BERT-A: Fine-tuning BERT with Adapters and Data Augmentation

Sina J. Semnani
Department of Electrical Engineering
Stanford University
sinaj@stanford.edu

Kaushik Ram Sadagopan
Department of Mechanical Engineering
Stanford University
kaushik7@stanford.edu

Fatma Tlili
Department of Computer Science
Stanford University
ftlili@stanford.edu

Abstract

We tackle the contextual question answering (QA) problem on the SQuAD 2.0 dataset. Our project has two main objectives. Firstly, we aim to build a model that achieves a reasonable performance while keeping the number of trainable parameters to a minimum. In this regard, we insert task-specific modules *inside* the pre-trained BERT model to control the flow of information between transformer blocks. Our proposed method for fine-tuning BERT achieves comparable performance to fine-tuning all BERT parameters while only training **0.57%** of them. Secondly, we use our findings in the previous task to achieve an EM score of **78.36** and an F1 score of **81.44** on the test set (ranked 3rd on the PCE test leaderboard).

1 Introduction

Question answering (QA) is considered to be a proxy for machine comprehension, or at least a way to quantify it, and has been a central topic in NLP research during the last decade. It, however, still remains a challenging task since it requires complex reasoning in order to determine relationships between words across long sequences of text. In this project, we tackle the contextual question answering problem. More specifically, given a context paragraph and a question, our model returns a span of the answer in the context. We use SQuAD 2.0 dataset [12] and build a system that is both efficient to train, has minimal parameter overhead and achieves F1 and exact match (EM) scores comparable to human performance.

Pre-trained models have gained immense use and popularity in neural NLP tasks in recent years. While pre-trained word vectors such as word2vec [8] and GloVe [9] have been successful in capturing the meaning of single words, by themselves they generally do not help in capturing the meaning of words in their context. The advent of pre-trained contextual models such ULMFiT [4], ELMo [10], OpenAI transformer [11] and BERT [1] has led to a paradigm shift in NLP research where anyone can use pre-trained representations that encode contextual information rather than training everything from scratch.

Another important trend in NLP research is developing systems that can perform multiple tasks at the same performance level as a single-task model. [7] and [16] provide two datasets and approaches to tackle this problem. Considering these two trends, an important question arises: can we use a pre-trained model like BERT in a multi-task setting? [3] and [15] try to answer this very question for the GLUE benchmark [16]. Our work, though inspired by them, is different in two ways: Firstly, tasks

in GLUE are mostly sentence-level.[1] We investigate the possibility of using similar techniques for contextual QA in which understanding the details of every part of the sentence is crucial. Secondly, we extend their approach to a single-task setting where performance is much more important than the number of parameters.

Inspired by [3], we propose a method to use pre-trained BERT for QA without adding much more than half a million trained parameters, while maintaing the same F1 and EM scores as fine-tuning approach described in [1].

2 Related work

2.1 BERT and fine-tuning

BERT [1] differs from OpenAI GPT [11] and ELMo [10] by virtue of its bidirectional encoder where each word attends to every other word in both directions. A Masked Language Modeling task (where a portion of words is masked out and the model is made to predict these words) is used for its pre-training. BERT also entails a next sentence prediction task to understand relationships between sentences gaining deeper context.

Most current systems that tackle a single NLP task, use BERT as a means to get contextual word embeddings, and build a secondary model on top of that[2]. For reference, using BiDAF [14] would add 2.6 million additional parameters that need to be trained from scratch, using QANet[19], a state-of-the-art model, would add 1.3 million, which requires long training time. Another approach is to fine-tune all parameters of BERT for each task. This approach, however fast in terms of training time, does not generalize to the multi-task learning setting since it requires a separate BERT model for each task. This is especially more challenging when we consider deployment of trained models on mobile or edge devices since their memory is usually limited.

2.2 Answer Pointer

Our baseline model predicts a start-token and an end-token which constitutes a span in the passage, and are produced independently. We adopt Answer Pointer proposed by [17] where the start-token prediction is fed to a GRU to point to the corresponding end-token.

