

COSC490LLModels

Homework 6: Adapting Transformers Language Models for Your Problems

Name: _____Hayden Komasz_____

Collaborators, if any: _____Doug Hoover_____

Sources used for your homework, if any: _____ChatGPT_____

This assignment will return to basics of Transformers and dig deeper into the fundamental concepts. Additionally, we will get our hands dirty by fine-tuning Transformer language models and comparing their performance. We will also be working more closely with GPU processors.

Homework goals: After completing this homework, you should be comfortable with:

- Training a pre-trained Language Model for your tasks
- You will explore different variations of model training

How to hand in your written work: via MyClasses.

Collaboration: You can do this homework as a team of 2 people. Basically, you can write a single solution for both people and upload it as a single PDF and a single .zip file as a group.

1 Concepts, intuitions and big picture

1. How does the BERT model expect a pair of sentences to be processed?

- ☐ Tokens-of-sentence-1 [SEP] Tokens-of-sentence-2
- ☐ [CLS] Tokens-of-sentence-1 Tokens-of-sentence-2
- ☐ [CLS] Tokens-of-sentence-1 [SEP] Tokens-of-sentence-2

Answer:

2. When should you train a new tokenizer?

- ☐ When your dataset is similar to that used by an existing pretrained model, and you want to pretrain a new model
- ☐ When your dataset is similar to that used by an existing pretrained model, and you want to fine-tune a new model using this pretrained model
- ☐ When your dataset is different from the one used by an existing pretrained model, and you want to pretrain a new model

Answer:

3. Select the sentences that apply to the BPE model of tokenization.

- ☒ BPE is a subword tokenization algorithm that starts with a small vocabulary and learns merge rules.
- ☐ BPE is a subword tokenization algorithm that starts with a big vocabulary and progressively removes tokens from it.
- ☐ BPE tokenizers learn merge rules by merging the pair of tokens that is the most frequent.
- ☐ A BPE tokenizer learns a merge rule by merging the pair of tokens that maximizes a score that privileges frequent pairs with less frequent individual parts.
- ☐ BPE tokenizes words into subwords by splitting them into characters and then applying the merge rules.
- ☐ BPE tokenizes words into subwords by finding the longest subword starting from the beginning that is in the vocabulary, then repeating the process for the rest of the text.

Answer:

4. What are the labels in a masked language modeling problem?

- ☐ Some of the tokens in the input sentence are randomly masked and the labels are the original input tokens.
- ☐ Some of the tokens in the input sentence are randomly masked and the labels are the original input tokens, shifted to the left.
- ☐ Some of the tokens in the input sentence are randomly masked, and the label is whether the sentence is positive or negative.
- ☐ Some of the tokens in the two input sentences are randomly masked, and the label is whether the two sentences are similar or not.

Answer:

5. When should you pretrain a new model?

- ☐ When there is no pretrained model available for your specific language
- ☐ When you have lots of data available, even if there is a pretrained model that could work on it
- ☐ When you have concerns about the bias of the pretrained model you are using

Answer:

6. Is there something wrong with the following code?

```
from transformers import AutoTokenizer, AutoModel

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
model = AutoModel.from_pretrained("gpt2")

encoded = tokenizer("Hey!", return_tensors="pt")
result = model(**encoded)
```

- ☐ No, it seems correct.
- ☐ The tokenizer and model should always be from the same checkpoint.
- ☐ It's good practice to pad and truncate with the tokenizer as every input is a batch.

Homework 6, a challenge so grand,
Transformers await at my command.
BERT, T5, and GPT stand tall,
Will my fine-tuning conquer them all?

Hyperparameters, a delicate art,
Adjusting with care, a model to start.
Learning rates dance, embeddings refine,
Seeking performance, both sharp and divine.

With every epoch, the losses decline,
Attention heads align in design.
Yet prompts demand a guiding hand,
To shape responses, thoughtful and planned.

Iteration by iteration, the models adapt,
Patterns emerge, connections are mapped.
Errors persist, but lessons unfold,
As insights deepen, both daring and bold.

