# Natural Language Processing, Assignment 4

**November 14th, 2023**

## 1 Introduction

This assignment consists of two parts. In Part 1, you will be doing some mathematical exploration, which will use some mathematical investigations to illuminate a few of the motivations of self-attention and Transformer networks. In Part 2, you will be finetuning a classifier based on BERT, and trying a trick: joint training with masked language modeling.

You are required to finish this homework in the Python (version 3) programming language. You are only allowed to use PyTorch, no other external libraries are allowed (e.g., transformers).

**You should submit the report, your code, and the scripts that can reproduce all your experiments in a .zip file!**

## 2 Part 1: Attention Exploration

**Problem 1 :** (21 points) Multi-headed self-attention is the core modeling component of Transformers. In this question, we'll get some practice working with the self-attention equations, and motivate why multi-headed self-attention can be preferable to single-headed self-attention.

1. (6 points) **Copying in attention**
   (a) (2 points) Recall that attention can be viewed as an operation on a query $q \in \mathbb{R}^d$, a set of value vectors $\{v_1, ..., v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, .., k_n\}, k_i \in \mathbb{R}^d$, specified as follows:

$$c = \sum_{i=1}^{n} v_i \alpha_i$$

$$\alpha_i = \frac{exp(k_i^T q)}{\sum_{j=1}^{n} exp(k_j^T q)}$$

   where $\alpha_i$ are frequently called the "attention weights", and the output $c \in \mathbb{R}^d$ is a correspondingly weighted average over the value vectors. We'll frst show that it's particularly simple for attention to "copy" a value vector to the output c. Describe (in one sentence) what properties of the inputs to the attention operation would result in the output $c$ being approximately equal to $v_j$ for some $j \in \{1, ..., n\}$. Specifically, what must be true about the query $q$, the values $\{v_1, ..., v_n\}$ and/or the keys $\{k_1, ..., k_n\}$?

   (b) (4 points) Consider a set of key vectors $\{k_1, ..., k_n\}$ where all key vectors are perpendicular, that is $k_i \perp k_j$ for all $i \neq j$. Let $\parallel k_i \parallel = 1$ for all i. Let $\{v_1, ..., v_n\}$ be a set of arbitrary value vectors. Let $v_a, v_b \in \{v_1, ..., v_n\}$ be two of the value vectors. Give an expression for a query vector $q$ such that the output $c$ is approximately equal to the average of $v_a$ and $v_b$, that is, $\frac{1}{2}(v_a + v_b)$ [1]. Note that you can reference the corresponding key vector of $v_a$

[1] Hint: while the softmax function will never exactly average the two vectors, you can get close by using a large scalar multiple in the expression.

and $v_b$ as $k_a$ and $k_b$.

2. (5 points) **Drawbacks of single-headed attention:** In the previous part, we saw how it was possible for a single-headed attention to focus equally on two values. The same concept could easily be extended to any subset of values. In this question we'll see why it's not a practical solution. Consider a set of key vectors $\{k_1, ..., k_n\}$ that are now randomly sampled, $k_i \sim N(\mu_i, \Sigma_i)$, where the means $\mu_i$ are known to you, but the covariances $\Sigma_i$ are unknown. Further, assume that the means $\mu_i$ are all perpendicular; $\mu_i^T \mu_j = 0$ if $i \neq j$, and unit norm, $\parallel \mu_i \parallel = 1$.
    (a) (2 points) Assume that the covariance matrices are $\Sigma_i = \alpha I$, for vanishingly small $\alpha$. Design a query $q$ in terms of the $\mu_i$ such that as before, $c \approx \frac{1}{2}(v_a + v_b)$, and provide a brief argument as to why it works.
    (b) (3 points) Though single-headed attention is resistant to small perturbations in the keys, some types of larger perturbations may pose a bigger issue. Specifically, in some cases, one key vector $k_a$ may be larger or smaller in norm than the others, while still pointing in the same direction as $\mu_a$. As an example, let us consider a covariance for item $a$ as $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^T)$ for vanishingly small $\alpha$. Further, let $\Sigma_i = \alpha I$ for all $i \neq a$. When you sample $\{k_1, ..., k_n\}$ multiple times, and use the $q$ vector that you defined in question 1.2 (a), what qualitatively do you expect the vector $c$ will look like for different samples?
3. (3 points) **Benefts of multi-headed attention:** Now we'll see some of the power of multi-headed attention. We'll consider a simple version of multi-headed attention which is identical to single-headed self-attention as we've presented it in this homework, except two query vectors ($q_1$ and $q_2$) are defined, which leads to a pair of vectors ($c_1$ and $c_2$), each the output of single-headed attention given its respective query vector. The final output of the multi-headed attention is their average, $\frac{1}{2}(c_1 + c_2)$. As in question 1.2, consider a set of key vectors $\{k_1, ..., k_n\}$ that are randomly sampled, $k_i \sim N(\mu_i, \Sigma_i)$, where the means $\mu_i$ are known to you, but the covariances $\Sigma_i$ are unknown. Also as before, assume that the means $\mu_i$ are mutually orthogonal; $\mu_i^T \mu_j = 0$ if $i \neq j$, and unit norm, $\parallel \mu_i \parallel = 1$.
    (a) (1 points) Assume that the covariance matrices are $\Sigma_i = \alpha I$, for vanishingly small $\alpha$. Design $q_1$ and $q_2$ such that $c$ is approximately equal to $\frac{1}{2}(v_a + v_b)$.
    (b) (2 points) Assume that the covariance matrices are $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^T)$ for vanishingly small $\alpha$, and $\Sigma_i = \alpha I$ for all $i \neq a$. Take the query vectors $q_1$ and $q_2$ that you designed in question 1.3 (a). What, qualitatively, do you expect the output c to look like across different samples of the key vectors? Please briefly explain why. You can ignore cases in which $q_i^T k_a \leq 0$.
4. (7 points) **Key-Query-Value self-attention in neural networks:** So far, we've discussed attention as a function on a set of key vectors, a set of value vectors, and a query vector. In Transformers, we perform self-attention, which roughly means that we draw the keys, values, and queries from the same data. More precisely, let $\{x_1, ..., x_n\}$ be a sequence of vectors in $\mathbb{R}^d$. Think of each $x_i$ as representing word $i$ in a sentence. One form of self-attention defines keys, queries, and values as follows. Let $V, K, Q \in \mathbb{R}^{d \times d}$ be parameter matrices. Then

