Enhancing MinBert Embeddings for Multiple Downstream Tasks

Stanford CS224N Default Project

Donald Stephens
Department of Computer Science
Stanford University
Stanford, CA 94305
`dsteph@stanford.edu`

Abstract

Bidirectional Encoder Representations from Transfers (BERT) is a transformer-based model that generates contextual word embeddings. The model contains 12 Encoder Transformer Layers each consisting of multi-head attention, followed by an additive and normalization layer with a residual connection, a feed-forward layer, and a final additive and normalization layer with a residual connection. I started with a minimal base version of BERT, called minbert, which I completed implementations of the Multi-head Self-Attention and the Transformer Layer, including a portion of the Adam stochastic optimization method. My goal was to make enhancements to minbert to obtain robust and generalizable sentence embeddings that perform well on the following three tasks: sentiment analysis, paraphrase detection and semantic textual similarity. I was able to achieve a 0.520 accuracy, 0.583 accuracy, and 0.528 correlation respectively (overall score of 0.589) on holdout development datasets of the aforementioned tasks.

1 Key Information to include

- Mentor: Rohan Taori
- External Collaborators (if you have any): N/A
- Sharing project: No

2 Introduction

My base model was a minimal implementation of Bidirectional Encoder Representations from Transformers (BERT) Devlin et al. (2018), called minbert. The model contains 12 Encoder Transformer Layers each consisting of multi-head attention, followed by an additive and normalization layer with a residual connection, a feed-forward layer, and a final additive and normalization layer with a residual connection. The transformer architecture is based on work from "Attention Is All You Need" Vaswani et al. (2017).

I completed implementations of the Multi-head Self-Attention and the Transformer Layer, including a portion of the Adam stochastic optimization method. I based my implementation of the step function for Adam optimization on the efficient version of Algorithm 1 as outlined in "Adam: A Method for Stochastic Optimization" Kingma and Ba (2014).

2.1 Multi-head Self-Attention

Attention functions take query vectors and a set of key-value pair vectors and map them to some output denoting the compatibility of the query with the given keys. Multi-head attention can be thought of as a group of attention functions running in parallel, and they allow the model to jointly attend to information from different representations, at different positions. Single attention heads typically do not allow this joint behavior. I summarize the mathematical formulas for Self-Attention and Multi-head Attention as described in Vaswani et al. (2017):

$$\text{Attention}\left(Q, K, V\right) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \tag{1}$$

$$\text{head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V,\right) \tag{2}$$

$$\text{MHA}\left(Q, K, V\right) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_h\right) W^O \tag{3}$$

2.2 Adam Optimizer

Adam is a method for stochastic optimization that "computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients" Kingma and Ba (2014). I used a variation of Adam, called AdamW, where the weight decay is performed only after controlling the parameter-wise step size - the regularization term is outside of the moving averages and proportional to the weight itself.

2.3 Downstream Tasks and Goal

The goal of the project was to explore and understand ways to obtain robust and generalizable embeddings that perform well in multiple downstream tasks. The tasks were:

1. Sentiment Analysis: The task of classifying text to understanding which affective state (e.g., positive, negative, ...) is expressed.
2. Paraphrase Detection: The task of finding paraphrases of texts in a large corpus of passages. At its essence, this task seeks to determine whether particular words or phrases convey the same semantic meaning. The ultimate outcome would be binary (i.e., "yes" one is a paraphrase of other, or "no" they are not).
3. Semantic Textual Similarity: The task seeks to capture the notion that some texts are more similar than others by measuring the degree of semantic equivalence. To account for the degree of semantic equivalence there is a scale from 0 (denoting not related) to 5 (denoting they have the same meaning).

2.4 Complexity and Difficulty

When BERT was first introduced it achieved incredible results in many language understanding tasks. Pre-trained word embeddings from BERT acted as generalized word embeddings to be used as features in downstream models, as well as re-used across different language tasks. The effectiveness of this approach limited the need to train new models from scratch. My base model, minbert, is a strict subset of the original BERT, lacking some of the capabilities that BERT contained.

BERT is pre-trained on Masked Language Model (MLM) and Next Sentence Prediction (NSP) using a very large cross domain corpus of training data. The MLM involves masking a randomly selected predefined percentage of words and then attempting to predict the missing word. NSP attempts to predict the next sequence of words (i.e., next sentence) to follow a given sentence. Effectively, MLM aims to understand relationships between words and NSP aims to understand dependencies across sentences. The base implementation of minbert does not support MLM nor NSP.

Being trained on a very large cross domain corpus of training data allows BERT to have a general understanding of context, but this can also cause BERT to reach a ceiling on performance for any domain specific task, such as text classification in the Healthcare field. Words do not necessarily have the same meaning across different domains. Minbert being a subset of BERT suffers from the same dilemma. Here, my goal was to create generalizable embeddings that perform well on multiple downstream tasks, each with significantly different domain data.

