# BERT Vision

**Siduo Jiang**
UC Berkeley

**William Casey King**
UC Berkeley

**Cristopher Benge**
UC Berkeley

## Abstract

For our purposes, we would like to find ways to substitute expensive BERT fine-tuning with parameter-efficient models by using the information in the hidden state activations of the BERT model, which is discarded during regular BERT inference. As a result, we would like to fine-tune as little as possible, ideally none, before building our models. However, using a variety of parameter-efficient CNNs and dense architectures, we found such models were unable to fit raw BERT hidden state activations without fine-tuning to our specific Q&A task SQuAD 2.0 (see Appendix). This discovery is consistent with the observations made in (Ma et al., 2019), where the authors found that fine-tuned BERT on either SNLI and in-domain text corpus consistently outperformed pre-trained BERT without fine-tuning by a large margin for various tasks, including Q&A (although not on SQuAD). As a result, we decided mild amounts of fine-tuning was necessary in order to generate usable embeddings.

## 1 Introduction

Introduction section.

## 2 Background

Background section

## 3 Methods

This section describes how we use the SQuAD 2.0 question-answering dataset as both span detection and classification, and our custom models for inference on BERT embeddings. We also detail two custom layers implemented for this task.

**SQuAD 2.0 for Span Detection**

**SQuAD 2.0 span detection and how 0,0**
**SQuAD 2.0 as classification task**

**Custom layers (Tenney and Adapter)**
**Data processing details**

In order to establish a baseline for our Q&A task, we fine-tuned BERT for 6 epochs, measuring dev set performance at the end of each epoch. Our setup is similar to Devlin et. al. [?]. For questions that do not have answers, we assign the start and end span of the answer to be at the CLS token (position 0). We used the Adam optimizer with an initial learning rate of 1e-5. One major deviation from Devlin was a result of hardware limitations, which is the batch size. We used a batch size of 8, the maximum we could achieve, compared to a batch size of 48. At inference time, we predict the start and end spans based on the argmax probability at each token position.

For training of our custom models, for a single SQuAD 2.0 example, the data point has a shape of (386,1024,25), where 386 represents the text length dimension, 1024 the BERT embeddings dimension, and 25 the hidden state activations with the final sequence outputs for BERT. We found that for simple models, including the sequence outputs mildly improved performance (see Appendix). The pooling method described in Tenney et. al. [?] can handle data of this shape, reducing the 25 dimension down to 1. Our custom implementation of adapter pooling can also handle 3D inputs by either sharing weights across the encoder dimension, or learning separate weights. As a result, our data can be fed directly into 2D CNNs like regular "images", be directly processed using pooling (reduce 25 to 1) and compression (reduce 1024 to adapter size), or processed by stacking CNNs with pooling and/or compression. We wanted to try all such combinations in order to find

the best architecture. For all models, we train for a single epoch for three reasons: 1. Performance was already desirable at a single epoch. 2. Further training typically did not help performance (see Appendix), 3. Due to our data management issue, loading the data usually takes more than 90% of the training time, which makes it impractical to train for extended numbers of epochs (See Data Processing section).

In order to process the data further, we explored two pooling techniques to reduce the dimensionality of our encoder dimension from 26 to 1, before stacking our 1xN CNN model on top.

### 3.1 Pooling

This was suggested by (Ma et al., 2019) to perform significantly better than max pooling. Although Ma et. al. only present results averaging the first and last layers of the encoders, the authors state that averaging all layers mildly improved performance, which is the approach we take here.

### 3.2 Adapters

The reason we wanted to evaluate 386 is that such a transformation results in embeddings with a square shape (386 x 386), which is the preferred input format for CNNs in computer vision. For SQuAD 1.1, Houlsby et. al. found that an adapter size of 64 provides the best F1 score when used in between transformer blocks. For our purposes, we tried adapter sizes of 64, and 386. For both adapter sizes of 64 and 386, we evaluated whether sharing parameters across each encoder layer would result in reasonable performance.
Next, we evaluated the possibility of using adapters first proposed by Houlsby et. al. This is desirable as sharing weights greatly reduces the number of parameters, by approximately 24x. Furthermore, with minimal parameter penalty, we also evaluated adding a skip connection between the final transformed adapter and the BERT input sequence used for inference.