$$v_i = Vx_i, i \in \{1, ..., n\}$$

$$k_i = Kx_i, i \in \{1, ..., n\}$$

$$q_i = Qx_i, i \in \{1, ..., n\}$$

Then we get a context vector for each input $i$; we have $c_i = \sum_{j=1}^{n} \alpha_{ij} v_j$, where $\alpha_{ij}$ is defined as $\alpha_{ij} = \frac{\exp(k_j^T q_i)}{\sum_{l=1}^{n} \exp(k_l^T q_i)}$. Note that this is single-headed self-attention.

In this question, we'll show how key-value-query attention like this allows the network to use different aspects of the input vectors $x_i$ in how it defines keys, queries, and values. Intuitively, this allows networks to choose different aspects of $x_i$ to be the "content" (value vector) versus what it uses to determine "where to look" for content (keys and queries.)

(a) (3 points) First, consider if we didn't have key-query-value attention. For keys, queries, and values we'll just use $x_i$; that is, $v_i = q_i = k_i = x_i$. We'll consider a specific set of $x_i$. In particular, let $u_a, u_b, u_c, u_d$ be mutually orthogonal vectors in $\mathbb{R}^d$, each with equal norm $\| u_a \| = \| u_b \| = \| u_c \| = \| u_d \| = \beta$, where $\beta$ is very large. Now, let our $x_i$ be:

$$x_1 = u_d + u_b$$

$$x_2 = u_a$$

$$x_3 = u_c + u_b$$

If we perform self-attention with these vectors, what vector does $c_2$ approximate? Would it be possible for $c_2$ to approximate $u_b$ by adding either $u_d$ or $u_c$ to $x_2$? Explain why or why not (either math or English is fine).

(b) (4 points) Now consider using key-query-value attention as we've defined it originally. Using the same definitions of $x_1$, $x_2$ and $x_3$ as in part (a), specify matrices $K$, $Q$, $V$ such that $c_2 \approx u_b$, and $c_1 \approx u_b - u_c$. There are many solutions to this problem, so it will be easier for you (and the graders), if you first find $V$ such that $v_1 = u_b$ and $v_3 = u_b - u_c$, then work on $Q$ and $K$. Some outer product properties may be helpful (as summarized in this footnote [2]).

# 3 Part 2: Conduct Text Classification with Transformer

In this section, we will learn how to finetune a BERT model for sentiment analysis. Considering that some students may not easily achieve the GPU resources, we will experiment with TinyBert (same architecture as BERT, but smaller size). You needn't care about the construction process of TinyBert. You can just view it as a small size version of BERT. If implemented correctly, you can achieve an accuracy above 0.55 with a CPU in an hour.

To generate BERT input, we will add a special token "[CLS]" at the beginning of each sentence and a special token "[SEP]" at the end of each sentence. If there are more than one sentences, we will separate them with "[SEP]". To build the classifier, use the final hidden vector $C \in \mathbb{R}^H$ corresponding to the first input token "[CLS]" as the aggregate representation, and then feed it into a fully-connected softmax layer with dropout.

**Problem 2 :** (20 points) **Build Text Classifier with Transformer**

Firstly, read the file `classifier.py`, and learn the process of training and evaluation. We will evaluate every *logging_steps*, and save the best model on the development set. After training, we will test its performance on the test set. You should implement the class `BertForSequenceClassification` in `mymodel.py`. The unfinished parts are marked with "#todo". If you do not finish them, "NotImplementedError" will be raised. After that, train the classifier and evaluate with the data (`yelp_small`) we provided in homework 3.

