operations to compute a scalar loss $\mathcal{L}$. For example, if the loss is a function of $\mathbf{Y}$, the output tensor, which is computed through multiple layers, the objective is to find the gradient of $\mathcal{L}$ with respect to $\mathbf{A}$. This requires the computation of the Jacobian matrix, which represents the gradient of each component of $\mathbf{Y}$ with respect to each component of $\mathbf{A}$. Using the chain rule of differentiation, the gradient of the loss with respect to $\mathbf{A}$ is given by:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \sum_i \frac{\partial \mathcal{L}}{\partial \mathbf{Y}_i} \cdot \frac{\partial \mathbf{Y}_i}{\partial \mathbf{A}} \tag{608}$$

This is an application of the **multivariable chain rule**, where $\frac{\partial \mathcal{L}}{\partial \mathbf{Y}_i}$ represents the gradient of the loss with respect to the output tensor at the i-th component, and $\frac{\partial \mathbf{Y}_i}{\partial \mathbf{A}}$ is the Jacobian matrix for the transformation from $\mathbf{A}$ to $\mathbf{Y}$. This computation is achieved by **backpropagating** the gradients through the computation graph that PyTorch builds dynamically. Every operation node in the graph has an associated gradient, which is propagated backward through the graph as we move from the loss back to the input parameters. For example, if $\mathbf{Y} = \mathbf{A} \cdot \mathbf{B}$, the gradient of the loss with respect to $\mathbf{A}$ would be:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \cdot \mathbf{B}^T \tag{609}$$

Similarly, the gradient with respect to $\mathbf{B}$ would be:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \cdot \mathbf{A}^T \tag{610}$$

This shows how the gradients are passed backward through the computation graph, utilizing the stored operations at each node to calculate the required derivatives. The advantage of this **dynamic construction** of the graph is that it does not require the entire graph to be constructed beforehand, as in the static graph approach. Instead, the graph is dynamically updated as operations are executed, making it both more **memory-efficient** and **computationally efficient**. An important feature of PyTorch's dynamic graph is its ability to handle **conditionals** within the computation. Consider a case where we have different branches in the computation based on a conditional statement. In a static graph, such conditionals would require the entire graph to be predetermined, including all possible branches. In contrast, PyTorch constructs the relevant part of the graph depending on the input data, effectively enabling a **branching computation**. For instance, suppose that we have a decision-making process in a neural network model, where the output depends on whether an input tensor exceeds a threshold $\mathbf{x}_i > \mathbf{t}$:

$$\mathbf{y}_i = \begin{cases} \mathbf{A} \cdot \mathbf{x}_i + \mathbf{b} & \text{if } \mathbf{x}_i > \mathbf{t} \\ \mathbf{C} \cdot \mathbf{x}_i + \mathbf{d} & \text{otherwise} \end{cases} \tag{611}$$

In a static graph, we would have to design two separate branches and potentially deal with the computational cost of unused branches. In PyTorch's dynamic graph, only the relevant branch is executed, and the graph is updated accordingly to reflect the necessary operations. The **memory efficiency** in PyTorch's dynamic graph construction is particularly evident when handling large models and training on **large datasets**. When building models like **deep neural networks (DNNs)**, the operations performed on each tensor during both the forward and backward passes are recorded in the computation graph. This allows for **efficient reuse** of intermediate results, and only the necessary memory is allocated for each tensor during the graph's construction. This stands in contrast to static computation graphs, where the full graph needs to be defined and memory allocated up front, potentially leading to unnecessary memory consumption.

To summarize, the **dynamic computation graph** in PyTorch is a powerful tool that allows for flexible model building and efficient computation. By constructing the graph incrementally during the execution of the forward pass, PyTorch is able to dynamically adjust to the input size, control flow, and variable-length sequences, leading to more efficient use of memory and computational resources. The **autograd** system enables **automatic differentiation**, applying the chain rule of calculus to compute gradients with respect to all model parameters. This flexibility is a key reason why PyTorch has gained popularity for deep learning research and production, as it combines **high performance** with **flexibility** and **transparency**, allowing researchers and engineers to experiment with dynamic architectures and complex control flows without sacrificing efficiency.

12.3 JAX

Literature Review: Li et. al. (2024) [313] introduced JAX-based differentiable density functional theory (DFT), enabling end-to-end differentiability in materials science simulations. This paper extends machine learning theory into quantum chemistry by leveraging JAX's automatic differentiation and parallelization capabilities for efficient optimization of density functional models. Bieberich and Li (2024) [314] explored quantum machine learning (QML) using JAX and Diffrax to solve neural differential equations efficiently. They developed a new theoretical model for quantum neural ODEs and discussed how JAX facilitates efficient GPU-based quantum simulations. Dagréou et. al. (2024) [315] analyzed the efficiency of Hessian-vector product (HVP) computation in JAX and PyTorch for deep learning. They established a mathematical foundation for computing second-order derivatives in deep learning and optimization, showcasing JAX's superior automatic differentiation. Lohoff and Neftci (2025) [316] developed a deep reinforcement learning (DRL) model that optimizes JAX's autograd engine for scientific computing. They demonstrated how reinforcement learning improves computational efficiency in JAX through a theoretical framework that eliminates redundant computations in deep learning. Legrand et. al. (2024) [317] introduced a JAX and Rust-based deep learning library for predictive coding networks (PCNs). They explored theoretical extensions of neural networks beyond traditional backpropagation, providing a formalized framework for hierarchical generative models. Alzás and Radev (2024) [318] used JAX to create differentiable models for nuclear reactions, demonstrating its power in high-energy physics simulations. They established a new differentiable framework for theoretical physics, utilizing JAX's gradient-based optimization to improve nuclear physics modeling. Edenhofer et. al. (2024) [319] developed a Gaussian Process and Variational Inference framework in JAX, extending traditional Bayesian methods. They bridged statistical physics and deep learning, formulating a theoretical link between Gaussian processes and deep neural networks using JAX. Chan et. al. (2024) [320] proposed a JAX-based quantum machine learning framework for long-tailed X-ray classification. They introduced a novel quantum transfer learning technique within JAX, demonstrating its advantages over classical deep learning models in medical imaging. Ye et. al. (2025) [321] used JAX to model electron transfer kinetics, bridging deep learning and density functional theory (DFT). They developed a new theoretical framework for modeling charge transfer reactions, leveraging JAX's high-performance computation for quantum chemistry applications. Khan et. al. (2024) [322] extended NODEs using JAX's efficient autodiff capabilities for high-dimensional dynamical systems. They established a rigorous mathematical framework for extending NODEs to stochastic and chaotic systems, leveraging JAX's high-speed parallelization.

JAX is an advanced numerical computing framework designed to optimize high-performance scientific computing tasks with particular emphasis on automatic differentiation, hardware acceleration, and just-in-time (JIT) compilation. These capabilities are essential for applications in machine learning, optimization, physical simulations, and computational science, where large-scale, high-dimensional computations must be executed with both speed and efficiency. At its core, JAX integrates a deep mathematical structure based on advanced concepts in linear algebra, optimization theory, tensor calculus, and numerical differentiation, providing the foundation for scalable computations across multi-core CPUs, GPUs, and TPUs. The framework leverages the power of reverse-mode differentiation and JIT compilation to significantly reduce computation time while ensuring correctness and accuracy. The following rigorous exploration will dissect these operations mathematically and conceptually, explaining their inner workings and theoretical implications.

JAX's **automatic differentiation** is central to its ability to compute gradients, Jacobians, Hessians, and other derivatives efficiently. For many applications, the function of interest involves computing gradients with respect to model parameters in optimization and machine learning tasks. Automatic differentiation allows for the efficient computation of these gradients using the **reverse-mode** differentiation technique. Let us consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$, and suppose we wish to compute the gradient of the scalar-valued output with respect to each input variable. The gradient of f, denoted as $\nabla_{\mathbf{x}} f$, is a vector of partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left(\frac{\partial f_1}{\partial x_1}, \frac{\partial f_1}{\partial x_2}, \dots, \frac{\partial f_1}{\partial x_n}, \dots, \frac{\partial f_m}{\partial x_n} \right), \tag{612}$$

where $f = (f_1, f_2, \dots, f_m)$ represents a vector of m scalar outputs, and $\mathbf{x} = (x_1, x_2, \dots, x_n)$ represents the input vector. Reverse-mode differentiation computes this gradient by applying the **chain rule** in reverse order. If f is composed of several intermediate functions, say $f = g \circ h$, where $g : \mathbb{R}^m \to \mathbb{R}^p$ and $h : \mathbb{R}^n \to \mathbb{R}^m$, the gradient of f with respect to $\mathbf{x}$ is computed recursively by applying the chain rule:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left(\frac{\partial g}{\partial h} \right) \cdot \left(\frac{\partial h}{\partial x_1}, \frac{\partial h}{\partial x_2}, \dots, \frac{\partial h}{\partial x_n} \right). \tag{613}$$

This recursive application of the chain rule ensures that each intermediate gradient computation is propagated backward through the function's layers, reducing the number of required passes compared to forward-mode differentiation. This technique becomes particularly beneficial for functions where the number of outputs m is much smaller than the number of inputs n, as it minimizes the computational complexity. In the context of JAX, automatic differentiation is utilized through functions like `jax.grad`, which can be applied to scalar-valued functions to return their gradients with respect to vector-valued inputs. To compute higher-order derivatives, such as the Hessian matrix, JAX allows for the computation of second- and higher-order derivatives using similar principles. The Hessian matrix H of a scalar function $f(\mathbf{x})$ is given by the matrix of second derivatives:

$$H = \left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right), \tag{614}$$

which is computed by applying the chain rule once again. The second-order derivatives can be computed efficiently by differentiating the gradient once more, and this process can be extended to higher-order derivatives by continuing the recursive application of the chain rule. A central concept in JAX's approach to high-performance computing is **JIT (just-in-time) compilation**, which provides substantial performance gains by compiling Python functions into optimized machine code tailored to the underlying hardware architecture. JIT compilation in JAX is built on the foundation of the **XLA (Accelerated Linear Algebra)** compiler. XLA optimizes the execution of tensor operations by fusing multiple operations into a single kernel, thereby reducing the overhead associated with launching individual computation kernels. This technique is particularly effective for matrix multiplications, convolutions, and other tensor operations commonly found in machine learning tasks. For example, consider a simple sequence of operations $f = \mathrm{Op}_1(\mathrm{Op}_2(\dots(\mathrm{Op}_n(\mathbf{x}))))$, where Op_i represents different mathematical operations applied to the input tensor $\mathbf{x}$. Without optimization, each operation would typically be executed separately, introducing significant overhead. JAX's JIT compiler, however, recognizes this sequence and applies a fusion transformation, resulting in a single composite operation:

$$\mathrm{Optimized}(f(\mathbf{x})) = \mathrm{Fused\ Op}(\mathbf{x}), \tag{615}$$

where Fused Op represents a highly optimized version of the original sequence of operations. This optimization minimizes the number of kernel launches and reduces memory access overhead, which in turn accelerates the computation. The JIT compiler analyzes the computational graph of the function and identifies opportunities to combine operations into a more efficient form, ultimately speeding up the computation on hardware accelerators such as GPUs or TPUs.

The **vectorization** capability provided by JAX through the `jax.vmap` operator is another essential optimization for high-performance computing. This feature automatically vectorizes functions across batches of data, allowing the same operation to be applied in parallel across multiple data points. Mathematically, for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ and a batch of inputs $\mathbf{X} \in \mathbb{R}^{B \times n}$, the vectorized function can be expressed as:

$$\mathbf{Y} = \mathrm{vmap}(f)(\mathbf{X}), \tag{616}$$

where B is the batch size and $\mathbf{Y}$ is the matrix in $\mathbb{R}^{B \times m}$, containing the results of applying f to each row of $\mathbf{X}$. The mathematical operation applied by JAX is the same as applying f to each individual row $\mathbf{X}_i$, but with the benefit that the entire batch is processed in parallel, exploiting the available hardware resources efficiently. The ability to **parallelize computations across multiple devices** is one of JAX's strongest features, and it is enabled through the `jax.pmap` operator. This operator allows for the parallel execution of functions across different devices, such as multiple GPUs or TPUs. Suppose we have a function $f : \mathbb{R}^n \to \mathbb{R}^m$ and a batch of inputs $\mathbf{X} = (\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_p)$, distributed across p devices. The parallelized execution of the function can be written as:

$$\mathbf{Y} = \mathrm{pmap}(f)(\mathbf{X}), \tag{617}$$

where each device independently computes its portion of the computation $f(\mathbf{X}_i)$, and the results are gathered into the final output $\mathbf{Y}$. This capability is essential for large-scale distributed training of machine learning models, where the model's parameters and data must be distributed across multiple devices to ensure efficient training. The parallelization effectively reduces computation time, as each device operates on a distinct subset of the data and model parameters. **GPU/TPU acceleration** is another crucial aspect of JAX's performance, and it is facilitated by libraries like cuBLAS for GPUs, which are specifically designed to optimize matrix operations. The primary operation used in many numerical computing tasks is matrix multiplication, and JAX optimizes this by leveraging hardware-accelerated implementations of these operations. Consider the matrix multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A} \in \mathbb{R}^{n\times m}$ and $\mathbf{B} \in \mathbb{R}^{m\times p}$, resulting in a matrix $\mathbf{C} \in \mathbb{R}^{n\times p}$:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}. \tag{618}$$

Using cuBLAS or a similar library, JAX can execute this operation on a GPU, utilizing the massive parallel processing power of the hardware to perform the multiplication efficiently. This operation can be further optimized by considering the specific memory hierarchies of GPUs, where large matrix multiplications are broken down into smaller tiles that fit into the GPU's high-speed memory. This technique minimizes memory bandwidth constraints, accelerating the computation. In addition to these core operations, JAX allows for the definition of **custom gradients** using the `jax.custom_jvp` decorator, which enables users to specify the Jacobian-vector products (JVPs) manually for more efficient gradient computation. This feature is especially useful in machine learning applications, where certain operations might have custom gradients that cannot be computed automatically. For instance, in a non-trivial activation function such as the softmax, the custom gradient function might be provided explicitly for efficiency:

$$\frac{\partial \text{softmax}(\mathbf{x})}{\partial \mathbf{x}} = \text{diag}(\text{softmax}(\mathbf{x})) - \text{softmax}(\mathbf{x}) \cdot \text{softmax}(\mathbf{x})^T. \tag{619}$$

Thus, JAX allows for both flexibility and performance, enabling scientific computing applications that require both efficiency and the ability to define complex, custom derivatives.

By providing advanced capabilities such as automatic differentiation, JIT compilation, vectorization, parallelization, hardware acceleration, and custom gradients, JAX is equipped to handle a wide range of high-performance computing tasks, making it an invaluable tool for solving complex scientific and engineering problems. The framework not only ensures the correctness of numerical methods but also leverages the power of modern hardware to achieve performance that is crucial for large-scale simulations, machine learning, and optimization tasks.

13 Appendix

13.1 Linear Algebra Essentials

13.1.1 Matrices and Vector Spaces

Definition of a Matrix: A **matrix** A is a rectangular array of numbers (or elements from a field $\mathbb{F}$), arranged in rows and columns:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{F}^{m\times n} \tag{620}$$

where a_{ij} denotes the **entry** of A at the i-th row and j-th column. A **square matrix** is one where $m = n$. A matrix is **diagonal** if all off-diagonal entries are zero. For matrices $A \in \mathbb{F}^{m\times n}$ and $B \in \mathbb{F}^{m\times n}$ the following are the matrix operations:

- **Addition:** Defined entrywise:
$$(A+B)_{ij} = A_{ij} + B_{ij} \tag{621}$$
- **Scalar Multiplication:** For $\alpha \in \mathbb{F}$,
$$(\alpha A)_{ij} = \alpha \cdot A_{ij} \tag{622}$$
- **Matrix Multiplication:** If $A \in \mathbb{F}^{m\times p}$ and $B \in \mathbb{F}^{p\times n}$, then the product $C = AB$ is given by:
$$C_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj} \tag{623}$$
This is **only defined when** the number of columns of A equals the number of rows of B.
- **Transpose:** The **transpose** of A, denoted A^T, satisfies:
$$(A^T)_{ij} = A_{ji} \tag{624}$$
- **Determinant:** If $A \in \mathbb{F}^{n\times n}$, then its **determinant** is given recursively by:
$$\det(A) = \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(A_{1j}) \tag{625}$$
where A_{1j} is the $(n-1) \times (n-1)$ submatrix obtained by removing the first row and j-th column.
- **Inverse:** A square matrix A is **invertible** if there exists A^{-1} such that:
$$AA^{-1} = A^{-1}A = I \tag{626}$$
where I is the identity matrix.