Homework 6, you tested my mind,
But left behind knowledge refined.
For now, I wield these Transformers with might,
Prompting with skill, to generate right.

–ChatGPT Feb 25, 2025

Answer:

2 Getting Your Attention with Self-Attention

Recall that the transformer architecture uses scaled dot-product attention to compute *attention weights*:

$$\alpha^{(t)} = \text{softmax} \left(\frac{\mathbf{q}_t \mathbf{K}^\top}{\sqrt{h}} \right) \in [0, 1]^n$$

The resulting embedding in the output of attention at position t are:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_t = \sum_{t'=1}^n \alpha_{t'}^{(t)} \mathbf{v}_{t'} \in \mathbb{R}^{1 \times h},$$

where $\alpha^{(t)} = [\alpha_0^{(t)}, \dots, \alpha_n^{(t)}]$. The same idea can be stated in a matrix form,

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q} \mathbf{K}^\top}{\sqrt{h}} \right) \mathbf{V} \in \mathbb{R}^{n \times h}.$$

In the above equations:

- h is the hidden state dimension and n is the input sequence length;
- $\mathbf{X} \in \mathbb{R}^{n \times h}$ is the input to the attention;
- $\mathbf{x}_t \in \mathbb{R}^{1 \times h}$ is the slice of \mathbf{X} at position t , i.e. vector representation (embedding) of the input token at position t ;
- $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{h \times h}$ are the projection matrices to build query, key and value representations;
- $\mathbf{Q} = \mathbf{X} \mathbf{W}_q \in \mathbb{R}^{n \times h}, \mathbf{K} = \mathbf{X} \mathbf{W}_k \in \mathbb{R}^{n \times h}, \mathbf{V} = \mathbf{X} \mathbf{W}_v \in \mathbb{R}^{n \times h}$ are the query, key and value representations;
- $\mathbf{q}_t = \mathbf{x}_t \mathbf{W}_q \in \mathbb{R}^{1 \times h}$ is the slice of \mathbf{Q} at position t . Similarly, $\mathbf{k}_t = \mathbf{x}_t \mathbf{W}_k \in \mathbb{R}^{1 \times h}$ and $\mathbf{v}_t = \mathbf{x}_t \mathbf{W}_v \in \mathbb{R}^{1 \times h}$.

2.1 Forget 'Softmax', 'Argmax' is the New Boss in Town!

Recall from lectures that we can think about attention as being queryable softmax pooling. In this problem, we ask you to consider a hypothetical 'argmax' version of attention where it returns exactly the value corresponding to the key that is most similar to the query, where similarity is measured using the traditional inner-product. In other words, here use a version of Attention that instead of using softmax we use argmax to generate outputs from the attention layer. For example, $\text{softmax}([1, 3, 2])$ becomes $\text{argmax}([1, 3, 2]) = [0, 1, 0]$.

1. Perform argmax attention with the following keys and values:

$$\text{Keys} = \left\{ \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ -2 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ -2 \end{bmatrix} \right\}, \text{Values} = \left\{ \begin{bmatrix} 3 \\ 2 \\ -4 \end{bmatrix}, \begin{bmatrix} 3 \\ -2 \\ 4 \end{bmatrix}, \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix} \right\}$$

using the following query:

$$\mathbf{q} = \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}$$

What would be the output of the attention layer for this query? Remember, to simplify calculations, use an argmax instead of softmax. Note about notation: each vector in 'Keys = {.'}' and 'Values{.'}' are key/value vectors corresponding to four, non-interchangeable positions (i.e., ordering of these vectors matter).

