BERTVision - A Parameter-Efficient Approach for Question Answering

Siduo Jiang
UC Berkeley
siduojiang@berkeley.edu

Cristopher Benge
UC Berkeley
cris.benge@berkeley.edu

William Casey King
UC Berkeley
caseyking@berkeley.edu

Abstract

We present a highly parameter-efficient approach for Question Answering (QA) that significantly reduces the need for extended BERT fine-tuning. Our method uses information from the hidden state activations of each BERT transformer layer, which is discarded during typical BERT inference. Our best model achieves maximal BERT performance at a fraction of the training time and GPU/TPU expense. Performance is further improved by ensembling our model with BERT's predictions. Furthermore, we find that near optimal performance can be achieved for QA span annotation using less training data. Our experiments show that this approach works well not only for span annotation, but also for classification, suggesting that it may be extensible to a wider range of tasks. [1] [2]

1 Introduction

The introduction of Transformers (Vaswani et al., 2017) has significantly advanced the state-of-the-art for many NLP tasks. The most well-known Transformer-based model is BERT (Devlin et al., 2019). The standard way to use BERT on a specific task is to first download pre-trained weights for the model, then fine-tune these weights on a supervised dataset. However, this procedure can be quite slow, and at times prohibitive for those without a powerful GPU/TPU, or those with limited CPU capacity. Smaller Transformers, such as DistilBERT (Sanh et al., 2019), can fine-tune up to 60% faster. However, such models tend to consistently underperform full-size BERT on a wide range of tasks. A method that reduces fine-tuning but maintains the same or better performance would make BERT more accessible for practical applications.

To develop our method, we drew inspiration from previous works that use BERT for feature extraction rather than for fine-tuning (Zhu et al., 2020; Chen et al., 2020). For example, Zhu et al. showed that the sequence outputs from the final BERT layer can be used as contextualized embeddings to supplement the self-attention mechanism in an encoder/decoder neural translation model. This led to an improvement over the standard Transformer model in all tested language pairs on standard metrics (BLEU score (Papineni et al., 2002)).

One characteristic these studies share with typical BERT inference is that only information from the final layer of BERT is used. However, a study by (Tenney et al., 2019a) suggests that all layers of BERT carry unique information. By training a series of classifiers within the edge probing framework (Tenney et al., 2019b), the authors computed how much each layer changes performance on eight labeling tasks, and the expected layer at which the model predicts the correct labels. The discovery is that syntactic information is encoded earlier in the network, while higher level semantic information comes later. Furthermore, classifier performance generally increases for all tasks when more layers are accounted for, starting from layer 0, suggesting that useful information is being incorporated at each progressive layer. Others, such as (van Aken et al., 2019), looked specifically at QA with SQuAD and published similar findings. Their work suggests that different layers encode different information important for QA.

(Ma et al., 2019) showed that a simple averaging of only the first and last layers of BERT results in contextualized embeddings that performed better than only using the final layer. The authors evaluated this approach on a variety of tasks such as text classification and question-answering. Together, these works suggest that the hidden state activations within BERT may contain unique information that can be used to augment the final layer. That said, the exact way of doing this requires further exploration.

[1] *BERTVision* - so named for our method of peering within BERT for the signal hidden therein.

[2] See GitHub repository: BERTVision

In this work, we leverage the findings by Tenney and Ma as inspiration for developing a solution with a two-fold goal: 1. Reduce expensive BERT fine-tuning; 2. Maintain or exceed BERT-level performance in the process. To do so, we extract the information-rich hidden states from each encoder layer and use the full embeddings as training data. We demonstrate that, for two question-answering tasks, even our simple architectures can match BERT performance at a fraction of the fine-tuning cost. Our best model saves on one full epoch of fine-tuning, and performs better than BERT, suggesting that our approach may be a desirable substitute to fine-tuning until convergence.

2 Methods

This section introduces our baseline BERT model, and custom models trained on BERT embeddings. We also describe how we use the SQuAD 2.0 question-answering dataset for both span annotation and classification.

2.1 Modeling approach

Our custom models use the BERT activations from each encoder layer as input data. For a single SQuAD 2.0 example, our data point has a shape of (X,1024, 25), where $X = 386$ for span annotation, and 1 for classification (see appendix [10] for details). We term this representation of the data as *"embeddings."*

2.2 Learned and average pooling

We implemented the pooling method described in (Tenney et al., 2019a), which contracts the last dimension from 25 to 1 through a learned linear combination of each layer. We call this approach learned pooling (LP). We also evaluate the pooling approach reported in (Ma et al., 2019), average pooling (AP), where each encoder layer is given equal weights.

2.3 Adapter compression

We use the term "compression" to refer to methods that reduce the 1024 dimension of the BERT embeddings. The method proposed in (Houlsby et al., 2019) is termed *"adapter,"* which is an auto-encoder type architecture wherein the embeddings are first projected into a lower dimension using a fully-connected layer. For our purposes, the adapter serves as a bridge between BERT embeddings and downstream layers. The architecture in Houlsby et. al. can only handle 3D tensors (including batch), because each adapter handles data from a single transformer layer. Since we work with the activations from multiple layers at once, we need to be able to handle 4D tensors. To do this, our adapter implementation can treat each transformer layer as independent, learning separate compression weights for each layer. Alternatively, we can also employ weight-sharing, so that the compression for each encoder layer follows the same set of transformations.

2.4 Custom CNNs

We also implemented various novel CNN architectures, as well as modified existing models such as Inception and Xception (Szegedy et al., 2014; Chollet, 2016). For span annotation, the key is that the CNN models must preserve the text length dimension of 386 (see appendix [A.9] for rationale). To view a notebook of all models we tried and their summaries, see: BERT Vision GitHub repo.

2.5 Fine-tuning BERT

For all experiments, we use the $BERT_{LARGE}$ uncased implementation from Hugging Face. To establish a baseline for our QA tasks, we fine-tuned BERT for 6 epochs with a setup similar to that described in (Devlin et al., 2019). For QA span annotation, questions that do not have answers have the start and end span positions assigned at the CLS token (position 0). We used the Adam optimizer with an initial learning rate of 1e-5. Due to hardware constraints, we use batch size of 4 rather than 48. At inference time, the most likely complete span is predicted based on the maximum softmax probability of the start- and end-span position. The setup is identical in classification, except that we used the pooled CLS token rather than the sequence outputs.