13.1.2 Vector Spaces and Linear Transformations

Vector Spaces A **vector space** over a field $\mathbb{F}$ is a set V with two operations:

- **Vector Addition:** $\mathbf{v} + \mathbf{w}$ for $\mathbf{v}, \mathbf{w} \in V$
- **Scalar Multiplication:** $\alpha\mathbf{v}$ for $\alpha \in \mathbb{F}$ and $\mathbf{v} \in V$

satisfying the **8 vector space axioms** (associativity, commutativity, existence of identity, etc.). A set $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ is a **basis** if:

- It is **linearly independent**:
$$\sum_{i=1}^{n} \alpha_i \mathbf{v}_i = 0 \Rightarrow \alpha_i = 0, \forall i \tag{627}$$
- It **spans** V, meaning every $\mathbf{v} \in V$ can be written as:
$$\mathbf{v} = \sum_{i=1}^{n} \beta_i \mathbf{v}_i \tag{628}$$

The **dimension** of V, denoted $\dim(V)$, is the number of basis vectors. **Linear Transformations:** A function $T : V \to W$ is **linear** if:
$$T(\alpha\mathbf{v} + \beta\mathbf{w}) = \alpha T(\mathbf{v}) + \beta T(\mathbf{w}) \tag{629}$$
The **matrix representation** of T is the matrix A such that:
$$T(\mathbf{x}) = A\mathbf{x} \tag{630}$$

13.1.3 Eigenvalues and Eigenvectors

Definition: For a square matrix $A \in \mathbb{F}^{n\times n}$, an **eigenvalue** λ and **eigenvector** $\mathbf{v} \neq \mathbf{0}$ satisfy:

$$A\mathbf{v} = \lambda\mathbf{v} \tag{631}$$

Characteristic Equation: The eigenvalues are found by solving:

$$\det(A - \lambda I) = 0 \tag{632}$$

which gives an **n-th degree polynomial** in λ. The set of all solutions $\mathbf{v}$ to $(A - \lambda I)\mathbf{v} = 0$ is the **eigenspace** associated with λ.

13.1.4 Singular Value Decomposition (SVD)

Definition: For any $A \in \mathbb{F}^{m\times n}$, the **Singular Value Decomposition** (SVD) states:

$$A = U\Sigma V^T \tag{633}$$

where $U \in \mathbb{F}^{m\times m}$ is **orthogonal** ($U^TU = I$), $V \in \mathbb{F}^{n\times n}$ is **orthogonal** ($V^TV = I$), Σ is an $m \times n$ **diagonal matrix**:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r \end{bmatrix} \tag{634}$$

where σ_i are the **singular values**, given by:

$$\sigma_i = \sqrt{\lambda_i} \tag{635}$$

where λ_i are the eigenvalues of A^TA.

13.2 Probability and Statistics

13.2.1 Probability Distributions

A **probability distribution** is a mathematical function that provides the probabilities of occurrence of different possible outcomes in an experiment. A random variable X can take values from a sample space S, and the probability distribution describes how the probabilities are distributed over these possible outcomes.

Discrete Probability Distributions: For a discrete random variable X, which takes values from a countable set, the **probability mass function** (PMF) is defined as:

$$P(X = x_i) = p(x_i), \quad \forall x_i \in S \tag{636}$$

The PMF satisfies the following properties:

- $0 \leq p(x_i) \leq 1$ for each $x_i \in S$.
- The sum of probabilities across all possible outcomes is 1:

$$\sum_{x_i \in S} p(x_i) = 1 \tag{637}$$

An example of a discrete probability distribution is the **binomial distribution**, which describes the number of successes in a fixed number of independent Bernoulli trials. The PMF for the binomial distribution is:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, \dots, n \tag{638}$$

Where n is the number of trials, p is the probability of success on each trial, and k is the number of successes.

Continuous Probability Distributions: For a continuous random variable X, which takes values from a continuous set (e.g., the real line), the **probability density function** (PDF) is used instead of the PMF. The PDF $f(x)$ is defined such that for any interval $[a, b]$, the probability that X lies in this interval is:

$$P(a \leq X \leq b) = \int_a^b f(x)\, dx \tag{639}$$

The PDF must satisfy:

- $f(x) \geq 0$ for all x.
- The total probability over the entire range of X is 1:

$$\int_{-\infty}^{\infty} f(x)\, dx = 1 \tag{640}$$

An example of a continuous probability distribution is the **normal distribution**, which is given by the PDF:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad x \in \mathbb{R} \tag{641}$$

Where μ is the mean and σ^2 is the variance of the distribution.

13.2.2 Bayes' Theorem

Bayes' theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event. It is a fundamental result in the field of probability theory and statistics.

Let A and B be two events. Then, Bayes' theorem gives the conditional probability of A given B:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{642}$$

where $P(A|B)$ is the posterior probability of A given B, $P(B|A)$ is the likelihood, the probability of observing B given A, $P(A)$ is the prior probability of A, $P(B)$ is the marginal likelihood of B, computed as:

$$P(B) = \sum_i P(B|A_i)P(A_i) \tag{643}$$

In the continuous case, Bayes' theorem is written as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(B|A)P(A)}{\int P(B|A')P(A')\, dA'} \tag{644}$$

This allows one to update beliefs about a hypothesis A based on observed evidence B. Let us consider a diagnostic test for a disease. Let A be the event that a person has the disease and B

be the event that the test is positive. We are interested in the probability that a person has the disease given that the test is positive, i.e., $P(A|B)$. By Bayes' theorem, we have:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{645}$$

where $P(B|A)$ is the probability of a positive test result given that the person has the disease (sensitivity), $P(A)$ is the prior probability of having the disease, $P(B)$ is the total probability of a positive test result.

13.2.3 Statistical Measures

Statistical measures summarize the properties of data or a probability distribution. Some key statistical measures are the **mean**, **variance**, **standard deviation**, and **skewness**.

A **statistical measure** is a function $M : \mathcal{S} \to \mathbb{R}$ that assigns a real-valued quantity to an element in a statistical space $\mathcal{S}$, where $\mathcal{S}$ can represent a dataset, a probability distribution, or a stochastic process. Mathematically, a statistical measure must satisfy certain properties such as **measurability**, **invariance under transformation**, and **convergence consistency** in order to be well-defined. Statistical measures can be broadly classified into:

1. **Measures of Central Tendency** (e.g., mean, median, mode)
2. **Measures of Dispersion** (e.g., variance, standard deviation, interquartile range)
3. **Measures of Shape** (e.g., skewness, kurtosis)
4. **Measures of Association** (e.g., covariance, correlation)
5. **Information-Theoretic Measures** (e.g., entropy, mutual information)

Each of these measures provides different insights into the characteristics of a dataset or a probability distribution. There are several Measures of Central Tendency. Given a probability space $(\Omega, \mathcal{F}, P)$ and a random variable $X : \Omega \to \mathbb{R}$, the **expectation** (or mean) is defined as:

$$\mathbb{E}[X] = \int_{\Omega} X(\omega) dP(\omega) \tag{646}$$

If X is a discrete random variable with probability mass function $p(x)$, then:

$$\mathbb{E}[X] = \sum_{x \in \mathbb{R}} x p(x) \tag{647}$$

If X is a continuous random variable with probability density function $f(x)$, then:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f(x) dx \tag{648}$$

The **median** m of a probability distribution is defined as:

$$P(X \leq m) \geq \frac{1}{2}, \quad P(X \geq m) \geq \frac{1}{2} \tag{649}$$

In terms of the cumulative distribution function $F(x)$, the median m satisfies:

$$F(m) = \frac{1}{2} \tag{650}$$

The **mode** is defined as the point x_m that maximizes the probability density function:

$$x_m = \arg\max_x f(x) \tag{651}$$

The variance σ^2 of a random variable X is given by:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] \tag{652}$$

Expanding this expression:

$$\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{653}$$

The **standard deviation** σ is defined as the square root of the variance:

$$\sigma = \sqrt{\text{Var}(X)} \tag{654}$$

If Q_1 and Q_3 denote the first and third quartiles of a dataset (where Q_1 is the 25th percentile and Q_3 is the 75th percentile), then the interquartile range is:

$$IQR = Q_3 - Q_1 \tag{655}$$

The **skewness** of a random variable X is defined as:

$$\gamma_1 = \frac{\mathbb{E}[(X - \mathbb{E}[X])^3]}{\sigma^3} \tag{656}$$

It quantifies the asymmetry of the probability distribution. The **kurtosis** is given by:

$$\gamma_2 = \frac{\mathbb{E}[(X - \mathbb{E}[X])^4]}{\sigma^4} \tag{657}$$

A normal distribution has $\gamma_2 = 3$, and deviations from this indicate whether a distribution has heavy or light tails. There are several Measures of Association. The Covariance is defined as follows: Given two random variables X and Y, their **covariance** is:

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] \tag{658}$$

Expanding:

$$\text{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \tag{659}$$

The Pearson Correlation Coefficient defined as:

$$\rho(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y} \tag{660}$$

where σ_X and σ_Y are the standard deviations of X and Y, respectively. The Information-Theoretic Measure is Entropy which is defined as the **entropy** of a discrete probability distribution $p(x)$ is given by:

$$H(X) = -\sum_x p(x) \log p(x) \tag{661}$$

For continuous distributions with density $f(x)$, the **differential entropy** is:

$$h(X) = -\int_{-\infty}^{\infty} f(x) \log f(x)\, dx \tag{662}$$

Given two random variables X and Y, their mutual information is:

$$I(X; Y) = H(X) + H(Y) - H(X, Y) \tag{663}$$

which measures how much knowing X reduces uncertainty about Y. Statistical Measures satisfy Linearity and Invariance i.e.

- **Expectation is linear:**
$$\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y] \tag{664}$$
- **Variance is translation invariant but scales quadratically:**
$$\mathrm{Var}(aX + b) = a^2\mathrm{Var}(X) \tag{665}$$

For the Convergence and Asymptotic Behavior, The **law of large numbers** ensures that empirical means converge to the expected value, while the **central limit theorem** states that sums of i.i.d. random variables converge in distribution to a normal distribution.

The **mean** or **expected value** of a random variable X, denoted by $\mathbb{E}[X]$, represents the average value of X. For a discrete random variable:

$$\mathbb{E}[X] = \sum_{x_i \in S} x_i p(x_i) \tag{666}$$

For a continuous random variable, the expected value is given by:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f(x)\, dx \tag{667}$$

The **variance** of a random variable X, denoted by $\mathrm{Var}(X)$, measures the spread or dispersion of the distribution. For a discrete random variable:

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{668}$$

For a continuous random variable:

$$\mathrm{Var}(X) = \int_{-\infty}^{\infty} x^2 f(x)\, dx - \left(\int_{-\infty}^{\infty} x f(x)\, dx \right)^2 \tag{669}$$

The **standard deviation** is the square root of the variance and provides a measure of the spread of the distribution in the same units as the random variable:

$$\mathrm{SD}(X) = \sqrt{\mathrm{Var}(X)} \tag{670}$$

The **skewness** of a random variable X quantifies the asymmetry of the probability distribution. It is defined as:

$$\mathrm{Skew}(X) = \frac{\mathbb{E}[(X - \mathbb{E}[X])^3]}{(\mathrm{Var}(X))^{3/2}} \tag{671}$$

A positive skew indicates that the distribution has a long tail on the right, while a negative skew indicates a long tail on the left. The **kurtosis** of a random variable X measures the "tailedness" of the distribution, i.e., how much of the probability mass is concentrated in the tails. It is defined as:

$$\mathrm{Kurt}(X) = \frac{\mathbb{E}[(X - \mathbb{E}[X])^4]}{(\mathrm{Var}(X))^2} \tag{672}$$

A distribution with high kurtosis has heavy tails, and one with low kurtosis has light tails compared to a normal distribution.

13.3 Optimization Techniques

13.3.1 Gradient Descent (GD)

Gradient Descent is an iterative optimization algorithm used to minimize a differentiable function. The goal is to find the point where the function achieves its minimum value. Mathematically, it can be formulated as follows. Given a differentiable objective function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient descent update rule is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k) \tag{673}$$

where:

- $\mathbf{x}_k \in \mathbb{R}^n$ is the current point in the n-dimensional space (iteration index k),
- $\nabla f(\mathbf{x}_k)$ is the gradient of the objective function at $\mathbf{x}_k$,
- η is the learning rate (step size).

To analyze the convergence of gradient descent, we assume f is **convex** and **differentiable** with a **Lipschitz continuous gradient**. That is, there exists a constant $L > 0$ such that:

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|, \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n. \tag{674}$$

This property ensures the gradient of f does not change too rapidly, which allows us to bound the convergence rate. The following is an upper bound on the decrease in the function value at each iteration:

$$f(\mathbf{x}_{k+1}) - f(\mathbf{x}^*) \leq (1 - \eta L)(f(\mathbf{x}_k) - f(\mathbf{x}^*)), \tag{675}$$

where $\mathbf{x}^*$ is the global minimum. Thus, we have the following convergence rate:

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq (1 - \eta L)^k (f(\mathbf{x}_0) - f(\mathbf{x}^*)). \tag{676}$$

For this to converge, we require $\eta L < 1$. Hence, the step size η must be chosen carefully to ensure convergence.

13.3.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent is a variant of gradient descent that approximates the gradient of the objective function using a randomly chosen subset (mini-batch) of the data at each iteration. This can significantly reduce the computational cost when the dataset is large.

Let the objective function be the sum of individual functions $f_i(\mathbf{x})$ corresponding to each data point:

$$f(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^{m} f_i(\mathbf{x}), \tag{677}$$

where m is the number of data points. In **Stochastic Gradient Descent**, the update rule becomes:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f_{i_k}(\mathbf{x}_k), \tag{678}$$

where i_k is a randomly chosen index at the k-th iteration, and $\nabla f_{i_k}(\mathbf{x})$ is the gradient of the function $f_{i_k}(\mathbf{x})$ corresponding to that randomly selected data point. The stochastic gradient is given by:

$$\nabla f_{i_k}(\mathbf{x}_k) = \nabla f_i(\mathbf{x}_k). \tag{679}$$

Given that the gradient is stochastic, the convergence analysis of SGD is more complex. Assuming that each f_i is convex and differentiable, and using the strong convexity assumption (i.e., there exists a constant $m > 0$ such that f satisfies the inequality):

$$f(\mathbf{x}) - f(\mathbf{y}) \geq m\|\mathbf{x} - \mathbf{y}\|^2, \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n, \tag{680}$$

SGD converges to the optimal solution at a rate of:

$$\mathbb{E}[f(\mathbf{x}_k) - f(\mathbf{x}^*)] \leq \frac{C}{k}, \tag{681}$$

where C is a constant depending on the step size η, the variance of the stochastic gradients, and the strong convexity constant m. This slower convergence rate is due to the inherent noise in the gradient estimates. Variance reduction techniques such as **mini-batch SGD** (using multiple data points per iteration) or **Momentum** (accumulating past gradients) are often employed to improve convergence speed and stability.

13.3.3 Second-Order Methods

Second-order methods make use of not just the gradient $\nabla f(\mathbf{x})$, but also the **Hessian matrix** $\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x})$, which is the matrix of second-order partial derivatives of the objective function. The update rule for second-order methods is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \mathbf{H}^{-1}(\mathbf{x}_k) \nabla f(\mathbf{x}_k), \tag{682}$$

where $\mathbf{H}^{-1}(\mathbf{x}_k)$ is the inverse of the Hessian matrix.