Given the complexity and size of minbert's neural architecture, significant compute would be needed to run a series of targeted experiments to determine which changes, if any, are effective. Given the short duration of this academic project (i.e., <4 weeks) both time and compute create additional challenges.

3 Related Work

"BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers" Devlin et al. (2018). The pre-trained BERT can be used for a variety of downstream tasks. Such tasks can be accomplished by adding a task (or multi-task) head on top of BERT, to create a task (or multi-task) output layer.

To better ensure success in the downstream tasks, one would either enhance the task layer or make the embeddings from BERT more robust. Such enhancements can be incorporated at either the pre-training stage (i.e., making the embeddings more robust) or the fine-tuning stage. Later in this document I discuss which enhancements I experimented with. In this section, I provide a cursory background on the motivations for experimenting with specific pre-training or fine-tuning approaches.

Researchers Sun, Qiu, Xu and Huang Sun et al. (2020) provided an overview of various approaches to fine-tune BERT for text classification downstream tasks. The contributions of their work include:

1. The authors proposed a general solution to fine-tune a pre-trained BERT:
   - Further pre-train BERT on within-task training data or in-domain data (see Figure 1),
   - Fine-tune BERT with multi-task learning objectives and data,
   - Fine-tune BERT for a specific task.
2. The authors investigated pre-processing of text methods, layer selection, layer-wise learning considerations and low-shot learning problems

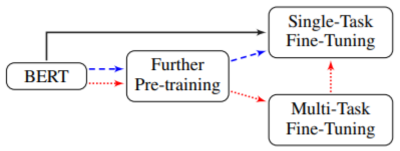


Figure 1: General Ways for fine-turning BERT

Their work introduced some very good ideas on practical considerations for improving minbert on the given tasks. Unfortunately, given the short (<4 weeks) amount of time and compute resources, I decided it was more prudent to be tactical and focus my efforts largely around pre-training on multi-task data, and some straight forward approaches to fine-tuning (impacting relatively more of the task layer for at least 2 of my downstream tasks).

Reading the work of Reimers and Gurevych in their development of Sentence-BERT (SBERT) Reimers and Gurevych (2019) they achieved success computing similarity scores between embeddings for each of two source sentence inputs. Though the authors incorporated a siamese network architecture, their work did show strong success with using a similarity measure like cosine similarity. As such, I chose to use a similar concept for the paraphrase and semantic text similarity downstream tasks of my project. Additionally, their work also motivated me to fine-tune using a cosine embedding loss function.

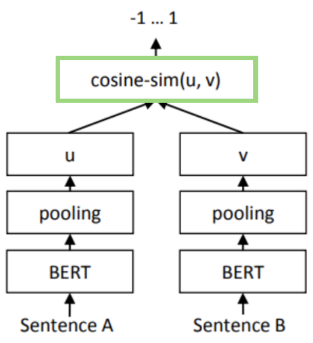


Figure 2: Computing Similarity Scores in SBERT during Inference

4 Approach

Base Transformer Model. The base model contains 12 Encoder Transformer Layers each consisting of multi-head attention, followed by an additive and normalization layer with a residual connection, a feed-forward layer, and a final additive and normalization layer with a residual connection. I added three prediction heads on top of the *same* base embedding model; one prediction head for each downstream task.

Sentiment Analysis. For sentiment analysis, the downstream task was formulated as a text classification problem. Using the final minbert embedding, which is the hidden state of the [CLS] token, I added a classification head composed of a linear layer and a dropout layer. The linear layer is based on 768 neurons, plus a bias neuron, and 5 output units (one for each sentiment class: negative, somewhat negative, neutral, somewhat positive and positive). The dropout layer is used as a regularization method to reduce overfitting. A dropout probability of 30% was used. Using a dropout layer could result in a lower training performance metric value, but should allow better generalization to any holdout development sets. During training I used cross entropy loss as the loss function. An argmax is applied to the logits from the linear classifier to get the ultimate sentiment prediction.

Paraphrase Detection. For paraphrase detection, the downstream task was formulated as a binary classification problem. The cosine similarity between the embeddings of the two given phrases is calculated and then passed to a sigmoid function returning probabilities for yes (i.e., the phrases are paraphrases) and no (i.e., the phrases are not paraphrases), providing the ultimate paraphrase detection prediction. During training I used binary cross entropy loss as the loss function.

Semantic Textual Similarity. For semantic textual similarity, the downstream task was formulated as a continuous regression problem. The cosine similarity between the embeddings of the two given phrases is calculated. During training I used cosine embedding loss as the loss function.

Enhancements to the base architecture and extensions to the training process (pre-training and fine-tuning) are discussed within the "Experimental details" section.