2.3 Adapters and PALs

Both PALs (Stickland and Murray [15]) and adapters ([3]) introduce task-specific modules in the BERT architecture for multi-task training. Adapters [3] add new bottleneck modules within the transformer layers of the pre-trained network which are fine-tuned for the given task. Similarly [15] add Projected Attention Layers within the transformer layer with shared weights across the different transformer layers of the BERT model. We describe these models in more details in the next section.

2.4 Transfer Learning from CoQA

Reddy et al. [13] have performed a comparative study of question answering datasets (CoQA, SQuAD 2.0 & QuAC) where it was shown that models benefit from pre-training on the CoQA dataset to SQuAD 2.0 (but not from QuAC). This prompts us to employ transfer learning from the CoQA dataset for our model as well.

3 Approach

In this section we describe our models and training approaches. Implementation of all these approaches (except for the baseline) and the idea and method of using adapters and PALs for QA are ours.

[1] Even QNLI part of GLUE which is based on SQuAD, is a simplified binary classification version.

[2] There are very few published work in this field due to its newness.

3.1 Baseline

We used the question answering model proposed in [1] to build our baseline. This model uses two vectors $S; E \ 2 \ \mathrm{R}^H$ for Start and End respectively where H is the size of output vectors for each token. Let $T_i \ 2 \ R^H$ be the final hidden vector from BERT of the i^{th} token, the probability of each word $i \ 2 \ fstart; endg$ is computed as follows:

$$P_i = \frac{e^{K:T_j}}{\sum_j e^{K:T_j}} \; j \; K \ 2 \ fS; Eg$$

Similar to the original implementation of BERT for SQuAD 2.0, we use a threshold to decide whether question is unanswerable. This is the simplest way to apply BERT to SQuAD as Devlin et al. [1] did, to intentionally avoid substantial task-specific architectural modifications. We use a modified version (to add training/validation tracking and other changes) of Huggingface Pytorch implementation [2] for the baseline.

3.2 Architecture Search

In this section we describe our modifications to the BERT architecture, and how they affect the performance. Our model choices are focused on two main goals:

1. Improving the overall performance on the SQuAD dataset
2. Keeping the storage overhead minimal

The three main parts of the our methods are summarized in Figures 1, 2 and 3 respectively. Our changes are both within the BERT-base transformer layers as well as at the output layer.

One of the changes that we made to the baseline model is to change the prediction of the start and end positions at the output layer and this is depicted in 3. Since predicting the start and end positions independently in BERT (see section 3.1) seems too simplistic, we employ the Boundary Model of the Answer Pointer Layer proposed by [17] to condition the end-position prediction on the start-position prediction.

The other two components, Adapters and PALs (Projected Attention Layers), add task-specific layers within the BERT model specifically in each of the 12 transformer blocks of the BERT-base. These methods are explained in more detail in the following parts. They are inspired by the papers mentioned below, however, since our task and goals are quite different from theirs, we only use their core idea as inspiration. The implementation, specific choice of additional layers and experiments conducted are ours.

3.2.1 Output Layer

Let S be the sequence output from the last block of BERT for a single example. $S \ 2 \ \mathrm{R}^{(L;H)}$ where L is the sequence length and H is the hidden size. This model generates the start-token and the end-token using attention weight factors $\ _{start}$ and $\ _{end} \ 2 \ \mathrm{R}^L$ on S. $\ _{start}$ and $\ _{end}$ represent the start-token and end-token softmax probabilities respectively. The computation of these factors depends on the following parameters which are to be learned: $V \ 2 \ \mathrm{R}^{H \quad H}; W^a \ 2 \ \mathrm{R}^{H \quad H}; b^a \ 2 \ \mathrm{R}^H; v \ 2 \ \mathrm{R}^H; c \ 2 \ \mathrm{R}$ and $h^a_{start} \ 2 \ \mathrm{R}^H$ ($h_0 \ 2 \ \mathrm{R}^H$ is initialized as a zero vector). e_L expands the first dimension of its left vector by repeating it L times. We start by computing $F_{start} \ 2 \ \mathrm{R}^{L \quad H}$:

$$F_{start} = \tanh(SV + (W^a h^a_0 + b^a) \quad e_L)$$

$$_{start} = \mathrm{softmax}(F_{start} v + c \quad e_L)$$

$$h^a_{start} = \mathrm{GRU}(S^T \ _{start}; h_0)$$

$$F_{end} = \tanh(SV + (W^a h^a_{start} + b^a) \quad e_L)$$

$$_{end} = \mathrm{softmax}(F_{end} v + c \quad e_L)$$

The loss function is calculated in the same way as the baseline model once we have the start_logits ($F_{start} v + c \quad e_L$) and the end_logits ($F_{end} v + c \quad e_L$).