Second-order methods typically have faster convergence rates compared to gradient descent, particularly when the function f has well-conditioned curvature. However, computing the Hessian is computationally expensive, which limits the scalability of these methods. Newton's method is a widely used second-order optimization technique that uses both the gradient and the Hessian. The update rule is given by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \mathbf{H}^{-1}(\mathbf{x}_k) \nabla f(\mathbf{x}_k). \tag{683}$$

Newton's method converges quadratically near the optimal point under the assumption that the objective function is twice continuously differentiable and the Hessian is positive definite. More formally, if $\mathbf{x}_k$ is sufficiently close to the optimal point $\mathbf{x}^*$, then the error $\|\mathbf{x}_k - \mathbf{x}^*\|$ decreases quadratically:

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq C \|\mathbf{x}_k - \mathbf{x}^*\|^2, \tag{684}$$

where C is a constant depending on the condition number of the Hessian.

Since directly computing the Hessian is expensive, quasi-Newton methods aim to approximate the inverse Hessian at each iteration. One of the most popular quasi-Newton methods is the **Broyden–Fletcher–Goldfarb–Shanno (BFGS)** method, which maintains an approximation to the inverse Hessian, updating it at each iteration. The Summary of what we discussed above are as follows:

- **Gradient Descent (GD):** An optimization algorithm that updates the parameter vector in the direction opposite to the gradient of the objective function. Convergence is guaranteed under convexity assumptions with an appropriately chosen step size.
- **Stochastic Gradient Descent (SGD):** A variant of GD that uses a random subset of the data to estimate the gradient at each iteration. While faster and less computationally intensive, its convergence is slower and more noisy, requiring variance reduction techniques for efficient training.
- **Second-Order Methods:** These methods use the Hessian (second derivatives of the objective function) to accelerate convergence, often exhibiting quadratic convergence near the optimum. However, the computational cost of calculating the Hessian restricts their practical use. Quasi-Newton methods, such as BFGS, approximate the Hessian to improve efficiency.

Each of these methods has its advantages and trade-offs, with gradient-based methods being widely used due to their simplicity and efficiency, and second-order methods providing faster convergence but at higher computational costs.

13.4 Matrix Calculus

13.4.1 Matrix Differentiation

Consider a matrix $\mathbf{A}$ of size $m \times n$, where $A = [a_{ij}]$. For the purposes of differentiation, we will focus on functions $f(\mathbf{A})$ that map matrices to scalars or other matrices. We aim to compute the derivative of $f(\mathbf{A})$ with respect to $\mathbf{A}$. Let $f(\mathbf{A})$ be a scalar function of the matrix $\mathbf{A}$. The derivative of this scalar function with respect to $\mathbf{A}$ is defined as:

$$\frac{\partial f(\mathbf{A})}{\partial \mathbf{A}} = \left[\frac{\partial f(\mathbf{A})}{\partial a_{ij}} \right] \tag{685}$$

This is a matrix where the (i, j)-th entry is the partial derivative of the scalar function with respect to the element a_{ij}. Let us take an example of Differentiating the Frobenius Norm. Consider the Frobenius norm of a matrix $\mathbf{A}$, defined as:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2} \tag{686}$$

To compute the derivative of $\|\mathbf{A}\|_F$ with respect to $\mathbf{A}$, we first apply the chain rule:

$$\frac{\partial \|\mathbf{A}\|_F}{\partial a_{ij}} = \frac{2a_{ij}}{2\|\mathbf{A}\|_F} = \frac{a_{ij}}{\|\mathbf{A}\|_F} \tag{687}$$

Thus, the gradient of the Frobenius norm is the matrix $\frac{\mathbf{A}}{\|\mathbf{A}\|_F}$. The Matrix Derivatives of Common Functions are as follows:

- **Matrix trace**: For a matrix $\mathbf{A}$, the derivative of the trace $\text{Tr}(\mathbf{A})$ with respect to $\mathbf{A}$ is the identity matrix:

$$\frac{\partial \text{Tr}(\mathbf{A})}{\partial \mathbf{A}} = \mathbf{I} \tag{688}$$

- **Matrix product**: Let $\mathbf{A}$ and $\mathbf{B}$ be matrices, and consider the product $f(\mathbf{A}) = \mathbf{AB}$. The derivative of this product with respect to $\mathbf{A}$ is:

$$\frac{\partial (\mathbf{AB})}{\partial \mathbf{A}} = \mathbf{B}^T \tag{689}$$

- **Matrix inverse**: The derivative of the inverse of $\mathbf{A}$ with respect to $\mathbf{A}$ is:

$$\frac{\partial (\mathbf{A}^{-1})}{\partial \mathbf{A}} = -\mathbf{A}^{-1} \left(\frac{\partial \mathbf{A}}{\partial \mathbf{A}} \right) \mathbf{A}^{-1} \tag{690}$$

13.4.2 Tensor Differentiation

A **tensor** is a multi-dimensional array of components that transform according to certain rules under a change of basis. For simplicity, let's focus on second-order tensors (which are matrices in $m \times n$ form), but the results can extend to higher-order tensors.

Let $\mathbf{T}$ be a tensor, represented by the array of components $T_{i_1, i_2, \ldots, i_k}$, where the indices $i_1, i_2, \ldots, i_k$ are the dimensions of the tensor. Let $f(\mathbf{T})$ be a scalar-valued function that depends on the tensor $\mathbf{T}$. The derivative of this function with respect to the tensor components $T_{i_1, \ldots, i_k}$ is given by:

$$\frac{\partial f(\mathbf{T})}{\partial T_{i_1, \ldots, i_k}} = \text{Jacobian of } f(\mathbf{T}) \text{ with respect to } T_{i_1, \ldots, i_k} \tag{691}$$

For example, consider a function of a second-order tensor, $f(\mathbf{T})$, where $\mathbf{T}$ is a matrix. The differentiation rule follows similar principles as matrix differentiation. The Jacobian is computed for each tensor component in the same fashion, based on the partial derivatives with respect to the individual tensor components.

Consider a second-order tensor $\mathbf{T}$, and let's compute the derivative of the Frobenius norm of $\mathbf{T}$:

$$\|\mathbf{T}\|_F = \sqrt{\sum_{i_1,i_2,\ldots,i_k} T^2_{i_1,\ldots,i_k}} \tag{692}$$

Differentiating with respect to $T_{i_1,\ldots,i_k}$, we get:

$$\frac{\partial\|\mathbf{T}\|_F}{\partial T_{i_1,\ldots,i_k}} = \frac{2T_{i_1,\ldots,i_k}}{2\|\mathbf{T}\|_F} = \frac{T_{i_1,\ldots,i_k}}{\|\mathbf{T}\|_F} \tag{693}$$

This is the gradient of the Frobenius norm, where each component $T_{i_1,\ldots,i_k}$ is normalized by the Frobenius norm. For higher-order tensors, differentiation follows the same principles but extends to multi-indexed components. If $\mathbf{T}$ is a third-order tensor, say T_{i_1,i_2,i_3}, the differentiation of $f(\mathbf{T})$ with respect to any component is given by:

$$\frac{\partial f(\mathbf{T})}{\partial T_{i_1,i_2,i_3}} = \text{Jacobian of}\, f(\mathbf{T})\, \text{with respect to the multi-index components.} \tag{694}$$

For the tensor product of two tensors $\mathbf{T}_1$ and $\mathbf{T}_2$, say of orders p and q, respectively, the product is another tensor of order $p+q$. Differentiation of the tensor product $\mathbf{T}_1 \otimes \mathbf{T}_2$ follows the product rule:

$$\frac{\partial(\mathbf{T}_1 \otimes \mathbf{T}_2)}{\partial \mathbf{T}_1} = \mathbf{T}_2, \quad \frac{\partial(\mathbf{T}_1 \otimes \mathbf{T}_2)}{\partial \mathbf{T}_2} = \mathbf{T}_1 \tag{695}$$

This tensor product rule applies for higher-order tensors, where differentiation follows tensor contraction rules. The process of differentiating matrices and tensors extends the rules of differentiation to multi-dimensional data structures, with careful application of chain rules, product rules, and understanding the Jacobian of the functions. For matrices, the derivative is a matrix of partial derivatives, while for tensors, the derivative is typically expressed as a tensor with respect to multi-index components. In higher-order tensor differentiation, we apply these principles recursively, accounting for multi-index notation, and respecting the tensor contraction rules that define how the components interact.

We start with the Differentiation of Scalar-Valued Functions with Matrix Arguments. Let $f : \mathbb{R}^{m\times n} \to \mathbb{R}$ be a scalar function of a matrix $\mathbf{X}$. The **differential** df of f is defined by:

$$df = \lim_{\|\mathbf{H}\|\to 0} \frac{f(\mathbf{X}+\mathbf{H}) - f(\mathbf{X})}{\|\mathbf{H}\|} \tag{696}$$

where $\mathbf{H}$ is an infinitesimal perturbation. The total derivative of f is given by:

$$df = \operatorname{tr}\left(\left(\frac{\partial f}{\partial \mathbf{X}}\right)^T d\mathbf{X}\right). \tag{697}$$

Definition of the Matrix Gradient: The **gradient $\mathbf{D}_\mathbf{X} f$** (or **Jacobian**) is the unique matrix satisfying:

$$df = \operatorname{tr}\left(\mathbf{D}_\mathbf{X}^T d\mathbf{X}\right). \tag{698}$$

This ensures that differentiation is **dual** to the Frobenius inner product $\langle \mathbf{A}, \mathbf{B}\rangle = \operatorname{tr}(\mathbf{A}^T\mathbf{B})$, giving a **Hilbert space structure**. Let's start with the example of Quadratic Form Differentiation. Let $f(\mathbf{X}) = \operatorname{tr}(\mathbf{X}^T\mathbf{A}\mathbf{X})$. Expanding in a small perturbation $\mathbf{H}$:

$$f(\mathbf{X}+\mathbf{H}) = \operatorname{tr}((\mathbf{X}+\mathbf{H})^T\mathbf{A}(\mathbf{X}+\mathbf{H})). \tag{699}$$

Expanding and isolating linear terms:

$$df = \text{tr}(\mathbf{H}^T\mathbf{A}\mathbf{X}) + \text{tr}(\mathbf{X}^T\mathbf{A}\mathbf{H}). \tag{700}$$

Using the cyclic property of the trace:

$$df = \text{tr}(\mathbf{H}^T(\mathbf{A}\mathbf{X} + \mathbf{A}^T\mathbf{X})). \tag{701}$$

Thus, the derivative is:

$$\frac{\partial f}{\partial \mathbf{X}} = \mathbf{A}\mathbf{X} + \mathbf{A}^T\mathbf{X}. \tag{702}$$

If $\mathbf{A}$ is symmetric ($\mathbf{A}^T = \mathbf{A}$), this simplifies to:

$$\frac{\partial f}{\partial \mathbf{X}} = 2\mathbf{A}\mathbf{X}. \tag{703}$$

Regarding the Differentiation of Matrix-Valued Functions. Consider a differentiable function $\mathbf{F} : \mathbb{R}^{m\times n} \to \mathbb{R}^{p\times q}$. The **Fréchet derivative $\mathcal{D}_{\mathbf{X}}\mathbf{F}$** is a **fourth-order tensor** satisfying:

$$d\mathbf{F} = \mathcal{D}_{\mathbf{X}}\mathbf{F} : d\mathbf{X}. \tag{704}$$

Regarding the Differentiation of the Matrix Inverse, for $\mathbf{F}(\mathbf{X}) = \mathbf{X}^{-1}$ we use the identity:

$$d(\mathbf{X}\mathbf{X}^{-1}) = 0 \Rightarrow d\mathbf{X}\mathbf{X}^{-1} + \mathbf{X}d\mathbf{X}^{-1} = 0. \tag{705}$$

Solving for $d\mathbf{X}^{-1}$:

$$d\mathbf{X}^{-1} = -\mathbf{X}^{-1}(d\mathbf{X})\mathbf{X}^{-1}. \tag{706}$$

Thus, the derivative is the **negative bilinear operator**:

$$\mathcal{D}_{\mathbf{X}}(\mathbf{X}^{-1}) = -(\mathbf{X}^{-1} \otimes \mathbf{X}^{-1}). \tag{707}$$

where $\otimes$ denotes the Kronecker product. For Differentiation of Tensor-Valued Functions. We need to have a differentiable tensor function $\mathcal{F} : \mathbb{R}^{m\times n\times p} \to \mathbb{R}^{a\times b\times c}$, the **Fréchet derivative** shall be a **higher-order tensor $\mathcal{D}_{\mathcal{X}}\mathcal{F}$** satisfying:

$$d\mathcal{F} = \mathcal{D}_{\mathcal{X}}\mathcal{F} : d\mathcal{X}. \tag{708}$$

Let's do a Differentiation of Tensor Contraction. If $f(\mathcal{X}) = \mathcal{X} : \mathcal{A}$, where $\mathcal{X}, \mathcal{A}$ are second-order tensors, then:

$$\frac{\partial}{\partial \mathcal{X}}(\mathcal{X} : \mathcal{A}) = \mathcal{A}. \tag{709}$$

For a fourth-order tensor $\mathcal{C}$, if $f(\mathcal{X}) = \mathcal{C} : \mathcal{X}$, then:

$$\frac{\partial}{\partial \mathcal{X}}(\mathcal{C} : \mathcal{X}) = \mathcal{C}. \tag{710}$$

Differentiation can be also done in Non-Euclidean Spaces. For a manifold $\mathcal{M}$, differentiation is defined via **tangent spaces $T_{\mathbf{X}}\mathcal{M}$**, with the **covariant derivative $\nabla_{\mathbf{X}}$** satisfying the **Levi-Civita connection**:

$$\nabla_{\mathbf{X}}\mathbf{Y} = \lim_{\epsilon\to 0} \frac{\text{Proj}_{T_{\mathbf{X}+\epsilon\mathbf{H}}\mathcal{M}}(\mathbf{Y}(\mathbf{X}+\epsilon\mathbf{H})) - \mathbf{Y}(\mathbf{X})}{\epsilon}. \tag{711}$$

We can do differentiation using Variational Principles also. If $f(\mathbf{X})$ is an energy functional, the differentiation that follows from **Gateaux derivatives** is:

$$\delta f = \lim_{\epsilon\to 0} \frac{f(\mathbf{X}+\epsilon\mathbf{H}) - f(\mathbf{X})}{\epsilon}. \tag{712}$$

For **functionals**, differentiation uses **Euler-Lagrange equations**:

$$\frac{d}{dt}\int_{\Omega} L(\mathbf{X}, \nabla\mathbf{X})\, dV = 0. \tag{713}$$

14 Acknowledgments

The authors acknowledge the contributions of researchers whose foundational work has shaped our understanding of Deep Learning.

References

[1] Rao, N., Farid, M., and Raiz, M. (2024). Symmetric Properties of λ-Szász Operators Coupled with Generalized Beta Functions and Approximation Theory. Symmetry, 16(12), 1703.

[2] Mukhopadhyay, S.N., Ray, S. (2025). Function Spaces. In: Measure and Integration. University Texts in the Mathematical Sciences. Springer, Singapore.

[3] Szołdra, T. (2024). Ergodicity breaking in quantum systems: from exact time evolution to machine learning (Doctoral dissertation).

[4] SONG, W. X., CHEN, H., CUI, C., LIU, Y. F., TONG, D., GUO, F., ... and XIAO, C. W. (2025). Theoretical, methodological, and implementation considerations for establishing a sustainable urban renewal model. JOURNAL OF NATURAL RESOURCES, 40(1), 20-38.

[5] El Mennaoui, O., Kharou, Y., and Laasri, H. (2025). Evolution families in the framework of maximal regularity. Evolution Equations and Control Theory, 0-0.

[6] Pedroza, G. (2024). On the Conditions for Domain Stability for Machine Learning: a Mathematical Approach. arXiv preprint arXiv:2412.00464.

[7] Cerreia-Vioglio, S., and Ok, E. A. (2024). Abstract integration of set-valued functions. Journal of Mathematical Analysis and Applications, 129169.

[8] Averin, A. (2024). Formulation and Proof of the Gravitational Entropy Bound. arXiv preprint arXiv:2412.02470.

[9] Potter, T. (2025). Subspaces of $L^2(\mathbb{R}^n)$ Invariant Under Crystallographic Shifts. arXiv e-prints, arXiv-2501.

