Abstract

Question Answering has become a central task in Natural Language Processing. In this work, we aim to propose improvements on some state-of-the-art Question Answering models and test their performance on the Stanford Question Answering Dataset 2.0. Two model families are discussed in this paper, i.e. BiDAF and BERT. For BiDAF model, we verified that character embedding layer contributes to the model prediction accuracy and outperforms the model without character embedding layer by 2-3%. We proposed a BiDAF model with ensembling techniques to split the prediction into two tasks. One is classification and the other is text span prediction. The experiment result shows the improvement of the prediction accuracy by about 1% compared with BiDAF model. For the BERT model family, we based our research on top of the bert-base-uncased model. After 5000 training steps, the fine-tuned BERT has already outperformed our best BiDAF model. Further we experimented with two sets of modifications on top of the baseline BERT model. The first set of models focuses on modifying the FC SQuAD output layer baseline BERT uses. The three models proposed in this category have steeper learning curves than the baseline BERT; however, in the end, the improvement is marginal (1%). The second set of models focuses on using pre-trained (no further optimization) BERT as word encoders. One model is proposed in this category. The experiment result shows that further work needs to be done, and the current model underperforms the fine-tuned baseline BERT.

Keywords: Natural Language Processing; Question Answering; BiDAF; BERT; SQuAD

1 Introduction

Question Answering (QA) is a challenging task for Natural Language Processing (NLP) researchers and practitioners. This is due to the complex relations between the context paragraph and its answer segment. To do well in reading comprehension task, skills such as inference, common sense reasoning, causal relations, and spatio-temporal relations are required. It also has a wide range of real-world applications, such as automated customer service, e-learning, web search and virtual assistants. The difficulty of the QA tasks has also increased from choosing the best answers among several possible options, to span prediction, to free-form generative QA. Such a reading comprehension task gives us a gauge in how well a NLP system can perform reading comprehension and information retrieval. The current best-performing models according to the SQuAD 2.0 leaderboard all use some form of pre-trained contextual embeddings, mostly from BERT.

The rest of the paper is organized as following: In Section 2, we describe the QA problem that we have solved and define it mathematically. In Section 3, we give a brief overview of the architectures of existing BiDAF and BERT models and any related work already done to both models. In Section 4, we introduce the design of our training infrastructure and software modules. We also list the pros and cons between Google Colab and AWS SageMaker that we experienced with during the project. In Section 5, we propose different models built on top of baseline BiDAF and BERT models and their architectures. In Section 6, we show and compare the experiment results and discuss our findings. In Section 7, we make suggestions for future work.

2 Problem Definition

The goal of the task in this project is to train a model to answer a question correctly by selecting the span of text from the given paragraph or to generate “N/A” if there is no answer in the paragraph. This means
that the model does not need to construct the answer text itself. It only needs to select the span of text (start and end positions) from the paragraph that answers the question.

Mathematically, QA problem is defined as the following. Given a sequence of words called context with length \( n_c: c = \{c_1, c_2, ..., c_{n_c}\} \) and a sequence of words called question with length \( n_q: q = \{q_1, q_2, ..., q_{n_q}\} \), the target task of the model is to learn a function \( f: (c, q) \rightarrow (s, e) \), where \( 1 \leq s \leq e \leq n_c \). (\( s, e \)) is a tuple of indices of the start and end positions of a span of text in the context \( c \). The span is the answer to the question. Further, if the model is extended to learn to detect questions with no answer, we simply padding a dummy token \( c_0 \) at the beginning of the context and the range of \( s, e \) is extended to \( 0 \leq s \leq e \leq n_c \). The model predicts no answer if the predicted span \((\hat{s}, \hat{e})\) is \((0, 0)\).

Cross-entropy loss is used as the loss function, i.e. the sum of the negative logarithm probabilities of the actual start and end positions of the answer by the predicted distributions, averaged over all examples:

\[
l(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \left( \log(p_{y^s_i}^e) + \log(p_{y^e_i}^e) \right)
\]

where \( \theta \) represents all trainable model parameters, \( N \) is the number of samples in the dataset, \( y^s_i, y^e_i \) are the actual start and end positions of the answer in the \( i^{th} \) input context, \( p_{y^s_i}^e \) is the predicted probability that the start position is \( y^s_i \), \( p_{y^e_i}^e \) is the predicted probability that the end position is \( y^e_i \).