3.2.2 Task-specific Layers

These two task-specific architectures were inspired from [3] and [15]. These papers aim at minimizing the number of parameters they train without hurting the performance of the model on the GLUE benchmark [16].

All layers we add, include some kind of bottleneck projection to lower-dimensional spaces to achieve a regularization effect, and a skip connection to minimize interference with other layers in BERT. We now describe these layers in more details. Let $\text{BERT}_w(x)$ be the function that represents the BERT model and $\text{BERTA}_{w;v}(x)$ be our models where v are the parameters of our additional layers. Based on our experiment, we learned that v should be initialized to make these layers near-identity i.e. $\text{BERTA}_{w;v_0}(x) \quad \text{BERT}_w(x)$.

Projected Attention Layers:

[15] introduced the idea of using task-specific attention mechanisms in BERT. In Figure 2, the inputs and outputs to the task-specific component are L vectors each in $\mathbb{R}^H$ where L is the sequence length and $H = 768$ is the hidden size. The task-specific layer is applied to each of these vectors and has the following formulae (A is the number of attention heads, a hyperparameter):

$$\text{Attention}_i(h_j) = \sum_k \text{softmax}(\frac{W_i^Q h_j \quad W_i^K h_k}{\sqrt{H/A}}) W_i^V h_k$$

$$\text{MultiHead}(h_j) = W^O[\text{Attention}_1(h_j); \text{Attention}_2(h_j); \ldots; \text{Attention}_A(h_j)]$$

$$\text{TaskSpecific}(h_j) = W^D \text{MultiHead}(W^E h_j)$$

Here, W^E projects the inputs to a smaller space, and W^D projects them back to the original size. These two matrices are the only parameters shared accross all transformer blocks. The size of the smaller space (bottleneck size) is a hyperparameter that needs to be tuned.

Adapters:

[3] proposes *adapters* as a more efficient alternative to fine tuning. The paper shows that adapter-based tuning requires training two orders of magnitude fewer parameters compared to fine-tuning the BERT model, while attaining similar performance on different tasks in the GLUE benchmark.

We adopt a similar approach for our question answering task. We implemented these bottleneck architectures and added them to the BERT architecture after the self-attention layer and after the feed-forward layer (see Figure 1). This module is added across all 12 transformer blocks with the goal to filter information and retain only the relevant information specific to Question Answering from the two main layers of the Transformer across all 12 blocks.

The Adapter has a simple bottleneck architecture. The bottleneck contains fewer parameters than the attention and the feed-forward layers.

The adapter layer can be expressed as follows:

$$\text{Adapter}(h_j) = W^D(\text{non-linearity}(W^E h_j + b_E)) + b_D$$

Here, similarity to the task-specific attention layers W^E projects the inputs to a smaller space, and W^D projects them back to the original size and b_E and b_D are the corresponding biases. The size of the smaller space is the only hyper-parameter that needs to be tuned.

3.3 Data Augmentation

One direction we considered is the effect of data augmentation on performance. We have experimented with two data augmentation techniques:

No-Answer Augmentation:

The paragraphs in SQuAD come from Wikipedia articles, and each paragraph has several questions. We retrieve the article from which the paragraph is chosen, choose two adjacent paragraphs from that article and add them with their questions, using *unanswerable* as the label. This increases the size of our training set by a factor of 3.

