Finetuning a multitask BERT for downstream tasks

Stanford CS224N Default Project

Chenchen Gu
Department of Computer Science
Stanford University
cygu@stanford.edu

Abstract

Multitask models are of interest and importance in NLP because of their ability to handle a variety of different tasks. In particular, Transformer-based models such as BERT are able to generate useful sentence-level representations, leading to strong performance on many downstream tasks. There are many proposed methods for using BERT and its embeddings for downstream tasks. However, there has been little direct comparison between these methods, holding constant BERT's model architecture. To address this gap, we perform experiments to find effective methods for building and finetuning a multitask BERT that simultaneously performs well on multiple tasks. We find that one-size-fits-all approach is not optimal—different tasks have different methods which work best. Among the methods we test, we find that cosine similarity works best for semantic textual similarity, while sentence concatenation works best for paraphrase detection. Also, we find that gradient surgery and model ensembling do not deliver significant performance gains, suggesting that training a multitask BERT may be a "natural" multitask learning problem, with few conflicting gradients between tasks.

1 Key Information to include

- Mentor: Sauren Khosla
- External Collaborators (if you have any): None
- Sharing project: No

2 Introduction

In recent years, Transformer-based models, initially proposed by Vaswani et al. (2017), have become dominant in natural language processing (NLP) research. In particular, BERT (Bidirectional Encoder Representations from Transformers), developed by Devlin et al. (2018), is one of the most well-known and widely-studied Transformer-based models. BERT generates useful representations, or embeddings, of words and sequences, which can be used for downstream tasks. However, there are many possible methods for extracting and using BERT's embeddings for downstream tasks. In the original BERT paper, Devlin et al. (2018) used a particular set of methods to use BERT for downstream tasks, but since then, much research has gone into other methods and extensions for improving BERT's performance on downstream tasks.

In this project, we investigate methods of using and finetuning BERT to achieve high performance on downstream tasks. Specifically, we aim to build a multitask BERT model that simultaneously performs well on several downstream tasks. Multitask models are important and interesting because they are more versatile and useful than single-task models, and performing well on multiple tasks suggests that the model has learned richer and more accurate representations of natural language.

Our main approach is to run experiments on different methods for performing different downstream tasks. We run these experiments by finetuning a pretrained BERT model on the downstream tasks.

Our key finding is that a one-size-fits-all approach is not optimal. Different tasks have different methods which work best to achieve high performance using BERT.

3 Related Work

First, we review existing work about finetuning and using BERT for downstream tasks. At a high level, the method used by Devlin et al. (2018) in the original BERT paper is simply feeding the input sequence into the BERT model. For single sentence tasks, they directly pass the sentences into BERT. For sentence-pair tasks, they concatenate the sentences, separated by the special `[SEP]` separator token, then pass this combined sequence into BERT. They also add a learned embedding to each token indicating which sentence it belongs to. After passing the input sequence through BERT, they obtain the final hidden vector of the special `[CLS]` (classification) token as a sentence-level representation for downstream tasks. Note that `[CLS]` is automatically prepended to the start of every BERT input. They then use simple linear transformations to obtain the appropriate output for the tasks.

Since the original BERT paper, there have been many other methods for performing downstream tasks. For example, Stickland and Murray (2019) use an additional linear layer and nonlinearity when processing the final hidden vector of the `[CLS]` token. They apply a $d \times d$ linear transformation, where d is the size of the hidden states (768 in BERT), followed by a nonlinearity, then a final linear transformation to the appropriate output for the task. Note that this is not the main aspect of their work—they propose projected attention layers to train a multitask BERT.

Reimers and Gurevych (2019) propose using cosine similarity for sentence comparison tasks, such as semantic similarity. For sentence-pair tasks, instead of concatenating the two sentences before passing into BERT, they pass the two sentences into BERT separately, then compute the cosine similarity between the outputs as a metric for semantic similarity. They also find that using the mean of all of the hidden vectors in BERT's final layer results in slightly better performance than using the last hidden vector of just `[CLS]`. Their approach matches the performance of the original BERT model (Devlin et al., 2018), but is significantly more efficient due to using cosine similarity.