[10] Lee, M. (2025). Emergence of Self-Identity in Artificial Intelligence: A Mathematical Framework and Empirical Study with Generative Large Language Models. Axioms, 14(1), 44.

[11] Wang, R., Cai, L., Wu, Q., and Niyato, D. (2025). Service Function Chain Deployment with Intrinsic Dynamic Defense Capability. IEEE Transactions on Mobile Computing.

[12] Duim, J. L., and Mesquita, D. P. (2025). Artificial Intelligence Value Alignment via Inverse Reinforcement Learning. Proceeding Series of the Brazilian Society of Computational and Applied Mathematics, 11(1), 1-2.

[13] Khayat, M., Barka, E., Serhani, M. A., Sallabi, F., Shuaib, K., and Khater, H. M. (2025). Empowering Security Operation Center with Artificial Intelligence and Machine Learning–A Systematic Literature Review. IEEE Access.

[14] Agrawal, R. (2025). 46 Detection of melanoma using DenseNet-based adaptive weighted loss function. Emerging Trends in Computer Science and Its Application, 283.

[15] Hailemichael, H., and Ayalew, B. Adaptive and Safe Fast Charging of Lithium-Ion Batteries Via Hybrid Model Learning and Control Barrier Functions. Available at SSRN 5110597.

[16] Nguyen, E., Xiao, J., Fan, Z., and Ruan, D. Contrast-free Full Intracranial Vessel Geometry Estimation from MRI with Metric Learning based Inference. In Medical Imaging with Deep Learning.

[17] Luo, Z., Bi, Y., Yang, X., Li, Y., Wang, S., and Ye, Q. A Novel Machine Vision-Based Collision Risk Warning Method for Unsignalized Intersections on Arterial Roads. Frontiers in Physics, 13, 1527956.

[18] Bousquet, N., Thomassé, S. (2015). VC-dimension and Erdős–Pósa property. Discrete Mathematics, 338(12), 2302-2317.

[19] Asian, O., Yildiz, O. T., Alpaydin, E. (2009, September). Calculating the VC-dimension of decision trees. In 2009 24th International Symposium on Computer and Information Sciences (pp. 193-198). IEEE.

[20] Zhang, C., Bian, W., Tao, D., Lin, W. (2012). Discretized-Vapnik-Chervonenkis dimension for analyzing complexity of real function classes. IEEE transactions on neural networks and learning systems, 23(9), 1461-1472.

[21] Riondato, M., Akdere, M., Çetintemel, U., Zdonik, S. B., Upfal, E. (2011). The VC-dimension of SQL queries and selectivity estimation through sampling. In Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part II 22 (pp. 661-676). Springer Berlin Heidelberg.

[22] Bane, M., Riggle, J., Sonderegger, M. (2010). The VC dimension of constraint-based grammars. Lingua, 120(5), 1194-1208.

[23] Anderson, A. (2023). Fuzzy VC Combinatorics and Distality in Continuous Logic. arXiv preprint arXiv:2310.04393.

[24] Fox, J., Pach, J., Suk, A. (2021). Bounded VC-dimension implies the Schur-Erdős conjecture. Combinatorica, 41(6), 803-813.

[25] Johnson, H. R. (2021). Binary strings of finite VC dimension. arXiv preprint arXiv:2101.06490.

[26] Janzing, D. (2018). Merging joint distributions via causal model classes with low VC dimension. arXiv preprint arXiv:1804.03206.

[27] Hüllermeier, E., Fallah Tehrani, A. (2012, July). On the vc-dimension of the choquet integral. In International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (pp. 42-50). Berlin, Heidelberg: Springer Berlin Heidelberg.

[28] Mohri, M. (2018). Foundations of machine learning.

[29] Cucker, F., Zhou, D. X. (2007). Learning theory: an approximation theory viewpoint (Vol. 24). Cambridge University Press.

[30] Shalev-Shwartz, S., Ben-David, S. (2014). Understanding machine learning: From theory to algorithms. Cambridge university press.

[31] Truong, L. V. (2022). On rademacher complexity-based generalization bounds for deep learning. arXiv preprint arXiv:2208.04284.

[32] Gnecco, G., and Sanguineti, M. (2008). Approximation error bounds via Rademacher complexity. Applied Mathematical Sciences, 2, 153-176.

[33] Astashkin, S. V. (2010). Rademacher functions in symmetric spaces. Journal of Mathematical Sciences, 169(6), 725-886.

[34] Ying and Campbell (2010). Rademacher chaos complexities for learning the kernel problem. Neural computation, 22(11), 2858-2886.

[35] Zhu, J., Gibson, B., and Rogers, T. T. (2009). Human rademacher complexity. Advances in neural information processing systems, 22.

[36] Astashkin, S. V., Astashkin, S. V., and Mazlum. (2020). The Rademacher system in function spaces. Basel: Birkhäuser.

[37] Sachs, S., van Erven, T., Hodgkinson, L., Khanna, R., and Şimşekli, U. (2023, July). Generalization Guarantees via Algorithm-dependent Rademacher Complexity. In The Thirty Sixth Annual Conference on Learning Theory (pp. 4863-4880). PMLR.

[38] Ma and Wang (2020). Rademacher complexity and the generalization error of residual networks. Communications in Mathematical Sciences, 18(6), 1755-1774.

[39] Bartlett, P. L., and Mendelson, S. (2002). Rademacher and Gaussian complexities: Risk bounds and structural results. Journal of Machine Learning Research, 3(Nov), 463-482.

[40] Bartlett, P. L., and Mendelson, S. (2002). Rademacher and Gaussian complexities: Risk bounds and structural results. Journal of Machine Learning Research, 3(Nov), 463-482.

[41] McDonald, D. J., and Shalizi, C. R. (2011). Rademacher complexity of stationary sequences. arXiv preprint arXiv:1106.0730.

[42] Abderachid, S., and Kenza, B. EMBEDDINGS IN RIEMANN–LIOUVILLE FRACTIONAL SOBOLEV SPACES AND APPLICATIONS.

[43] Giang, T. H., Tri, N. M., and Tuan, D. A. (2024). On some Sobolev and Pólya-Sezgö type inequalities with weights and applications. arXiv preprint arXiv:2412.15490.

[44] Ruiz, P. A., and Fragkiadaki, V. (2024). Fractional Sobolev embeddings and algebra property: A dyadic view. arXiv preprint arXiv:2412.12051.

[45] Bilalov, B., Mamedov, E., Sezer, Y., and Nasibova, N. (2025). Compactness in Banach function spaces: Poincaré and Friedrichs inequalities. Rendiconti del Circolo Matematico di Palermo Series 2, 74(1), 68.

[46] Cheng, M., and Shao, K. (2025). Ground states of the inhomogeneous nonlinear fractional Schrödinger-Poisson equations. Complex Variables and Elliptic Equations, 1-17.

[47] Wei, J., and Zhang, L. (2025). Ground State Solutions of Nehari-Pohozaev Type for Schrödinger-Poisson Equation with Zero-Mass and Weighted Hardy Sobolev Subcritical Exponent. The Journal of Geometric Analysis, 35(2), 48.

[48] Zhang, X., and Qi, W. (2025). Multiplicity result on a class of nonhomogeneous quasilinear elliptic system with small perturbations in $\mathbb{R}^N$. arXiv preprint arXiv:2501.01602.

[49] Xiao, J., and Yue, C. (2025). A Trace Principle for Fractional Laplacian with an Application to Image Processing. La Matematica, 1-26.

[50] Pesce, A., and Portaro, S. (2025). Fractional Sobolev spaces related to an ultraparabolic operator. arXiv preprint arXiv:2501.05898.

[51] LASSOUED, D. (2026). A STUDY OF FUNCTIONS ON THE TORUS AND MULTIPERIODIC FUNCTIONS. Kragujevac Journal of Mathematics, 50(2), 297-337.

[52] Chen, H., Chen, H. G., and Li, J. N. (2024). Sharp embedding results and geometric inequalities for Hö rmander vector fields. arXiv preprint arXiv:2404.19393.

[53] Adams, R. A., and Fournier, J. J. (2003). Sobolev spaces. Elsevier.

[54] Brezis, H., and Brézis, H. (2011). Functional analysis, Sobolev spaces and partial differential equations (Vol. 2, No. 3, p. 5). New York: Springer.

[55] Evans, L. C. (2022). Partial differential equations (Vol. 19). American Mathematical Society.

[56] Maz'â, V. G. (2011). Sobolev Spaces: With Applications to Elliptic Partial Differential Equations. Springer.

[57] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, 2(5), 359-366.

[58] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems, 2(4), 303-314.

[59] Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. IEEE Transactions on Information theory, 39(3), 930-945.

[60] Pinkus, A. (1999). Approximation theory of the MLP model in neural networks. Acta numerica, 8, 143-195.

[61] Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. (2017). The expressive power of neural networks: A view from the width. Advances in neural information processing systems, 30.

[62] Hanin, B., and Sellke, M. (2017). Approximating continuous functions by relu nets of minimal width. arXiv preprint arXiv:1710.11278.

[63] Garcıa-Cervera, C. J., Kessler, M., Pedregal, P., and Periago, F. Universal approximation of set-valued maps and DeepONet approximation of the controllability map.

[64] Majee, S., Abhishek, A., Strauss, T., and Khan, T. (2024). MCMC-Net: Accelerating Markov Chain Monte Carlo with Neural Networks for Inverse Problems. arXiv preprint arXiv:2412.16883.

[65] Toscano, J. D., Wang, L. L., and Karniadakis, G. E. (2024). KKANs: Kurkova-Kolmogorov-Arnold Networks and Their Learning Dynamics. arXiv preprint arXiv:2412.16738.

[66] Son, H. (2025). ELM-DeepONets: Backpropagation-Free Training of Deep Operator Networks via Extreme Learning Machines. arXiv preprint arXiv:2501.09395.

[67] Rudin, W. (1964). Principles of mathematical analysis (Vol. 3). New York: McGraw-hill.

[68] Stein, E. M., and Shakarchi, R. (2009). Real analysis: measure theory, integration, and Hilbert spaces. Princeton University Press.

[69] Conway, J. B. (2019). A course in functional analysis (Vol. 96). Springer.

[70] Dieudonné, J. (2020). History of Functional Analyais. In Functional Analysis, Holomorphy, and Approximation Theory (pp. 119-129). CRC Press.

[71] Folland, G. B. (1999). Real analysis: modern techniques and their applications (Vol. 40). John Wiley and Sons.

[72] Sugiura, S. (2024). On the Universality of Reservoir Computing for Uniform Approximation.

[73] LIU, Y., LIU, S., HUANG, Z., and ZHOU, P. NORMED MODULES AND THE CATEGORIFICATION OF INTEGRATIONS, SERIES EXPANSIONS, AND DIFFERENTIATIONS.

[74] Barreto, D. M. (2025). Stone-Weierstrass Theorem.

[75] Chang, S. Y., and Wei, Y. (2024). Generalized Choi–Davis–Jensen's Operator Inequalities and Their Applications. Symmetry, 16(9), 1176.

[76] Caballer, M., Dantas, S., and Rodríguez-Vidanes, D. L. (2024). Searching for linear structures in the failure of the Stone-Weierstrass theorem. arXiv preprint arXiv:2405.06453.

[77] Chen, D. (2024). The Machado–Bishop theorem in the uniform topology. Journal of Approximation Theory, 304, 106085.

[78] Rafiei, H., and Akbarzadeh-T, M. R. (2024). Hedge-embedded Linguistic Fuzzy Neural Networks for Systems Identification and Control. IEEE Transactions on Artificial Intelligence.

[79] Kolmogorov, A. N. (1957). On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. In Doklady Akademii Nauk (Vol. 114, No. 5, pp. 953-956). Russian Academy of Sciences.

[80] Arnold, V. I. (2009). On the representation of functions of several variables as a superposition of functions of a smaller number of variables. Collected works: Representations of functions, celestial mechanics and KAM theory, 1957–1965, 25-46.

[81] Lorentz, G. G. (1966). Approximation of functions, athena series. Selected Topics in Mathematics.

[82] Guilhoto, L. F., and Perdikaris, P. (2024). Deep learning alternatives of the Kolmogorov superposition theorem. arXiv preprint arXiv:2410.01990.

[83] Alhafiz, M. R., Zakaria, K., Dung, D. V., Palar, P. S., Dwianto, Y. B., and Zuhal, L. R. (2025). Kolmogorov-Arnold Networks for Data-Driven Turbulence Modeling. In AIAA SCITECH 2025 Forum (p. 2047).

[84] Lorencin, I., Mrzljak, V., Poljak, I., and Etinger, D. (2024, September). Prediction of CODLAG Propulsion System Parameters Using Kolmogorov-Arnold Network. In 2024 IEEE 22nd Jubilee International Symposium on Intelligent Systems and Informatics (SISY) (pp. 173-178). IEEE.

[85] Trevisan, D., Cassara, P., Agazzi, A., and Scardera, S. NTK Analysis of Knowledge Distillation.

[86] Bonfanti, A., Bruno, G., and Cipriani, C. (2024). The Challenges of the Nonlinear Regime for Physics-Informed Neural Networks. arXiv preprint arXiv:2402.03864.

[87] Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. Advances in neural information processing systems, 31.

[88] Lee, J., Xiao, L., Schoenholz, S., Bahri, Y., Novak, R., Sohl-Dickstein, J., and Pennington, J. (2019). Wide neural networks of any depth evolve as linear models under gradient descent. Advances in neural information processing systems, 32.

[89] Yang, G., and Hu, E. J. (2020). Feature learning in infinite-width neural networks. arXiv preprint arXiv:2011.14522.

[90] Xiang, L., Dudziak, Ł., Abdelfattah, M. S., Chau, T., Lane, N. D., and Wen, H. (2021). Zero-Cost Operation Scoring in Differentiable Architecture Search. arXiv preprint arXiv:2106.06799.

[91] Lee, J., Xiao, L., Schoenholz, S., Bahri, Y., Novak, R., Sohl-Dickstein, J., and Pennington, J. (2019). Wide neural networks of any depth evolve as linear models under gradient descent. Advances in neural information processing systems, 32.

[92] McAllester, D. A. (1999, July). PAC-Bayesian model averaging. In Proceedings of the twelfth annual conference on Computational learning theory (pp. 164-170).

[93] Catoni, O. (2007). PAC-Bayesian supervised classification: the thermodynamics of statistical learning. arXiv preprint arXiv:0712.0248.

[94] Germain, P., Lacasse, A., Laviolette, F., and Marchand, M. (2009, June). PAC-Bayesian learning of linear classifiers. In Proceedings of the 26th Annual International Conference on Machine Learning (pp. 353-360).

[95] Seeger, M. (2002). PAC-Bayesian generalisation error bounds for Gaussian process classification. Journal of machine learning research, 3(Oct), 233-269.

[96] Alquier, P., Ridgway, J., and Chopin, N. (2016). On the properties of variational approximations of Gibbs posteriors. Journal of Machine Learning Research, 17(236), 1-41.

[97] Dziugaite, G. K., and Roy, D. M. (2017). Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. arXiv preprint arXiv:1703.11008.

[98] Rivasplata, O., Kuzborskij, I., Szepesvári, C., and Shawe-Taylor, J. (2020). PAC-Bayes analysis beyond the usual bounds. Advances in Neural Information Processing Systems, 33, 16833-16845.

[99] Lever, G., Laviolette, F., and Shawe-Taylor, J. (2013). Tighter PAC-Bayes bounds through distribution-dependent priors. Theoretical Computer Science, 473, 4-28.

[100] Rivasplata, O., Parrado-Hernández, E., Shawe-Taylor, J. S., Sun, S., and Szepesvári, C. (2018). PAC-Bayes bounds for stable algorithms with instance-dependent priors. Advances in Neural Information Processing Systems, 31.

[101] Lindemann, L., Zhao, Y., Yu, X., Pappas, G. J., and Deshmukh, J. V. (2024). Formal verification and control with conformal prediction. arXiv preprint arXiv:2409.00536.

[102] Jin, G., Wu, S., Liu, J., Huang, T., and Mu, R. (2025). Enhancing Robust Fairness via Confusional Spectral Regularization. arXiv preprint arXiv:2501.13273.