2.6 Ensembling

We evaluated multiple ensemble approaches for span annotation. The most successful method takes the element-wise max of the softmax probabilities output by each model in the ensemble. Let Z and Y be two model softmax probabilities for start or end span position. Then, the predicted position is:

$$\arg\max\left(\max_{Z_i,Y_i}, \quad \text{for i in } \{1,2,...,386\}\right) \qquad (1)$$

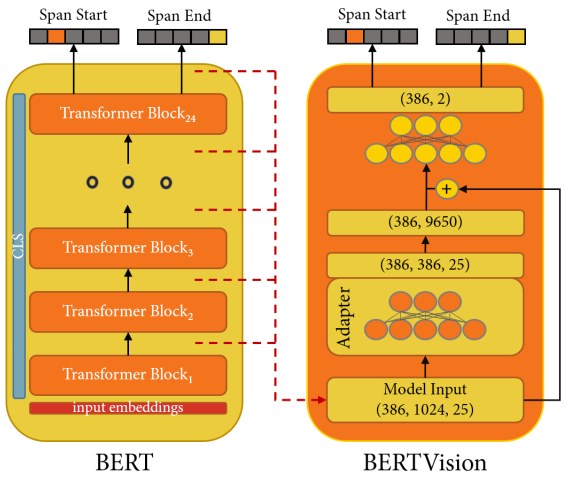


Figure 1: Architecture of our best model in relation to BERT. **Left**: BERT and its span annotation inference process. **Right**: For our model, BERT embeddings are first transformed through our custom adapter layer. Next, the last two dimensions are flattened. Optionally, a skip connection is added between the sequence outputs of the final BERT layer and this flattened representation. This is present in the best model discovered at 3/10 of an epoch, but was not necessary for the best model discovered at 1 full epoch. This tensor is then projected down to (386, 2) with a densely connected layer and split on the last axis into two model heads. These represent the logits of the start-span and end-span position.

2.7 Data processing and evaluation

We use SQuAD 2.0 for our QA dataset. Standard Exact Match (EM) and F1-score (F1) are used for evaluation as outlined in (Rajpurkar et al., 2018). For classification, EM is equivalent to accuracy as we are predicting a single binary outcome. For our BERT models, we restrict the maximum token length to 386 (See appendix [A.8] for rationale). Question-context pairs that exceed this maximum sequence are split into multiple segments (as many as needed) with an overlap between each segment of 128 tokens. For the splits that do not contain the answer, the labels for that split are set to "no answer." At inference time, we take the argmax of the span probabilities predicted by each split as the final prediction for the example.

3 Results

3.1 Span Annotation

For span annotation, we sought to generate useful embeddings with as little fine-tuning as possible. However, we were unable to find models that can fit BERT embeddings without some fine-tuning (see appendix [A.12] for details). As a result, we decided that a small amount of fine-tuning would be necessary.

3.1.1 BERT fine-tuning as baseline

To establish a baseline, we fine-tuned BERT for up to 6 epochs with results shown in Figure [A.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 performance 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 as measured by both EM and F1. (For comparison with other published works, see appendix [A.10]) For span detection we extracted embeddings at 3/10 of an epoch and at one full epoch. (for a rationale for this decision see appendix [A.7])

3.1.2 Models trained at 3/10 epoch embeddings

Using the embeddings at 3/10 of an epoch, we explored over 20 parameter-efficient models using a combination of CNNs, pooling, and compression techniques. [3] Table [1] shows that our best models outperform baseline BERT fine-tuned to 3/10 of an epoch. Results of all models are in appendix [A.2].

First, we compare two models that use different pooling strategies, LP (learned pooling) and AP (average pooling) [2]. We found that the AP model achieved similar performance as BERT itself, while LP improved performance. An analysis of the learned weights suggests that a non-uniform distribution of pooling weights is optimal, with the distribution favoring later layers of BERT compared to earlier layers (see appendix [A.14]).

Next, we evaluated models leveraging adapters (see Methods). We found that our modified adapters of all flavors improved model performance compared to baseline BERT at 3/10 of an epoch, with our best model improving EM by 4.5 percentage points, and F1 by 3.8. As indicated in Table [1], without weight-sharing, the number of parameters increases by ∼24x, with a penalty to model performance, making weight-sharing superior in every respect. In addition, (Houlsby et al., 2019) found that an adapter size of 64 provides the best F1-score when used between transformer blocks.

[3] See appendix [A.6] for training time and strategy

Our modified implementation did not perform well at this size with or without weight-sharing.

While we extensively explored stacking pooling with adapters and CNNs, we did not find a model which performed better than our best adapter model. Table [1] shows the results, and the appendix [A.2] contains the rest. For one model where we stacked a modified Xception network, the number of parameters increased by 16x, but performance was no better than pooling alone.

Model	% Params BERT$_{large}$	SQuAD2.0 EM	SQuAD2.0 F1
BERT $\frac{3}{10}e$	100%	0.654	0.702
learned pooling (LP)	0.001%	0.675	0.721
average pooling (AP)	0.001%	0.657	0.700
adapter size 386	**0.124%**	**0.699**	**0.740**
- weights not shared	2.957%	0.676	0.723
adapter size 64	0.021%	0.680	0.732
- weights not shared	0.491%	0.684	0.716
LP adapter size 386 & xception	1.970%	0.668	0.711

Table 1: Models trained on embeddings at $\frac{3}{10}$ epochs

3.1.3 Best models on top at 1 epoch

Using 1 epoch embeddings as our training data, we fit the same set of models as with the 3/10 epoch embeddings (see the appendix [A.2] for full results). In most cases, we found that our models outperformed BERT at the same level of fine-tuning. The best model is nearly identical to that found with 3/10 epoch embeddings (see table [2]), outperforming BERT by 2.1 percentage points in EM, and 1.3 in F1. This model's performance is also competitive with maximum performance achieved with BERT on this task, which is BERT fine-tuned to 2 epochs. This result shows that we are able to reduce BERT fine-tuning by 1 epoch and still achieve comparable performance with a model of only about 0.124% of the number of BERT parameters. The implications of our findings suggest that near-optimal BERT performance can be achieved in a fraction of the training time and GPU/TPU expense.