There has also been research into different methods for simultaneously training a model on multiple tasks. A simple approach used by Bi et al. (2022) is to add the loss functions and gradients for each task. However, this additive approach may run into difficulties when gradients for different tasks conflict with each other, i.e. they point in different directions. To combat this issue, Yu et al. (2020) propose PCGrad, a method of gradient surgery which projects task gradients onto the normal plane of a conflicting gradient from another task. They show that on difficult multitask learning problems, gradient surgery improves performance and efficiency.

One limitation with this existing work is that it does not focus on finding optimal methods for using BERT for downstream tasks. Most of these papers do not explicitly experiment with different methods for extracting sentence-level representations from BERT or for using these representations for downstream tasks. These papers usually propose and focus on some key modification or extension to model architecture, so performance differences between models are likely primarily due to architecture differences, rather than differences in methods for using BERT for downstream tasks.

We seek to address this gap in knowledge by performing thorough experimentation on different methods for finetuning and using BERT for multiple downstream tasks. We keep the BERT model architecture constant for these experiments, so performance differences can directly be attributed to differences in methods for using BERT for downstream tasks.

4 Approach

For this project, we use the BERT model architecture, as described by Devlin et al. (2018) and the default project handout[1]. The fundamental building block of BERT is the self-attention based Transformer layer, first introduced by Vaswani et al. (2017). BERT consists of 12 Transformer encoder layers, which each consist of multi-head attention, an additive and normalization (add-and-norm) layer with a residual connection, a feed-forward layer, and a final add-and-norm layer with a residual connection. More details can be found in the default project handout, including relevant figures and equations.

[1] http://web.stanford.edu/class/cs224n/project/final-report-instructions-2023.pdf

To train our models, we use the AdamW optimizer as described by Kingma and Ba (2014), with the addition of decoupled weight decay regularization described by Loshchilov and Hutter (2017) (hence the W in AdamW). The AdamW optimizer is a variant of gradient descent, using adaptive learning rates for different parameters based on estimates of the first and second moments of the gradient, which are obtained via exponential moving averages of the gradient and element-wise squared gradient respectively. To implement BERT and AdamW, we completed the partial implementations provided in the default project starter code[2], which also provides minimalist starter code for using and training BERT for our downstream tasks of interest.

Next, we briefly describe the three downstream tasks we use in this project: sentiment classification, paraphrase detection, and semantic textual similarity (STS). Sentiment classification is identifying the polarity of the opinion expressed in a sentence. We use a classification task, namely assigning one of five sentiment labels. Paraphrase detection is determining whether two sentences are paraphrases of each other. We use a binary classification task, where sentence pairs are either paraphrases or not paraphrases. STS is determining the degree of semantic similarity between two sentences. We use a regression task, so the similarity value is a numeric value from 0 to 5.

4.1 Baseline model

Next, we implemented a simple multitask BERT using the initial approaches suggested in the default project handout, which serves as a baseline for comparison with our later extensions and experimentation. For this baseline, for the downstream tasks, we pass the input sentences through BERT and obtain the final hidden vectors of `[CLS]` as output. For the sentence pair tasks, we separately pass the two input sentences through BERT, then concatenate the outputs. Then, we apply dropout and a simple linear transformation layer with learnable parameters that maps the BERT output into logits appropriate for the task.

For the sentiment classification task, there is one logit for each label. The logits are converted to probabilities via softmax, and the logit with the highest probability is the outputted label. We use cross entropy loss on the unnormalized logits. For the paraphrase detection binary classification task, there is only one logit. We pass the logit through the sigmoid function and round it to either 0 or 1 as the outputted binary label. We use binary cross entropy loss. For the semantic similarity regression task, there is only one logit, whose value is directly taken as the outputted similarity score. We use the mean squared error (MSE) loss against the true similarity score.