5 Experiments

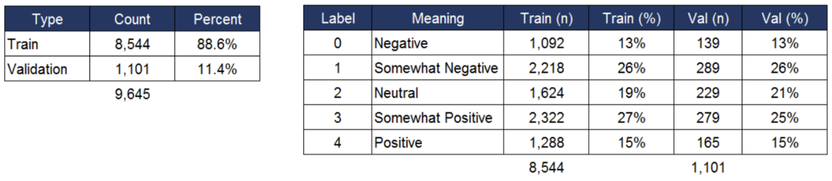
5.1 Data

5.1.1 Data for Sentiment Analysis

I used an excerpt of the Stanford Sentiment Treebank (SST) dataset. The original dataset consists of 11,855 single sentences extracted from movie reviews. The dataset was parsed with the Stanford parser and includes a total of 215,154 unique phrases from those parse trees, each annotated by 3 human judges. Each phrase has a label for the following:

1. Negative
2. Somewhat Negative
3. Neutral
4. Somewhat Positive
5. Positive

The labels previously described represent varying levels of positive, negative, and neutral affective states. Below I provide a distribution of the train and validation split, along with a distribution of the composition of labels.



Type	Count	Percent
Train	8,544	88.6%
Validation	1,101	11.4%
	9,645	

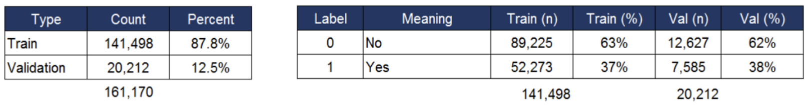
Label	Meaning	Train (n)	Train (%)	Val (n)	Val (%)
0	Negative	1,092	13%	139	13%
1	Somewhat Negative	2,218	26%	289	26%
2	Neutral	1,624	19%	229	21%
3	Somewhat Positive	2,322	27%	279	25%
4	Positive	1,288	15%	165	15%
		8,544		1,101	

Figure 3: Stanford Sentiment Treebank Data Distribution

Note the presence of 5 affective states versus 2 states (Negative and Positive) or 3 states (Negative, Neutral and Positive).

5.1.2 Data for Paraphrase Detection

I used the Quora dataset containing question pairs in training for Paraphase Detection. Below I provide a distribution of the train and validation split, along with a distribution of the composition of labels.



Type	Count	Percent
Train	141,498	87.8%
Validation	20,212	12.5%
	161,170	

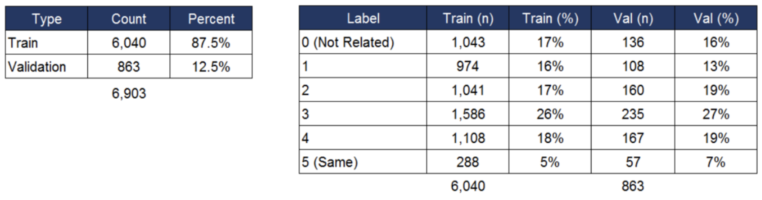
Label	Meaning	Train (n)	Train (%)	Val (n)	Val (%)
0	No	89,225	63%	12,627	62%
1	Yes	52,273	37%	7,585	38%
		141,498		20,212	

Figure 4: Quora Data Distribution

Note the data imbalance between "No" and "Yes".

5.1.3 Data for Semantic Textual Similarity

I used the provided SemEval STS Benchmark dataset in training for Semantic Textual Similarity. Below I provide a distribution of the train and validation split, along with a distribution of the composition of labels.



Type	Count	Percent
Train	6,040	87.5%
Validation	863	12.5%
	6,903	

Label	Train (n)	Train (%)	Val (n)	Val (%)
0 (Not Related)	1,043	17%	136	16%
1	974	16%	108	13%
2	1,041	17%	160	19%
3	1,586	26%	235	27%
4	1,108	18%	167	19%
5 (Same)	288	5%	57	7%
	6,040		863	

Figure 5: SemEval STS Data Distribution

Note the slight data imbalance for the last STS label compared to the other labels.

5.2 Evaluation method

The primary evaluation approaches:

- **Accuracy** for Sentiment Analysis and Paraphrase Detection. (TP = True Positive, FP = False Positive, TN = True Negative, FN = False Negative)

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

- **Pearson Correlation** for Semantic Textual Similarity.

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 (y_i - \bar{y})^2}} \tag{5}$$

5.3 Experimental details

5.3.1 Performance Enhancement Strategy

To better ensure success in the downstream tasks, one would either enhance the task layer or make the embeddings from minbert more robust. Such enhancements can be incorporated at either the pre-training stage (i.e., making the embeddings more robust) or the fine-tuning stage (i.e., enhancing the downstream task prediction heads). As a tactical strategy (given the short timeframe to work on the project) I focused primary on further pre-training of BERT, but also explored approaches for fine-tuning.

5.3.2 Further Pre-Training - Masked Language Model

To enhance the quality of the word embeddings, I implemented a masked language objective which I used to further pre-train minbert. I started with an existing pretrained bert model (called “bert-base-uncased”), and further pre-trained using the the downstream multi-task training data.

This process masked 15% of the training data (excluding PADDED and MASKED tokens), attempted to predict the masked data, then updated the weights and embeddings based on the cross entropy loss function. This provided a significant boost to the quality of the embeddings. The masked language objective provided the most impactful boost of all of the performance enhancement experiments. I included an excerpt of my masked language implementation in the appendix.

