BERT Fine-tuning with Meta Learning

Stanford CS224N Default Project

Hui Xue
Department of Computer Science
Stanford University
hxue@stanford.edu

Abstract

The Bidirectional Encoder Representations from Transformers (BERT [Devlin et al. (2018)]) is a powerful and popular model to generate the contextual language embedding for a broad set of NLP applications. The typical method to train BERT for a new application is the pre-training + fine-tuning scheme. As a result, a new task requires to generate one fine-tuning model by collecting specific training data. While managing multiple models can be tedious and error-prone, a more severe disadvantage is data collected for multiple tasks cannot help each other in the task-specific fine-tuning scheme. This study is to investigate whether a meta-learning scheme can help boost model performance via the multi-task training. In specific, the reptile [Nichol et al. (2018)] algorithm is implemented on top of the BERT model to jointly train three different NLP tasks.

1 Key Information to include

- External collaborators: None
- External mentor: None
- Sharing project: None

2 Introduction

The Bidirectional Encoder Representations from Transformers (BERT [Devlin et al. (2018)]) is a popular model to capture bi-directional context information from input texts and produce the context-aware feature vectors for sentences. BERT model is pre-trained with a large body of high-quality texts and available publicly. Often, performance boost can be achieved for a new natural language processing (NLP) task by collecting task-specific labeled data and conducting fine-tuning on top of the pre-trained BERT model. This process involves adding task specific layers to process the feature tensor produced by the BERT model and output logits for different tasks and corresponding loss functions. During the backprop process, optimizer can choose to update the added layers with or without the pre-trained model parameters. After training the augmented model for a few epochs, the pre-trained BERT model is adapted to this new NLP task.

This fine-tuning process is very effective to boost NLP task performance by utilizing the rich representation offered by the pre-training model. It has advantages to be adaptive and efficient and become a main stream method to improved deep learning performance in NLP [Huang et al. (2022)]. While its success is prominent, training and maintaining multiple models is tedious, computationally costly and error-prone. A better scheme is to have one model capable of handling multiple NLP tasks. This is the so-called multi-task learning [Zhang and Yang (2017)]. The straight-forward multi-task learning can be achieved with the joint-learning scheme. Given a set of N NLP tasks with corresponding loss function $L_i$ and task-specific datasets $D_i$, where $i$ is the task index (from 0 to $N - 1$). The joint-learning is to solve the following optimization problem:
\[ \theta^* = \min_{\theta} \frac{1}{N} \sum_{i=0}^{N-1} \lambda_i l_i(\theta_s, \theta_i, D_i) \]  

(1)

where \( \theta = \{\theta_s, \theta_0, \ldots, \theta_{N-1}\} \) is the all-set of parameters for the multi-task model. \( \theta_s \) is the shared parameters from the pre-trained backbone BERT and \( \theta_i \) is the task specific parameter. \( \lambda_i \) is the weight to balance all loss functions. The optimization process of joint-learning is to evaluate all loss functions and compute the weighted mean and performance one backprop to update model parameters. This simple scheme is proved to be effective and broadly adopted in NLP deep learning [Devlin et al. (2018)].

The joint-learning method minimizes the averaged loss function and benefits from the reduced total number of parameters and shared task datasets; but it cannot guarantee the improvement of all tasks. In other words, the updated gradient can favor some tasks more than others, leading to overall improvement in the total loss, but inferior performance for some tasks. The model can also forget what was learned and over-fit the new task (aka catastrophic forgetting [Kirkpatrick et al. (2017)]).

Meta-learning is a set of deep learning techniques for learning-to-learn [Hospedales et al. (2022)]. Meta-learning aims to find a learning scheme to utilize information from multiple tasks to improve learning algorithms. While meta-learning can be achieved by modifying the model parameters, it often comes with a double-loop structure. The inner optimizer aims to improve task-specific loss function and estimates the task-specific gradient. The outer optimizer or meta optimizer will update the meta model based on task specific gradients. Given a set of tasks, meta-learning can learn a good model sitting in a favorable place in the parameter manifold closer to optimal points for all tasks. That is a model good for multiple tasks.