To finetune this baseline model, we use a simple round-robin training approach, inspired by the multitask learning used by Bi et al. (2022). In each epoch, we independently train on each of the three tasks/datasets and make gradient updates for each using the appropriate loss function. Since we are finetuning, all parameters of the model are updated, including the linear layers and the BERT parameters. We initialize the BERT parameters to the pretrained bert-base-uncased[3] weights, which we use for all experiments in this project.

4.2 Extension 1: cosine similarity for semantic textual similarity

As our first extension, we experiment with using cosine similarity for the semantic textual similarity (STS) regression task, as described by Reimers and Gurevych (2019). For this method, we first separately pass the two sentences through the same BERT model. Denote the BERT output vectors for the two sentences as x, y. We compute the cosine similarity between x and y, which measures the similarity between the directions the vectors are pointing in, as given by the following equation[4]:

$$\text{cosine similarity} = \frac{x \cdot y}{||x|| \cdot ||y||} \tag{1}$$

where $x \cdot y$ denotes the dot product between x and y, and $||x||$ denotes the Euclidean norm of x. The value of cosine similarity ranges from -1 to 1. However, the cosine similarity between two BERT outputs for real sentence pairs almost always falls between 0 and 1 (Gao et al., 2022). So, to project cosine similarity onto our STS scores, which range from 0 to 5, we first apply ReLU to

[2]https://github.com/gpoesia/minbert-default-final-project

[3]https://huggingface.co/bert-base-uncased

[4]See https://pytorch.org/docs/stable/generated/torch.nn.CosineSimilarity.html. As an implementation detail, the denominator is converted to a small constant if $||x|| \cdot ||y|| = 0$, to prevent division by zero.

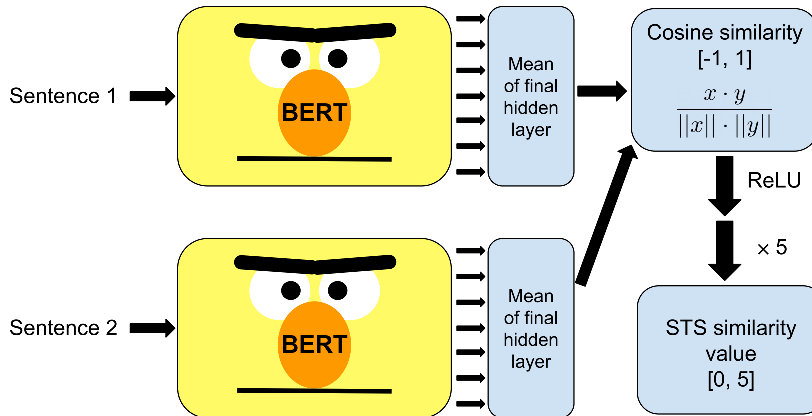

Figure 1: Cosine similarity method for STS, using "MEAN" pooling

the cosine similarity to clamp it between 0 and 1, then multiply by 5 to project to the range $[0, 5]$. As in the baseline model, we use the mean squared error loss against the true similarity score. We also experiment with two different pooling strategies to obtain the outputs x, y from the hidden states of the final BERT layer. The "CLS" pooling strategy is to take the final hidden vector of the `[CLS]` token, as we used in the baseline model. The "MEAN" pooling strategy is to take the element-wise mean of all the hidden states in the final layer of BERT.