5.3.3 Fine-Tuning - Cosine Embedding Loss

The cosine embedding loss measures the loss given input tensors x_1 and x_2 and a tensor label y with values 1 or -1. This is used for measuring whether two inputs are similar or dissimilar, using the cosine similarity, and is typically used for learning nonlinear embeddings or semi-supervised learning. This was used during fine-tuning of the semantic text similarity downstream task.

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{6}$$

$$\text{cosine\_embedding\_loss}\,(x, y) = \begin{cases} 1 - \cos(x_1, x_2) & , \text{ if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}) & , \text{ if } y = -1 \end{cases} \tag{7}$$

where $-1 \leq \text{margin} \leq 1$, but $0 \leq \text{margin} \leq 0.5$ is often suggested.

5.3.4 Learning Rate Schedule

Instead of using a static learning rate, and though the AdamW optimizer does have some adjustment to the learning rate, I added an exponential learning rate decay scheduler to the optimizer.

$$\text{learning rate}_0 = 0.001 \tag{8}$$

$$\text{gamma} = 0.96 \tag{9}$$

$$\text{learning rate}_{\text{epoch}} = \text{learning rate}_0 * \text{gamma}^{\text{epoch}} \tag{10}$$

Figure 6 provides a visual illustration of the exponential decay schedule.

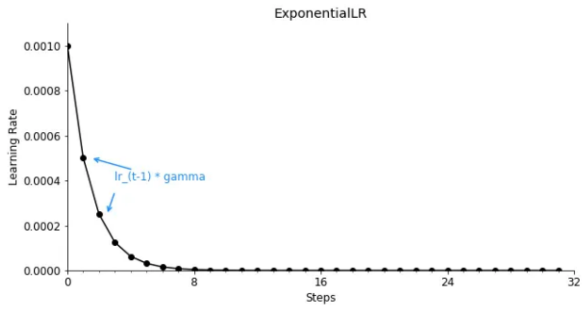


Figure 6: Exponential Decay Learning Rate

5.3.5 Hyperparameter Tuning

I employed a standard grid search for hyperparameter tuning. But I did not change the number of layers in the model architecture nor the number of hidden nodes.

$$\begin{aligned}
\text{learning rate} &= 0.001 \text{ for pre-training} && (11)\\
\text{learning rate} &= 0.0002 \text{ for fine-training} && (12)\\
\text{learning rate} &= 0.0003 && (13)\\
&\quad \text{for pre-training using masked}\\
&\quad \text{language}\\
\text{gamma} &= 0.96 && (14)\\
&\quad \text{for the exponential learning}\\
&\quad \text{rate decay schedule}\\
\text{batch size} &= 64 && (15)
\end{aligned}$$

5.3.6 Experimentation Process

The enhancements and extensions previously described were the ones where I observed positive impact to performance. There were some other experiments performed that either lowered predictive performance or provided no additional benefit, and hence were not retained.

Mapping Cosine Similarity to Discrete Classes. I attempted to statically map the cosine similarity score to varying levels of similarity classes for semantic text similarity.

$$\text{logit} = \begin{cases} 0, & \text{if similarity} < 0.16667 \\ 1, & \text{if similarity} < 0.33334 \\ 2, & \text{if similarity} < 0.50001 \\ 3, & \text{if similarity} < 0.66668 \\ 4, & \text{if similarity} < 0.83335 \\ 5, & \text{otherwise} \end{cases}$$

I also attempted to statically map the cosine similarity score to varying levels of yes / no for paraphrase detection:

$$\text{logit} = \begin{cases} 0, & \text{if similarity} < 0.5 \\ 1, & \text{otherwise} \end{cases}$$

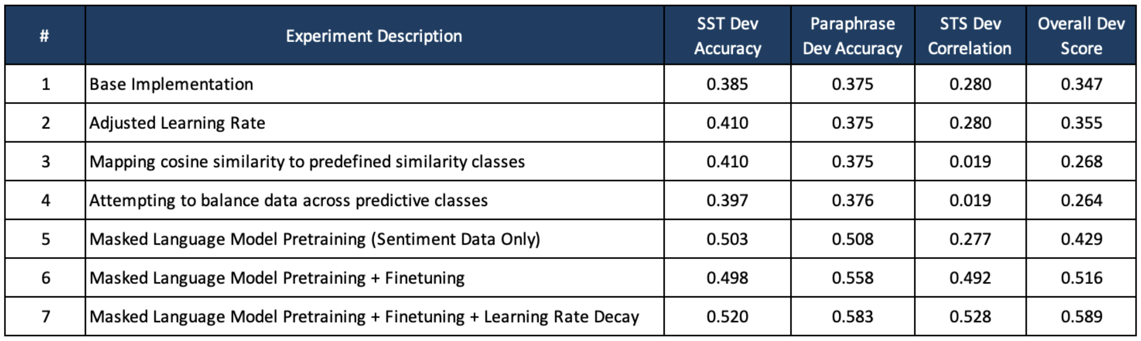
5.4 Results

The completed base implementation achieved a 0.385 accuracy, 0.375 accuracy and 0.280 correlation respectively (overall score of 0.347) on holdout datasets of the aforementioned tasks.