We found that for both sizes of 64 and 386, and for both shared and unshared weights, adding a skip connection mildly helps performance. For an adapter size of 64, sharing weights across each encoder does not significantly degrade performance. We achieve an EM of 0.680 and F1 of 0.732 with shared weights, and an EM of 0.689 and F1 of 0.733 with unshared weights. Surprisingly, for an adapter size of 386, sharing weights performed better than unshared weights when a skip connection is used, with an EM of 0.686 (compared to 0.676) and F1 of 0.732 (compared to 0.723). While an adapter size of 64 with shared weights performed the best on its own, the mild increase also comes with 4x penalty in the number of parameters. As a result, for downstream tests, we favor an adapter size of 386 with shared weights.

## 4 Results

### 4.1 BERT Fine-Tuning Partial & Full Epochs

In order to establish a baseline performance, we fine-tuned BERT for up to 6 epochs with results shown in Figure [1]. We measure performance for every 10th of a fractional epoch between 0 and 1 epochs, as well as full epochs up to 6. We observed that performanced peaked at 2 epochs, achieving an Exact Match (EM) of 0.747, and an F1 score of 0.792. Between 0 and 1 epochs, performance consistently increased both in terms of EM and F1. For comparison with other published works, see Appendix.

In order to select where to extract our hidden state activations, we chose 2 time points to balance training time and model performance. The first is a full training epoch. On our GPU, fine-tuning for 1 epoch is expensive and takes around 5 hours, but performance is already relatively decent compared to peak performance at 2 epochs. This time point gives us high quality embeddings for model development. The second time point is 3/10 of an epoch, which takes about 1.5hr to fine-tune. While cheaper, at this time point, the model has yet to achieve the large jump in performance observed between 3/10 and 4/10 of an epoch. This provides us an opportunity for improvement using our models.

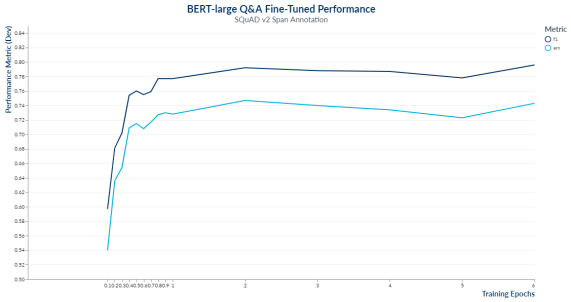


Figure 1: QnA Performance, BERT SQuAD 2.0

### 4.2 Models trained at 3/10 epoch BERT embeddings

Using the embeddings at 3/10 of epoch, we explored over 20 parameter-efficient models using a combination of CNNs, pooling and compression techniques. Table [?] shows the performance of our most promising models, and results of all models are in the Appendix.

First, we compared two pooling strategies. Our LP (learned pooling) model is identical to the pooling model outlined in Tenney et. al. [?], where weights for combining each encoder layer is learned. The AP (average pooling) model proposed by Ma et. al. [?] on the other hand weights each encoder layer equally. We found that an LP model achieved similar performance as BERT itself, while average pooling significantly decreased performance. An analysis of the learned weights suggests that a non-uniform distribution of pooling weights is optimal, with the distribution slightly favoring later layers of BERT compared to earlier layers (see Appendix).

Next, we evaluated the possibility of using adapters modified from Houlsby et. al (see Methods). We found that our modified adapters of all flavors improved model performance compared to baseline BERT. Our best model uses an adapter of size of 386 with weight sharing enabled for each encoder layer, achieving an EM of 0.699 and F1 of 0.740. A skip connection between the final encoder layer and our model's penultimate hidden layer is also included (see Figure [?]). Without weight-sharing, the number of parameters increases by about 24x, and model performance slightly decreases. While Houlsby et. al. found that an adapter size of 64 provides the best F1 score when used in between transformer blocks, for our purposes, a size of 64 performed worse with or without weight-sharing. Our best adapter model significantly beats BERT fine-tuned to the same level of 3/10 of an epoch by about 4 percentage points in both EM and F1, although we failed to reach performance at 1 full epoch.