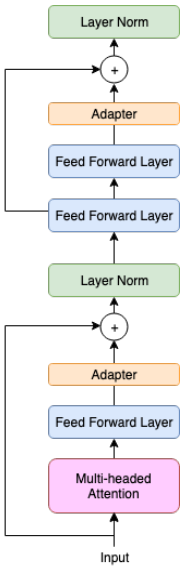

Figure 1: One transformer block in BERT. The adapter part is shown in orange

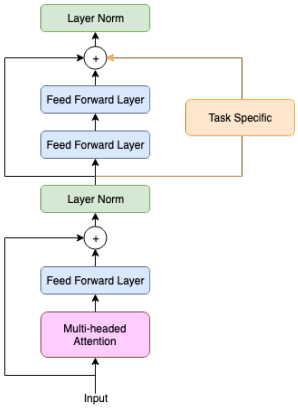

Figure 2: One transformer block in BERT. The projected attention layers are shown in orange.

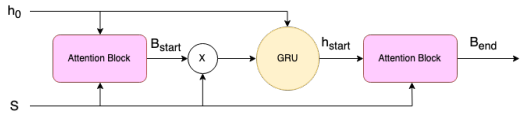

Figure 3: Answer Pointer Architecture

Transfer Learning from CoQA:

Inspired by [18] which compares different question answering datasets, we perform transfer learning from the model pre-trained on the CoQA dataset [13] (Conversational Question Answering systems) to the SQuAD datasest. This paper [18] shows that the datasets are ineffective for direct transfer but can improve the performance significantly when the model is pre-trained on a different dataset then normaly trained on the target dataset. Therefore we pre-train our model on the CoQA dataset prior to fine-turning it on the SQuAD dataset. We do not decrease the initial learning rates when fine-tuning on the SQuAD dataset.

4 Storage Efficiency

In this section, we explain how to achieve a good performance without greatly increasing the number of trainable parameters. We train our models on SQuAD 2.0 dataset only, i.e. without any augmentation. We do not use Answer Pointers here. We evaluated our models using F1 and EM metrics as well as the total number of parameters to be trained.

4.1 Experiments

We used BERT-base (the smaller version) to make training faster on the moderately powerful GPUs we have access to. We use mixed precision[3] for better speed and lower GPU memory consumption. Unless otherwise stated, we use Adam optimizer with $_1 = 0{:}9$, $_2 = 0{:}99$, learning rate of $2 \quad 10^{\ 5}$ with learning rate warm-up during the first 10% of the training. We train each model for 2 epochs with batch size 16. The results for each of the models are reported in table 1. The numbers are reported before tuning the answer/no-answer threshold since it increases overfitting to the dev set.

Note that if bottleneck size is b, hidden size is H and the number of transformer blocks is L, ignoring the bias parameters, adapters adds $4LHb$, layer norms have $4HL$ and PALs add $2Hb + 3Lb^2$ parameters in total.

[3] https://github.com/NVIDIA/apex

Table 1: Storage Efficient fine-tuning

Model	F1	EM	# Parameters (overhead)
baseline (fine-tuned)	76.5	73.5	110 M (+100%)
baseline (top block fine-tuned)	54.0	51.7	9.2 M (+8.3%)
baseline (frozen)	51.1	51.0	1.5 K (+0.001%)
baseline (frozen) + PALs(120)	63.9	60.7	704 K (+0.64%)
baseline (frozen) + Adapters(768)	70.9	67.4	592.9 K (+0.54%)
baseline (frozen) + Adapters(768) + LayerNorm	74.7	72.3	629.7 K (+0.57%)

In the table, baseline means fine-tuning all BERT parameters (and the final output layer) and baseline (frozen) means all BERT parameters (other than the final output layer) are frozen. For comparison, we have included a naive approach to reducing trainable parameters, which is only fine-tuning the last transformer block. baseline (frozen)+ means we freeze all pretrained parameters of BERT, and only train parameters of the module after +. The number in parenthesis is the size of the bottleneck. We have tried bottleneck sizes in $[64; 120; 768]$ and the latter achieved the best performance. To have a fair comparison, we use smaller bottleneck for PALs so that is has roughly the same number of parameters as adapters. Since adapters consistently outperformed PALs in QA, we report and analyze adapters in more details.