4.3 Extension 2: cosine similarity, concatenation with [SEP] for paraphrase detection

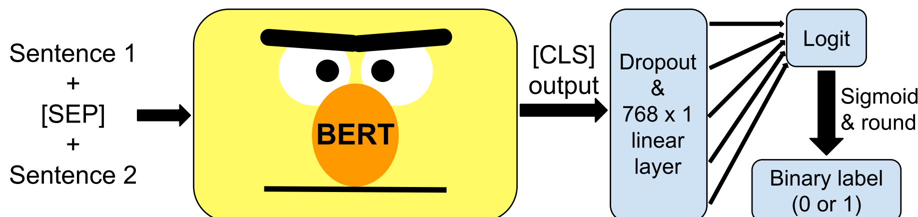

Figure 2: Concatenation w/ [SEP] method for paraphrase detection, using "CLS" pooling and "1-layer" classifier

Next, we experiment with different methods for performing the paraphrase detection binary classification task. Because paraphrase detection is very similar to the STS task, one approach we take is to also use cosine similarity for paraphrase detection, as described above for the STS task. We pass both sentences through BERT, then compute the cosine similarity between the output vectors. We again experiment with the "CLS" and "MEAN" pooling strategies for obtaining the output vectors. However, since paraphrase detection is a binary classification task, we output either 1 (paraphrase) or 0 (not paraphrase) depending on if the cosine similarity is above a certain threshold. We experiment with threshold values. We again use the mean squared error loss between the cosine similarity and the true label.

For paraphrase detection, we also experiment with the approach for sentence-pair tasks used by Devlin et al. (2018), as described in the related work section. We refer to this method as "concatenation with [SEP]". We concatenate the two sentences, separated by the special `[SEP]` token, then pass the combined sequence into BERT. However, unlike Devlin et al. (2018), we do not add learned segment embeddings indicating whether each token is in the first or second sentence. To obtain the sentence-level representation vector from BERT, we again experiment with the "CLS" and "MEAN" pooling strategies. Then, we apply learnable layers to project the BERT output into a single logit. This logit is then passed through the sigmoid function and rounded to 0 or 1 as the outputted label. We experiment with two classifier layer configurations. "1-layer" classifier consists of dropout, then

a 768×1 linear transformation to the logit. "2-layer" classifier consists of dropout, then a 768×768 linear transformation followed by the GELU activation function to an hidden layer of size 768, then a final 768×1 linear transformation from the hidden layer to the output logit.

4.4 Extension 3: PCGrad gradient surgery, ensembling for multitask training

We experiment with different methods for training our model on multiple tasks and datasets. Our baseline method is the round-robin training described in §4.1, which consists of independently training and making gradient updates on each task. However, this simple additive approach may run into difficulty if the gradients for different tasks conflict with each other. We experiment with projecting conflicting gradients (PCGrad) (Yu et al., 2020), a method of gradient surgery which seeks to address this by projecting gradients onto the normal plane of a conflicting gradient. Specifically, if $\mathbf{g}_i$ and $\mathbf{g}_j$ are the gradients for two different tasks, and the cosine similarity between $\mathbf{g}_i$ and $\mathbf{g}_j$ is negative, indicating a conflicting gradient, PCGrad replaces $\mathbf{g}_i$ with its projection onto the normal plane of $\mathbf{g}_j$:

$$\mathbf{g}_i := \mathbf{g}_i - \frac{\mathbf{g}_i \cdot \mathbf{g}_j}{||\mathbf{g}_j||^2}\mathbf{g}_j \tag{2}$$

To use gradient surgery, we use an open-source reimplementation[5] of PCGrad by Tseng (2020). We also experiment with ensembling to address conflicting gradients, where we use separate BERT models with independent weights for each task. In ensembling, no tasks will share any parameters, so there cannot be any conflicting gradients.

5 Experiments

In this section, we describe the experiments we performed using the different approaches we described in the previous section.

5.1 Data

As described earlier, the three downstream tasks we use are sentiment classification, paraphrase detection, and semantic textual similarity. Here are the datasets we use.