Answer:

$$\begin{aligned} \mathbf{q} \cdot \mathbf{k}_1 &= (-2)(0) + (1)(1) + (1)(2) = 0 + 1 + 2 = 3 \\ \mathbf{q} \cdot \mathbf{k}_2 &= (-2)(1) + (1)(1) + (1)(2) = -2 + 1 + 2 = 1 \\ \mathbf{q} \cdot \mathbf{k}_3 &= (-2)(-1) + (1)(1) + (1)(-2) = 2 + 1 - 2 = 1 \\ \mathbf{q} \cdot \mathbf{k}_4 &= (-2)(0) + (1)(-1) + (1)(-2) = 0 - 1 - 2 = -3 \end{aligned}$$

Maximum dot product is with $\mathbf{k}_1 \Rightarrow$ output is \mathbf{v}_1 :

$$\begin{bmatrix} 3 \\ 2 \\ -4 \end{bmatrix}$$

2. How does this design choice affect our ability to usefully train models involving attention? (**Hint:** think about how the gradients flow through the network in the backward pass. Can we learn to improve our queries or keys during the training process?)

Answer: Using $\arg\max$ creates a non-differentiable function, which prevents gradients from flowing backwards. This makes it impossible to train the attention weights using gradient descent.

2.2 Superposition of Information in Self-Attention

We will show that the attention mechanism is able to copy the information from its input, whenever needed.*

1. **‘Copying’ mechanism in self-attention:** We’ll first show that it is particularly simple for attention to “copy” a value vector to the output. Describe (in 1-2 sentences) what properties of the inputs to the attention operation would result in the output $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_t$ being approximately equal to \mathbf{v}_j for some $j \in [0 \dots n]$.

Hint: Show that there can exist $j \in [0 \dots n]$ such that $\alpha_j^{(t)} \gg \sum_{i \neq j} \alpha_i^{(t)}$, if certain property holds for the query \mathbf{q}_t , and the keys $\{\mathbf{k}_0 \dots \mathbf{k}_n\}$.

Answer: If the query \mathbf{q}_t is very similar to a single key \mathbf{k}_j (and dissimilar to others), then:

$$\alpha_j(t) \gg \sum_{i \neq j} \alpha_i(t)$$

Because the softmax will assign nearly all attention weight to \mathbf{v}_j , so:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_t \approx \mathbf{v}_j$$

2. **Extracting signals after averaging them:** Instead of focusing on just one vector \mathbf{v}_j , attention mechanism might want to incorporate information from multiple source vectors. Consider the case where we instead want to incorporate information from two vectors \mathbf{v}_a and \mathbf{v}_b , with corresponding key vectors \mathbf{k}_a and \mathbf{k}_b . How should we combine information from two value vectors \mathbf{v}_a and \mathbf{v}_b ? A common way to combine values vectors is to average them: $\bar{\mathbf{v}} = \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$. However, after such averaging it is not quite clear how to tease apart the original information in value vectors \mathbf{v}_a and \mathbf{v}_b . Unless ... some special properties hold!

Suppose that although we don’t know \mathbf{v}_a or \mathbf{v}_b , we do know that \mathbf{v}_a lies in a subspace[†] A formed by the m basis vectors[‡] $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m\}$, while \mathbf{v}_b lies in a subspace B formed by the p basis vectors $\{\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_p\}$. This means that any \mathbf{v}_a can be expressed as a linear combination of its basis vectors, as can \mathbf{v}_b . All basis vectors have norm 1 and are orthogonal to each other. Additionally, suppose that the two subspaces are orthogonal; i.e. $\mathbf{a}_j^\top \mathbf{b}_k = 0$ for all j, k . Using the basis vectors $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m\}$, construct a matrix M such that for arbitrary vectors $\mathbf{v}_a \in A$ and $\mathbf{v}_b \in B$, we can use M to extract \mathbf{v}_a from the average vector $\bar{\mathbf{v}}$. In other words, we want to construct M such that for any $\mathbf{v}_a, \mathbf{v}_b$, $M\bar{\mathbf{v}} = \mathbf{v}_a$. Show that $M\bar{\mathbf{v}} = \mathbf{v}_a$ holds for your M .