Our experiments show that freezing layer norms inside BERT negatively affect the performance, therefore our best model also trains them. If we used data augmentation, we could potentially get the same scores as the baseline while maintaining an extremely low number of parameers.

4.2 Analysis

We visualize the weights of W^E and W^D from each of two adapter modules in the last transformer block after training in figure 4. The general patterns of these matrices are similar in all transformer blocks: The first adapter (bottom one in figure 1) seems to be unnecessary since its weights do not show a particular pattern.

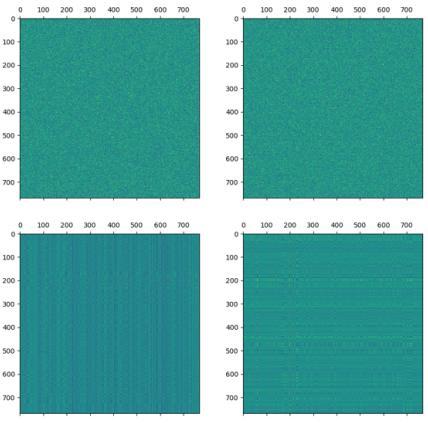

Figure 4: Learned weights from the last transformer block's adapter module. Top row: weights from adapters in self-attention submodule. Bottom row: weights from output submodule

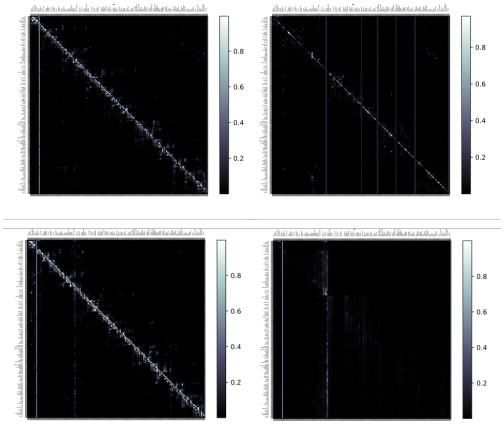

Figure 5: Top row: 6th transformer block. Bottom row: Last (12th) transformer block. Plots on the left are before training and plots on the right are after training

5 Performance

In this section we use some of our findings from the previous section and combine them with data augmentation to achieve higher performance by sacrificing the storage efficiency (we fine-tune all BERT weights).

5.1 Experiments

The setting and hyper-parameters are similar to experiments in the previous section.

In order to improve our model's performance on the SQuAD dataset, we performed several experiments. Our models were trained on a combination of the SQuAD dataset, the CoQA dataset, as well as the no-answer augmented dataset. We evaluated our models using F1 and EM metrics as well as the overall training time (in minutes). In the table, +Adapter means we trained the model before +, *then froze all BERT parameters* and inserted adapter modules, then trained adapter weights on the SQuAD dataset for 2 more epochs. Other methods like training adapters and BERT weighs simultaneously perform poorly.

The results for each of the models are reported in table 2. We have made a final submission to PCE test leader-board under the name sfk and we achieved an F1 score of 81.442 and EM score of 78.36. The results are better than expected. As shown in the table, the data-augmentation and adapters gave the majority of the performance boost. Our approach is thus fairly competitive with the recent approaches for the SQuAD competition. We observe good generalizability based on our results on

Table 2: Performance of various combinations of our techniques

Model	F1	EM	Training time (minutes)
baseline (fine-tuned)	76.5	73.5	377
baseline + Answer Pointer	76.7	73.5	388
baseline + Data Augmentation	77.9	75.5	1110
baseline + Pre-training on CoQA	78.5	75.7	836
baseline + Pre-training on CoQA + Adapter(768)	79.2	76.3	1240
baseline + Pre-training on CoQA + Data Augmentation + Answer Pointer	79.5	76.5	1722
baseline + Pre-training on CoQA + Data Augmentation + Answer Pointer+Adapter(768)	80.5	77.5	2151

dev and test set. This can be attributed to the fact that we freeze layers before additional training, and the bottleneck characteristics of adapters.

5.2 Analysis

Here we analyze the output of our best model from the Performance section.