3.1.4 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, we explore whether the same performance can be achieved with significantly less data. To answer this question, we trained our best models on varying amounts of data, ranging from

Model	% Params BERT$_{large}$	SQuAD2.0 EM	SQuAD2.0 F1
BERT $1e$	100%	0.728	0.777
BERT $2e$ (*best*)	100%	0.747	0.792
adapter size 386	**0.124%**	**0.749**	**0.790**
- weights not shared	2.957%	0.716	0.767
adapter size 64	0.021%	0.739	0.785
- weights not shared	0.491%	0.736	0.784
LP adapter size 386 & xception	1.970%	0.741	0.786

Table 2: Models trained on embeddings at 1 epoch

0% to 100%. As with the full dataset, all models were trained for 1 epoch. Figure [2] shows the results. In both cases, we rapidly approach strong performance early, beating BERT at ∼30% of the data. Even by 10%, we approach peak performance. This shows that we can achieve strong performance even when using less training data.

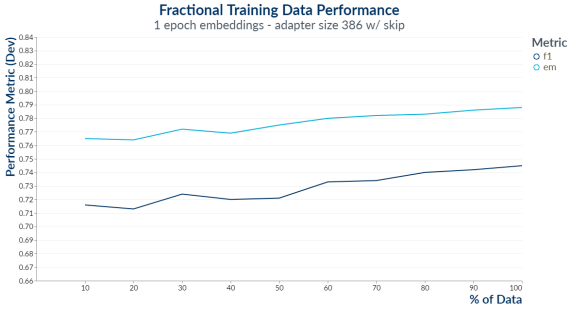


Figure 2: Training with less data

3.1.5 Ensembling our models with BERT

Since our approach requires fine-tuning BERT in order to derive workable embeddings, the BERT predictions come for free. Thus, we can ensemble our models with BERT predictions with no additional training required. Table [3] presents the results of this exercise. Ensembling the models at 3/10 of an epoch did not lead to an improvement, but ensembling our 1 epoch model with BERT at 1 epoch led to more than a half percentage point improvement. This suggests that our model and BERT supply complementary information that together leads to better predictions.

3.2 Classification

The previous section on QA span annotation uses the full sequence embeddings generated by BERT. Another common way to apply BERT is to text classification, which only uses the CLS token. Our goal in this section is to explore the efficacy of leveraging the CLS token hidden state activations

Model	SQuAD2.0 EM	SQuAD2.0 F1
BERT $\frac{3}{10}e$	0.654	0.702
our model $\frac{3}{10}e$	0.699	0.740
BERT $1e$	0.728	0.777
our model $1e$	0.749	0.790
ensemble BERT+our model $\frac{3}{10}e$	0.691	0.734
ensemble BERT+our model $1e$	**0.756**	**0.798**

Table 3: QA ensembling results

in an analogous manner to our approach with the span annotation task.

3.2.1 BERT fine-tuning as baseline

Similar to QA span annotation, we use BERT fine-tuning as a baseline. We fine-tuned BERT for 6 epochs using the CLS token, with the EM and F1 for each epoch shown in Figure [5]. Our best performance was achieved at 3 epochs. We utilized embeddings at 2/10 of an epoch and 1 full epoch for the binary classification task (see appendix [A.12] for rationale).

3.2.2 Models trained at 2/10 epoch embeddings

At 2/10ths of an epoch, BERT achieved an F1 of 0.635 and EM of 0.687 [5]. We tried training various parameter-efficient model architectures similar to span annotation, most of which were CNN-based or simple linear. Our best performing model leverages a simple linear weighting that contracts across the channels dimension through learned weights inspired by (Tenney et al., 2019a). We trained for 10 epochs and recorded performance of the model at each epoch evaluated against the dev set. At every epoch, the model outperformed the BERT baseline at 2/10 of an epoch [3].

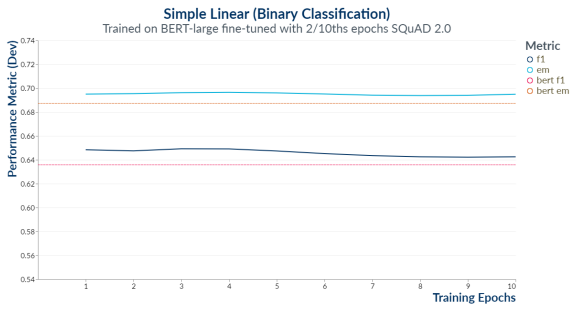


Figure 3: 2/10ths epoch model vs BERT

3.2.3 Models trained at 1 epoch embeddings

With 1 epoch embeddings, our results show significant improvement over the 2/10ths epoch models. Surprisingly, the best results were achieved at 1 epoch of training with the simple linear model consisting of only 0.001% of BERT's parameters. This model outperforms BERT at both 1 and 2 epochs of fine-tuning. This implies that our parameter efficient model can save on 1 full epoch of BERT fine-tuning. However, performance does fall short of maximal BERT performance achieved at 3 epochs, and we were unable to find a model architecture that achieves this [4]. Nevertheless, this suggests that with text classification, we can use the CLS token hidden state activations in the same way we used the full sequence hidden state activations in the span annotation task.

e	BERT F1	BERT EM	Our F1	Our EM	F1 Δ	EM Δ
1	0.720	0.761	0.763	0.782	**+0.043**	**+0.021**
3	0.790	0.804	0.795	0.811	**+0.005**	**+0.007**

Table 4: Comparison of BERT and our models performance at 1 and 3 epochs on binary classification task

3.2.4 Modeling with 3 full epoch embeddings

BERT achieved its best performance against the SQuAD 2.0 binary classification task at 3 full epochs (0.79 F1, 0.804 EM) [5]. Given that the simpler linear model outperformed BERT at the same level of fine-tuning, we wanted to evaluate if the simple model's trend in performance would continue as BERT exhausts its ability to learn the binary classification task. In this final evaluation, the linear model again outperformed BERT, achieving the top overall score for our task, albeit at a smaller margin [4].