[103] Ye, F., Xiao, J., Ma, W., Jin, S., and Yang, Y. (2025). Detecting small clusters in the stochastic block model. Statistical Papers, 66(2), 37.

[104] Bhattacharjee, A., and Bharadwaj, P. (2025). Coherent Spectral Feature Extraction Using Symmetric Autoencoders. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.

[105] Wu, Q., Hu, B., Liu, C. et al. (2025). Velocity Analysis Using High-resolution Hyperbolic Radon Transform with $L_{q_1} - L_{q_2}$ Regularization. Pure Appl. Geophys.

[106] Ortega, I., Hannigan, J. W., Baier, B. C., McKain, K., and Smale, D. (2025). Advancing CH 4 and N 2 O retrieval strategies for NDACC/IRWG high-resolution direct-sun FTIR Observations. EGUsphere, 2025, 1-32.

[107] Kazmi, S. H. A., Hassan, R., Qamar, F., Nisar, K., and Al-Betar, M. A. (2025). Federated Conditional Variational Auto Encoders for Cyber Threat Intelligence: Tackling Non-IID Data in SDN Environments. IEEE Access.

[108] Zhao, Y., Bi, Z., Zhu, P., Yuan, A., and Li, X. (2025). Deep Spectral Clustering with Projected Adaptive Feature Selection. IEEE Transactions on Geoscience and Remote Sensing.

[109] Saranya, S., and Menaka, R. (2025). A Quantum-Based Machine Learning Approach for Autism Detection using Common Spatial Patterns of EEG Signals. IEEE Access.

[110] Dhalbisoi, S., Mohapatra, A., and Rout, A. (2024, March). Design of Cell-Free Massive MIMO for Beyond 5G Systems with MMSE and RZF Processing. In International Conference on Machine Learning, IoT and Big Data (pp. 263-273). Singapore: Springer Nature Singapore.

[111] Wei, C., Li, Z., Hu, T., Zhao, M., Sun, Z., Jia, K., ... and Jiang, S. (2025). Model-based convolution neural network for 3D Near-infrared spectral tomography. IEEE Transactions on Medical Imaging.

[112] Goodfellow, I. (2016). Deep learning (Vol. 196). MIT press.

[113] Haykin, S. (2009). Neural networks and learning machines, 3/E. Pearson Education India.

[114] Schmidhuber, J. (2015). Deep learning in neural networks: An overview.

[115] Bishop, C. M., and Nasrabadi, N. M. (2006). Pattern recognition and machine learning (Vol. 4, No. 4, p. 738). New York: springer.

[116] Poggio, T., and Smale, S. (2003). The mathematics of learning: Dealing with data. Notices of the AMS, 50(5), 537-544.

[117] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. nature, 521(7553), 436-444.

[118] Tishby, N., and Zaslavsky, N. (2015, April). Deep learning and the information bottleneck principle. In 2015 ieee information theory workshop (itw) (pp. 1-5). IEEE.

[119] Sorrenson, P. (2025). Free-Form Flows: Generative Models for Scientific Applications (Doctoral dissertation).

[120] Liu, W., and Shi, X. (2025). An Enhanced Neural Network Forecasting System for the July Precipitation over the Middle-Lower Reaches of the Yangtze River.

[121] Das, P., Mondal, D., Islam, M. A., Al Mohotadi, M. A., and Roy, P. C. (2025). Analytical Finite-Integral-Transform and Gradient-Enhanced Machine Learning Approach for Thermoelastic Analysis of FGM Spherical Structures with Arbitrary Properties. Theoretical and Applied Mechanics Letters, 100576.

[122] Zhang, R. (2025). Physics-informed Parallel Neural Networks for the Identification of Continuous Structural Systems.

[123] Ali, S., and Hussain, A. (2025). A neuro-intelligent heuristic approach for performance prediction of triangular fuzzy flow system. Proceedings of the Institution of Mechanical Engineers, Part N: Journal of Nanomaterials, Nanoengineering and Nanosystems, 23977914241310569.

[124] Li, S. (2025). Scalable, generalizable, and offline methods for imperfect-information extensive-form games.

[125] Hu, T., Jin, B., and Wang, F. (2025). An Iterative Deep Ritz Method for Monotone Elliptic Problems. Journal of Computational Physics, 113791.

[126] Chen, P., Zhang, A., Zhang, S., Dong, T., Zeng, X., Chen, S., ... and Zhou, Q. (2025). Maritime near-miss prediction framework and model interpretation analysis method based on Transformer neural network model with multi-task classification variables. Reliability Engineering and System Safety, 110845.

[127] Sun, G., Liu, Z., Gan, L., Su, H., Li, T., Zhao, W., and Sun, B. (2025). SpikeNAS-Bench: Benchmarking NAS Algorithms for Spiking Neural Network Architecture. IEEE Transactions on Artificial Intelligence.

[128] Zhang, Z., Wang, X., Shen, J., Zhang, M., Yang, S., Zhao, W., ... and Wang, J. (2025). Unfixed Bias Iterator: A New Iterative Format. IEEE Access.

[129] Rosa, G. J. (2010). The Elements of Statistical Learning: Data Mining, Inference, and Prediction by HASTIE, T., TIBSHIRANI, R., and FRIEDMAN, J.

[130] Murphy, K. P. (2012). Machine learning: a probabilistic perspective. MIT press.

[131] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

[132] Zou, H., and Hastie, T. (2005). Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society Series B: Statistical Methodology, 67(2), 301-320.

[133] Vapnik, V. (2013). The nature of statistical learning theory. Springer science and business media.

[134] Ng, A. Y. (2004, July). Feature selection, L 1 vs. L 2 regularization, and rotational invariance. In Proceedings of the twenty-first international conference on Machine learning (p. 78).

[135] Li, T. (2025). Optimization of Clinical Trial Strategies for Anti-HER2 Drugs Based on Bayesian Optimization and Deep Learning.

[136] Yasuda, M., and Sekimoto, K. (2024). Gaussian-discrete restricted Boltzmann machine with sparse-regularized hidden layer. Behaviormetrika, 1-19.

[137] Xiaodong Luo, William C. Cruz, Xin-Lei Zhang, Heng Xiao, (2023), Hyper-parameter optimization for improving the performance of localization in an iterative ensemble smoother, Geoenergy Science and Engineering, Volume 231, Part B, 212404

[138] Alrayes, F.S., Maray, M., Alshuhail, A. et al. (2025) Privacy-preserving approach for IoT networks using statistical learning with optimization algorithm on high-dimensional big data environment. Sci Rep 15, 3338. https://doi.org/10.1038/s41598-025-87454-1

[139] Cho, H., Kim, Y., Lee, E., Choi, D., Lee, Y., and Rhee, W. (2020). Basic enhancement strategies when using Bayesian optimization for hyperparameter tuning of deep neural networks. IEEE access, 8, 52588-52608.

[140] IBRAHIM, M. M. W. (2025). Optimizing Tuberculosis Treatment Predictions: A Comparative Study of XGBoost with Hyperparameter in Penang, Malaysia. Sains Malaysiana, 54(1), 3741-3752.

[141] Abdel-salam, M., Elhoseny, M. and El-hasnony, I.M. Intelligent and Secure Evolved Framework for Vaccine Supply Chain Management Using Machine Learning and Blockchain. SN COMPUT. SCI. 6, 121 (2025). https://doi.org/10.1007/s42979-024-03609-3

[142] Vali, M. H. (2025). Vector quantization in deep neural networks for speech and image processing.

[143] Vincent, A.M., Jidesh, P. An improved hyperparameter optimization framework for AutoML systems using evolutionary algorithms. Sci Rep 13, 4737 (2023). https://doi.org/10.1038/s41598-023-32027-3

[144] Razavi-Termeh, S. V., Sadeghi-Niaraki, A., Ali, F., and Choi, S. M. (2025). Improving flood-prone areas mapping using geospatial artificial intelligence (GeoAI): A non-parametric algorithm enhanced by math-based metaheuristic algorithms. Journal of Environmental Management, 375, 124238.

[145] Kiran, M., and Ozyildirim, M. (2022). Hyperparameter tuning for deep reinforcement learning applications. arXiv preprint arXiv:2201.11182.

[146] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25.

[147] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90.

[148] Simonyan, K., and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[149] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[150] Cohen, T., and Welling, M. (2016, June). Group equivariant convolutional networks. In International conference on machine learning (pp. 2990-2999). PMLR.

[151] Zeiler, M. D., and Fergus, R. (2014). Visualizing and understanding convolutional networks. In Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13 (pp. 818-833). Springer International Publishing.

[152] Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ... and Guo, B. (2021). Swin transformer: Hierarchical vision transformer using shifted windows. In Proceedings of the IEEE/CVF international conference on computer vision (pp. 10012-10022).

[153] Lin, M. (2013). Network in network. arXiv preprint arXiv:1312.4400.

[154] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. nature, 323(6088), 533-536.

[155] Bensaid, B., Poëtte, G., and Turpault, R. (2024). Convergence of the Iterates for Momentum and RMSProp for Local Smooth Functions: Adaptation is the Key. arXiv preprint arXiv:2407.15471.

[156] Liu, Q., and Ma, W. (2024). The Epochal Sawtooth Effect: Unveiling Training Loss Oscillations in Adam and Other Optimizers. arXiv preprint arXiv:2410.10056.

[157] Li, H. (2024). Smoothness and Adaptivity in Nonlinear Optimization for Machine Learning Applications (Doctoral dissertation, Massachusetts Institute of Technology).

[158] Heredia, C. (2024). Modeling AdaGrad, RMSProp, and Adam with Integro-Differential Equations. arXiv preprint arXiv:2411.09734.

[159] Ye, Q. (2024). Preconditioning for Accelerated Gradient Descent Optimization and Regularization. arXiv preprint arXiv:2410.00232.

[160] Compagnoni, E. M., Liu, T., Islamov, R., Proske, F. N., Orvieto, A., and Lucchi, A. (2024). Adaptive Methods through the Lens of SDEs: Theoretical Insights on the Role of Noise. arXiv preprint arXiv:2411.15958.

[161] Yao, B., Zhang, Q., Feng, R., and Wang, X. (2024). System response curve based first-order optimization algorithms for cyber-physical-social intelligence. Concurrency and Computation: Practice and Experience, 36(21), e8197.

[162] Wen, X., and Lei, Y. (2024, June). A Fast ADMM Framework for Training Deep Neural Networks Without Gradients. In 2024 International Joint Conference on Neural Networks (IJCNN) (pp. 1-8). IEEE.

[163] Hannibal, S., Jentzen, A., and Thang, D. M. (2024). Non-convergence to global minimizers in data driven supervised deep learning: Adam and stochastic gradient descent optimization provably fail to converge to global minimizers in the training of deep neural networks with ReLU activation. arXiv preprint arXiv:2410.10533.

[164] Yang, Z. (2025). Adaptive Biased Stochastic Optimization. IEEE Transactions on Pattern Analysis and Machine Intelligence.

[165] Kingma, D. P., and Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[166] Reddi, S. J., Kale, S., and Kumar, S. (2019). On the convergence of adam and beyond. arXiv preprint arXiv:1904.09237.

[167] Jin, L., Nong, H., Chen, L., and Su, Z. (2024). A Method for Enhancing Generalization of Adam by Multiple Integrations. arXiv preprint arXiv:2412.12473.

[168] Adly, A. M. (2024). EXAdam: The Power of Adaptive Cross-Moments. arXiv preprint arXiv:2412.20302.

[169] Liu, Y., Cao, Y., and Lin, J. Convergence Analysis of the ADAM Algorithm for Linear Inverse Problems.

[170] Yang, Z. (2025). Adaptive Biased Stochastic Optimization. IEEE Transactions on Pattern Analysis and Machine Intelligence.

[171] Park, K., and Lee, S. (2024). SMMF: Square-Matricized Momentum Factorization for Memory-Efficient Optimization. arXiv preprint arXiv:2412.08894.

[172] Mahjoubi, M. A., Lamrani, D., Saleh, S., Moutaouakil, W., Ouhmida, A., Hamida, S., ... and Raihani, A. (2025). Optimizing ResNet50 Performance Using Stochastic Gradient Descent on MRI Images for Alzheimer's Disease Classification. Intelligence-Based Medicine, 100219.

[173] Seini, A. B., and Adam, I. O. (2024). HUMAN-AI COLLABORATION FOR ADAPTIVE WORKING AND LEARNING OUTCOMES: AN ACTIVITY THEORY PERSPECTIVE.

[174] Teessar, J. (2024). The Complexities of Truthful Responding in Questionnaire-Based Research: A Comprehensive Analysis.

[175] Lauand, C. K., and Meyn, S. (2025). Markovian Foundations for Quasi-Stochastic Approximation. SIAM Journal on Control and Optimization, 63(1), 402-430.

[176] Maranjyan, A., Tyurin, A., and Richtárik, P. (2025). Ringmaster ASGD: The First Asynchronous SGD with Optimal Time Complexity. arXiv preprint arXiv:2501.16168.

[177] Gao, Z., and Gündüz, D. (2025). Graph Neural Networks over the Air for Decentralized Tasks in Wireless Networks. IEEE Transactions on Signal Processing.

[178] Yoon, T., Choudhury, S., and Loizou, N. (2025). Multiplayer Federated Learning: Reaching Equilibrium with Less Communication. arXiv preprint arXiv:2501.08263.

[179] Verma, K., and Maiti, A. (2025). Sine and cosine based learning rate for gradient descent method. Applied Intelligence, 55(5), 352.

[180] Borowski, M., and Miasojedow, B. (2025). Convergence of projected stochastic approximation algorithm. arXiv e-prints, arXiv-2501.

[181] Dong, K., Chen, S., Dan, Y., Zhang, L., Li, X., Liang, W., ... and Sun, Y. (2025). A new perspective on brain stimulation interventions: Optimal stochastic tracking control of brain network dynamics. arXiv preprint arXiv:2501.08567.

[182] Jiang, Y., Kang, H., Liu, J., and Xu, D. (2025). On the Convergence of Decentralized Stochastic Gradient Descent with Biased Gradients. IEEE Transactions on Signal Processing.

[183] Sonobe, N., Momozaki, T., and Nakagawa, T. (2025). Sampling from Density power divergence-based Generalized posterior distribution via Stochastic optimization. arXiv preprint arXiv:2501.07790.

[184] Zhang, X., and Jia, G. (2025). Convergence of Policy Gradient for Stochastic Linear Quadratic Optimal Control Problems in Infinite Horizon. Journal of Mathematical Analysis and Applications, 129264.

[185] Thiriveedhi, A., Ghanta, S., Biswas, S., and Pradhan, A. K. (2025). ALL-Net: integrating CNN and explainable-AI for enhanced diagnosis and interpretation of acute lymphoblastic leukemia. PeerJ Computer Science, 11, e2600.

[186] Ramos-Briceño, D. A., Flammia-D'Aleo, A., Fernández-López, G., Carrión-Nessi, F. S., and Forero-Peña, D. A. (2025). Deep learning-based malaria parasite detection: convolutional neural networks model for accurate species identification of Plasmodium falciparum and Plasmodium vivax. Scientific Reports, 15(1), 3746.

[187] Espino-Salinas, C. H., Luna-García, H., Cepeda-Argüelles, A., Trejo-Vázquez, K., Flores-Chaires, L. A., Mercado Reyna, J., ... and Villalba-Condori, K. O. (2025). Convolutional Neural Network for Depression and Schizophrenia Detection. Diagnostics, 15(3), 319.

[188] Ran, T., Huang, W., Qin, X., Xie, X., Deng, Y., Pan, Y., ... and Zou, D. (2025). Liquid-based cytological diagnosis of pancreatic neuroendocrine tumors using hyperspectral imaging and deep learning. EngMedicine, 2(1), 100059.

[189] Araujo, B. V. S., Rodrigues, G. A., de Oliveira, J. H. P., Xavier, G. V. R., Lebre, U., Cordeiro, C., ... and Ferreira, T. V. (2025). Monitoring ZnO surge arresters using convolutional neural networks and image processing techniques combined with signal alignment. Measurement, 116889.

[190] Sari, I. P., Elvitaria, L., and Rudiansyah, R. (2025). Data-driven approach for batik pattern classification using convolutional neural network (CNN). Jurnal Mandiri IT, 13(3), 323-331.