We visualize self-attention probabilities of two blocks when question with id 68cf05f67fd29c6f129fe2fb9 is fed into the network. The question ("In what country is Normandy located?") comes first in the input sequence and the paragraph comes second. For conciseness, we have taken the maximum over 12 attention heads. We see that after training, attentions in the last layer are higher for the key word in the question ("Normandy") and around the answer ("France"). WE observe that attentions on later blocks change more drastically.

We also analysis and compare the performance of our final model and the baseline model using the following plots inspired by the paper [5] (the implementation is ours). Figure 6 shows the percentage of the correctly and the incorrectly answered questions that have length at most k.

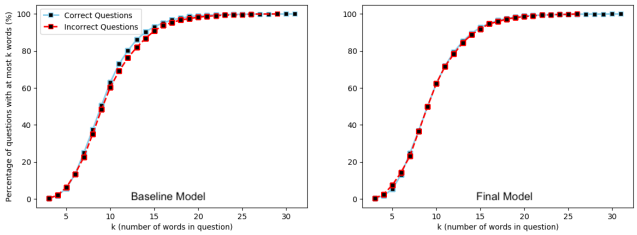

Figure 6: Question Lengths analysis for the baseline model and the final model

We notice that in the first graph, the blue curve is slightly above the red curve for k 18 and equal for $k > 18$. This means that a bigger fraction of the correctly answered questions have short questions than the incorrect ones, which implies that the model performs slightly better on the short questions. We see that this is no longer the case in the second graph as the two curves are equal this means that the model has improved at answering longer questions.

Figure 7 shows the fraction of questions with at least an n-gram match between the question and the the original paragraph for the model's successes and failures. We compare these percentages for the baseline model and the final model and for each value of n.

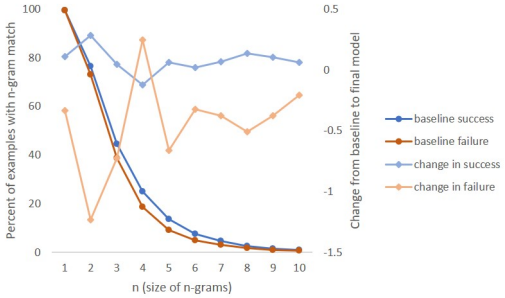

Figure 7: Fraction of n-gram match of question and context for successes and failures

We notice that successes are more likely to have an n-gram match than failures in both our models and we can see that the differences between the percentages of n-gram match for the correctly and the incorrectly answered questions have increased from the baseline model to the final model. This means that the success percentages slightly increased and failure percentages slightly decreased. This implies that our final model is smarter in finding answers to non-trivial questions. i.e the questions that are not very similar to the context. However this improvement is not very significant therefore we can further address this problem in the future.

Lastly, we evaluate our model as a binary classifier for answer/no-answer questions. We report below the confusion matrices and the classification accuracy for the baseline model and the final model.

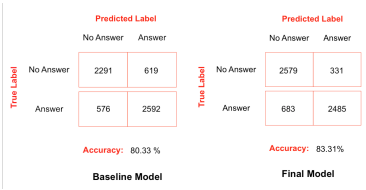

Figure 8: Confusion Matrix and Accuracy

Figure 8 shows that the true negatives have increased and the false positives have decreased however the false negatives have increased and the true positives have decreased. This means that our model improved the no answer predictions but worsened the answers predictions. The overall accuracy of the model has also improved.

6 Conclusion

In this project, we conducted numerous experiments to find a better way of using BERT pretrained weights without having to tune all parameters of the model. We showed that for QA, adapters outperform other approaches such as tuning the last transformer block of BERT or using PALs. We have also provided a method to use adapters to achieve high F1 measure.

One possible future direction is using the very large TriviaQA [6] dataset for pretraining instead of CoQA. The sheer size of this dataset can help the performance. Furthermore, we believe adding appropriate task-specific modules inside BERT can help interpreting its performance on the task. For example, one could add task-specific attentions where we used linear projection, and try to visualize the attention mechanism. This visualization would not be mixed with what BERT attentions have already learned during their pre-training on language modeling tasks and therefore can provide a better understanding of the model.