A principled meta-learning method (e.g. MAML [Finn et al. (2017)]) will optimize the model performance on the new datasets, by minimizing the following loss:

\[ \min_{\theta} \frac{\lambda_i}{N} \sum_{i=0}^{N-1} l_i(\theta - \alpha \nabla_{\theta} \frac{\lambda_j}{N} \sum_{j=0}^{N-1} l_j(D_j, \theta), D_i) \]  

(2)

The MAML optimization first performs a gradient step to minimize loss defined in equation 1 and then make sure the resulting model parameters further minimize loss for a new task \( i \). If a model was trained by solving the minimization problem defined in the equation 2, this model should be able to adapt to a new task, as this is exactly optimized by the MAML loss.

Minimizing equation 2 involves evaluation of parameter Jacobin matrix, incurring elevated computational and memory cost. Reptile [Nichol et al. (2018)] is a one-order approximation of the MAML method, requiring combining the gradients over batches from multiple tasks:

Given batches \( B_j, j = 0 \ldots N - 1 \) for \( N \) tasks, the gradient is computed as:

\[ W = \frac{\lambda_j}{N} \sum_{j=0}^{N-1} \nabla \theta l_j(B_j, \theta) \]  

(3)

\[ \theta \leftarrow \theta + \alpha (W - \theta) \]  

(4)

This study is to investigate whether the Reptile meta-learning training can improve the multi-task learning performance on three NLP tasks. The meta-learning results are compared to joint-learning scheme and to train each task independently.

3 Related Work

Different schemes have been proposed to train a model to handle multiple tasks. Transfer learning [Zhuang et al. (2020)] was proposed to reuse the learned features from a past dataset on a new task. This scheme is a single-layer optimization method, without accessing the old dataset. Domain adaption [Nozza et al. (2016)] methods are a step forward, to train the model for better generalization on the
new task. It requires to access the previous training data, but still does not have the explicit meta optimization. In parallel, the multi-task learning (Zhang and Yang, 2017) based on the joint-learning method was proposed. Meta-learning, on the other hand, is a step-forward beyond all these previous development, by adding the explicit meta optimization to modify meta model to work well on the multiple tasks.

Given the promised advantages of meta-learning, it has been applied to training the BERT model (Bansal et al., 2020; Murty et al., 2021; van der Heijden et al., 2021). Previous studies differ on the architecture added on top of BERT model, loss functions and training schedule. The performance boost was reported, especially when the training source is limited.

4 Approach

The miniBERT model (CS224N, 2023) was extended with three sets of heads to handle three NLP tasks: sentiment analysis (sst), paraphrase detection (para) and semantic textual similarity (sts). The sst task is to classify a sentence to be negative, somewhat negative, neutral, somewhat positive, or positive. The para task is to decide whether two sentences are paraphrases of each other. The sts task is to estimate the degree of semantic equivalence, as a float value from 0 to 5.

4.1 Model

Figure 1 gives the model architecture, built on top of the BERT model to handle three tasks.

![Figure 1: Model architecture for the three-task BERT. While the BERT backbone is shared, every task has its specific head. The sst task only takes in one sentence.](image)

sst: For the sst task, the last hidden state vectors first go through the mean pooling layer through the time dimension. The pooled feature vectors are further transformed with two linear layers, with a
gelu nonlinearity [Hendrycks and Gimpel (2020)] in the middle. The final logits are in the shape of $[B, 5]$, $B$ is the number of batches and 5 is for five sentimental categories. The cross-entropy loss is used for training.

**para**: Two sentences first go through the BERT backbone model. Resulting hidden state vectors go through the mean pooling. Two pooled tensors are concatenated and processed with two linear and one gelu layers. The output logit have the shape of $[B, 1]$ for the binary classification. The binary cross-entropy loss is minimized.

**sts**: Similar to **para**, two sentences go through the BERT and hidden state vectors are mean-pooled. After a linear layer, the resulting tensors of two inputs are inputted into the cosine similarity computation. Since the cosine similarity has the range of $[-1, 1]$, it goes through a relu [Agarap (2019)] and scaled by 5.0 to match the expected measure of sts equivalence. The MSE loss is used for this regression task.