4 Model analysis

4.1 Question types

Model performance on SQuAD 2.0 can be categorized based on whether the question-context pair has an answer (Has Answer versus No Answer). The dev set is very balanced in this sense as it contains 5,928 (∼49.93%) pairs with answers and 5,945 (∼50.07%) pairs without answers. Figure [4] shows the fraction of questions of each type that BERT and our best-performing models correctly answers. As BERT is fine-tuned, the number of correctly answered questions in both categories increases, but questions with "No Answer" increases more rapidly. Our best models answer more "No Answer" questions correctly than BERT, but un-

derperform BERT in terms of "Has Answer" questions.

To investigate further, we looked at our 3/10 epoch model's incorrect predictions on "Has Answer" questions, and found that a large percentage, ∼65%, were predicted as having no answer (rather than with the wrong answer). These results suggest that our models may be more liberal than BERT at predicting "no answer."

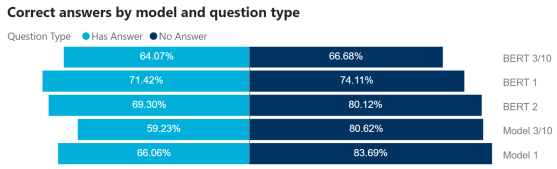


Figure 4: Correctly answered percentages by model

4.2 Second most likely answer

For questions where our 3/10 epoch model's most likely answer was incorrect, we considered the second most likely answer predicted by the model. We found that the second most likely answer is correct 44% of the time when the most likely answer is incorrect. We also note 47% of these were also "Has Answer" questions, which is a more balanced distribution compared to the most likely answer (where only 42% of correctly answered questions have an answer). This suggests that even though our model at 3/10 is liberal at predicting no answer, its second most likely answer is quite frequently correct and less biased.

4.3 Context Length

The previous section shows that our models are better at recognizing questions with no answers compared to BERT. Here, we look at which models perform better on longer sequences. At 3/10 of an epoch of fine-tuning, BERT and our models performed similarly (BERT average length: 165.86, our model average length: 164.50). However, we believe this is because both models answered many of the same questions correctly. When looking at questions that were uniquely answered correctly by each model, a clear difference emerges: the average sequence length for BERT is longer (BERT: 175.30, our model: 161.58). This suggests that our models are better at answering questions with shorter contexts than BERT. The same trend holds at 1 epoch of fine-tuning.

4.4 Do 2 epochs perform even better?

Our previous results show that models built on BERT embeddings at 1 epoch achieves maximal BERT performance at 2 epochs. Here, we investigate if we can further improve performance by training directly on embeddings derived at 2 epochs. While this approach does not reduce BERT fine-tuning time, it does provide a way to investigate whether our approach can achieve even better performance with longer fine-tuning.

Table [5] shows the results. As expected, the 2-epoch models outperform the same models trained on 1-epoch embeddings. Since ensembling improved performance at 1 epoch, we applied the same approach to the models at 2 epochs. Surprisingly, the 2 epoch ensemble does not outperform the 1 epoch ensemble in terms of either EM (0.749 vs 0.756) or F1 (0.795 vs 0.798). This suggests that additional fine-tuning does not guarantee better performance. Our 1-epoch models, in addition with ensembling, not only saves on BERT fine-tuning time, but also achieves the best possible performance, suggesting a total lack of need for fine-tuning beyond 1 epoch.

Model	**SQuAD2.0**	
	EM	**F1**
BERT $2e$ (*best*)	0.747	0.792
best model architecture from $1e$	0.753	0.797
best model architecture from $\frac{3}{10}e$	0.749	0.795
ensemble BERT $2e$ + best $1e$	0.751	0.796

Table 5: Models trained on embeddings at 2 epochs

5 Conclusion and future work

In this paper, we propose a parameter-efficient approach that achieves maximal BERT performance for QA span annotation and greatly reduces fine-tuning time required for BERT. Our models are trained on BERT hidden state activations (embeddings), and consistently outperform BERT at the same level of fine-tuning. By using an ensemble of our model with BERT's predictions, we further surpass BERT performance, reducing the need for fine-tuning by 1 epoch. We achieved similarly promising results for QA classification, which suggests that this approach works well with both the full sequence BERT embeddings and with the CLS token embedding. Future work might look at reducing fine-tuning even further and applying this approach to other NLP tasks.

Acknowledgments

We are incredibly thankful for the faculty and instructors of W266: *Natural Language Processing 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, PhD without whom we would not have escaped the tangled maze of BERT hidden activations in time.

References

Betty van Aken, Benjamin Winter, Alexander Löser, and Felix A. Gers. 2019. How does BERT answer questions? *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*.

Zhiyu Chen, Mohamed Trabelsi, Jeff Heflin, Yinan Xu, and Brian D. Davison. 2020. Table search using a deep contextualized language model. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*.

François Chollet. 2016. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357.

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*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. *CoRR*, abs/1902.00751.

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

Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. pages 311–318.

Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *CoRR*, abs/1806.03822.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going deeper with convolutions. *CoRR*, abs/1409.4842.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019a. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601, Florence, Italy. Association for Computational Linguistics.

Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R. Thomas McCoy, Najoung Kim, Benjamin Van Durme, Samuel R. Bowman, Dipanjan Das, and Ellie Pavlick. 2019b. What do you learn from context? Probing for sentence structure in contextualized word representations. *CoRR*, abs/1905.06316.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

Jinhua Zhu, Yingce Xia, Lijun Wu, Di He, Tao Qin, Wengang Zhou, Houqiang Li, and Tie-Yan Liu. 2020. Incorporating BERT into neural machine translation. *ArXiv*, abs/2002.06823.

A Appendices

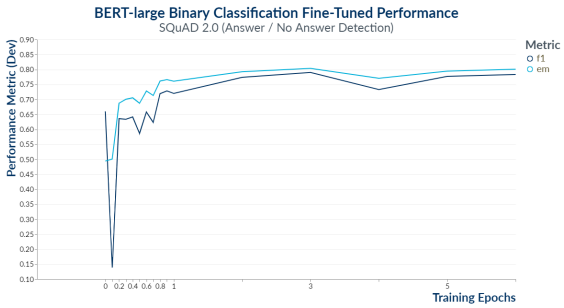
A.1 BERT fine-tuning figures

BERT$_{large}$ was fine-tuned for both span annotation and classification tasks; below are tables and blots indicating the performance for each task.

Model : BERT-large
Task : Binary Classification (SQuAD v2 - Answer / No Answer)
Parameters : 335,141,888

Baseline	Condition	Epochs	EM	F1	Time to Train (sec)
	BERT-large (no tuning)	0	0.4942306073	0.6599467694	n/a
	BERT-large fine-tuning	1/10	0.5008843595	0.1389131067	1421
X X	BERT-large fine-tuning	2/10	0.6871894214	0.6357395057	2840
	BERT-large fine-tuning	3/10	0.7004969258	0.6337796087	4258
	BERT-large fine-tuning	4/10	0.7052977344	0.6414591659	5676
	BERT-large fine-tuning	5/10	0.6871051967	0.5860724234	7096
	BERT-large fine-tuning	6/10	0.7279541818	0.6579839051	8514
	BERT-large fine-tuning	7/10	0.7129621831	0.6230922362	9932
	BERT-large fine-tuning	8/10	0.7612229428	0.7192791365	11350
	BERT-large fine-tuning	9/10	0.7659395267	0.7282151589	12769
X X	BERT-large fine-tuning	1	0.7609702687	0.7201735358	14105
	BERT-large fine-tuning	2	0.7925545355	0.7736421285	28293
	BERT-large fine-tuning	3	0.8035879727	0.7900990099	42478
	BERT-large fine-tuning	4	0.7704876611	0.7327645386	56665
	BERT-large fine-tuning	5	0.7944917039	0.7768836869	70864
	BERT-large fine-tuning	6	0.8006401078	0.7831027215	85054

(a) BERT$_{large}$ fine-tuning table : classification



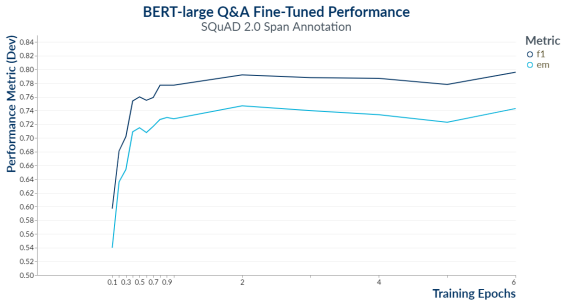
(b) BERT$_{large}$ fine-tuning plot : classification

Figure 5: BERT$_{large}$ fine-tuning for classification

Model : BERT-large
Task : Span Annotation (SQuAD v2 - Start/End Span Positions)
Parameters : 335,143,938

Baseline	Condition	Epochs	EM	F1	Time to Train (sec)
	BERT-large (no tuning)	0	0.036	0.051	n/a
	BERT-large fine-tuning	1/10	0.728	0.777	1375
	BERT-large fine-tuning	2/10	0.747	0.792	2750
X X	BERT-large fine-tuning	3/10	0.74	0.788	4125
	BERT-large fine-tuning	4/10	0.734	0.787	5500
	BERT-large fine-tuning	5/10	0.723	0.778	6875
	BERT-large fine-tuning	6/10	0.743	0.796	8250
	BERT-large fine-tuning	7/10	0.54	0.597	9625
	BERT-large fine-tuning	8/10	0.636	0.681	11000
	BERT-large fine-tuning	9/10	0.654	0.702	12375
X X	BERT-large fine-tuning	1	0.709	0.754	13750
	BERT-large fine-tuning	2	0.715	0.76	27537
	BERT-large fine-tuning	3	0.708	0.755	41324
	BERT-large fine-tuning	4	0.717	0.759	55111
	BERT-large fine-tuning	5	0.727	0.777	68898
	BERT-large fine-tuning	6	0.73	0.777	82685

(a) BERT$_{large}$ fine-tuning table : span annotation



(b) BERT$_{large}$ fine-tuning plot : span annotation

Figure 6: BERT$_{large}$ fine-tuning for span annotation

A.2 Table of all model results for span annotation

Models were trained on embeddings derived from 3/10 of an epoch and 1 full epoch. Performance on evaluation of the dev set is presented below.

Model Description	% BERT Params	EM $\frac{3}{10}$	F1 $\frac{3}{10}$	EM 1	F1 1
1xN CNN	0.099	0.673	0.720	0.745	0.789
1xN CNNs with dilation	0.121	0.655	0.701	N/A	N/A
1xN CNN without dropout	0.099	0.670	0.717	0.744	0.789
Learned Pooling	0.001	0.670	0.712	0.731	0.766
Learned Pooling with 1xN CNN	0.093	0.675	0.721	0.745	0.789
Average Pooling with 1xN CNN	0.093	0.657	0.700	0.742	0.786
Adapter 64 shared weights	0.021	0.677	0.730	0.739	0.785
Adapter 64 shared weights with skip	0.021	0.680	0.732	0.742	0.787
Adapter 64 not shared weights	0.491	0.684	0.716	0.736	0.782
Adapter 64 not shared weights with skip	0.491	0.689	0.733	0.736	0.784
Adapter 386 shared weights	0.124	0.660	0.703	0.749	0.790
Adapter 386 shared weights with skip	0.124	0.699	0.740	0.74	0.786
Adapter 386 not shared weights	2.957	0.676	0.723	0.714	0.769
Adapter 386 not shared weights with skip	2.958	0.676	0.723	0.716	0.767
Learned Pooling, Adapter 386 shared	0.119	0.691	0.734	0.741	0.786
Adapter 386 shared skip Tenney Pooling	0.242	0.691	0.734	0.745	0.789
Learned pooling, adapter with 1xn CNN	0.207	0.670	0.720	0.748	0.789
Learned pooling, adapter with NxN CNN	0.332	0.690	0.731	0.747	0.786
Learned pooling, adapter, Inception	1.994	0.501	0.501	N/A	N/A
Learned pooling, adapter, XCeptionMod	1.970	0.668	0.711	0.741	0.786

Figure 7: BERT$_{large}$ All model performance : span annotation

A.3 Best model performance for classification task

The best model evaluated for the classification task was a simple linear model that performed channel contraction using learned weights followed by a simple dense layer connection with no softmax. As indicated below, this model outperformed BERT$_{large}$ for almost every metric save for F1 on 3 epoch fine-tuned embeddings.

Model : Tenney Small
Task : Binary Classification (SQuAD v2 - Answer / No Answer)
Parameters : 2,077

v BERT EM	v BERT F1	Data	Epochs	EM	F1	Time to Train (sec)
1	1	2/10th BERT embeddings	1	0.6950223195	0.6484124672	19
1	1	2/10th BERT embeddings	2	0.6953592184	0.6475002436	38
1	1	2/10th BERT embeddings	3	0.6962014655	0.6491586422	57
1	1	2/10th BERT embeddings	4	0.6965383644	0.6491381829	76
1	1	2/10th BERT embeddings	5	0.6960330161	0.6473864191	95
1	1	2/10th BERT embeddings	6	0.6951065443	0.6452371619	115
1	1	2/10th BERT embeddings	7	0.6940958477	0.6435021594	136
1	1	2/10th BERT embeddings	8	0.6938431736	0.6425410562	159
1	1	2/10th BERT embeddings	9	0.694011623	0.6422451994	180
1	1	2/10th BERT embeddings	10	0.6948538701	0.6424553439	201
1	1	1 epoch BERT embeddings	1	0.781605323	0.763261207	20
1	1	1 epoch BERT embeddings	2	0.7799208288	0.7604730039	40
1	1	1 epoch BERT embeddings	3	0.780426177	0.7608914978	60
1	1	1 epoch BERT embeddings	4	0.7805104018	0.7608735548	80
1	1	1 epoch BERT embeddings	5	0.7803419523	0.7603381731	101
1	1	1 epoch BERT embeddings	6	0.780426177	0.7602758621	122
1	1	1 epoch BERT embeddings	7	0.7806788512	0.7601768281	143
1	1	1 epoch BERT embeddings	8	0.7806788512	0.7598229109	164
1	1	1 epoch BERT embeddings	9	0.7809315253	0.7600332134	185
1	1	1 epoch BERT embeddings	10	0.7812684242	0.7602695468	205
1	1	3 epoch BERT embeddings	1	0.811083972	0.7947286538	20
1	1	3 epoch BERT embeddings	2	0.8098206014	0.7922723091	40
1	0	3 epoch BERT embeddings	3	0.8080518824	0.7896631287	60
1	0	3 epoch BERT embeddings	4	0.8073780847	0.788456202	80
1	0	3 epoch BERT embeddings	5	0.8065358376	0.7869795048	101
1	0	3 epoch BERT embeddings	6	0.8059462646	0.7860326895	122
1	0	3 epoch BERT embeddings	7	0.8055251411	0.7853091585	143
1	0	3 epoch BERT embeddings	8	0.8056093658	0.7853023256	164
1	0	3 epoch BERT embeddings	9	0.8049355681	0.7845180499	185
1	0	3 epoch BERT embeddings	10	0.8045986692	0.7840655249	205

Figure 8: BERT$_{large}$ Best model performance : classification task

A.4 Table of all model results for classification

Models were trained on BERT$_{large}$ fine-tuned embeddings derived from 2/10 of an epoch and 1 full epoch. Additionally, the most successful model was trained on 3 epochs fine-tuned embeddings. Performance on evaluation of the dev set is presented below.

Model	% Params BERT$_{large}$	SQuAD2.0 EM	SQuAD2.0 F1
adapter pooler tenney	0.119%	0.685	0.646
xception abbr	0.079%	0.673	0.702
xception	**6.181%**	**0.707**	**0.710**
xception abbr cls	0.080%	0.672	0.564
adapter pooler avg	0.119%	0.691	0.624
tenney small	0.001%	0.697	0.650

Table 6: Models trained on embeddings at $\frac{2}{10}e$

Model	% Params BERT$_{large}$	SQuAD2.0 EM	SQuAD2.0 F1
adapter pooler tenney	0.119%	0.781	0.765
xception abbr	**0.079%**	**0.785**	**0.782**
xception	6.181%	0.783	0.766
xception abbr cls	0.080%	0.783	0.780
adapter pooler avg	0.119%	0.779	0.765
tenney small	0.001%	0.782	0.763

Table 7: Models trained on embeddings at $1e$

Model	% Params BERT$_{large}$	SQuAD2.0 EM	SQuAD2.0 F1
tenney small	0.001%	0.811	0.795

Table 8: Models trained on embeddings at $3e$

A.5 Data challenges and training time

During training, in order to obtain the internal BERT embeddings for each example (which is the input "data" into our downstream models), we had to either: 1. Pre-generate the embeddings, or 2. Generate the embeddings for each example on the fly from a frozen BERT model. Due to the size of BERT, we quickly ran out of GPU memory with the second method, so we had to resort to the first. Working with BERT embeddings on the SQuAD 2.0 data set presented a data engineering problem due to the size of the data. Using the 4-byte float representation, the entire dataset with each example having shape (386, 1024, 25) is approximately 5 TB in size, which is far too large to store into memory on our hardware. Instead, this data was written to disk, and a custom Keras data generator used to retrieve this data in batch sizes of 16 for training (in a shuffled random order).

While this method presents no cost to accuracy, the I/O time for loading the data was significant, much longer in most cases than the actual training time for fitting any of our models. For all models, training per epoch was around 5 hours, but we estimate that on average over 95% of the time is spent on loading data, and only 5% of the time fitting the model. As a result, in this work, in order to separate out infrastructure issues with model training, as a proxy for training cost, we compare the number of parameters in our model rather than wall clock training time. We also explore the potential to use less embedding data during fitting, and for future work, a faster storage system should be explored in order to advance this work for practical applications.

We also note that for the classification task, we did not experience any data management issues. Since we only use the CLS token for classification rather than the entire sequence length of 386, the full embeddings of all examples were a little over 13GB, which easily fit into CPU memory. Training for each epoch for on the order of minutes rather than hours.

A.6 Model training strategy

We train all of these models for a single epoch for three reasons: 1. Performance was already desirable at a single epoch. 2. Further training typically did not help performance further, 3. Due to our data management issue, loading the data usually takes more than 95% of the training time, which makes it difficult to train for extended numbers of epochs (See [A.5]).

A.7 Explanation on the shape of input embeddings data

For span annotation, for a single SQuAD 2.0 example, the data point has a shape of (386,1024,25). Here, 386 represents the text length dimension, and 1024 the BERT embedding dimension for each encoder layer. The 25 comes from the stacking of the contextual wordpiece (Wu et al., 2016) text embeddings, the 23 hidden state activations, and the final sequence outputs (24th layer) for BERT.

For classification, an example has input shape (1,1024, 26). Here, 1 represents the lone CLS token, 1024 the BERT embedding dimension. The 26 comes from the stacking of the contextual wordpiece text embeddings, 23 hidden state activations, the final CLS token for the output encoder layer (24th layer), and the pooled CLS token from the final pooler layer.

A.8 Explanation on max sequence length input into BERT

BERT$_{large}$ has a maximum input sequence length of 512. While we could have truncated our question-context pairs at this length, we found that a large majority of examples were much shorter than this length. The average length of the input is about 171 tokens long. The length we chose was 386, which is between 98-99th percentile. As a result, without much loss, we can significantly save on the number of mostly meaningless "[PAD]" tokens we needed to store, especially for full 25-layers of BERT embeddings.

A.9 CNN models preserving max sequence length

Our CNN models were all designed to preserve the sequence length of the input question-context pair. This was achieved with 1xN convolutions so that these filters only compress along the 1024 dimension, or with NxN convolutions with padding along the sequence token dimension. In the case of 1xN convolutions, this is similar to a unigram model that treats each token separately. The NxN convolutions were n-gram models, ranging from 1 - 7 (depending on the size of N). The reason we did this is because we also tried convolution blocks from Inception [1] that gradually shrinks the 386 dimension; however, this model failed to learn to even fit the training data. As a result, we believe that for span annotation, since the data starts with a 386 dimension representing token position, and ends by predicting a probability for each position, we need this sentence length dimension throughout the entire model. A traditional computer vision CNN model such as Inception shrinks the image along this dimension, which destroys the

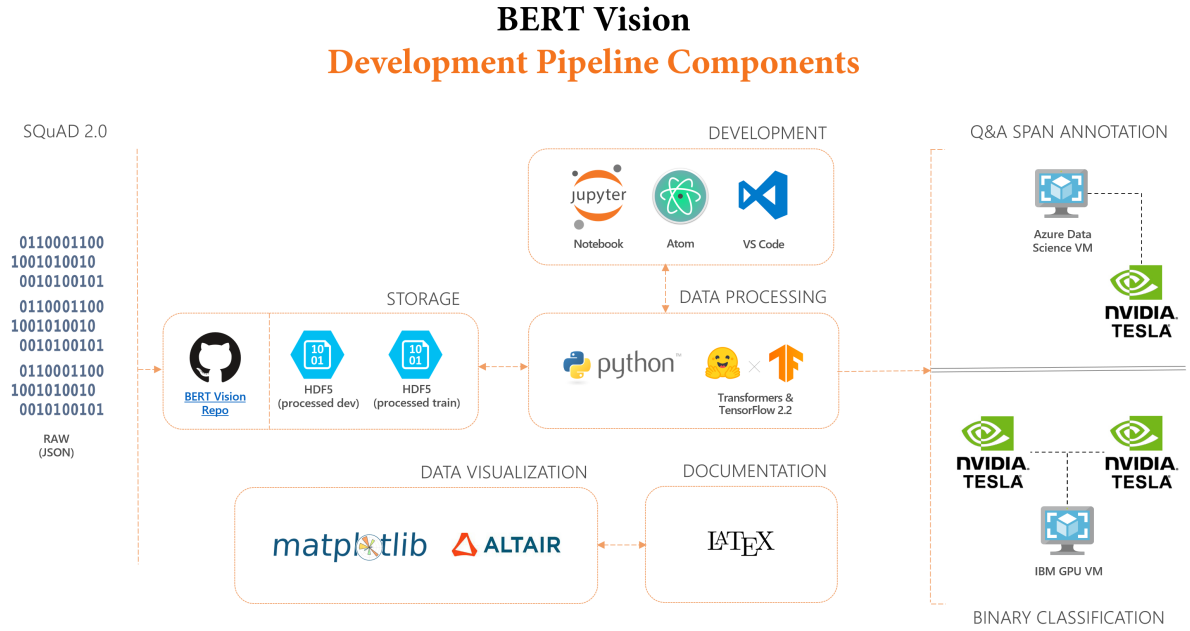


Figure 9: BERTVision development pipeline

structure necessary for span detection. Shrinking the text length and later expanding it again loses information, resulting in a failure to fit the data.

A.10 Comparison of our BERT performance with Devlin

For our fine-tuning procedure, we observed that performance peaked at 2 epochs, achieving an Exact Match (EM) of 0.747, and an F1 score of 0.792. This is slightly worse than that reported by (Devlin et al., 2019) at an EM of 78.7 and F1 of 81.9. We hypothesize that these difference might arise due to our training batch size, random initializations, and the fact we do not favor the null answer by a "θ" threshold, where "θ" was optimized based on dev set performance (see Section 4.3 in Devlin). We use purely the softmax probabilities output by the model without favoring no answer by an optimized threshold based on the dev set itself.

A.11 Where to exact embeddings

We wished to extract embeddings at two stages in the training process: 1. Early-on so that BERT fine-tuning is cheap, and the embeddings are amenable to use as data for modeling, 2. At a slightly later stage before convergence so that our models have a chance to achieve or outcompete the best performing BERT model.