After further pre-training on task specific data by training on a masked language model objective, fine-tuning using cosine embedding loss, applying a learning rate decay schedule, and hyperparameter tuning, my final model achieved:

- 0.520 accuracy for the sentiment task
- 0.583 accuracy for the paraphrase detection task
- 0.528 correlation for the semantic text similarity task

The final model achieved an overall average score of 0.589. The following table displays various experiments performed and their measured performance:



#	Experiment Description	SST Dev Accuracy	Paraphrase Dev Accuracy	STS Dev Correlation	Overall Dev Score
1	Base Implementation	0.385	0.375	0.280	0.347
2	Adjusted Learning Rate	0.410	0.375	0.280	0.355
3	Mapping cosine similarity to predefined similarity classes	0.410	0.375	0.019	0.268
4	Attempting to balance data across predictive classes	0.397	0.376	0.019	0.264
5	Masked Language Model Pretraining (Sentiment Data Only)	0.503	0.508	0.277	0.429
6	Masked Language Model Pretraining + Finetuning	0.498	0.558	0.492	0.516
7	Masked Language Model Pretraining + Finetuning + Learning Rate Decay	0.520	0.583	0.528	0.589

Figure 7: Experimental Results Summary

The results are inline with my expectations because:

(a) There is a significant increase in performance when the masked language objective was applied to further pre-train minbert. Such an objective helps minbert understand relationships between words.

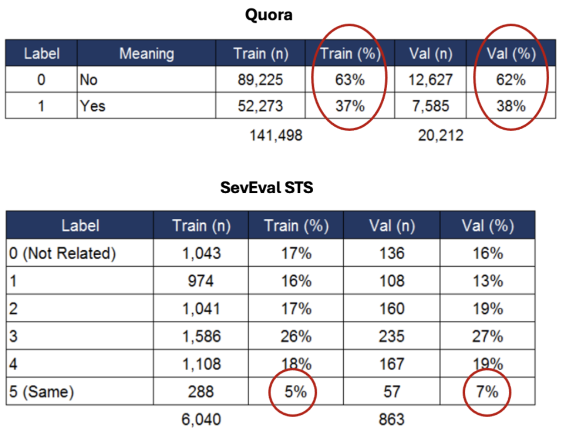
(b) Applying a learning rate schedule allows the step size to change, effectively decaying over time. Lowering the step size in this way can improve both convergence and model performance.

(c) None of the experiments (due to time constraints) incorporated balanced data, additional domain data, nor a combined loss function (aggregating loss gradients of the three downstream tasks). As such, I expected a ceiling on performance. These limitations are further discussed later within the "Limitations" section.

6 Analysis

6.1 Data Imbalance and Performance Distributions

The Stanford Sentiment Treebank dataset for sentiment was generally balanced across the classes (i.e., Negative, . . . , Positive). However, I observed high imbalance in the Quora dataset for paraphrase detection. There was some imbalance in the SevEval STS dataset for semantic textual similarity, but nothing substantial.



Quora

Label	Meaning	Train (n)	Train (%)	Val (n)	Val (%)
0	No	89,225	63%	12,627	62%
1	Yes	52,273	37%	7,585	38%
		141,498		20,212	

SevEval STS

Label	Train (n)	Train (%)	Val (n)	Val (%)
0 (Not Related)	1,043	17%	136	16%
1	974	16%	108	13%
2	1,041	17%	160	19%
3	1,586	26%	235	27%
4	1,108	18%	167	19%
5 (Same)	288	5%	57	7%
	6,040		863	

Figure 8: Imbalanced Data Distributions

Performing an error analysis to understand whether the distribution of predictions may have some correlation with the imbalance in the training data. Take for example the Quora dataset, referring to the data distribution described earlier we see that the “no” class makes up 62% of the labeled data. A distribution of the model predictions (refer to Figure 11) shows that majority of the predictions for paraphrase detection are also “no”. The optimization in the training with a single objective function will favor best predictions on average, which can get influenced with extremely class imbalance.