While we extensively explored stacking pooling with adapters and CNNs, we did not find a model which performed better than our best adapter model. For example, Table [1] shows a model where we stacked learned pooling with our best adapter model and abbreviated Xception network. The number of parameters increased by 16x, but performance was no better than pooling alone.

| **Model** (% params of $BERT_{LARGE}$) | **EM** $\frac{3}{10}e$ | **F1** $\frac{3}{10}e$ | **EM** $1e$ | **F1** $1e$ |
|---|---|---|---|---|
| BERT $\frac{3}{10}e$ $(100\%)$ | 0.654 | 0.702 | 0.728 | 0.777 |
| BERT $1e$ $(100\%)$ | 0.670 | 0.712 | 0.728 | 0.777 |
| learned pooling $(0.001\%)$ | 0.660 | 0.703 | 0.725 | 0.759 |
| average pooling $(0.001\%)$ | 0.657 | 0.700 | 0.736 | 0.741 |
| adapter shared 386 skip $(0.124\%)$ | **0.699** | **0.740** | **0.745** | **0.791** |
| adapter independent 386 $(2.957\%)$ | 0.676 | 0.723 | 0.730 | 0.778 |
| adapter shared 64 $(0.021\%)$ | 0.680 | 0.732 | 0.722 | 0.761 |
| adapter independent 64$(0.491\%)$ | 0.684 | 0.716 | 0.736 | 0.782 |
| pooling adapter 386 & xception $(1.970\%)$ | 0.668 | 0.711 | 0.699 | 0.751 |

Table 1: Models at $\frac{3}{10}$ epochs but evaluated at 1 epoch

### 4.3 Best Models on top at 1 epoch

Using the embeddings at 1 epoch of fine-tuning, we trained the same set of 20 models as at 3/10 of an epoch of fine-tuning (see the Appendix for a full set of results). In most cases, we found that our models outperformed BERT at 1 epoch of fine-tuning. One exception is our adapter 386 model with independent weights, which suffered about 1 percentage point drop. Table [**??**] shows that adapter models of size 64, with or without weight-sharing, both beat BERT, but the best model found is almost identical to that found at 3/10 of an epoch utilizing an adapter size of 386. The only exception that a skip connection is not needed in this case. This model's performance is also competitive with maximum performance achieved with BERT fine-tuning, with a slightly improved EM (0.749 vs 0.747) and a slightly decreased F1 (0.790 vs 0.792). This result shows that we are able to reduce BERT fine-tuning by 1 epoch and still achieve similar performance with a model of only about 0.124% of the number of BERT parameters.

### 4.4 Performance of models trained at 3/10 epoch on higher quality embeddings

We wanted to better understand why models trained on embeddings derived at 1 epoch of fine-tuning performed better than models trained

| Model | % params of $BERT_{LARGE}$ | EM | F1 |
|---|---|---|---|
| BERT $1e$ | 100% | 0.728 | 0.777 |
| BERT $2e$ (*best*) | 100% | 0.747 | 0.792 |
| adapter shared 386 | 0.124% | 0.749 | 0.790 |
| adapter independent 386 | 2.957% | **0.716** | **0.767** |
| adapter shared 64 | 0.021% | 0.739 | 0.785 |
| adapter independent 64 | 0.491% | 0.736 | 0.784 |
| tenney adapter 386 & xception | 1.970% | 0.741 | 0.786 |

Table 2: Dev set performance

on embeddings derived at 3/10 of an epoch. In order to do this, we took all of our models trained on the 3/10 embeddings, and evaluated their performance on the dev set with embeddings derived at 1 epoch. To our surprise, all models experienced a significant jump in performance compared to evaluation at embeddings derived at 3/10 of an epoch. Figure [?] shows a comparison of these results on our top models, and Appendix S[?] contains data for all models.

For example, our best model trained at 3/10 of an epoch achieved a performance of 0.745 EM and 0.791 F1 when evaluated on embeddings from 1 epoch. This is a significant boost compared to evaluating the exact same model on embeddings derived from 3/10 of an epoch (0.699 EM and 0.740EM). This model performs similarly to our best model trained on 1 epoch embeddings, and by a transitive relation, also performs similarly to maximum BERT performance at 2 epochs. We consider this a significant achievement for a model that has only seen data from 3/10 of an epoch of fine-tuning.

Our result suggests that models trained at 3/10 of an epoch are learning transformations that are transferable even as BERT is further fine-tuned. This implies that BERT is maintaining some level of constancy in the structure of its internal representations. Practically, this means that our models can be trained in parallel with BERT fine-tuning. As BERT learns better internal representations of the data, models trained on earlier representations can leverage this improvement and achieve better performance on the task on hand.

### 4.5 Training with less data

The previous sections show that our models perform well compared to BERT when trained on the full dataset. However, since supervised data can be difficult to obtain for certain tasks, we wanted to explore whether our models can perform well with significantly less data. In addition in our case, since embeddings can be expensive to calculate and store, reducing training data also helps with our data management issue (see Methods). To answer this question, we trained our best models (for both 3/10 epoch and 1 epoch embeddings) on varying amounts of data, ranging from 0% to 100% of the dataset at step sizes of 10%. All models were trained for 1 epoch, the same as using the full dataset. Here, 0% represents randomized weights with no training. For the model at 3/10 of an epoch, we measured its dev set performance on both the 3/10 epoch embeddings and 1 epoch embeddings.