### 4.2 Training

The Reptile algorithm was implemented for the three-task training scenario. Following algorithm illustrates the training process. The meta iteration is the outer optimization loop. The meta optimizer gathers the updated gradient (equation 3) for each iteration and updates the meta model parameters. For each meta iteration, one of the three tasks are randomly picked. The probability to choose each tasks can be specified. A inner model is created as a clone of the meta model and optimized for a few steps. The meta model parameters are updated using equation 4.

**Reptile training for three NLP tasks**

```plaintext
Set up meta model $m$ and meta optimizer $m_{-}optim$

for $k = 0 \ldots meta\_iter - 1$ do
    Select a task, according to the probabilities $[sst\_prob, para\_prob, sts\_prob]$
    Create a clone of meta model, $inner_{m}$
    Set up inner optimizer $optim$ and number of iteration $task\_iter$
    for $t = 0 \ldots task\_iter - 1$ do
        Select a batches $B$
        forward pass, compute loss
        optim.step()
    end for
    Update $m$ gradient as $inner_{m}.parameters() - m.parameters()$
    $m_{-}optim.step()$
end for
```

According to the data size for each task and relative difficulties estimated by the individual training experiments, the probabilities to pick each task can be adjusted, together with batch size (sst_batch, para_batch, sts_batch) and number of inner optimization steps (e.g. sst_iter, para_iter and sts_iter). After every 20 meta iterations, the meta model is tested on the dev datasets for three tasks.

### 5 Experiments

Step-by-step experiments were conducted to explore the hyper-parameter space and to develop better understanding for the datasets. First, the training was conducted for each task independently. The purpose is to test model capacity and establish a top line of performance. Second, the joint-learning scheme was implemented (equation 1), as the baseline. Third, the Reptile meta-learning was trained on the three tasks.

#### 5.1 Data

Three NLP tasks were tested in this study. **sst**: The Stanford Sentiment Treebank dataset [Socher et al. (2013)] includes 8544 samples for training and 1101 samples for evaluation. Each sample is classified into one of five categories. **para**: A subset of the Quora dataset [Shankar Iyer and Csernai (2019)]
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_iter</td>
<td>number of outer iteration for meta optimization</td>
<td>400</td>
</tr>
<tr>
<td>meta_lr</td>
<td>learning rate for meta optimization</td>
<td>1.0</td>
</tr>
<tr>
<td>meta_optimizer</td>
<td>optimizer type (SGD, AdamW, Adam or NAdam)</td>
<td>SGD</td>
</tr>
<tr>
<td>meta_scheduler</td>
<td>lr scheduler (Plateau, Steplr, OneCycle, CosAnneal)</td>
<td>CosineAnnealingLR</td>
</tr>
<tr>
<td>lr</td>
<td>learning rate for inner optimization</td>
<td>1e-5</td>
</tr>
<tr>
<td>scheduler</td>
<td>inner lr scheduler</td>
<td>Linear, multiply 0.8, every 30 iters</td>
</tr>
<tr>
<td>optimizer</td>
<td>inner optimizer</td>
<td>AdamW</td>
</tr>
<tr>
<td>sst_iter</td>
<td>number of inner iteration for sst</td>
<td>4</td>
</tr>
<tr>
<td>para_iter</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>sts_iter</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>sst_batch</td>
<td>batch size for inner iteration, sst</td>
<td>128</td>
</tr>
<tr>
<td>para_batch</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>sts_batch</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>sst_prob</td>
<td>probability to pick sst task, sst</td>
<td>0.4</td>
</tr>
<tr>
<td>para_prob</td>
<td></td>
<td>0.3</td>
</tr>
<tr>
<td>sts_prob</td>
<td></td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 1: Key hyper-parameters.

(2012) was assembled, consisting of 141498 samples for training and 20212 for evaluation. Each sample includes a pair of sentences and is classified as either paraphrase or not. **sts:** The SemEval STS Benchmark dataset [Agirre et al. (2013)] has 6040 training samples and 863 evaluation sample. The similarity for two sentences in every sample is measured as a float number between 0.0 and 5.0.

5.2 Evaluation method

The classification accuracy is used to evaluate the sst and para tasks. The Pearson correlation coefficient is computed to evaluate the sts task.