To minimize the loss function \( l(\theta) \) is equivalent to choose an answer span \((y^s_i, y^e_i)\) where \( y^s_i \leq y^e_i \) with the maximum value of \( p_{y^s_i}^e p_{y^e_i}^e \).

3 Related Work

Our project mainly leverages on two main models: Bidirectional Attention Flow (BiDAF) and Bidirectional Encoder Representations from Transformers (BERT).

Before BERT [5] was developed in 2018, Bidirectional Attention Flow (BiDAF) [8] was one of the mainstream models, and many researchers have proposed changes to this baseline model over the years to achieve the state-of-the-art accuracy at the time. The central idea many researchers leveraged on is the Attention Flow layer that allows the attention flow bidirectionally. It improved the previous mainstream attention paradigms by a) enabling the attention to flow through to the subsequent modeling layer, which reduces the information loss caused by early summarization; b) simplifying the mechanism so that the attention does not directly depend on the attention at the previous time step; and more importantly c) enabling attention to flow bidirectionally from query to context and context to query [8]. The BiDAF model [8] is composed of:

- **Character embedding layer.** This layer maps each word in context or question to a vector space at the character level through a 1D-CNN.
- **Word embedding layer.** This layer maps each word in context or question to a vector space using a pre-trained word embedding model. In the original paper, the author used GloVe.
- **Contextual embedding layer.** This layer uses information surrounding words to refine the embedding of words.
- **Attention flow layer.** This layer couples the context and question vectors and produces a set of query-aware feature vectors for each word in the context.
- **Modeling layer.** This layer uses a RNN to scan the context.
- **Output layer.** This layer outputs the answer of the query.

The Figure (1) shows the high-level model architecture. We can see from the architecture that BiDAF model stacks lots of layers on top of each other and each one is designed to have different modeling purposes. We do not try to challenge the model itself or to break it up and finetune each component. Instead, since the BiDAF model itself is well-crafted, we tried to utilize it as the baseline and built layers on top of it and finetuned the model using the SQuAD dataset in this project. The open-sourced code [16] provides implementation of this BiDAF model but without the character embedding layer. We tried to implement
that as our first task in the project. It gives us the opportunity to test and compare the effectiveness of the character embedding layer and let us get familiar with the problem, the SQuAD dataset, the codebase, and the training and analysis procedure. We also use this opportunity to test our development environment to make sure it is robust enough to moving forward to develop more complicated BERT model.

The introduction of Bidirectional Encoder Representations from Transformers (BERT) improved the question-answering accuracy to a new level. A single BERT model consists of:

- A BERT Embedding Layer
  The embedding layer takes in a sequence of tokens and projects them to word, position and token type embeddings.

- A series of BERT Layers ($L := \text{number of layers, } H := \text{hidden size, } A := \text{number of self-attention heads}$)
  A BERT layer is a bi-directional transformer.

- A pooler layer
  This is a dense linear layer of size $H \times H$ followed by $tanh$ activation.

The Figure (2) shows the high-level model architecture. The open source github [17] provides code to build BERT models alongside with pre-trained weights for the following two BERT models:

- base-bert-uncased
  $L = 12, H = 768, A = 12$

- large-bert-uncased
  $L = 24, H = 1024, A = 16$

In the project, we load the pre-trained bert-base and develop our work on top of it. Devlin et al (2019) showed that one could use the pre-trained BERT and adapt it to SQuAD v2.0 by simply adding a fully-connected dense linear layer at the end. The adapted model consists of:

- A single BERT that outputs its final hidden vector
- A fully connected layer of size $H \times 2$
The main difference is the additional two learnable parameters introduced in the fully connected layer. Each column of the matrix maps the final hidden vector from the single BERT \( T_i \) for the \( i^{th} \) input token to a start logit and an end logit:

\[
\text{start logit}(i) = S \ast T_i, \text{end logit}(i) = E \ast T_i,
\]