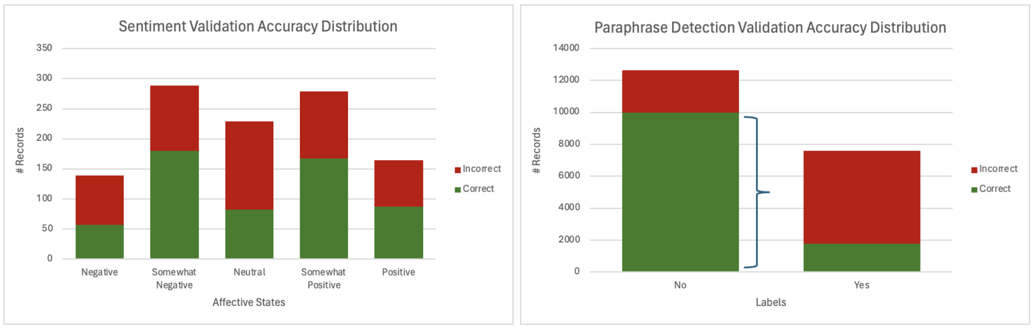


Figure 9: Accuracy Distributions

6.2 Training Loss Curves

One of the extensions included the addition of a masked language objective to help minbert better understand relationships between words inorder to boost the quality of the embeddings. In Figure 11, I provided the loss curve of minbert’s pre-training using the masked language model (based on a learning rate of 0.0004). Early on during the training there is a steep drop in loss before beginning to flatten out, signaling that the language model is better understanding the domain specific wording.

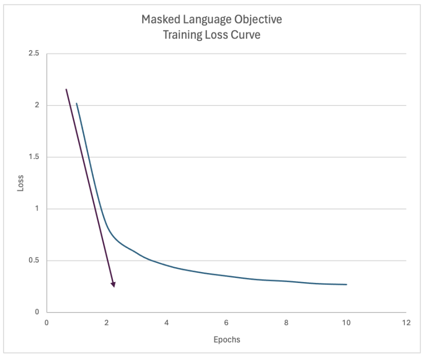


Figure 10: Masked Language Pretraining - Loss Curve

6.3 Overfitting

During finetuning I collected accuracy measurements of predictions on training and validation data. I observed a significant increase in accuracy for the training, but little variance in changes to the accuracy on validation data. This is likely a visual indication of severe overfitting. Please refer to Figure 11:

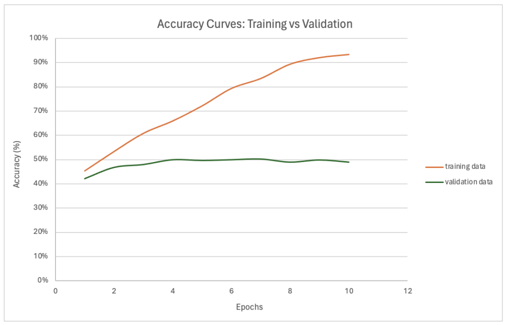


Figure 11: Plot of Training Data vs Validation Data Accuracy

7 Conclusion

7.1 Limitations

My implementation does not use a combined loss approach. Model iterations are saved based on individual downstream task performance during training. This limitation places a ceiling on the overall performance that could be achieved with this model. By combining the losses of individual downstream tasks during fine-tuning the implementation would be closer to supporting true multi-task learning.

Another limitation is that my implementation lacks regularized optimization, specifically in the loss function. As a result the implementation has increased chances of overfitting.

7.2 Possible Extensions

There are a number of potential extensions not explored in this project. Below are some additional things to be explored in future work on this project:

(a) Down-sampling to get more balanced training datasets across the task domain datasets. For example, the number of training records for each class / category of each task domain dataset could be down-sampled to be 900 records; similar remarks to the validation records but with 130 records.

(b) Fine-tune the model using Multiple Negative Ranking Loss. Such an approach would minimize the distance between similar training data records and maximize the distance between dissimilar training data records.

(c) Incorporation of regularized optimization in the loss functions, such as smoothness-inducing regularization and bergman proximal point optimization.

(d) Pre-train using Next Sentence Prediction for the paraphrase detection downstream task. I included an excerpt of my next sentence prediction implementation in the appendix.

8 Authorship Statement

I'd like to extend a big thanks to Assistant Professors Dr. Diyi Yang and Dr. Tatsunori Hashimoto; both in the Computer Science Department, Stanford University. Additionally I'd like to thank Rohan Taori and all of the other Teaching Assistants of CS224N Natural Language Processing with Deep Learning. Their suggestions and guidance were helpful and appreciated. Any opinions, findings, and conclusions expressed in this document are that of the authors and do not necessarily reflect the views of Stanford University.

References

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

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

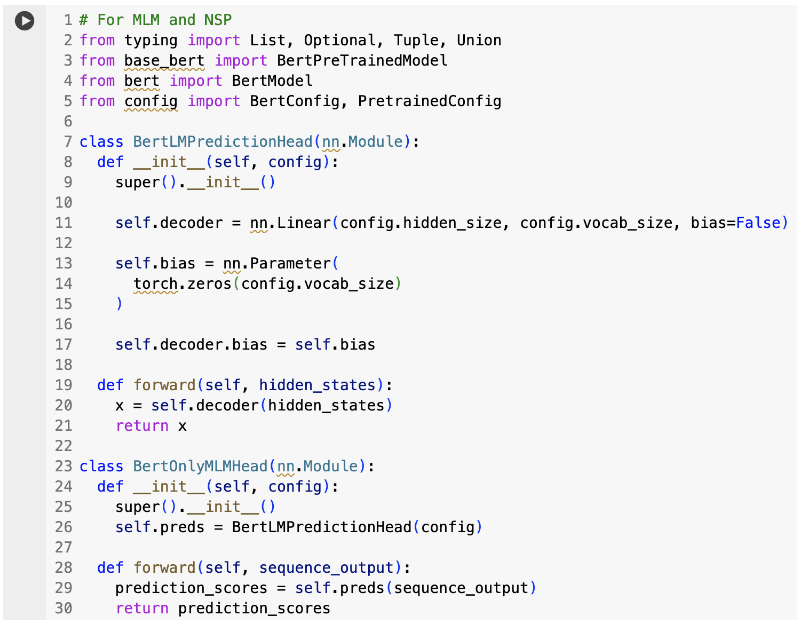
Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks.

Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2020. How to fine-tune bert for text classification?

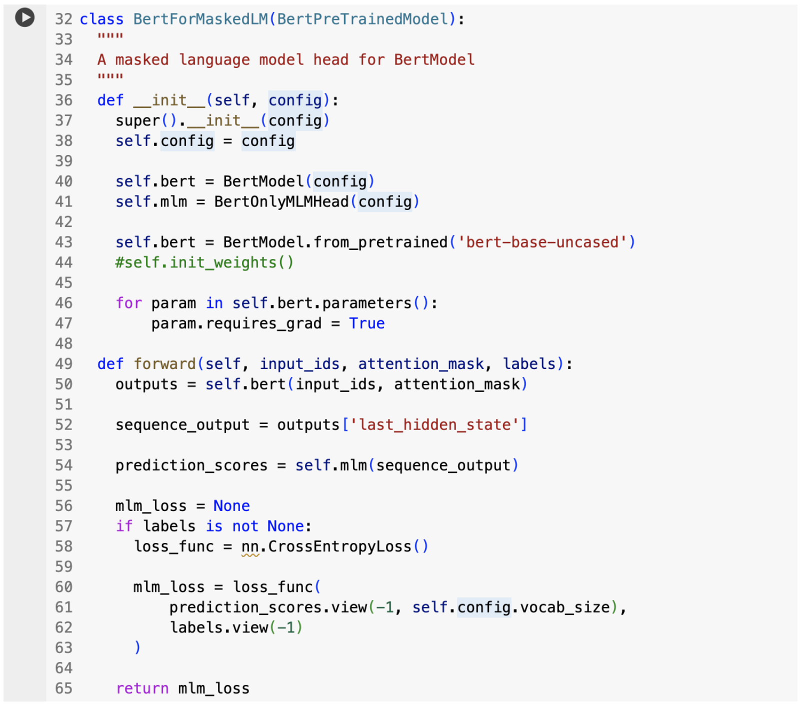
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

A Appendix (optional)