[191] Wang, D., An, K., Mo, Y., Zhang, H., Guo, W., and Wang, B. Cf-Wiad: Consistency Fusion with Weighted Instance and Adaptive Distribution for Enhanced Semi-Supervised Skin Lesion Classification. Available at SSRN 5109182.

[192] Cai, P., Zhang, Y., He, H., Lei, Z., and Gao, S. (2025). DFNet: A Differential Feature-Incorporated Residual Network for Image Recognition. Journal of Bionic Engineering, 1-14.

[193] Vishwakarma, A. K., and Deshmukh, M. (2025). CNNM-FDI: Novel Convolutional Neural Network Model for Fire Detection in Images. IETE Journal of Research, 1-14.

[194] Ranjan, P., Kaushal, A., Girdhar, A., and Kumar, R. (2025). Revolutionizing hyperspectral image classification for limited labeled data: unifying autoencoder-enhanced GANs with convolutional neural networks and zero-shot learning. Earth Science Informatics, 18(2), 1-26.

[195] Naseer, A., and Jalal, A. Multimodal Deep Learning Framework for Enhanced Semantic Scene Classification Using RGB-D Images.

[196] Wang, Z., and Wang, J. (2025). Personalized Icon Design Model Based on Improved Faster-RCNN. Systems and Soft Computing, 200193.

[197] Ramana, R., Vasudevan, V., and Murugan, B. S. (2025). Spectral Pyramid Pooling and Fused Keypoint Generation in ResNet-50 for Robust 3D Object Detection. IETE Journal of Research, 1-13.

[198] Shin, S., Land, O., Seider, W., Lee, J., and Lee, D. (2025). Artificial Intelligence-Empowered Automated Double Emulsion Droplet Library Generation.

[199] Taca, B. S., Lau, D., and Rieder, R. (2025). A comparative study between deep learning approaches for aphid classification. IEEE Latin America Transactions, 23(3), 198-204.

[200] Ulaş, B., Szklenár, T., and Szabó, R. (2025). Detection of Oscillation-like Patterns in Eclipsing Binary Light Curves using Neural Network-based Object Detection Algorithms. arXiv preprint arXiv:2501.17538.

[201] Valensi, D., Lupu, L., Adam, D., and Topilsky, Y. Semi-Supervised Learning, Foundation Models and Image Processing for Pleural Line Detection and Segmentation in Lung Ultrasound. Foundation Models and Image Processing for Pleural Line Detection and Segmentation in Lung Ultrasound.

[202] V, A., V, P. and Kumar, D. An effective object detection via BS2ResNet and LTK-Bi-LSTM. Multimed Tools Appl (2025). https://doi.org/10.1007/s11042-024-20433-2

[203] Zhu, X., Chen, W., and Jiang, Q. (2025). High-transferability black-box attack of binary image segmentation via adversarial example augmentation. Displays, 102957.

[204] Guo, X., Zhu, Y., Li, S., Wu, S., and Liu, S. (2025). Research and Implementation of Agronomic Entity and Attribute Extraction Based on Target Localization. Agronomy, 15(2), 354.

[205] Yousif, M., Jassam, N. M., Salim, A., Bardan, H. A., Mutlak, A. F., Sallibi, A. D., and Ataalla, A. F. Melanoma Skin Cancer Detection Using Deep Learning Methods and Binary GWO Algorithm.

[206] Rahman, S. I. U., Abbas, N., Ali, S., Salman, M., Alkhayat, A., Khan, J., ... and Gu, Y. H. (2025). Deep Learning and Artificial Intelligence-Driven Advanced Methods for Acute Lymphoblastic Leukemia Identification and Classification: A Systematic Review. Comput Model Eng Sci, 142(2).

[207] Pratap Joshi, K., Gowda, V. B., Bidare Divakarachari, P., Siddappa Parameshwarappa, P., and Patra, R. K. (2025). VSA-GCNN: Attention Guided Graph Neural Networks for Brain Tumor Segmentation and Classification. Big Data and Cognitive Computing, 9(2), 29.

[208] Ng, B., Eyre, K., and Chetrit, M. (2025). Prediction of ischemic cardiomyopathy using a deep neural network with non-contrast cine cardiac magnetic resonance images. Journal of Cardiovascular Magnetic Resonance, 27.

[209] Nguyen, H. T., Lam, T. B., Truong, T. T. N., Duong, T. D., and Dinh, V. Q. Mv-Trams: An Efficient Tumor Region-Adapted Mammography Synthesis Under Multi-View Diagnosis. Available at SSRN 5109180.

[210] Chen, W., Xu, T., and Zhou, W. (2025). Task-based Regularization in Penalized Least-Squares for Binary Signal Detection Tasks in Medical Image Denoising. arXiv preprint arXiv:2501.18418.

[211] Pradhan, P. D., Talmale, G., and Wazalwar, S. Deep dive into precision (DDiP): Unleashing advanced deep learning approaches in diabetic retinopathy research for enhanced detection and classification of retinal abnormalities. In Recent Advances in Sciences, Engineering, Information Technology and Management (pp. 518-530). CRC Press.

[212] Örenç, S., Acar, E., Özerdem, M. S., Şahin, S., and Kaya, A. (2025). Automatic Identification of Adenoid Hypertrophy via Ensemble Deep Learning Models Employing X-ray Adenoid Images. Journal of Imaging Informatics in Medicine, 1-15.

[213] Jiang, M., Wang, S., Chan, K. H., Sun, Y., Xu, Y., Zhang, Z., ... and Tan, T. (2025). Multimodal Cross Global Learnable Attention Network for MR images denoising with arbitrary modal missing. Computerized Medical Imaging and Graphics, 102497.

[214] Al-Haidri, W., Levchuk, A., Zotov, N., Belousova, K., Ryzhkov, A., Fokin, V., ... and Brui, E. (2025). Quantitative analysis of myocardial fibrosis using a deep learning-based framework applied to the 17-Segment model. Biomedical Signal Processing and Control, 105, 107555.

[215] Osorio, S. L. J., Ruiz, M. A. R., Mendez-Vazquez, A., and Rodriguez-Tello, E. (2024). Fourier Series Guided Design of Quantum Convolutional Neural Networks for Enhanced Time Series Forecasting. arXiv preprint arXiv:2404.15377.

[216] Umeano, C., and Kyriienko, O. (2024). Ground state-based quantum feature maps. arXiv preprint arXiv:2404.07174.

[217] Liu, N., He, X., Laurent, T., Di Giovanni, F., Bronstein, M. M., and Bresson, X. (2024). Advancing Graph Convolutional Networks via General Spectral Wavelets. arXiv preprint arXiv:2405.13806.

[218] Vlasic, A. (2024). Quantum Circuits, Feature Maps, and Expanded Pseudo-Entropy: A Categorical Theoretic Analysis of Encoding Real-World Data into a Quantum Computer. arXiv preprint arXiv:2410.22084.

[219] Kim, M., Hioka, Y., and Witbrock, M. (2024). Neural Fourier Modelling: A Highly Compact Approach to Time-Series Analysis. arXiv preprint arXiv:2410.04703.

[220] Xie, Y., Daigavane, A., Kotak, M., and Smidt, T. (2024). The price of freedom: Exploring tradeoffs between expressivity and computational efficiency in equivariant tensor products. In ICML 2024 Workshop on Geometry-grounded Representation Learning and Generative Modeling.

[221] Liu, G., Wei, Z., Zhang, H., Wang, R., Yuan, A., Liu, C., ... and Cao, G. (2024, April). Extending Implicit Neural Representations for Text-to-Image Generation. In ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 3650-3654). IEEE.

[222] Zhang, M. (2024). Lock-in spectrum: a tool for representing long-term evolution of bearing fault in the time–frequency domain using vibration signal. Sensor Review, 44(5), 598-610.

[223] Hamed, M., and Lachiri, Z. (2024, July). Expressivity Transfer In Transformer-Based Text-To-Speech Synthesis. In 2024 IEEE 7th International Conference on Advanced Technologies, Signal and Image Processing (ATSIP) (Vol. 1, pp. 443-448). IEEE.

[224] Lehmann, F., Gatti, F., Bertin, M., Grenié, D., and Clouteau, D. (2024). Uncertainty propagation from crustal geologies to rock-site ground motion with a Fourier Neural Operator. European Journal of Environmental and Civil Engineering, 28(13), 3088-3105.

[225] Jurafsky, D. (2000). Speech and language processing.

[226] Manning, C., and Schutze, H. (1999). Foundations of statistical natural language processing. MIT press.

[227] Liu, Y., and Zhang, M. (2018). Neural network methods for natural language processing.

[228] Allen, J. (1988). Natural language understanding. Benjamin-Cummings Publishing Co., Inc..

[229] Li, Z., Zhao, Y., Zhang, X., Han, H., and Huang, C. (2025). Word embedding factor based multi-head attention. Artificial Intelligence Review, 58(4), 1-21.

[230] Hempelmann, C. F., Rayz, J., Dong, T., and Miller, T. (2025, January). Proceedings of the 1st Workshop on Computational Humor (CHum). In Proceedings of the 1st Workshop on Computational Humor (CHum).

[231] Koehn, P. (2009). Statistical machine translation. Cambridge University Press.

[232] Eisenstein, J. (2019). Introduction to natural language processing. The MIT Press.

[233] Otter, D. W., Medina, J. R., and Kalita, J. K. (2020). A survey of the usages of deep learning for natural language processing. IEEE transactions on neural networks and learning systems, 32(2), 604-624.

[234] Mitkov, R. (Ed.). (2022). The Oxford handbook of computational linguistics. Oxford university press.

[235] Liu, X., Tao, Z., Jiang, T., Chang, H., Ma, Y., and Huang, X. (2024). ToDA: Target-oriented Diffusion Attacker against Recommendation System. arXiv preprint arXiv:2401.12578.

[236] Çekik, R. (2025). Effective Text Classification Through Supervised Rough Set-Based Term Weighting. Symmetry, 17(1), 90.

[237] Zhu, H., Xia, J., Liu, R., and Deng, B. (2025). SPIRIT: Structural Entropy Guided Prefix Tuning for Hierarchical Text Classification. Entropy, 27(2), 128.

[238] Matrane, Y., Benabbou, F., and Ellaky, Z. (2024). Enhancing Moroccan Dialect Sentiment Analysis through Optimized Preprocessing and transfer learning Techniques. IEEE Access.

[239] Moqbel, M., and Jain, A. (2025). Mining the truth: A text mining approach to understanding perceived deceptive counterfeits and online ratings. Journal of Retailing and Consumer Services, 84, 104149.

[240] Kumar, V., Iqbal, M. I., and Rathore, R. (2025). Natural Language Processing (NLP) in Disease Detection—A Discussion of How NLP Techniques Can Be Used to Analyze and Classify Medical Text Data for Disease Diagnosis. AI in Disease Detection: Advancements and Applications, 53-75.

[241] Yin, S. (2024). The Current State and Challenges of Aspect-Based Sentiment Analysis. Applied and Computational Engineering, 114, 25-31.

[242] Raghavan, M. (2024). Are you who AI says you are? Exploring the role of Natural Language Processing algorithms for "predicting" personality traits from text (Doctoral dissertation, University of South Florida).

[243] Semeraro, A., Vilella, S., Improta, R., De Duro, E. S., Mohammad, S. M., Ruffo, G., and Stella, M. (2025). EmoAtlas: An emotional network analyzer of texts that merges psychological lexicons, artificial intelligence, and network science. Behavior Research Methods, 57(2), 77.

[244] Cai, F., and Liu, X. Data Analytics for Discourse Analysis with Python: The Case of Therapy Talk, by Dennis Tay. New York: Routledge, 2024. ISBN: 9781032419015 (HB: USD 41.24), xiii+ 182 pages. Natural Language Processing, 1-4.

[245] Wu, Yonghui. "Google's neural machine translation system: Bridging the gap between human and machine translation." arXiv preprint arXiv:1609.08144 (2016).

[246] Hettiarachchi, H., Ranasinghe, T., Rayson, P., Mitkov, R., Gaber, M., Premasiri, D., ... and Uyangodage, L. (2024). Overview of the First Workshop on Language Models for Low-Resource Languages (LoResLM 2025). arXiv preprint arXiv:2412.16365.

[247] Das, B. R., and Sahoo, R. (2024). Word Alignment in Statistical Machine Translation: Issues and Challenges. Nov Joun of Appl Sci Res, 1 (6), 01-03.

[248] Oluwatoki, T. G., Adetunmbi, O. A., and Boyinbode, O. K. A Transformer-Based Yoruba to English Machine Translation (TYEMT) System with Rouge Score.

[249] UÇKAN, T., and KURT, E. Word Embeddings in NLP. PIONEER AND INNOVATIVE STUDIES IN COMPUTER SCIENCES AND ENGINEERING, 58.

[250] Pastor, G. C., Monti, J., Mitkov, R., and Hidalgo-Ternero, C. M. (2024). Recent Advances in Multiword Units in Machine Translation and Translation Technology. Recent Advances in Multiword Units in Machine Translation and Translation Technology.

[251] Fernandes, R. M. Decoding spatial semantics: a comparative analysis of the performance of open-source LLMs against NMT systems in translating EN-PT-BR subtitles (Doctoral dissertation, Universidade de SÃ£ o Paulo).

[252] Jozić, K. (2024). Testing ChatGPT's Capabilities as an English-Croatian Machine Translation System in a Real-World Setting: eTranslation versus ChatGPT at the European Central Bank (Doctoral dissertation, University of Zagreb. Faculty of Humanities and Social Sciences. Department of English language and literature).

[253] Yang, M. (2025). Adaptive Recognition of English Translation Errors Based on Improved Machine Learning Methods. International Journal of High Speed Electronics and Systems, 2540236.

[254] Linnemann, G. A., and Reimann, L. E. (2024). Artificial Intelligence as a New Field of Activity for Applied Social Psychology–A Reasoning for Broadening the Scope.

[255] Merkel, S., and Schorr, S. OPP: APPLICATION FIELDS and INNOVATIVE TECHNOLOGIES.

[256] Kushwaha, N. S., and Singh, P. (2022). Artificial Intelligence based Chatbot: A Case Study. Journal of Management and Service Science (JMSS), 2(1), 1-13.

[257] Macedo, P., Madeira, R. N., Santos, P. A., Mota, P., Alves, B., and Pereira, C. M. (2024). A Conversational Agent for Empowering People with Parkinson's Disease in Exercising Through Motivation and Support. Applied Sciences, 15(1), 223.

[258] Gupta, R., Nair, K., Mishra, M., Ibrahim, B., and Bhardwaj, S. (2024). Adoption and impacts of generative artificial intelligence: Theoretical underpinnings and research agenda. International Journal of Information Management Data Insights, 4(1), 100232.

[259] Foroughi, B., Iranmanesh, M., Yadegaridehkordi, E., Wen, J., Ghobakhloo, M., Senali, M. G., and Annamalai, N. (2025). Factors Affecting the Use of ChatGPT for Obtaining Shopping Information. International Journal of Consumer Studies, 49(1), e70008.

[260] Jandhyala, V. S. V. (2024). BUILDING AI CHATBOTS AND VIRTUAL ASSISTANTS: A TECHNICAL GUIDE FOR ASPIRING PROFESSIONALS. INTERNATIONAL JOURNAL OF RESEARCH IN COMPUTER APPLICATIONS AND INFORMATION TECHNOLOGY (IJRCAIT), 7(2), 448-463.

[261] Pavlović, N., and Savić, M. (2024). The Impact of the ChatGPT Platform on Consumer Experience in Digital Marketing and User Satisfaction. Theoretical and Practical Research in Economic Fields, 15(3), 636-646.

[262] Mannava, V., Mitrevski, A., and Plöger, P. G. (2024, August). Exploring the Suitability of Conversational AI for Child-Robot Interaction. In 2024 33rd IEEE International Conference on Robot and Human Interactive Communication (ROMAN) (pp. 1821-1827). IEEE.

[263] Sherstinova, T., Mikhaylovskiy, N., Kolpashchikova, E., and Kruglikova, V. (2024, April). Bridging Gaps in Russian Language Processing: AI and Everyday Conversations. In 2024 35th Conference of Open Innovations Association (FRUCT) (pp. 665-674). IEEE.