\( S \) is the first column vector, and \( E \) is the second column vector of the weight matrix. The probabilities that the \( i^{th} \) token is the start/end of an answer span are therefore:

\[
P_{\text{START}}(i) = \frac{e^{S \ast T_i}}{\sum_j e^{S \ast T_j}}, P_{\text{END}}(i) = \frac{e^{E \ast T_i}}{\sum_j e^{E \ast T_j}}
\]

This architecture is shown in Figure (3).

As one can tell, this model simply adapts BERT to SQuAD by adding a logistic regression at the end, later we refer to this model as BERT + 1 FC.

It is fascinating that a simple FC layer at the end can give SOTA result at the time. We did not find much literature that explain why one FC layer is good enough. We therefore focused to explore if we could improve this architecture.

## 4 Model Training Infrastructure and Modules Overview

Designing a good model training infrastructure is important for smoothing model training experience and result analysis steps later on. We have several decisions to make before we start writing or running any code. This is even more important if we are going to develop large-scale software products or develop any large computational intensive models.

### 4.1 Google Colab v.s. AWS SageMaker

We have two vendor platforms to select from, i.e. Google Colab, which is used in homework and we have some experience with it, and AWS SageMaker, which is suggested by Professor Chaudhary and we have the coupon to use, but take some times to learn. We learned and tried both platforms at the beginning of the
project and list some pros and cons of both below to share our use experience.

Google Colab:

Pros:
- Easy to setup and link to gmail account, google drive (gdrive), etc. Most of us have google accounts.
- Free TPU/CPU hours, RAM, Disk space. User can run a program consecutively for 12 hours and use 12 GB RAM and 100 GB disk plus gdrive space mounted. There is also a paid plan which user can run a program consecutively for 24 hours and use 25 GB RAM, 100 GB disk plus gdrive space mounted.

Cons:
- Limited computing resource options: user can only use 1 GPU or 1 TPU, and select Standard or High RAM.
- Limited pre-installed ML libraries. Need to pip install many missing packages.
- Not stable connection. If internet gets disconnected or closing browser by accident or due to local machine problems, the running program stops. Any unsaved work is lost.

AWS SageMaker:

Pros:
- Various instance types fit different computation needs.
- Many pre-installed ML libraries, algorithms, packages.
- Connect to S3 storage.
- Has programmable interface using python boto3 library. User can not only talk to AWS services through console but also by writing python code to do it. This is an important feature for developing any real-world software.
- User can control start/stop computing instance without worrying about losing unsaved work. AWS takes care of all data backups and recovery if any issues happen.

Cons:
- Need to register an AWS account which most of us does not have.
- Need to learn how to use each AWS services through console and by writing python code. We started from reading manual and tutorials. Then reading boto3 library documentation [23]. It takes a lot of time to learn but it is worth doing it. It benefits us in the long term.
We experimented on different instance types by running the starter code from [16]. This can give us concrete evidences showing the power of each computing resources so that we can choose the one fitting our needs. Table (1) shows the instance types we tried together with some statistics to help us making the decision.

Table 1: Instance Types and Statistics for Google Colab and AWS EC2/SageMaker Computing Resources.

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>CPU/GPU Type</th>
<th>Cost (per hour)</th>
<th>Runtime per epoch BiDAF model</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 t2.xlarge</td>
<td>4xIntel Xeon CPU</td>
<td>$0.1856</td>
<td>90 mins</td>
</tr>
<tr>
<td>SageMaker ml.p2.xlarge</td>
<td>1xK80 GPU</td>
<td>$1.26</td>
<td>20 mins</td>
</tr>
<tr>
<td>SageMaker ml.p2.8xlarge</td>
<td>8xK80 GPU</td>
<td>$10.08</td>
<td>7 mins</td>
</tr>
<tr>
<td>SageMaker ml.p3.2xlarge</td>
<td>1xV100 GPU</td>
<td>$4.284</td>
<td>3 mins</td>
</tr>
<tr>
<td>Google Colab</td>
<td>1xK80/T4/P4/P100 GPU</td>
<td>Free</td>
<td>7 mins</td>
</tr>
</tbody>
</table>