A.1 Masked Language Model (MLM) Code

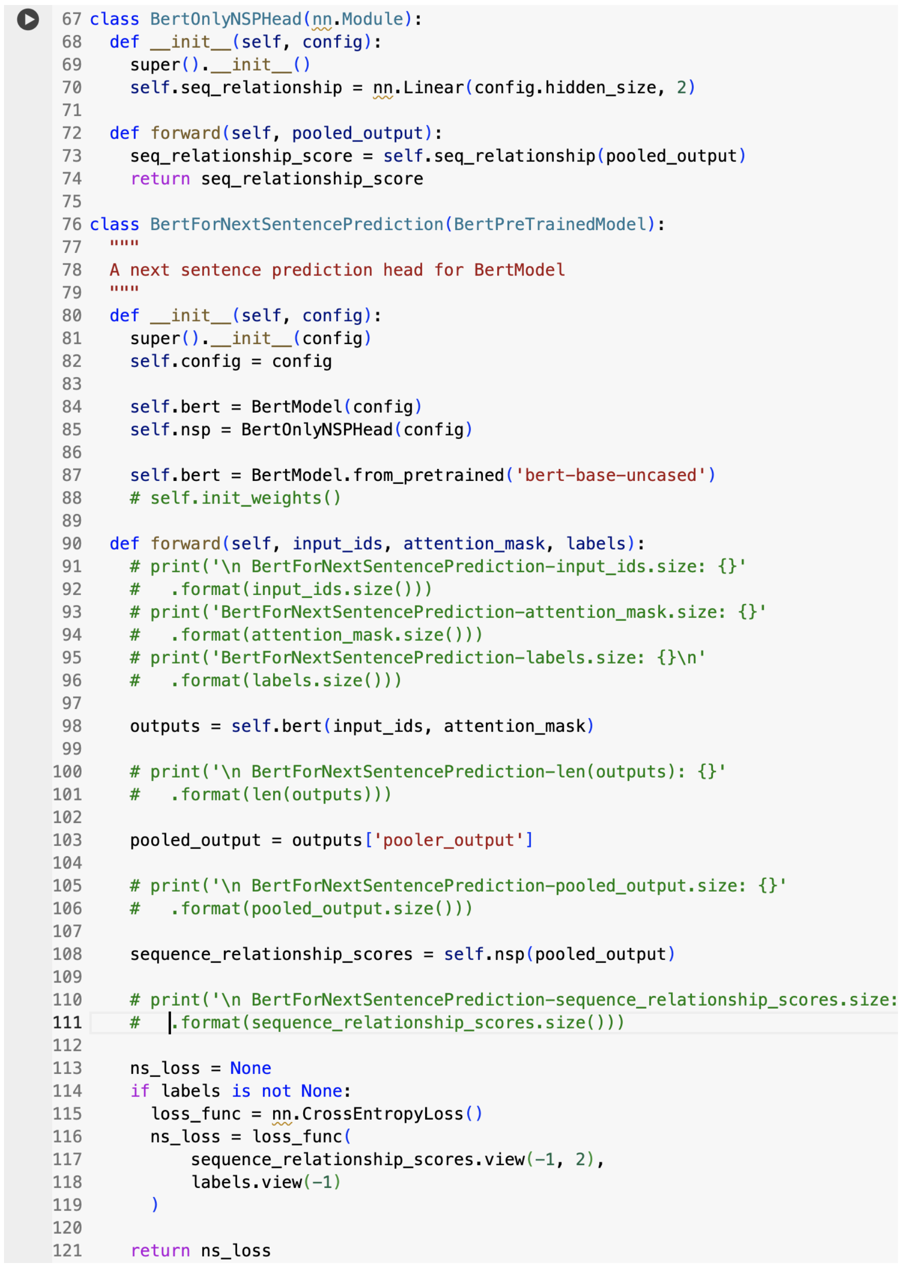


```python
# For MLM and NSP
from typing import List, Optional, Tuple, Union
from base_bert import BertPreTrainedModel
from bert import BertModel
from config import BertConfig, PretrainedConfig

class BertLMPredictionHead(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.decoder = nn.Linear(config.hidden_size, config.vocab_size, bias=False)

    self.bias = nn.Parameter(
      torch.zeros(config.vocab_size)
    )

    self.decoder.bias = self.bias

  def forward(self, hidden_states):
    x = self.decoder(hidden_states)
    return x

class BertOnlyMLMHead(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.preds = BertLMPredictionHead(config)

  def forward(self, sequence_output):
    prediction_scores = self.preds(sequence_output)
    return prediction_scores
```



```python
class BertForMaskedLM(BertPreTrainedModel):
  """
  A masked language model head for BertModel
  """
  def __init__(self, config):
    super().__init__(config)
    self.config = config

    self.bert = BertModel(config)
    self.mlm = BertOnlyMLMHead(config)

    self.bert = BertModel.from_pretrained('bert-base-uncased')
    #self.init_weights()

    for param in self.bert.parameters():
        param.requires_grad = True

  def forward(self, input_ids, attention_mask, labels):
    outputs = self.bert(input_ids, attention_mask)

    sequence_output = outputs['last_hidden_state']

    prediction_scores = self.mlm(sequence_output)

    mlm_loss = None
    if labels is not None:
      loss_func = nn.CrossEntropyLoss()

      mlm_loss = loss_func(
          prediction_scores.view(-1, self.config.vocab_size),
          labels.view(-1)
      )

    return mlm_loss
```

A.2 Next Sentence Prediction (NSP) Code



```
class BertOnlyNSPHead(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.seq_relationship = nn.Linear(config.hidden_size, 2)

  def forward(self, pooled_output):
    seq_relationship_score = self.seq_relationship(pooled_output)
    return seq_relationship_score

class BertForNextSentencePrediction(BertPreTrainedModel):
  """
  A next sentence prediction head for BertModel
  """
  def __init__(self, config):
    super().__init__(config)
    self.config = config

    self.bert = BertModel(config)
    self.nsp = BertOnlyNSPHead(config)

    self.bert = BertModel.from_pretrained('bert-base-uncased')
    # self.init_weights()

  def forward(self, input_ids, attention_mask, labels):
    # print('\n BertForNextSentencePrediction-input_ids.size: {}'
    #   .format(input_ids.size()))
    # print('BertForNextSentencePrediction-attention_mask.size: {}'
    #   .format(attention_mask.size()))
    # print('BertForNextSentencePrediction-labels.size: {}\n'
    #   .format(labels.size()))

    outputs = self.bert(input_ids, attention_mask)

    # print('\n BertForNextSentencePrediction-len(outputs): {}'
    #   .format(len(outputs)))

    pooled_output = outputs['pooler_output']

    # print('\n BertForNextSentencePrediction-pooled_output.size: {}'
    #   .format(pooled_output.size()))

    sequence_relationship_scores = self.nsp(pooled_output)

    # print('\n BertForNextSentencePrediction-sequence_relationship_scores.size:
    #   .format(sequence_relationship_scores.size()))

    ns_loss = None
    if labels is not None:
      loss_func = nn.CrossEntropyLoss()
      ns_loss = loss_func(
          sequence_relationship_scores.view(-1, 2),
          labels.view(-1)
      )

    return ns_loss
```