[264] Lipton, Z. C. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning. arXiv Preprint, CoRR, abs/1506.00019.

[265] Pascanu, R. (2013). On the difficulty of training recurrent neural networks. arXiv preprint arXiv:1211.5063.

[266] Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. Bonn, Germany: German National Research Center for Information Technology GMD Technical Report, 148(34), 13.

[267] Hochreiter, S. (1997). Long Short-term Memory. Neural Computation MIT-Press.

[268] Kawakami, K. (2008). Supervised sequence labelling with recurrent neural networks (Doctoral dissertation, Ph. D. thesis).

[269] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2), 157-166.

[270] Bhattamishra, S., Patel, A., and Goyal, N. (2020). On the computational power of transformers and its implications in sequence modeling. arXiv preprint arXiv:2006.09286.

[271] Siegelmann, H. T. (1993). Theoretical foundations of recurrent neural networks.

[272] Sutton, R. S. (2018). Reinforcement learning: An introduction. A Bradford Book.

[273] Barto, A. G. (2021). Reinforcement Learning: An Introduction. By Richard's Sutton. SIAM Rev, 6(2), 423.

[274] Bertsekas, D. P. (1996). Neuro-dynamic programming. Athena Scientific.

[275] Kakade, S. M. (2003). On the sample complexity of reinforcement learning. University of London, University College London (United Kingdom).

[276] Szepesvári, C. (2022). Algorithms for reinforcement learning. Springer nature.

[277] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018, July). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In International conference on machine learning (pp. 1861-1870). PMLR.

[278] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... and Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

[279] Konda, V., and Tsitsiklis, J. (1999). Actor-critic algorithms. Advances in neural information processing systems, 12.

[280] Levine, S. (2018). Reinforcement learning and control as probabilistic inference: Tutorial and review. arXiv preprint arXiv:1805.00909.

[281] Mannor, S., Mansour, Y., and Tamar, A. (2022). Reinforcement Learning: Foundations. Online manuscript.

[282] Borkar, V. S., and Borkar, V. S. (2008). Stochastic approximation: a dynamical systems viewpoint (Vol. 9). Cambridge: Cambridge University Press.

[283] Takhsha, Amir Reza, Maryam Rastgarpour, and Mozhgan Naderi. "A Feature-Level Ensemble Model for COVID-19 Identification in CXR Images using Choquet Integral and Differential Evolution Optimization." arXiv preprint arXiv:2501.08241 (2025).

[284] Singh, P., and Raman, B. (2025). Graph Neural Networks: Extending Deep Learning to Graphs. In Deep Learning Through the Prism of Tensors (pp. 423-482). Singapore: Springer Nature Singapore.

[285] Yao, L., Shi, Q., Yang, Z., Shao, S., and Hariri, S. (2024). Development of an Edge Resilient ML Ensemble to Tolerate ICS Adversarial Attacks. arXiv preprint arXiv:2409.18244.

[286] Chen, K., Bi, Z., Niu, Q., Liu, J., Peng, B., Zhang, S., ... and Feng, P. (2024). Deep learning and machine learning, advancing big data analytics and management: Tensorflow pretrained models. arXiv preprint arXiv:2409.13566.

[287] Dumić, E. (2024). Learning neural network design with TensorFlow and Keras. In ICERI2024 Proceedings (pp. 10689-10696). IATED.

[288] Bajaj, K., Bordoloi, D., Tripathy, R., Mohapatra, S. K., Sarangi, P. K., and Sharma, P. (2024, September). Convolutional Neural Network Based on TensorFlow for the Recognition of Handwritten Digits in the Odia. In 2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET) (pp. 1-5). IEEE.

[289] Abbass, A. M., and Fyath, R. S. (2024). Enhanced approach for artificial neural network-based optical fiber channel modeling: Geometric constellation shaping WDM system as a case study. Journal of Applied Research and Technology, 22(6), 768-780.

[290] Prabha, D., Subramanian, R. S., Dinesh, M. G., and Girija, P. (2024). Sustainable Farming Through AI-Enabled Precision Agriculture. In Artificial Intelligence for Precision Agriculture (pp. 159-182). Auerbach Publications.

[291] Abdelmadjid, S. A. A. D., and Abdeldjallil, A. I. D. I. (2024, November). Optimized Deep Learning Models For Edge Computing: A Comparative Study on Raspberry PI4 For Real-Time Plant Disease Detection. In 2024 4th International Conference on Embedded and Distributed Systems (EDiS) (pp. 273-278). IEEE.

[292] Mlambo, F. (2024). What are Bayesian Neural Networks?.

[293] Team, G. Y. Bifang: A New Free-Flying Cubic Robot for Space Station.

[294] Tabel, L. (2024). Delay Learning in Spiking.

[295] Naderi, S., Chen, B., Yang, T., Xiang, J., Heaney, C. E., Latham, J. P., ... and Pain, C. C. (2024). A discrete element solution method embedded within a Neural Network. Powder Technology, 448, 120258.

[296] Polaka, S. K. R. (2024). Verifica delle reti neurali per l'apprendimento rinforzato sicuro.

[297] Erdogan, L. E., Kanakagiri, V. A. R., Keutzer, K., and Dong, Z. (2024). Stochastic Communication Avoidance for Recommendation Systems. arXiv preprint arXiv:2411.01611.

[298] Liao, F., Tang, Y., Du, Q., Wang, J., Li, M., and Zheng, J. (2024). Domain Progressive Low-dose CT Imaging using Iterative Partial Diffusion Model. IEEE Transactions on Medical Imaging.

[299] Sekhavat, Y. (2024). Looking for creative basis of artificial intelligence art in the midst of order and chaos based on Nietzsche's theories. Theoretical Principles of Visual Arts.

[300] Cai, H., Yang, Y., Tang, Y., Sun, Z., and Zhang, W. (2025). Shapley value-based class activation mapping for improved explainability in neural networks. The Visual Computer, 1-19.

[301] Na, W. (2024). Rach-Space: Novel Ensemble Learning Method With Applications in Weakly Supervised Learning (Master's thesis, Tufts University).

[302] Khajah, M. M. (2024). Supercharging BKT with Multidimensional Generalizable IRT and Skill Discovery. Journal of Educational Data Mining, 16(1), 233-278.

[303] Zhang, Y., Duan, Z., Huang, Y., and Zhu, F. (2024). Theoretical Bound-Guided Hierarchical VAE for Neural Image Codecs. arXiv preprint arXiv:2403.18535.

[304] Wang, L., and Huang, W. (2025). On the convergence analysis of over-parameterized variational autoencoders: a neural tangent kernel perspective. Machine Learning, 114(1), 15.

[305] Li, C. N., Liang, H. P., Zhao, B. Q., Wei, S. H., and Zhang, X. (2024). Machine learning assisted crystal structure prediction made simple. Journal of Materials Informatics, 4(3), N-A.

[306] Huang, Y. (2024). Research Advanced in Image Generation Based on Diffusion Probability Model. Highlights in Science, Engineering and Technology, 85, 452-456.

[307] Chenebuah, E. T. (2024). Artificial Intelligence Simulation and Design of Energy Materials with Targeted Properties (Doctoral dissertation, Université d'Ottawa— University of Ottawa).

[308] Furth, N., Imel, A., and Zawodzinski, T. A. (2024, November). Graph Encoders for Redox Potentials and Solubility Predictions. In Electrochemical Society Meeting Abstracts prime2024 (No. 3, pp. 344-344). The Electrochemical Society, Inc..

[309] Gong, J., Deng, Z., Xie, H., Qiu, Z., Zhao, Z., and Tang, B. Z. (2025). Deciphering Design of Aggregation-Induced Emission Materials by Data Interpretation. Advanced Science, 12(3), 2411345.

[310] Kim, H., Lee, C. H., and Hong, C. (2024, July). VATMAN: Video Anomaly Transformer for Monitoring Accidents and Nefariousness. In 2024 IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS) (pp. 1-7). IEEE.

[311] Albert, S. W., Doostan, A., and Schaub, H. (2024). Dimensionality Reduction for Onboard Modeling of Uncertain Atmospheres. Journal of Spacecraft and Rockets, 1-13.

[312] Sharma, D. K., Hota, H. S., and Rababaah, A. R. (2024). Machine Learning for Real World Applications (Doctoral dissertation, Department of Computer Science and Engineering, Indian Institute of Technology Patna).

[313] Li, T., Shi, Z., Dale, S. G., Vignale, G., and Lin, M. Jrystal: A JAX-based Differentiable Density Functional Theory Framework for Materials.

[314] Bieberich, S., Li, P., Ngai, J., Patel, K., Vogt, R., Ranade, P., ... and Stafford, S. (2024). Conducting Quantum Machine Learning Through The Lens of Solving Neural Differential Equations On A Theoretical Fault Tolerant Quantum Computer: Calibration and Benchmarking.

[315] Dagréou, M., Ablin, P., Vaiter, S., and Moreau, T. (2024). How to compute Hessian-vector products?. In The Third Blogpost Track at ICLR 2024.

[316] Lohoff, J., and Neftci, E. (2024). Optimizing Automatic Differentiation with Deep Reinforcement Learning. arXiv preprint arXiv:2406.05027.

[317] Legrand, N., Weber, L., Waade, P. T., Daugaard, A. H. M., Khodadadi, M., Mikuš, N., and Mathys, C. (2024). pyhgf: A neural network library for predictive coding. arXiv preprint arXiv:2410.09206.

[318] Alzás, P. B., and Radev, R. (2024). Differentiable nuclear deexcitation simulation for low energy neutrino physics. arXiv preprint arXiv:2404.00180.

[319] Edenhofer, G., Frank, P., Roth, J., Leike, R. H., Guerdi, M., Scheel-Platz, L. I., ... and Enßlin, T. A. (2024). Re-envisioning numerical information field theory (NIFTy. re): A library for Gaussian processes and variational inference. arXiv preprint arXiv:2402.16683.

[320] Chan, S., Kulkarni, P., Paul, H. Y., and Parekh, V. S. (2024, September). Expanding the Horizon: Enabling Hybrid Quantum Transfer Learning for Long-Tailed Chest X-Ray Classification. In 2024 IEEE International Conference on Quantum Computing and Engineering (QCE) (Vol. 1, pp. 572-582). IEEE.

[321] Ye, H., Hu, Z., Yin, R., Boyko, T. D., Liu, Y., Li, Y., ... and Li, Y. (2025). Electron transfer at birnessite/organic compound interfaces: mechanism, regulation, and two-stage kinetic discrepancy in structural rearrangement and decomposition. Geochimica et Cosmochimica Acta, 388, 253-267.

[322] Khan, M., Ludl, A. A., Bankier, S., Björkegren, J. L., and Michoel, T. (2024). Prediction of causal genes at GWAS loci with pleiotropic gene regulatory effects using sets of correlated instrumental variables. PLoS genetics, 20(11), e1011473.

[323] Ojala, K., and Zhou, C. (2024). Determination of outdoor object distances from monocular thermal images.

[324] Popordanoska, T., and Blaschko, M. (2024). Advancing Calibration in Deep Learning: Theory, Methods, and Applications.

[325] Alfieri, A., Cortes, J. M. P., Pastore, E., Castiglione, C., and Rey, G. M. Z. A Deep Q-Network Approach to Job Shop Scheduling with Transport Resources.

[326] Zanardelli, R. (2025). Statistical learning methods for decision-making, with applications in Industry 4.0.

[327] Norouzi, M., Hosseini, S. H., Khoshnevisan, M., and Moshiri, B. (2025). Applications of pre-trained CNN models and data fusion techniques in Unity3D for connected vehicles. Applied Intelligence, 55(6), 390.

[328] Wang, R., Yang, T., Liang, C., Wang, M., and Ci, Y. (2025). Reliable Autonomous Driving Environment Perception: Uncertainty Quantification of Semantic Segmentation. Journal of Transportation Engineering, Part A: Systems, 151(3), 04024117.

[329] Xia, Q., Chen, P., Xu, G., Sun, H., Li, L., and Yu, G. (2024). Adaptive Path-Tracking Controller Embedded With Reinforcement Learning and Preview Model for Autonomous Driving. IEEE Transactions on Vehicular Technology.

[330] Liu, Q., Tang, Y., Li, X., Yang, F., Wang, K., and Li, Z. (2024). MV-STGHAT: Multi-View Spatial-Temporal Graph Hybrid Attention Network for Decision-Making of Connected and Autonomous Vehicles. IEEE Transactions on Vehicular Technology.

[331] Chakraborty, D., and Deka, B. (2025). Deep Learning-based Selective Feature Fusion for Litchi Fruit Detection using Multimodal UAV Sensor Measurements. IEEE Transactions on Artificial Intelligence.

[332] Mirindi, D., Khang, A., and Mirindi, F. (2025). Artificial Intelligence (AI) and Automation for Driving Green Transportation Systems: A Comprehensive Review. Driving Green Transportation System Through Artificial Intelligence and Automation: Approaches, Technologies and Applications, 1-19.

[333] Choudhury, B., Rajakumar, K., Badhale, A. A., Roy, A., Sahoo, R., and Margret, I. N. (2024, June). Comparative Analysis of Advanced Models for Satellite-Based Aircraft Identification. In 2024 International Conference on Smart Systems for Electrical, Electronics, Communication and Computer Engineering (ICSSEECC) (pp. 483-488). IEEE.

[334] Almubarok, W., Rosiani, U. D., and Asmara, R. A. (2024, November). MobileNetV2 Pruning for Improved Efficiency in Catfish Classification on Resource-Limited Devices. In 2024 IEEE 10th Information Technology International Seminar (ITIS) (pp. 271-277). IEEE.

[335] Ding, Q. (2024, February). Classification Techniques of Tongue Manifestation Based on Deep Learning. In 2024 IEEE 3rd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA) (pp. 802-810). IEEE.

[336] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[337] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25.

[338] Sultana, F., Sufian, A., and Dutta, P. (2018, November). Advancements in image classification using convolutional neural network. In 2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN) (pp. 122-129). IEEE.

[339] Sattler, T., Zhou, Q., Pollefeys, M., and Leal-Taixe, L. (2019). Understanding the limitations of cnn-based absolute camera pose regression. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 3302-3312).

[340] Vaswani, A. (2017). Attention is all you need. Advances in Neural Information Processing Systems.

[341] Nannepagu, M., Babu, D. B., and Madhuri, C. B. Leveraging Hybrid AI Models: DQN, Prophet, BERT, ART-NN, and Transformer-Based Approaches for Advanced Stock Market Forecasting.

[342] De Rose, L., Andresini, G., Appice, A., and Malerba, D. (2024). VINCENT: Cyber-threat detection through vision transformers and knowledge distillation. Computers and Security, 103926.

[343] Buehler, M. J. (2025). Graph-Aware Isomorphic Attention for Adaptive Dynamics in Transformers. arXiv preprint arXiv:2501.02393.

[344] Tabibpour, S. A., and Madanizadeh, S. A. (2024). Solving High-Dimensional Dynamic Programming Using Set Transformer. Available at SSRN 5040295.

[345] Li, S., and Dong, P. (2024, October). Mixed Attention Transformer Enhanced Channel Estimation for Extremely Large-Scale MIMO Systems. In 2024 16th International Conference on Wireless Communications and Signal Processing (WCSP) (pp. 394-399). IEEE.

[346] Asefa, S. H., and Assabie, Y. (2024). Transformer-Based Amharic-to-English Machine Translation with Character Embedding and Combined Regularization Techniques. IEEE Access.

[347] Liao, M., and Chen, M. (2024, November). A new deepfake detection method by vision transformers. In International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2024) (Vol. 13403, pp. 953-957). SPIE.

[348] Jiang, L., Cui, J., Xu, Y., Deng, X., Wu, X., Zhou, J., and Wang, Y. (2024, August). SC-Former: Spatial and Channel-wise Transformer with Contrastive Learning for High-Quality PET Image Reconstruction. In 2024 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE International Conference on Robotics, Automation and Mechatronics (RAM) (pp. 26-31). IEEE.

[349] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... and Bengio, Y. (2014). Generative adversarial nets. Advances in neural information processing systems, 27.

[350] CHAPPIDI, J., and SUNDARAM, D. M. (2024). DUAL Q-LEARNING WITH GRAPH NEURAL NETWORKS: A NOVEL APPROACH TO ANIMAL DETECTION IN CHALLENGING ECOSYSTEMS. Journal of Theoretical and Applied Information Technology, 102(23).

[351] Joni, R. (2024). Delving into Deep Learning: Illuminating Techniques and Visual Clarity for Image Analysis (No. 12808). EasyChair.

[352] Kalaiarasi, G., Sudharani, B., Jonnalagadda, S. C., Battula, H. V., and Sanagala, B. (2024, July). A Comprehensive Survey of Image Steganography. In 2024 2nd International Conference on Sustainable Computing and Smart Systems (ICSCSS) (pp. 1225-1230). IEEE.

[353] Arjmandi-Tash, A. M., Mansourian, A., Rahsepar, F. R., and Abdi, Y. (2024). Predicting Photodetector Responsivity through Machine Learning. Advanced Theory and Simulations, 2301219.

[354] Gao, Y. (2024). Neural networks meet applied mathematics: GANs, PINNs, and transformers. HKU Theses Online (HKUTO).

[355] Hisama, K., Ishikawa, A., Aspera, S. M., and Koyama, M. (2024). Theoretical Catalyst Screening of Multielement Alloy Catalysts for Ammonia Synthesis Using Machine Learning Potential and Generative Artificial Intelligence. The Journal of Physical Chemistry C, 128(44), 18750-18758.

[356] Wang, M., and Zhang, Y. (2024). Image Segmentation in Complex Backgrounds using an Improved Generative Adversarial Network. International Journal of Advanced Computer Science and Applications, 15(5).

[357] Alonso, N. I., and Arias, F. (2025). The Mathematics of Q-Learning and the Hamilton-Jacobi-Bellman Equation. Fernando, The Mathematics of Q-Learning and the Hamilton-Jacobi-Bellman Equation (January 05, 2025).

[358] Lu, C., Shi, L., Chen, Z., Wu, C., and Wierman, A. (2024). Overcoming the Curse of Dimensionality in Reinforcement Learning Through Approximate Factorization. arXiv preprint arXiv:2411.07591.

[359] Humayoo, M. (2024). Time-Scale Separation in Q-Learning: Extending TD ($\triangle$) for Action-Value Function Decomposition. arXiv preprint arXiv:2411.14019.

[360] Jia, L., Qi, N., Su, Z., Chu, F., Fang, S., Wong, K. K., and Chae, C. B. (2024). Game theory and reinforcement learning for anti-jamming defense in wireless communications: Current research, challenges, and solutions. IEEE Communications Surveys and Tutorials.

[361] Chai, J., Chen, E., and Fan, J. (2025). Deep Transfer Q-Learning for Offline Non-Stationary Reinforcement Learning. arXiv preprint arXiv:2501.04870.

[362] Yao, J., and Gong, X. (2024, October). Communication-Efficient and Resilient Distributed Deep Reinforcement Learning for Multi-Agent Systems. In 2024 IEEE International Conference on Unmanned Systems (ICUS) (pp. 1521-1526). IEEE.

[363] Liu, Y., Yang, T., Tian, L., and Pei, J. (2025). SGD-TripleQNet: An Integrated Deep Reinforcement Learning Model for Vehicle Lane-Change Decision. Mathematics, 13(2), 235.

[364] Masood, F., Ahmad, J., Al Mazroa, A., Alasbali, N., Alazeb, A., and Alshehri, M. S. (2025). Multi IRS-Aided Low-Carbon Power Management for Green Communication in 6G Smart Agriculture Using Deep Game Theory. Computational Intelligence, 41(1), e70022.

[365] Patrick, B. Reinforcement Learning for Dynamic Economic Models.

[366] El Mimouni, I., and Avrachenkov, K. (2025, January). Deep Q-Learning with Whittle Index for Contextual Restless Bandits: Application to Email Recommender Systems. In Northern Lights Deep Learning Conference 2025.

[367] Shefin, R. S., Rahman, M. A., Le, T., and Alqahtani, S. (2024). xSRL: Safety-Aware Explainable Reinforcement Learning–Safety as a Product of Explainability. arXiv preprint arXiv:2412.19311.

[368] Khlifi, A., Othmani, M., and Kherallah, M. (2025). A Novel Approach to Autonomous Driving Using DDQN-Based Deep Reinforcement Learning.

[369] Kuczkowski, D. (2024). Energy efficient multi-objective reinforcement learning algorithm for traffic simulation.

[370] Krauss, R., Zielasko, J., and Drechsler, R. Large-Scale Evolutionary Optimization of Artificial Neural Networks Using Adaptive Mutations.

[371] Ahamed, M. S., Pey, J. J. J., Samarakoon, S. B. P., Muthugala, M. V. J., and Elara, M. R. (2025). Reinforcement Learning for Reconfigurable Robotic Soccer. IEEE Access.

[372] Elmquist, A., Serban, R., and Negrut, D. (2024). A methodology to quantify simulation-vs-reality differences in images for autonomous robots. IEEE Sensors Journal.

[373] Kobanda, A., Portelas, R., Maillard, O. A., and Denoyer, L. (2024). Hierarchical Subspaces of Policies for Continual Offline Reinforcement Learning. arXiv preprint arXiv:2412.14865.

[374] Xu, J., Xie, G., Zhang, Z., Hou, X., Zhang, S., Ren, Y., and Niyato, D. (2025). UPEGSim: An RL-Enabled Simulator for Unmanned Underwater Vehicles Dedicated in the Underwater Pursuit-Evasion Game. IEEE Internet of Things Journal, 12(3), 2334-2346.

[375] Patadiya, K., Jain, R., Moteriya, J., Palaniappan, D., Kumar, P., and Premavathi, T. (2024, December). Application of Deep Learning to Generate Auto Player Mode in Car Based Game. In 2024 IEEE 16th International Conference on Computational Intelligence and Communication Networks (CICN) (pp. 233-237). IEEE.

[376] Janjua, J. I., Kousar, S., Khan, A., Ihsan, A., Abbas, T., and Saeed, A. Q. (2024, December). Enhancing Scalability in Reinforcement Learning for Open Spaces. In 2024 International Conference on Decision Aid Sciences and Applications (DASA) (pp. 1-8). IEEE.

[377] Yang, L., Li, Y., Wang, J., and Sherratt, R. S. (2020). Sentiment analysis for E-commerce product reviews in Chinese based on sentiment lexicon and deep learning. IEEE access, 8, 23522-23530.

[378] Manikandan, C., Kumar, P. S., Nikitha, N., Sanjana, P. G., and Dileep, Y. Filtering Emails Using Natural Language Processing.

[379] ISIAKA, S. O., BABATUNDE, R. S., and ISIAKA, R. M. Exploring Artificial Intelligence (AI) Technologies in Predictive Medicine: A Systematic Review.

[380] Petrov, A., Zhao, D., Smith, J., Volkov, S., Wang, J., and Ivanov, D. Deep Learning Approaches for Emotional State Classification in Textual Data.

[381] Liang, M. (2025). Leveraging natural language processing for automated assessment and feedback production in virtual education settings. Journal of Computational Methods in Sciences and Engineering, 14727978251314556.

[382] Jin, L. (2025). Research on Optimization Strategies of Artificial Intelligence Algorithms for the Integration and Dissemination of Pharmaceutical Science Popularization Knowledge. Scientific Journal of Technology, 7(1), 45-55.

[383] McNicholas, B. A., Madden, M. G., and Laffey, J. G. (2025). Natural language processing in critical care: opportunities, challenges, and future directions. Intensive Care Medicine, 1-5.

[384] Abd Al Abbas, M., and Khammas, B. M. (2024). Efficient IoT Malware Detection Technique Using Recurrent Neural Network. Iraqi Journal of Information and Communication Technology, 7(3), 29-42.

[385] Kalonia, S., and Upadhyay, A. (2025). Deep learning-based approach to predict software faults. In Artificial Intelligence and Machine Learning Applications for Sustainable Development (pp. 326-348). CRC Press.

[386] Han, S. C., Weld, H., Li, Y., Lee, J., and Poon, J. Natural Language Understanding in Conversational AI with Deep Learning.

[387] Potter, K., and Egon, A. RECURRENT NEURAL NETWORKS (RNNS) FOR TIME SERIES FORECASTING.

[388] Yatkin, M. A., Kõrgesaar, M., and Işlak, Ü. (2025). A Topological Approach to Enhancing Consistency in Machine Learning via Recurrent Neural Networks. Applied Sciences, 15(2), 933.

[389] Saifullah, S. (2024). Comparative Analysis of LSTM and GRU Models for Chicken Egg Fertility Classification using Deep Learning.

[390] Noguer I Alonso, Miquel, The Mathematics of Recurrent Neural Networks (October 27, 2024). Available at SSRN: https://ssrn.com/abstract=5001243 or http://dx.doi.org/10.2139/ssrn.5001243

[391] Tu, Z., Jeffries, S. D., Morse, J., and Hemmerling, T. M. (2024). Comparison of time-series models for predicting physiological metrics under sedation. Journal of Clinical Monitoring and Computing, 1-11.

[392] Zuo, Y., Jiang, J., and Yada, K. (2025). Application of hybrid gate recurrent unit for in-store trajectory prediction based on indoor location system. Scientific Reports, 15(1), 1055.

[393] Lima, R., Scardua, L. A., and De Almeida, G. M. (2024). Predicting Temperatures Inside a Steel Slab Reheating Furnace Using Neural Networks. Authorea Preprints.

[394] Khan, S., Muhammad, Y., Jadoon, I., Awan, S. E., and Raja, M. A. Z. (2025). Leveraging LSTM-SMI and ARIMA architecture for robust wind power plant forecasting. Applied Soft Computing, 112765.

[395] Guo, Z., and Feng, L. (2024). Multi-step prediction of greenhouse temperature and humidity based on temporal position attention LSTM. Stochastic Environmental Research and Risk Assessment, 1-28.

[396] Abdelhamid, N. M., Khechekhouche, A., Mostefa, K., Brahim, L., and Talal, G. (2024). Deep-RNN based model for short-time forecasting photovoltaic power generation using IoT. Studies in Engineering and Exact Sciences, 5(2), e11461-e11461.

[397] Rohman, F. N., and Farikhin, B. S. Hyperparameter Tuning of Random Forest Algorithm for Diabetes Classification.

[398] Rahman, M. Utilizing Machine Learning Techniques for Early Brain Tumor Detection.

[399] Nandi, A., Singh, H., Majumdar, A., Shaw, A., and Maiti, A. Optimizing Baby Sound Recognition using Deep Learning through Class Balancing and Model Tuning.

[400] Sianga, B. E., Mbago, M. C., and Msengwa, A. S. (2025). PREDICTING THE PREVALENCE OF CARDIOVASCULAR DISEASES USING MACHINE LEARNING ALGORITHMS. Intelligence-Based Medicine, 100199.

[401] Li, L., Hu, Y., Yang, Z., Luo, Z., Wang, J., Wang, W., ... and Zhang, Z. (2025). Exploring the assessment of post-cardiac valve surgery pulmonary complication risks through the integration of wearable continuous physiological and clinical data. BMC Medical Informatics and Decision Making, 25(1), 1-11.

[402] Lázaro, F. L., Madeira, T., Melicio, R., Valério, D., and Santos, L. F. (2025). Identifying Human Factors in Aviation Accidents with Natural Language Processing and Machine Learning Models. Aerospace, 12(2), 106.

[403] Li, Z., Zhong, J., Wang, H., Xu, J., Li, Y., You, J., ... and Dev, S. (2025). RAINER: A Robust Ensemble Learning Grid Search-Tuned Framework for Rainfall Patterns Prediction. arXiv preprint arXiv:2501.16900.

[404] Khurshid, M. R., Manzoor, S., Sadiq, T., Hussain, L., Khan, M. S., and Dutta, A. K. (2025). Unveiling diabetes onset: Optimized XGBoost with Bayesian optimization for enhanced prediction. PloS one, 20(1), e0310218.

[405] Kanwar, M., Pokharel, B., and Lim, S. (2025). A new random forest method for landslide susceptibility mapping using hyperparameter optimization and grid search techniques. International Journal of Environmental Science and Technology, 1-16.

[406] Fadil, M., Akrom, M., and Herowati, W. (2025). Utilization of Machine Learning for Predicting Corrosion Inhibition by Quinoxaline Compounds. Journal of Applied Informatics and Computing, 9(1), 173-177.

[407] Emmanuel, J., Isewon, I., and Oyelade, J. (2025). An Optimized Deep-Forest Algorithm Using a Modified Differential Evolution Optimization Algorithm: A Case of Host-Pathogen Protein-Protein Interaction Prediction. Computational and Structural Biotechnology Journal.

[408] Gaurav, A., Gupta, B. B., Attar, R. W., Alhomoud, A., Arya, V., and Chui, K. T. (2025). Driver identification in advanced transportation systems using osprey and salp swarm optimized random forest model. Scientific Reports, 15(1), 2453.

[409] Ning, C., Ouyang, H., Xiao, J., Wu, D., Sun, Z., Liu, B., ... and Huang, G. (2025). Development and validation of an explainable machine learning model for mortality prediction among patients with infected pancreatic necrosis. eClinicalMedicine, 80.

[410] Muñoz, V., Ballester, C., Copaci, D., Moreno, L., and Blanco, D. (2025). Accelerating hyperparameter optimization with a secretary. Neurocomputing, 129455.

[411] Balcan, M. F., Nguyen, A. T., and Sharma, D. (2025). Sample complexity of data-driven tuning of model hyperparameters in neural networks with structured parameter-dependent dual function. arXiv preprint arXiv:2501.13734.

[412] Azimi, H., Kalhor, E. G., Nabavi, S. R., Behbahani, M., and Vardini, M. T. (2025). Data-based modeling for prediction of supercapacitor capacity: Integrated machine learning and metaheuristic algorithms. Journal of the Taiwan Institute of Chemical Engineers, 170, 105996.

[413] Shibina, V., and Thasleema, T. M. (2025). Voice feature-based diagnosis of Parkinson's disease using nature inspired squirrel search algorithm with ensemble learning classifiers. Iran Journal of Computer Science, 1-25.

[414] Chang, F., Dong, S., Yin, H., Ye, X., Wu, Z., Zhang, W., and Zhu, H. (2025). 3D displacement time series prediction of a north-facing reservoir landslide powered by InSAR and machine learning. Journal of Rock Mechanics and Geotechnical Engineering.

[415] Cihan, P. (2025). Bayesian Hyperparameter Optimization of Machine Learning Models for Predicting Biomass Gasification Gases. Applied Sciences, 15(3), 1018.

[416] Makomere, R., Rutto, H., Alugongo, A., Koech, L., Suter, E., and Kohitlhetse, I. (2025). Enhanced dry SO_2 capture estimation using Python-driven computational frameworks with hyperparameter tuning and data augmentation. Unconventional Resources, 100145.

[417] Bakır, H. (2025). A new method for tuning the CNN pre-trained models as a feature extractor for malware detection. Pattern Analysis and Applications, 28(1), 26.

[418] Liu, Y., Yin, H., and Li, Q. (2025). Sound absorption performance prediction of multi-dimensional Helmholtz resonators based on deep learning and hyperparameter optimization. Physica Scripta.

[419] Ma, Z., Zhao, M., Dai, X., and Chen, Y. (2025). Anomaly detection for high-speed machining using hybrid regularized support vector data description. Robotics and Computer-Integrated Manufacturing, 94, 102962.

[420] El-Bouzaidi, Y. E. I., Hibbi, F. Z., and Abdoun, O. (2025). Optimizing Convolutional Neural Network Impact of Hyperparameter Tuning and Transfer Learning. In Innovations in Optimization and Machine Learning (pp. 301-326). IGI Global Scientific Publishing.

[421] Mustapha, B., Zhou, Y., Shan, C., and Xiao, Z. (2025). Enhanced Pneumonia Detection in Chest X-Rays Using Hybrid Convolutional and Vision Transformer Networks. Current Medical Imaging, e15734056326685.