Figure [2] shows the results. In all cases, we rapidly approach strong performance early on, beating BERT at around 30% of the data. Even at 10% of the data, we are only about 3 percentage points in terms of EM, and 2 from F1, from maximum peak performance. This shows that we can in fact achieve strong performance even when using training less.

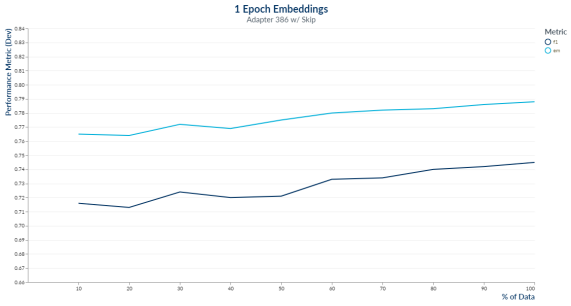


Figure 2: 1 epoch embeddings - adapter 386 w/ skip

## 5 Analysis

Work in Progress

## 6 Future Work

Work in Progress

## Acknowledgments

We are incredibly thankful for the faculty and instructors of W266 : *Natural Language Process-*

| % data | EM | F1 | EM 1$e$ embeddings | F1 1$e$ embeddings |
|---|---|---|---|---|
| 10% | 0.633 | 0.692 | 0.712 | 0.768 |
| 20% | 0.633 | 0.690 | 0.714 | 0.767 |
| 30% | 0.660 | 0.711 | 0.735 | 0.782 |
| 40% | 0.663 | 0.695 | 0.737 | 0.775 |
| 50% | 0.673 | 0.712 | 0.741 | 0.783 |
| 60% | 0.681 | 0.719 | 0.733 | 0.772 |
| 70% | 0.683 | 0.729 | 0.731 | 0.779 |
| 80% | 0.686 | 0.724 | 0.741 | **0.785** |
| 90% | 0.687 | 0.734 | 0.733 | 0.783 |
| 100% | **0.689** | **0.733** | **0.736** | 0.783 |

Table 3: $\frac{3}{10}$ Epochs embeddings - adapter 386

*ing with Deep Learning* for their guidance, patience, and detailed feedback over the course of this project. In particular, we call out special thanks to Daniel Cer, PhD and Joachim Rahmfeld without whom we would not have escaped the tangled maze of BERT hidden activations in time.

## References

Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. 2016. Character-Aware neural language models. In *30th AAAI Conference on Artificial Intelligence, AAAI 2016*.

Girish Limaye, Manish Pandit, and Sawal Vinay. 2019. BertNet: Combining BERT language representation with Attention and CNN for Reading Comprehension.

Xiaofei Ma, Zhiguo Wang, Patrick Ng, Ramesh Nallapati, and Bing Xiang. 2019. Universal text representation from bert: An empirical study.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuad: 100,000+ questions for machine comprehension of text. In *EMNLP 2016 - Conference on Empirical Methods in Natural Language Processing, Proceedings*.

## A Appendices

Appendices are material that can be read, and include lemmas, formulas, proofs, and tables that are not critical to the reading and understanding of the paper. Appendices should be **uploaded as supplementary material** when submitting the paper for review. Upon acceptance, the appendices come after the references, as shown here.

**LaTeX-specific details:** Use `\appendix` before any appendix section to switch the section numbering over to letters.

## B Supplemental Material

Submissions may include non-readable supplementary material used in the work and described in the paper. Any accompanying software and/or data should include licenses and documentation of research review as appropriate. Supplementary material may report preprocessing decisions, model parameters, and other details necessary for the replication of the experiments reported in the paper. Seemingly small preprocessing decisions can sometimes make a large difference in performance, so it is crucial to record such decisions to precisely characterize state-of-the-art methods.

Nonetheless, supplementary material should be supplementary (rather than central) to the paper. **Submissions that misuse the supplementary material may be rejected without review.** Supplementary material may include explanations or details of proofs or derivations that do not fit into the paper, lists of features or feature templates, sample inputs and outputs for a system, pseudocode or source code, and data. (Source code and data should be separate uploads, rather than part of the paper).

The paper should not rely on the supplementary material: while the paper may refer to and cite the supplementary material and the supplementary material will be available to the reviewers, they will not be asked to review the supplementary material.