Answer: We are told:

$$\mathbf{v}_a \in A = \text{span}(\mathbf{a}_1, \dots, \mathbf{a}_m)$$

$$\mathbf{v}_b \in B = \text{span}(\mathbf{b}_1, \dots, \mathbf{b}_p)$$

All \mathbf{a}_i and \mathbf{b}_j are orthonormal. Subspaces A and B are orthogonal:

$$\mathbf{a}_i^\top \mathbf{b}_j = 0$$

*Question credit: John Hewitt

[†]https://en.wikipedia.org/wiki/Linear_subspace

[‡][https://en.wikipedia.org/wiki/Basis_\(linear_algebra\)](https://en.wikipedia.org/wiki/Basis_(linear_algebra))

We want a matrix M that projects onto subspace A and extracts \mathbf{v}_a from the average. Construct projection matrix onto A :

$$M = \sum_{i=1}^m \mathbf{a}_i \mathbf{a}_i^\top$$

Then apply to the average:

$$M\bar{\mathbf{v}} = M\left(\frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)\right) = \frac{1}{2}(M\mathbf{v}_a + M\mathbf{v}_b)$$

Since M is the projection onto A :

$$M\mathbf{v}_a = \mathbf{v}_a \quad \text{and} \quad M\mathbf{v}_b = 0 \quad (\text{since } \mathbf{v}_b \in B \perp A)$$

So:

$$M\bar{\mathbf{v}} = \frac{1}{2}(\mathbf{v}_a + 0) = \mathbf{v}_a$$

3. **Averaging pairs of representations:** As before, let \mathbf{v}_a and \mathbf{v}_b be two value vectors corresponding to key vectors \mathbf{k}_a and \mathbf{k}_b , respectively. Assume that (1) all key vectors are orthogonal, so $\mathbf{k}_i^\top \mathbf{k}_j = 0$ for all $i \neq j$; and (2) all key vectors have norm 1. Find an expression for a query vector \mathbf{q}_t such that $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_t = \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$ and justify your answer.

Answer: We want:

$$\text{Attention}(Q, K, V)_t = \frac{1}{2}(v_a + v_b)$$

Given: Keys k_a, k_b are orthonormal: $\|k_i\| = 1$, and $k_i^\top k_j = 0$ for $i \neq j$. Let the query be:

$$q_t = \frac{1}{2}(k_a + k_b)$$

Then compute dot products:

$$q_t \cdot k_a = \frac{1}{2}(k_a^\top k_a + k_b^\top k_a) = \frac{1}{2}(1 + 0) = \frac{1}{2}$$

$$q_t \cdot k_b = \frac{1}{2}(k_a^\top k_b + k_b^\top k_b) = \frac{1}{2}(0 + 1) = \frac{1}{2}$$

These will yield equal attention scores, so softmax gives $\frac{1}{2}$ to each. Therefore, the output is:

$$\text{Attention}(Q, K, V)_t = \frac{1}{2}(v_a + v_b)$$

3 Programming

In this programming homework, we will finetune encoder LMs for a classification task with yes/no questions. In particular, we will

- Implement full-parameter finetuning.
- Explore various parameter-efficient finetuning strategies.

Skeleton Code and Structure: The code base for this homework can be found at MyClasses/Files under the hw6 directory. Your task is to fill in the missing parts in the skeleton code, following the requirements, guidance, and tips provided in this pdf and the comments in the corresponding .py files. The code base has the following structure:

- `base_classification.py`: implements the basic full finetuning
- `classification.py`: implements different parameter-efficient finetuning

TODOs — Your tasks include 1) generate plots and/or write short answers based on the results of running the code; 2) fill in the blanks in the skeleton to complete the code. We will explicitly mark these plotting, written answer, and filling-in-the-blank tasks as **TODOs** in the following descriptions, as well as a `# TODO` at the corresponding blank in the code.

Submission: Your submission should contain two parts: 1) plots and short answers under the corresponding questions below; and 2) your completion of the skeleton code base, in a .zip file, which includes the two provided skeleton files, and additionally a `classification_lora.py` (details in [item 6](#)).

Overview and Dataset In this programming assignment, you will get your hands dirtier with building self-supervised models. You will build a classifier using the Huggingface Transformer library and PyTorch. This classifier is expected to solve the BoolQ dataset [Clark et al., 2019]. You can find this dataset on Huggingface’s dataset hub: <https://huggingface.co/datasets/boolq>. In this task, each input consists of a question statement and a passage that may contain the answer to this question. The output is the answer to the question which may be one of ‘Yes’ or ‘No’ labels. Here is an example:

```
{
  "passage": "\"All biomass goes through at least some of these steps: it needs to be grown,
                                                    collected, dried, fermented, distilled, and
                                                    burned...\"",
  "question": "does ethanol take more energy make that produces"
  "answer": false,
}
```

The data comes with training and validation subsets. We will select a subset of the training data as ‘test’ set. So for the main experiments we will use 8k instances for training and the rest for testing.

Python Scripts Arguments Many of the **TODOs** below involves setting different training configurations and hyper-parameters, this could be achieved through the arguments provided in `main` of both `base_classification.py` and `classification.py` - they should be self-explanatory.

General Hint Follow the specifications and requirements in the code comments to complete missing lines.

3.1 Finetuning: Basics

1. Let’s start with the basic full finetuning, where all the parameters will be updated during the training. Specifically, we will use `AutoModelForSequenceClassification` which implements a classifier that maps the representation of encoder LMs (BERT, RoBERTa, etc.) into a fixed set of labels. Use this class to build a binary classifier using the representations of `distilbert-base-uncased` [Sanh et al., 2020]. The starter code for full finetuning in `base_classification.py` has a bunch of basic functionalities implemented, but it is missing

details.

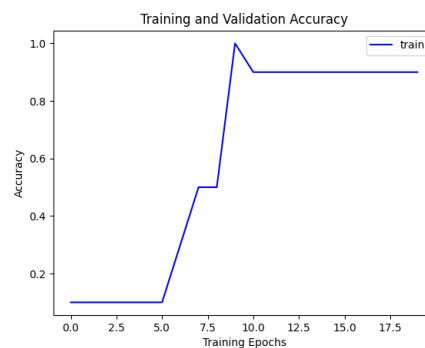
TODOs: Read and complete the missing lines in `evaluate_model` and `train` functions in `base_classification.py` to complete the functionalities of full finetuning.

To make sure your implementation is correct, we can train the model on 10 training instances for a few epochs (say, a total of 20 epochs). Note that over-fitting a few training examples is a good first test to make sure your implementation can successfully learn the patterns of your data. You should be able to complete the starter code and run it on a few instances in your local machine. You should be able to see that your classifier successfully over-fits its few training examples.

TODOs Run `base_classification.py` with the following command on your local machine, paste the generated plot of training accuracy vs. epochs plots below.

```
> python base_classification.py --small_subset --device cuda --model "distilbert-base-uncased"
--batch_size "64" --lr 1e-4 --num_epochs 20
```

your plot

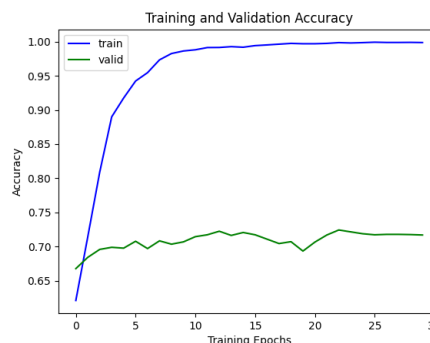


2. Now, rather than running training on a subset of the data, try training with all the data on GPUs. Make sure to find the largest batch size that can fit in the Henson HPL GPUs. You can start with a reasonable number, say 32. If it fits, double it to 64. If it does not fit, halve the value to 16.

TODOs Train for 30 epochs and paste the generated plot of train and dev accuracies vs. epochs below. Also report the largest batch size.

Hint: You should see that training accuracy goes high, while the dev accuracy will go high and then turn downhill as the model overfits the data.

your plot



Answer: Max batch size: 128

3. Let's implement hyper-parameter selection by exploring the combination of different hyper-parameter choices.

TODOs: Train models for various choices of learning rates (1e-4, 5e-4, 1e-3) and training epochs (7, 9). Select the model with the highest accuracy on the dev set and report its accuracy on the test set. Also report the corresponding choice of hyperparameters

Answer:

Best accuracy of distilBERT-base-uncased on the dev set: 0.71834

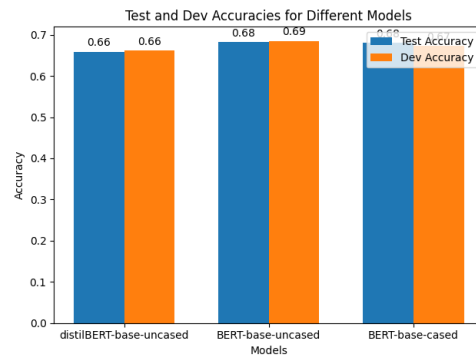
Best accuracy of distilBERT-base-uncased on the test set: 0.74211

Learning rate: 1e-4

Training epochs: 9

4. **TODOs** Now repeat the previous experiment with other models. In particular, create a **bar plot** that shows test/dev accuracies for two of your favorite of the following models: distilBERT-base-uncased, BERT-base-uncased, BERT-large-uncased, BERT-base-cased, BERT-large-cased, RoBERTa-base, RoBERTa-large. Note, it is possible that some of these models wouldn't fit in Henson HPL GPUs for any choice of batch size, in which case report 0 for their performance.

your plot



3.2 Parameter-Efficient Fine-Tuning

In this section, we will implement two more efficient ways of tuning language models: **Head-Tuning**, which freezes all parameters but the classification head, and **Prefix-Tuning**, which also freezes all parameters but the classification head, and additionally appends a fixed length ℓ of trainable embeddings to the input embeddings. See figure 1 for an illustration of which parameters are getting updated in each fine-tuning method we will explore in this homework.

Note that "output features" tensor for a single sequence of input should be of shape (length, hidden size), or equivalently $\mathbb{R}^{N \times d}$. However, the classifier expects a tensor of shape $\mathbb{R}^{1 \times d}$. There are multiple ways to transform the output features to $\mathbb{R}^{1 \times d}$, but in this work, we will take the **averaged** token representation.

Instead of using `AutoModelForSequenceClassification`, we now use the base `AutoModel` class to tailor the model to our needs. But don't worry, a skeleton code is provided to you in `classification.py`, you would only need to modify the details.

1. For comparison, let's redo the full finetuning first. Change the base model to RoBERTa-base. Implement full fine-tuning by adding a binary classifier on the averaged sentence representation.

TODOs: Read and complete the missing lines

(a) in `evaluate_model` function

(b) after `for index, batch in tqdm(enumerate(train_dataloader)):` in `train` function

in `classification.py` to complete the functionalities of full finetuning.

Hint: Your implementation should be the same as that of `base_classification.py` for `evaluate_model` and almost the same except for how we get the logits for train (until loss backward).