1. Stanford Sentiment Treebank (SST)[6]: We use SST for sentiment classification. SST consists of 11,855 single sentences from movie reviews. Each sentence has one of five labels: 0 (negative), 1 (somewhat negative), 2 (neutral), 3 (somewhat positive), and 4 (positive).
2. Quora Question Pairs (QQP)[7]: We use QQP for paraphrase detection. QQP consists of 400,000 question pairs. Each pair has a binary label indicating whether the two questions are paraphrases of each other.
3. SemEval STS Benchmark (Agirre et al., 2013): We use the SemEval STS dataset for semantic textual similarity. The SemEval STS dataset consists of 8,628 sentence pairs of varying semantic similarity on a scale from 0 (unrelated) to 5 (same meaning). The similarity scores have a granularity of 0.2, i.e. the possible similarity scores are 0, 0.2, 0.4, etc.

Each dataset is split into train, dev, and test portions. More details and examples from these datasets can be found in the default project handout.

5.2 Evaluation method

For sentiment classification on SST and paraphrase detection on QQP, our evaluation metric is raw accuracy, because these are classification tasks. For the semantic similarity regression task on the SemEval STS dataset, our evaluation metric is the Pearson correlation score of the true similarity values against the predicted similarity values. For the experiments we run, we calculate these metrics across the dev portions of the datasets. To evaluate our final model, we use these metrics across the test datasets.

[5] https://github.com/WeiChengTseng/Pytorch-PCGrad
[6] https://nlp.stanford.edu/sentiment/treebank.html
[7] https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs

5.3 Experimental details

For our experiments, we initialize BERT to the pretrained bert-base-uncased[8] parameters, then finetune each model using the AdamW optimizer with hyperparameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. During finetuning, all parameters are updated, including BERT. Except where stated otherwise, we use the round-robin training method as described in §4.1 on the train portions of the three datasets. Since the QQP training dataset is significantly larger than the other two, in each epoch, we only train on 1,000 randomly selected batches from QQP. This is both to ensure the model trains on each dataset equally, and so training runs in a reasonable amount of time. Unless stated otherwise, we use the following training hyperparameters: learning rate of 2e-5, 10 epochs, batch size of 8, and hidden dropout probability of 0.3. Each epoch took around 4 minutes to train on 8 AWS EC2 g5.2xlarge vCPUs, so training took around 40 minutes total.

5.4 Experiment 1: cosine similarity for STS

We experiment with cosine similarity for STS as described in §4.2, using both "CLS" and "MEAN" pooling. We add these methods to the baseline model, perform finetuning as described in §5.3, and observe the results on the dev portions of the datasets, shown in Table 1.

Method	SST	QQP	**STS**	Average score
Baseline	0.507	0.732	0.390	0.543
Cosine similarity, "CLS" pooling	0.500	0.728	0.762	0.663
Cosine similarity, "MEAN" pooling	0.512	0.745	**0.876**	0.711

Table 1: Experiment 1 results, on the dev portions of the datasets. We focus on the STS Pearson correlation, since we are experimenting with the cosine similarity method for the STS task.

We find that cosine similarity using "MEAN" pooling, as illustrated in Figure **??**, obtains the best results on the STS task. This is consistent with the findings of Reimers and Gurevych (2019), as discussed earlier.

5.5 Experiment 2: cosine similarity, concatenation with [SEP] for paraphrase detection

We experiment with the cosine similarity and concatenation with [SEP] methods for paraphrase detection, as described in §4.3. We experiment with "CLS" and "MEAN" pooling, different classification thresholds for cosine similarity, and the "1-layer" and "2-layer" classifier configurations for concatenation with [SEP]. We add these methods to the baseline model, with the addition of cosine similarity with "MEAN" pooling for STS, perform finetuning as described in §5.3, and observe the results on the dev portions of the datasets, shown in Table 2.

Method	SST	**QQP**	STS	Average score
Baseline	0.512	0.745	0.876	0.711
Cosine sim., "CLS" pooling, 0.5 threshold	0.493	0.733	0.874	0.700
Cosine sim., "MEAN" pooling, 0.5 threshold	0.504	0.777	0.875	0.719
Cosine sim., "CLS" pooling, 0.75 threshold	0.474	0.798	0.869	0.714
Cosine sim., "MEAN" pooling, 0.75 threshold	0.513	**0.811**	0.867	0.730
Concat. w/ [SEP], "CLS" pooling, 1-layer	0.478	**0.856**	0.871	0.735
Concat. w/ [SEP], "MEAN" pooling, 1-layer	0.495	0.839	0.870	0.735
Concat. w/ [SEP], "CLS" pooling, 2-layer	0.510	0.849	0.868	0.742
Concat. w/ [SEP], "MEAN" pooling, 2-layer	0.507	0.839	0.868	0.738

Table 2: Experiment 2 results, on the dev portions of the datasets. We focus on the QQP accuracy, since we are experimenting with methods for the paraphrase detection task.

We find that using cosine similarity with an appropriate classification threshold for paraphrase detection obtains better accuracy on QQP than the baseline. However, concatenation with [SEP] for

[8]https://huggingface.co/bert-base-uncased

paraphrase detection obtains the best results. Concatenation with [SEP] using "CLS" pooling and the "1-layer" classifier, as illustrated in Figure 2, obtains the best QQP accuracy, but the different configurations of concatenation with [SEP] all obtain very similar results.

5.6 Experiment 3: PCGrad gradient surgery, ensembling for multitask training

We experiment with the different methods for multitask training as described in §4.4, namely round-robin training (which we have been using thus far), PCGrad gradient surgery, and ensembling. We use these methods to train the same model, which uses the best methods we found in experiments 1 and 2, namely cosine similarity with "MEAN" pooling for STS and concatenation with [SEP] using "CLS" pooling and a 1-layer classifier for paraphrase detection. We observe the results in Table 3.

Method	SST	QQP	STS	Average score
Round-robin training (baseline)	0.478	**0.856**	0.871	0.735
PCGrad gradient surgery	**0.515**	0.850	**0.873**	**0.746**
Ensembling	0.501	0.835	0.867	0.734

Table 3: Experiment 3 results, on the dev portions of the datasets.

We find that all three multitask training methods result in similar performance. PCGrad achieves a slightly higher average score, almost entirely due to a higher SST accuracy. This may be the result of successfully dealing with conflicting gradients, but we note that the SST accuracy tends to fluctuate from epoch to epoch in our experiments, so this slight difference may be due to random chance. So, we do not observe significant difference in performance between the multitask training methods.

This is slightly surprising, since Yu et al. (2020) find that PCGrad significantly improves performance on difficult multitask training problems, and ensembling appears to be more powerful due to having separate BERT parameters for each task. Perhaps our multitask BERT is not a difficult multitask training problem, so gradients for the different tasks tend to point in the same direction. So PCGrad would not significantly improve performance due to conflicting gradients being rare, and ensembling would not significantly improve performance if all the tasks can already perform well using the same BERT parameters.

5.7 Final model: training on entire dataset

As discussed in §5.3, in our experiments, we do not train on the entire QQP dataset. For our final model and test set leaderboard submission, to further boost performance, we train on the entire QQP training set each epoch. We use round-robin training and the best methods we found in experiments 1 and 2, namely cosine similarity with "MEAN" pooling for STS and concatenation with [SEP] using "CLS" pooling and a 1-layer classifier for paraphrase detection. We still use the baseline method for sentiment classification. Table 4 shows our final model's performance on the *test* sets.

	SST	QQP	STS	Average score
Final model	**0.515**	**0.891**	**0.856**	**0.754**

Table 4: Final model performance on the test portions of the datasets. We obtain these results on the test set leaderboard.

6 Analysis

6.1 Ablation study: concatenation *without* [SEP] for paraphrase detection

In experiment 2, we found that concatenation with [SEP] is the best method for paraphrase detection. However, [SEP] is an arbitrary non-word special token, and it might be desirable to have a method which only relies on real worlds. We experiment with the method of direct concatenation for paraphrase detection, which concatenates the two sentences without any separating token. We use the same experimental details as experiment 2. Results are shown in Table 5.

Method	SST	**QQP**	STS	Average score
Direct concatenation, “CLS” pooling, 1-layer	0.501	**0.849**	0.871	0.740
Baseline	0.512	0.745	0.876	0.711
Cosine sim., “MEAN” pooling, 0.75 threshold	0.513	0.811	0.867	0.730
Concat. w/ [SEP], “CLS” pooling, 1-layer	0.478	**0.856**	0.871	0.735

Table 5: Ablation study results on the dev portions of the datasets, using direct concatenation for paraphrase detection. We compare with results from Table 2. We focus on the QQP accuracy.

We find that direct concatenation essentially matches the performance of concatenation with [SEP] for paraphrase detection, and still outperforms the other methods used in experiment 2. This suggests that separating the two sentences with the [SEP] token is not an important aspect of the concatenation method. Perhaps the model can adapt to use some other feature to separate the sentence pair, such as punctuation. Our findings demonstrate the robustness of the concatenation method for paraphrase detection, since it does not rely on the presence of the [SEP] token.

6.2 Error analysis: paraphrase detection

	Overall QQP	Paraphrases	Non-paraphrases
Final model	0.888	0.857	0.906

Table 6: Accuracy of final model on certain subsets of QQP dev set. Paraphrases are sentence pairs with a true label of 1, and non-paraphrases are sentence pairs with a true label of 0.

We perform error analysis of our final model on the QQP dev set. As seen in Table 6, our model performs noticeably worse on true paraphrases than non-paraphrases. Upon inspection, we observe some common features among errors. A common false negative (incorrectly classifying paraphrases as non-paraphrases) is when one question uses “I” and the other uses “you”, e.g. “How do I cure my pimple? / How do you cure pimples?” and “How can I get into WWE? / How do you get into WWE?” This may be hard for the model because changing the subject changes the semantic meaning if you take the question literally, but humans understand that “you” and “I” can be used interchangeably when asking questions. A common pattern among false positives (incorrectly classifying non-paraphrases as paraphrases) is when the two questions are almost identical, except for one or two crucial words. For example, “Are cats narcissists? / Why are cats narcissists?” and “Why do people have dogs? / Why do people like dogs?” These errors may occur if our BERT model fails to pay enough “attention” (in the technical sense of self-attention) to these key words.

7 Conclusion

In this project, we perform experiments to find good methods for building and constructing a multitask BERT that achieves strong performance on downstream tasks. We find that cosine similarity with “MEAN” pooling works well for semantic textual similarity, and sentence concatenation works well for paraphrase detection. We also find that PCGrad gradient surgery and ensembling do not deliver significant performance improvements, suggesting that multitask BERT is not a “difficult” multitask learning problem with many conflicting gradients.

Our findings indicate that a one-size-fits-all approach is not optimal for a multitask BERT—different tasks have different methods which work best. This suggests that models which use the same methods for every downstream task, which many papers use, may have some room for performance improvements, even if they already achieve state-of-the-art results. However, a limitation of our work is that we cannot generalize our claims to what methods work best for tasks other than those we experiment with in this project. An avenue for future work is to determine which methods work best for groups of tasks that share common features, rather than just specific single tasks.

References

Eneko Agirre, Daniel Cer, Mona Diab, Aitor Gonzalez-Agirre, and Weiwei Guo. 2013. *SEM 2013 shared task: Semantic textual similarity. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*, pages 32–43, Atlanta, Georgia, USA. Association for Computational Linguistics.

Qiwei Bi, Jian Li, Lifeng Shang, Xin Jiang, Qun Liu, and Hanfang Yang. 2022. Mtrec: Multi-task learning over bert for news recommendation. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2663–2669.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2022. Simcse: Simple contrastive learning of sentence embeddings.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks.

Asa Cooper Stickland and Iain Murray. 2019. Bert and pals: Projected attention layers for efficient adaptation in multi-task learning.

Wei-Cheng Tseng. 2020. Weichengtseng/pytorch-pcgrad.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. 2020. Gradient surgery for multi-task learning.