From Table (1), we could see GPU should be used for training our model. However, whether to choose Colab or SageMaker is not very clear. As mentioned before, Colab is free to use and it provides comparable good computing power to AWS. AWS is costly but the p3.2xlarge instance is twice as fast as Colab which can reduce training time by half. Based on the cost and benefit analysis, we decided to choose Colab as main playground platform, but once we finalized on model infrastructure (especially for BERT), we deployed the code to and run on AWS SageMaker p3.2xlarge instance.

### 4.2 High-level Infrastructure Design

The below shows the design of our software infrastructure for model training. There are three components: “GPU” that represents the remote computing resources, “Storage” which is used for storing and sharing files, e.g. AWS S3 or Google Drive, a user local machine which controls the program process through Web browser or another program. They are connected and communicate through the network.

![Figure 4: High-level Infrastructure Design](image)

### 4.3 Program Modules

Our codebase replies heavily on the open-sourced starter code from [16] for BiDAF model and Transformers package [17] for BERT model. The detailed code documentation is in the github README file as well as in the source file comments. There are two main modules in our codebase. One is for BiDAF models which is based on starter code from [16]. The other one is for BERT models which leverages Transformers package [17].

For the BiDAF module, we directly modify the starter code to incorporate other models so that we can train, experiment and compare models together. The project handout [22] has a good description of what each file does. Further, to implement the infrastructure designed above, we wrote notebook files and uses
aws boto3 functions to communicate with Colab and SageMaker. Those functions are written as utilities in the BiDAF modules. Finally, we wrote a short Question-Answering application script which is the one we demoed in the presentation.

For the BERT module, we wrote a python script to implement all models we proposed in section 5.2, and we modified the training code from [17] so we can incorporate our models and visualize training progress in tensorboard.

5 Original Work

5.1 Baseline Model - BiDAFs

As we mentioned in section 3, we planned to use the given BiDAF model from [16] as the baseline and built layers and models on top of it and try to see if we can get any improvement on model prediction accuracy. We refer the model in the starter code as BiDAF-Baseline. We would like to experiment with two models that build on top of BiDAF-Baseline. As we mentioned in section 3 that the starter code [16] misses the character level embedding implementation, the first task is to implement the character embedding implementation and fill in the gap. We refer this model as BiDAF-Full. The second idea we proposed is ensembling. It is a two sub-models built under the ensembler. One sub-model is the BiDAF-Baseline model which is used to predict the span of text. The other sub-model is a binary classifier model referred as BiDAF-Classifier which solely used for answer classification prediction.

5.1.1 Model 1: BiDAF-Full

Our first task is to add a character-level embedding layer and compare the model prediction performance with and without the character embedding. Word vectors and character vectors are from GloVe [20] pre-trained result. Following Kim [18][19], we add an additional character level embedding using 1D Convolutional Neural Networks (CNN) to the input layer of the BiDAF-Baseline model. The extra character-level embedding can be helpful to solve the Out-of-Vocabulary (OOV) problem and supposed to increase the model’s performance.

5.1.2 Model 2: BiDAF-Ensemble

SQuAD 2.0 task is difficult for two reasons: (1) systems must determine when no answer is available, (2) systems must correctly return the span of text which answers the question when possible. Therefore, we were thinking if it is worth having a separate model to do binary classification (answer/no answer) and fuse two models together? In the BiDAF model, attention could be distraction if it pays attention to too many things at the same time or to many conflicting information. We could design an ensembler and let one model focus on the classification task and let the other one continue to do the span prediction and provides span to the ensembler if it decides to predict an answer.

The mathematical thought process is as follows. By Bayesian rule, the probability of a model predicting the answer correctly is

\[
P(\text{predict answer correctly})
= P(\text{predict span } (s, e) \text{ correctly}|\text{predict an answer exists})
\times P(\text{predict an answer exists}|\text{ground truth has an answer})
\times P(\text{ground truth has an answer})
+ P(\text{predict no answer}|\text{ground truth has no answer})
\times P(\text{ground truth has no answer})
\]

Since the two probability terms \(P(\text{ground truth has an answer})\) and \(P(\text{ground truth has no answer})\) are about the distribution of the dataset which is out of the modeling purpose, we only focus on the remaining 3 probability terms.

We denote \(P(\text{predict an answer exists}|\text{ground truth has an answer})\) as term (1),
\(P(\text{predict no answer}|\text{ground truth has no answer})\) as term (2),
and \(P(\text{predict span } (s, e) \text{ correctly}|\text{predict an answer exists})\) as term (3).
The (1) and (2) terms are the probability of classification accuracy. The third one is a complex language
modeling problem which requires the model to be well-trained to understand the information among context
and question text. According to the Bayesian rule and the property of conditional probability, term (1)
and (3) are independent from each other. Therefore, we could potentially model those two distributions
separately and then ensemble them together.

Based on the thought process above, we first design a binary classifier model called BiDAF-Classifier. It
builds on top of BiDAF-Full model and we modified the output layer to fit a classification job for predicting
answer/no answer. While the original model’s outputs are two probability distributions predicting the answer
span, the output of the classifier simply keep the concatenation of attention layer output and modeling layer
output corresponding to position 0 (no answer). A logistic regression layer is used to generate the answer/no
answer prediction. The BiDAF-Classifier model consists of:
• A BiDAF-Full model that outputs its attention vector and modeling vector
• A fully connected linear layer with softmax for binary classification

Once we have the BiDAF-Classifier model, we can add an ensembler on top of the BiDAF-Full and BiDAF-
Classifier models. If the classifier predicts No Answer, the ensembler returns No Answer. If the classifier
predicts Has Answer, the ensembler returns the predicted start and end positions scores from BiDAF-Full.
The architecture is shown in Figure (5). We refer this model as BiDAF-Ensemble.

Figure 5: Model 2: BiDAF-Ensemble

5.2 Research Model - BERTs

As we mentioned in section 3, we are fascinated that a single FC layer at the end is enough to give SOTA
result on SQuAD 1.1 at the time, but we could not find much literature explains the reasons behind this
design choice. We therefore would like to experiment with several alternative designs inspired from our study
of the GRU architecture and word embeddings.

In total, we proposed four different models that built on top of BERT. These models can be categorized into
two groups:

1) The first group of the models focused on modifying the FC layer. We proposed three models in this
category.

2) The second group of the models explored the possiblity of using BERT as a pre-trained and fixed word
encoder. We proposed one model in this category.
5.2.1 Model 1: BERT + 1 FF

The BERT + 1 FC model seems to assume all features are linearly separable and a single fully connected layer is already good enough for SQuAD task. In this model, we test that hypothesis by replacing the FC layer with an 1-hidden-layer Feed Forward neural network. This architecture is shown in Figure (6), it consists of:

- A single BERT that outputs its final hidden vector
- An 1-hidden-layer FF network.

We refer to this model as BERT + 1 FF.

![Figure 6: Model 1: BERT + 1 FF SQuAD Head](image)

5.2.2 Model 2: BERT + HiddenCat

The BERT + 1 FC model only uses the last hidden output vector from BERT. We are curious whether performance can be improved if all hidden layers outputs from BERT are used. In this model, we simply concatenate all hidden layers into a wide matrix, and pass the wide matrix to a final linear layer. This architecture is shown in Figure (7), it consists of:

- A single BERT that outputs all hidden layers to be concatenated
- An 1-hidden-layer FF network which consumes the concatenated hidden vectors

We refer to this model as BERT + HiddenCat.

5.2.3 Model 3: BERT + GRUMixer

We continue exploring down a similar path as section 5.2.2. Our hypothesis is that previous hidden states could contain information that is useful for question answering. The approach takes in section 5.2.2 is to simply expose all hidden states from the BERT to the downstream SQuAD processor. However, there could be irrelevant information presented in the hidden states, and if we could filter them out, we might end up with a better model.

To test this hypothesis, we use an architecture very similar to the GRU architecture we learned in class. We introduce an update gate $g_i$ and a cell state $c_i$ for each hidden layer $h_i$ from BERT. The update gate decides what information should be filtered out, and what information should be retained:

$$c_1 = h_1$$

$$c_i = (1 - g_i) \circ c_{i-1} + g_i \circ \phi(W_i h_i + b_i)$$
Figure 7: Model 2: BERT + HiddenCat

\[ g_i = \sigma(W^g_i h_i + U^g_i c_{i-1} + b^g_i) \]

In the end, we pass the final cell state to a FF network. This architecture is shown in Figure (8), it consists of:

- A single BERT that outputs all hidden layers
- A GRU-like mixer that takes in all hidden layers and compute a final cell state
- An 1-hidden-layer FF network

We refer to this model as BERT + GRUMixer.

### 5.2.4 Model 4: BERT As Encoder

BERT is trained for masked language modeling (MLM) [5]. Conceptually, we think BERT should provide information useful for natural language understanding. Hence we are curious if we can just use the pre-trained BERT out of box (no further fine-tuning) as an input embedding layer, and add question-answering architectures on top.

Hence, we designed the model as following:

- A single BERT that outputs features for question
- A single BERT that outputs features for context
- A classic attention that uses context as key and values, and question-text as queries
- An 1-hidden-layer FF network

This architecture is shown in Figure (9). We refer to this model as BERTAsEncoder.

### 6 Results and Analysis

#### 6.1 Model Data

The dataset for training and developing models in this project is the Stanford Question Answering Dataset (SQuAD 2.0)[1]. It is one of the largest available machine comprehension datasets with human-written
questions and serves as a great test bed for our models. Figure (10, 11) are two example context and questions with and without an answer from SQuAD 2.0.
6.2 Model Evaluation

To evaluate model prediction performance, two standard metrics of Exact Match (EM) and F1 Score (F1) are used for SQuAD.

EM measures the percentage of predictions that match any one of the ground truths exactly. The EM metric for a single sample is 1 if the prediction matches the ground truth exactly, and 0 otherwise. The final EM score for a set of samples is calculated as below.

\[ EM = \frac{\text{number of exact-match predictions}}{\text{total number of predictions}} \]

F1 measures the average overlap between the prediction and the ground truth, using a harmonic mean of precision and recall, where each prediction/answer pair is treated as a bag of tokens. Precision refers to what percentage of words in predicted answer is in the ground truth answer, and recall refers to what percentage of words in the ground truth answer is in predicted answer. The formula is

\[ F1 = \frac{2 \times \text{prediction} \times \text{recall}}{\text{prediction} + \text{recall}} \]

The maximum F1 score is taken for questions with multiple answers and the final score is the average F1 across all sample questions.

For any question without an answer, both metrics are 1 if the model predicts no answer, and 0 otherwise. F1 is less strict than EM. Therefore, for any dataset, the F1 score must be greater than or equal to EM score.

6.3 Baseline Model - BiDAFs

To make results comparable, we use the same set of hyper-parameters for all models. Below we list the key hyper-parameters.

- learning rate: 0.5
- optimizer: Adadelta
- learning rate decay: 0.999
- dropout probability: 0.2
This set of hyper-parameters are provided by the starter code [16]. We tried to use the same parameters set and is able to reproduce the results shown in the project handout [22]. Further, since the purpose of this project is to research and try our ideas to improve the given model, we did not spend time to finetune any hyper-parameters to produce the best prediction. We want to get started to try our own models as quickly as possible given the limited time of the project. If time permitted, we will try to finetune model parameters for each model too.

Table (2) shows the key F1 score of the BiDAF-Baseline, BiDAF-Full and BiDAF-Ensemble models. It also highlighted the best Dev F1 score at different training steps.