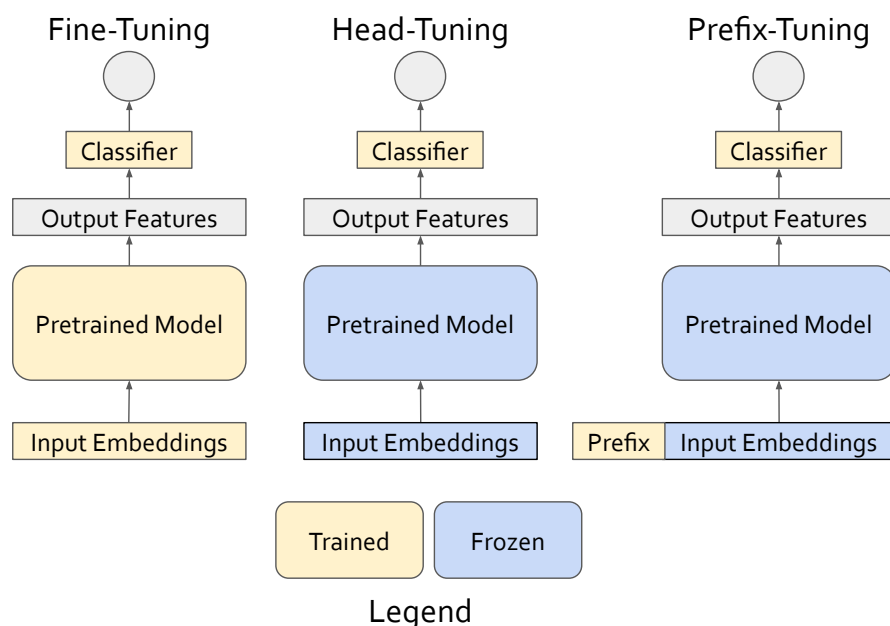


Figure 1: Overview of fine-tuning methods. Blue indicates parts that are frozen (no gradient update) during the fine-tuning process. Yellow indicates parts that are trained (with gradient updates).

TODOs: Read and complete the missing lines after `if self.type == 'full'` in the forward function of the `CustomModelForSequenceClassification` class in `classification.py`. Run the code with `type="full"` to train the model with full finetuning. Report the final accuracy you obtained along with the hyper-parameters you used (number of epochs and learning rate).

Answer:

dev accuracy: 0.62171

test accuracy: 0.62298

epochs: 7

learning rate: 5e-5

2. **TODOs:** Write down the number of tuned parameters for head-tuning for BoolQ, given that the model hidden size is d .

Answer: 1538

Now let's implement head-tuning by adding a binary classifier on the averaged sentence representation.

TODOs: Read and complete the missing lines after `if self.type == 'head'` in the forward function of the `CustomModelForSequenceClassification` class in `classification.py`. You would also need to modify the optimizer by adding code after `mymodel.type == 'head':` in the train function in `classification.py`. Run the code with `type="head"` to train the model with head-tuning. Report the final accuracy on full head-tuning along with the hyper-parameters you used[§].

Answer:

dev accuracy: 0.62171

test accuracy: 0.65311

epochs: 7

learning rate: 5e-5

3. **TODOs:** Write down the number of tuned parameters for our prefix-tuning on BoolQ that appends trainable prefix embeddings of length 128, given the model hidden size is d .

[§]You would be only graded by the correctness of your implementation, but if you want good performance, Try using a larger learning rate and longer training epochs.

Answer: 1538

It's time to implement prefix tuning!

TODOs: Read and complete the missing lines after `if self.type == 'prefix'` in the forward function of the `CustomModelForSequenceClassification` class in `classification.py`. You would also need to modify the optimizer by adding code after `mymodel.type == "prefix"` in the `train` function in `classification.py`. Run the code with `type="prefix"` to train the model with prefix-tuning. Report the final accuracy of prefix-tuning along with the hyper-parameters you used [¶].

Answer:

dev accuracy: 0.62171

test accuracy: 0.61387

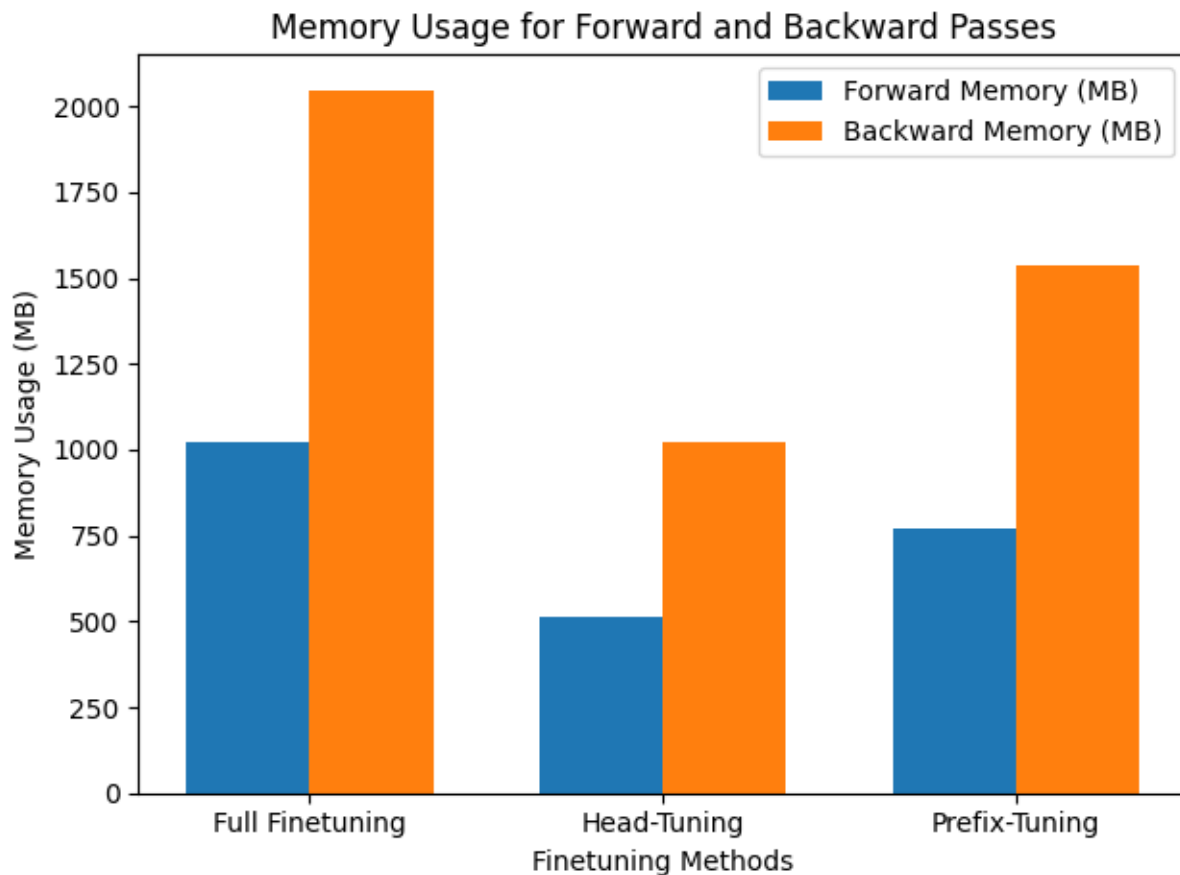
epochs: 7

learning rate: 5e-5

4. Here is a simple [tutorial](#) on how to use `torch.profiler` to record the memory usage of forward and backward operations.

TODOs Generate a bar plot of the forward/backward memory usage of full-finetuning, head-tuning, and prefix-tuning. Note that you don't need to upload any code for this.

your plot



5. **LoRA** fine-tuning [Hu et al., 2022] decomposes the parameter updates of a large dense matrix to two low-rank matrices. For a matrix M of size $\mathbb{R}^{d \times d}$, the full updates require storing the gradients and optimizer states of

[¶]Again, you would be only graded by the correctness of your implementation, but if you want good performance, Try using a larger learning rate and longer training epochs.

$d \times d$ parameters.

TODOs: However, if we decompose the gradient update on the full matrix $\Delta M \approx AB$ where $A \in \mathbb{R}^{d \times n}$ and $B \in \mathbb{R}^{n \times d}$, write down the number of parameters whose gradients and optimizer states we need to store:
Answer: 12288 .

6. Implement LoRA fine-tuning.

TODOs: Create `classification_lora.py` and copy `classification_base.py` to it and implement the LoRA fine-tuning method using the `peft` library. This should be just changing a few lines of code. Report the trainable parameters and final accuracy of your LoRA fine-tuning, as well as your hyper-parameters.

Answer:

dev accuracy: 0.68531

test accuracy: 0.68010

epochs: 7

learning rate: 5e-5

References

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1300. URL <https://aclanthology.org/N19-1300>.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. URL <https://openreview.net/forum?id=nZeVKeeFYf9>.