A default command is as follows. By running the default command, you should get a test accuracy score above 0.55. Adjust some hyper-parameters, and analyze their performance.

```
python classifier.py \
```

[2] For orthogonal vectors $u, v, w \in \mathbb{R}^d$, the outer product $uv^T$ is a matrix in $\mathbb{R}^{d \times d}$, and $(uv^T)v = u(v^T v) = u\| v \|_2^2$, and $(uv^T)w = u(v^T w) = u * 0$. (The last equality is because $v$ and $w$ are orthogonal.)

```
--model_type bert \
--model_name_or_path huawei-noah/TinyBERT_General_4L_312D \
--task_name yelp \
--do_train \
--do_eval \
--data_dir ${YOUR_DATA_DIR} \
--per_gpu_train_batch_size 32 \
--learning_rate ${CLS_LR} \
--num_train_epochs ${CLS_EPOCHS} \
--max_seq_length ${MAX_LENGTH} \
--output_dir ${YOUR_OUTPUT_DIR} \
--evaluate_during_training \
--overwrite_output_dir
```

Here, "YOUR_DATA_DIR" denotes where your data is saved. "YOUR_OUTPUT_DIR" denotes your output path. "CLS_LR" denotes your learning rate for training (for example, 3e-5), "CLS_EPOCHS" denotes the epochs you train (for example, 3), and "MAX_LENGTH" denotes the length limit of your sequence (for example, 128).

After you finished your experiments, the following contents should be reported:

1. Your code snippet of the class `BertForSequenceClassification` in `mymodel.py`.
2. The hyper-parameters you choose, the best validation score, test score, and your analysis.

**Problem 3 :** (30 points) **Joint Training of MLM and CLS** Sometimes, joint training the desired task and the language model on unlabeled data within the same domain may improve the performance of downstream tasks. Here we want to explore its effectiveness.

Masked Language Modeling (MLM) task is masking $r\%$ (for example, 15%) WordPiece tokens in each sequence at random and then trying to predict them. In practice, we may not replace all chosen words with "[MASK]", but replace it with (1) the [MASK] token 80% of the time (2) a random token 10% of the time (3) the unchanged token 10% of the time.

To be specific, to get a classification model joint trained with MLM, the following steps are required: (1) Prepare two dataloaders, one for classification, and another for masked language model (which takes the masked sentences as inputs and masked words as targets). We have implemented the two dataloaders in our code (see `data_utils.py` and `mlm_utils.py` for more details). (2) Train the classifier and MLM model jointly, i.e., you should calculate the classification loss and MLM loss at the same time. Then update the parameters of the model w.r.t. the two losses (can be balanced with a hyper-parameter). (3) Choose the best checkpoint with the development set of the classification task. Evaluate the classification model on test data.

We have finished most of the codes, and provided an unlabeled dataset named "unlabeled_train_50000.txt", which can be downloaded in https://cloud.tsinghua.edu.cn/f/7a34965b2169416bb8b5/?dl=1. (Note that you do not have to use all the unlabeled data.) You need to:

1. Complete the code:
   (a) implement the class `BertForMaskedLanguageModel` in `mymodel.py`.
   (b) get the MLM inputs in `classifier_mlm.py` with the help of our codes for the dataloader and the function of masking tokens.
2. Run the model with the following commands, and observe whether this trick can improve the performance. You can edit the file "classifier_mlm.py" as you like to get better performance.
3. Cut the number of the labeled training data to 50, retrain the basic classifier and the CLS-MLM

joint classifier. Observe whether this trick can improve performance. Note that training with a smaller dataset may need training for more epochs.

A default command is as follows.

```
python classifier_mlm.py \
    --model_type bert \
    --model_name_or_path huawei-noah/TinyBERT_General_4L_312D \
    --task_name yelp \
    --do_train \
    --do_eval \
    --data_dir ${YOUR_DATA_DIR} \
    --per_gpu_train_batch_size 32 \
    --learning_rate ${CLS_LR} \
    --num_train_epochs ${CLS_EPOCHS} \
    --logging_steps 50 \
    --max_seq_length ${Max_LENGTH} \
    --output_dir ${YOUR_OUTPUT_DIR} \
    --evaluate_during_training \
    --train_data_file ${YOUR_UNLABELED_DATA_DIR}/unlabeled_train_50000.txt \
    --block_size 128 \
    --mlm_alpha ${MLM_ALPHA} \
    --line_by_line \
    --overwrite_output_dir
```

In the command above, "YOUR_DATA_DIR", "YOUR_OUTPUT_DIR", "CLS_LR", "CLS_EPOCHS", and "Max_LENGTH" are the same as in Problem 3. "YOUR_UNLABELED_DATA_DIR" denotes where you save your unlabeled data. "MLM_ALPHA" is the hyper-parameter to balance the classification loss and MLM loss.

After you finished your experiments, the following contents should be reported:

1. Your code snippet of the class `BertForCLSMLM` in `mymodel.py`.
2. For each size of the dataset, report the hyper-parameters you choose, the validation score, test score, and your analysis.