For span detection, our learning curve for full-epoch BERT fine-tuning shows that 1 epoch is relatively decent compared to peak performance at 2 epochs, which gives us a chance to outperform our best observed BERT performance (goal 2). At the same time, fine-tuning for a full epoch on all of SQuAD 2.0 takes around 5 hours, which is already very expensive. To this end, we explored BERT performance every 1/10 of a fractional epoch between 0 and 1. We see that both EM and F1 are increasing steadily in our single run, with a large jump between fractional epochs 3 and 4. In addition to 1 epoch, we extracted embeddings at 3/10 of an epoch immediately before this jump, which took approximately 1.5 hours of fine-tuning. We believe this is a decent spot because such partial fine-tuning is relatively cheap, and performance has yet to jump, giving us an opportunity to improve performance using our models (goal 1).

For classification, looking at our learning curve for full-epoch fine-tuning, 1 epoch is yet again a clear candidate for extracting the embeddings (goal 2). Performance is already relatively decent compared to peak performance at 3 epochs, but takes $\frac{1}{3}$ time and compute resources to train. For goal 1, we extracted embeddings at 2/10 of a fractional epoch between 0 and 1. We see a clear jump in performance between 1/10 of a fractional epoch and 2/10, a jump we do not observe again until 8/10. Therefore, from the dev set perspective, 2/10 is a high value fractional epoch, whether 3/10 - 7/10 does not add much to the performance of the model.

A.12 Need to fine-tune

Using a variety of parameter-efficient CNNs and dense architectures, we found such models were unable to fit BERT hidden state activations without fine-tuning to our specific QA task SQuAD 2.0. 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 QA (although not on SQuAD). As a result, we decided mild amounts of fine-tuning was necessary in order to generate usable embeddings.

A.13 Training hardware and development pipeline

All span annotation training and inferencing was performed on a Microsoft Azure data science virtual machine with 112GiB onboard ram and a single NVIDIA Tesla TITAN series V100 Tensor Core GPU capable of 7 TFLOPS double-precision and tensor performance of 112 TFLOPS and possessed 16 GiB onboard RAM. All binary classification training and experimentation was performed on an IBM virtual machine equipped with two NVIDIA Tesla TITAN series V100 Tensor Core GPU's (see [9]).

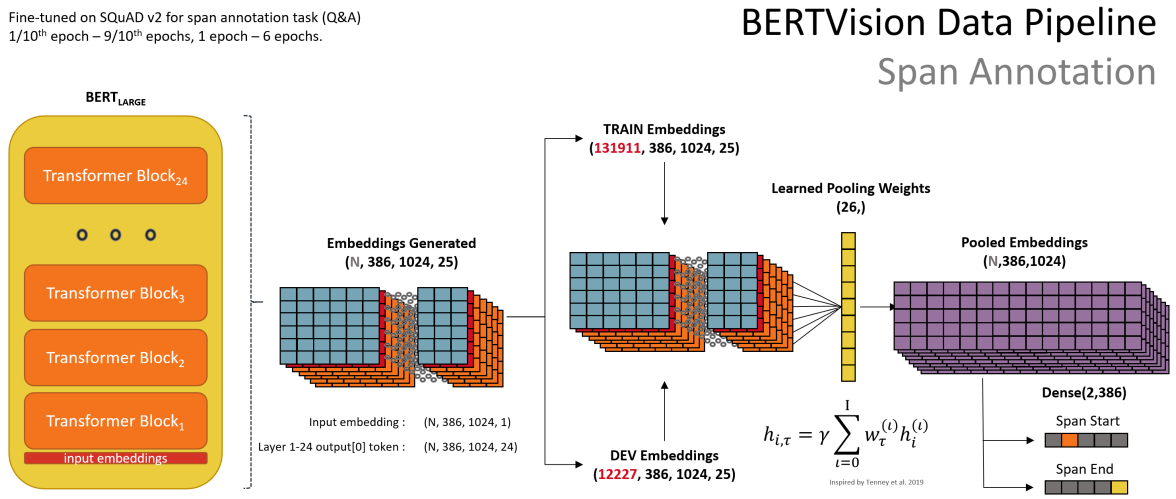


Figure 10: BERTVision span annotation data pipeline

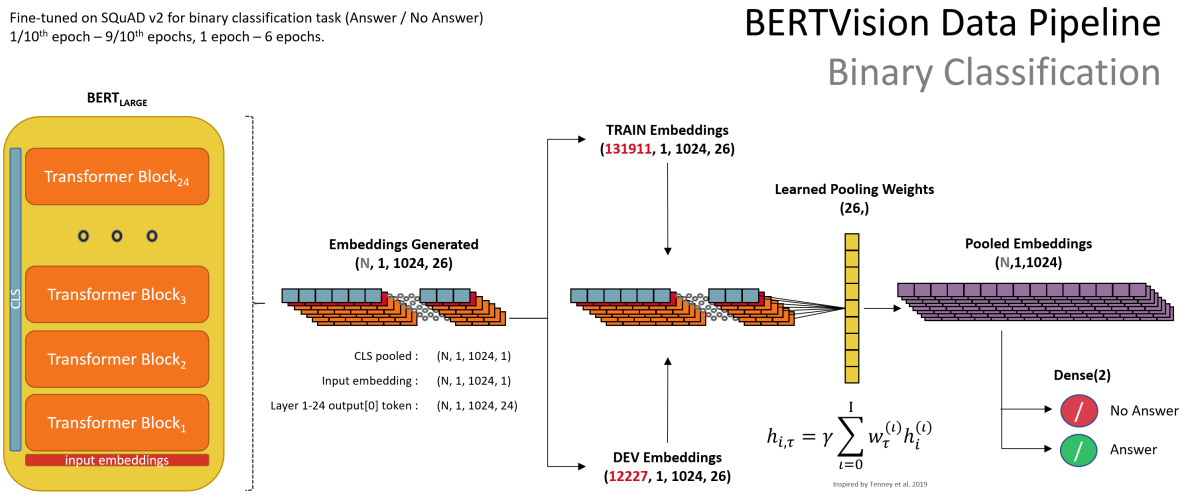


Figure 11: BERTVision classification data pipeline

A.14 Non-uniform weights

Learned weights for learning pooling using the approach described in (Tenney et al., 2019a). The model favors using later layers especially after layer 17. This is consistent with the observation in (van Aken et al., 2019) that the last layers of BERT-base are much more accurate at supporting fact identification (which is the author's proxy for span identification).