<table>
<thead>
<tr>
<th>Step</th>
<th>BiDAF-Baseline</th>
<th>BiDAF-Full</th>
<th>BiDAF-Ensemble</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>51.24</td>
<td>50.98</td>
<td>49.07</td>
</tr>
<tr>
<td>500000</td>
<td>53.64</td>
<td>54.73</td>
<td>55.22</td>
</tr>
<tr>
<td>1000000</td>
<td>58.16</td>
<td>60.30</td>
<td>62.09</td>
</tr>
<tr>
<td>1500000</td>
<td>58.68</td>
<td>61.33</td>
<td>62.66</td>
</tr>
<tr>
<td>2000000</td>
<td>59.63</td>
<td>62.45</td>
<td>63.75</td>
</tr>
<tr>
<td>2500000</td>
<td>60.07</td>
<td>62.48</td>
<td>63.90</td>
</tr>
</tbody>
</table>

From the result in the Table (2), we could have the following conclusions. First, adding character embedding layer to the baseline model improves the model accuracy by about 2-3%. Possible reasons are the following. First, it helps the model when the word is not in the vocabulary (OOV), while word embedding can only handle those seen words. Second, it a good fits for misspelling words, errors, typos, new words. Third, there are only limited amount of vectors for characters as compared to potentially unlimited words, it reduces model complexity and improving the performance (in terms of speed). However, the improvement depends on how many words are in the vocabulary, and the word embeddings we use are from Glove which is already representative enough: 840B tokens, 2.2M vocab, cased, 300d vectors. Further, we assume that when SQuAD compose the dataset, it could do some word error checks or corrections. Therefore, the benefits mentioned above are not prominent in our case and the accuracy improvement is by only about 2-3%.

Second, we see that the BiDAF-Ensemble outperforms BiDAF-Full by about 1%. It is not a significant improvement. Since using ensembling is known to be better than using stand-alone model, the results is not too exciting. Also since the model training time doubles because of training two sub-models at the same time, using ensemble does not provide much benefit if training time is a concern for any task or application.

### 6.4 Research Model - BERTs

We use the same set of hyper-parameters for all models. Below we list the key hyper-parameters.

- **learning rate**: $5e-5$
- **optimizer**: AdamW
- **number of epochs**: 2
- **batch size**: 8
- **doc_stride**: 128
This set of hyper-parameters is hand-picked to match the original BERT fine-tuning process from [17]. Since the hyper-parameters are consistent across the various runs and models, we can examine the effect of training speed vs. the F1 scores for different models.

Table (3) shows the key F1 score of all four models above and the base fine-tuned BERT model. It also highlighted the best Dev F1 score at different training steps.

<table>
<thead>
<tr>
<th>Step</th>
<th>BERT+1FC</th>
<th>BERT+1FF</th>
<th>BERT+HiddenCat</th>
<th>BERT+GRUMixer</th>
<th>BERTAsEncoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>54.22</td>
<td>61.40</td>
<td>63.21</td>
<td>62.95</td>
<td>39.47</td>
</tr>
<tr>
<td>1000</td>
<td>58.08</td>
<td>65.93</td>
<td>69.14</td>
<td>67.80</td>
<td>48.55</td>
</tr>
<tr>
<td>2000</td>
<td>59.55</td>
<td>70.70</td>
<td>74.28</td>
<td>74.12</td>
<td>49.46</td>
</tr>
<tr>
<td>5000</td>
<td>66.82</td>
<td>74.94</td>
<td>75.19</td>
<td>75.66</td>
<td>50.23</td>
</tr>
<tr>
<td>10000</td>
<td>72.00</td>
<td>74.60</td>
<td>75.38</td>
<td>75.93</td>
<td>49.48</td>
</tr>
<tr>
<td>END Of 1 EPOCH</td>
<td>74.30</td>
<td>74.17</td>
<td>75.38</td>
<td>75.54</td>
<td>49.67</td>
</tr>
</tbody>
</table>

*Note:* Due to limited time, we only trained our models for 1 EPOCH.

All BERT models outperform the baseline BiDAF model, and the GRU Mixer model slightly outperforms the BERT+1FC benchmark by 1% at the end of first epoch. Interestingly, adding additional hidden layer before the dense output layer does not improve the performance (-0.2%), although it has a steeper learning curve than BERT+1FC. A possible explanation is that, although the final pretrained BERT output could have non-linearly-separable features for QA tasks, the stacked multi-layer transformer model architecture allows it to eventually capture and learn non-linear features in earlier hidden layers, and prepare a final output that can be directly used by a FC layer. This can also help explain why both BERT+HiddenCat and BERT+GRUMixer are also trained much faster. As base-line BERT shuffling features from various layers and learning to prepare a final feature that can be passed to the end, these two models see all hidden features and might be able to skip some of the shuffling efforts.

Figure (12) shows two snapshots of the predicted answers from BERT+1FC and BERT+GRUMixer (taken at the 500-th training step). In this example, BERT+1FC seems to be reacting to a typical pattern - a sentence followed by a colon (":"). This pattern is usually used when authors try to explain concepts. Typically we should find that the detailed explanation of the concept directly follows the colon sign. This feature certainly makes sense, but apparently in this example, that is not the right answer. The model could have just looked at words closest to the question key word “The economy of Victoria” and it would give a right answer. BERT+GRUMixer clearly works better here. It appears as if BERT+1 FC didn’t pay attention to the question key words. We believe it is more likely that BERT+1FC in fact sees the nearby words, however, that attention feature locates in an earlier layer and it does not yet make its way to the final FC layer. On the other hand, BERT+GRUMixer benefits from the direct links from earlier hidden layers, so it sees the feature right away.

Although BERT+GRU learns faster, in the end, it only outperforms the baseline BERT marginally. Finally it might not be very surprised to see that the BERTAsEncoder perform worse than the other BERT models, as it has a much smaller trainable parameters set. However at a closer look, it also under performs our baseline BiDAF. We think there could be several explanations. Firstly, the model is intentionally designed to be very lightweight. The output features from BERT encoders are used directly in a multi-headed attention layer and are not optimized at all. This means that we are not utilizing BERT at its full potential. The BERT model is not only trained on the MSM task, but also on the NSP task. The output feature from BERT therefore has information on sentence prediction, but we are clearly throwing it away. This further implies that to achieve a better result, we might need to add additional CNN/FF networks on top of the BERT encoders to augment the relevant weights that is trained for MSM tasks. Secondly and more importantly, the model just has a very lightweight attention layer. The result suggests that one cannot fully rely on encoders and should always tailor their architectures for different tasks.
Future Work

For BiDAF, since we get a little improvement on model prediction accuracy in terms of F1 and EM metrics on SQuAD 2.0 using our ideas, we are interested to see how the model performs when applying to other dataset such as TriviaQA [21]. One limitation of our work is the time and resources required to train the model. If we have more computing resources and time, we would like to finetune the model hyper-parameters to see if they can play a role to improve the model performance. Further, since the F1 metric for BiDAF on the SQuAD leaderboard is reported as 67.9%, we are interested to see how they achieve that and may try to replicate their results.

For BERT, while trying to compare our fine-tuning result against other public literature, we noticed that most people prefer to use bert-large and albert-xxlarge for fine-tuning on SQuAD 2.0. Since the bert-large model has considerably more parameters to train, we did not consider that to be our baseline model. However, we would very much like to test our model using bert-large to see if we can better match the paper results, and if we still see that the BERT+GRUMixer model converges fast.

Also we have seen that the light-weight BertAsEncoder model does not perform very well. We think we have several leads with respect to why this is the case as discussed in section 6. If more time is given, we would like to try to improve the performance by adding additional layers after the base BERT encoder and replacing the multi-headed uni-directional attention layer by a better layer such as the one used in BiDAF. Last but not least, we would like to see if there is any good ideas from both BiDAF and BERT models that can be learned from to form a more powerful model. Ensembling those two together is a way to try. There could be some pieces or components in each of those two models that might be useful and used as building blocks for a different model.
References