5.3 Experimental details

The fine-tuning strategy is used in all experiments, so the parameters in the BERT model are updated, together with task specific headers. As the amount of training data in the para dataset is an order of magnitude more than other two tasks, there is data imbalance for the multi-tasking training. Among all hyper-parameters, one key step is how to balance the amount of training samples explored during the model training, which is determined by the product of inner iteration steps and batch size. On the other hand, no all tasks are equally difficult to learn. An easier task may require less training data to achieve good performance while harder task can benefit from larger number of training data explored.

Initial training runs were performed to select a proper range for learning rates and optimizer with a learning rate scheduling strategy. Based on the findings, all results in the Results section used a learning rate of 2e-5 for individual and joint-multi-task training. All tasks were trained with 20 epoches, as accuracy plateaued. The Adam optimizer [Kingma and Ba (2017)] with the ReduceLRonPlateau scheduler [Pytorch (2023)] was used with 0.5 reduction factor. The joint-multi-task training used a batch size 64 for all three tasking, meaning the same number of training samples were explored for every task.

A reduced learning rate 1e-5 was used for meta-learning inner optimization, with a linear scheduling of reduction factor 0.8. The meta learning was performed for 400 iterations. An inner optimization was performed for a randomly picked task for a number of steps. Table 1 summarized the parameters to produce the results.

All training was conducted on a linux server with a single A100 GPU. The training time for individual tasks were ~10mins for sst and sts tasks and ~2hours for para task. The meta-learning took ~90mins.

5.4 Results
The proposed model was first trained on each task independently. Figure 2 gives the accuracy/correlation vs. epoch curves for the dev datasets. The best dev accuracies were 0.489 for sst and 0.820 for para. The correlation was 0.629 for sts. In all cases, the model reached peak accuracy in <10 epochs and performance plateaued. There are over-fitting, indicated by the much higher performance on training datasets than the dev set.

The joint-multi-tasking training was less accurate than the individual training. The dev accuracy was 0.474 for sst and 0.600 for para. The dev correlation was 0.589 for sts.

The Reptile meta-learning improved performance over the multi-task training. The dev accuracy was 0.488 for sst and 0.730 for para. The correlation was 0.743 for sts. The test accuracies were 0.472 for sst and 0.718 for para. The sts test correlation was 0.711. Figure 3 plots the histogram of training loss for Reptile training, showing the less stable reduction of loss, due to the two-layer optimization scheme.

Since the meta-learning accuracy of para task is much lower than the individual fine-tuning, the meta training was repeated by increasing the number of para training samples explored during the training (increasing para_iter to 8). The resulting accuracies were 0.400 and 0.746 for sst and para and the correlation was 0.412 for sts.

The meta-learning may bring more improvements for smaller training sets. To test this, 10% of training data were selected for three tasks. Individual training for sst, para and sts gave 0.260, 0.707 and 0.530. Multi-task learning had 0.243, 0.627 and 0.512. The meta-learning had the best performance, 0.272, 0.707 and 0.704.

6 Analysis
Adding the meta-learning to multi-task learning introduces a new dimension to outperform the joint learning. It is a more flexible framework to control numbers of explored samples from every task; thus, a way to adjust task-specific performance. The training parameters for inner and outer loop can have impact. It is not clear how to find the optimal training parameters, especially if tasks are different in nature (as in this study). Results show increasing performance for one task can reduce performance for others.

The benefits of double loop optimization are related to the amount of training data in the downstream tasks. Results show task specific fine-tuning worked well when training data is abundant. When 10% of training data was used, the meta-learning had advantage. This leads to an interesting uscase. If a large number of downstream tasks were available and each is with limited labeled data, a meta-learning can offer extra gain by utilizing across task relevance.

Due to the limited resource, the hyper-parameter space was not fully explored and model architecture may not be optimal. There are other improvements, e.g. better loss function and regularization methods, to be combined.

7 Conclusion
In this study, a meta-learning training scheme was implemented for training three NLP tasks on the BERT backbone, using the Reptile algorithm. The results show meta-learning offered better performance than the joint-multitask training. When the amount of training data is limited, the meta-learning can outperform the task-specific fine-tuning.
Figure 3: Loss histogram of Reptile training.

References


Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve.


Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning.