# XCS224N Problem Set 5 Self-attention, Transformers, Pretraining

**Due Sunday, May 4 at 11:59pm PT.**

## Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs224n-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

## Submission Instructions

**Written Submission:** Some extra credit questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

**Coding Submission:** Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

## Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

## Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

**Note.** Here are some things to keep in mind as you plan your time for this assignment.

- The total amount of PyTorch code to write, and code complexity, of this assignment is lower than Assignment 4. However, you're also given less guidance or scaffolding in how to write the code.
- This assignment involves a pretraining step that takes approximately 1 hour to perform on Azure, and you'll have to do it twice. The 1 hour timeline is an upper bound on the training time assuming older/slower GPU. On faster GPUs, the pretraining can finish in around 30-40 minutes.

This assignment is an investigation into Transformer self-attention building blocks, and the effects of pretraining. It covers mathematical properties of Transformers and self-attention through written questions. Further, you'll get experience with practical system-building through repurposing an existing codebase. The assignment is split into a coding part and an extra credit written (mathematical) part. Here's a quick summary:

1. **Extending a research codebase:** In this portion of the assignment, you'll get some experience and intuition for a cutting-edge research topic in NLP: teaching NLP models facts about the world through pretraining, and accessing that knowledge through finetuning. You'll train a Transformer model to attempt to answer simple questions of the form "Where was person [x] born?" – without providing any input text from which to draw the answer. You'll find that models are able to learn some facts about where people were born through pretraining, and access that information during fine-tuning to answer the questions.

   Then, you'll take a harder look at the system you built, and reason about the implications and concerns about relying on such implicit pretrained knowledge.

2. **Mathematical exploration:** What kinds of operations can self-attention easily implement? Why should we use fancier things like multi-headed self-attention? This section will use some mathematical investigations to illuminate a few of the motivations of self-attention and Transformer networks. **Note:** for all questions, you should justify your answer with mathematical reasoning when required.

# 1 Pretrained Transformer models and knowledge access

You'll train a Transformer to perform a task that involves accessing knowledge about the world – knowledge which isn't provided via the task's training data (at least if you want to generalize outside the training set). You'll find that it more or less fails entirely at the task. You'll then learn how to pretrain that Transformer on Wikipedia text that contains world knowledge, and find that finetuning that Transformer on the same knowledge-intensive task enables the model to access some of the knowledge learned at pretraining time. You'll find that this enables models to perform considerably above chance on a held out development set.

The code you're provided with is a fork of Andrej Karpathy's `minGPT`. It's nicer than most research code in that it's relatively simple and transparent. The "GPT" in `minGPT` refers to the Transformer language model of OpenAI, originally described in this paper [1].

As in previous assignments, you will want to develop on your machine locally, then run training on Azure. You can use the same conda environment from previous assignments for local development, and the same process for training on Azure (see the *Practical Guide for Using the VM* section of the XCS224N Azure Guide for a refresher). You might find the troubleshooting section useful if you see any issue in conda environment and GPU usage. Specifically, you'll still be running "`conda activate XCS224N_CUDA`" on the Azure machine. You'll need around 2 hours for training, so budget your time accordingly!

Your work with this codebase is as follows:

(a) **[0 points (Coding)] Review the minGPT demo code (no need to submit code or written)**

*Note that you do not have to write any code or submit written answers for this part.*

In the `src/submission/mingpt-demo/` folder, there is a Jupyter notebook (`play_char.ipynb`) that trains and samples from a Transformer language model. Take a look at it locally on your computer and you might need to install Jupyter notebootk `pip install jupyter` or use vscode [2] to get somewhat familiar with the code how it defines and trains models. *You don't need to run the train locally, because training will take long time on CPU on your local environment.* Some of the code you are writing below will be inspired by what you see in this notebook.

(b) **[0 points (Coding)] Read through `NameDataset` in `src/submission/dataset.py`, our dataset for reading name-birth place pairs.**
The task we'll be working on with our pretrained models is attempting to access the birth place of a notable person, as written in their Wikipedia page. We'll think of this as a particularly simple form of question answering:

*Q: Where was [person] born?*
*A: [place]*

From now on, you'll be working with the `src/submission` folder. **The code in `mingpt-demo/` won't be changed or evaluated for this assignment.** In `dataset.py`, you'll find the the class `NameDataset`, which reads a TSV (tab-separated values) file of name/place pairs and produces examples of the above form that we can feed to our Transformer model.

To get a sense of the examples we'll be working with, if you run the following code, it'll load your `NameDataset` on the training set `birth_places_train.tsv` and print out a few examples.

```
cd src/submission
python dataset.py namedata
```

Note that you do not have to write any code or submit written answers for this part.

(c) **[20 points (Coding)] Define a *span corruption* function for pretraining.**
In the file `src/submission/dataset.py`, implement the `__getitem__()` function for the dataset class `CharCorruptionDataset`. Follow the instructions provided in the comments in `dataset.py`. Span corruption

[1] https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
[2] https://code.visualstudio.com/docs/datascience/jupyter-notebooks

is explored in the T5 paper [3]. It randomly selects spans of text in a document and replaces them with unique tokens (noising). Models take this noised text, and are required to output a pattern of each unique sentinel followed by the tokens that were replaced by that sentinel in the input. In this question, you'll implement a simplification that only masks out a single sequence of characters.

This question will be graded via autograder based on your whether span corruption function implements some basic properties of our spec. We'll instantiate the `CharCorruptionDataset` with our own data, and draw examples from it.

To help you debug, if you run the following code, it'll sample a few examples from your `CharCorruptionDataset` on the pretraining dataset `wiki.txt` and print them out for you.

```
    cd src/submission
    python dataset.py charcorruption
```

No written answer is required for this part.

(d) **[4 points (Coding)] Implement finetuning (without pretraining).**
Take a look at `src/submission/helper.py`, which is used by `src/run.py`.

It has some skeleton code you will implement to *pretrain*, *finetune* or *evaluate* a model. For now, we'll focus on the finetuning function, in the case without pretraining.

Taking inspiration from the training code in the `src/submission/mingpt-demo/play_char.ipynb` jupyter notebook file, write code to finetune a Transformer model on the name/birth place dataset, via examples from the `NameDataset` class. For now, implement the case without pretraining (i.e. create a model from scratch and train it on the birth-place prediction task from part (b)). You'll have to modify three sections, marked `[part d]` in the code: one to initialize the model, one to finetune it, and one to train it. Note that you only need to initialize the model in the case labeled "vanilla" for now (later in section (g), we will explore a model variant). Use the hyperparameters for the `Trainer` specified in the `src/submission/helper.py` code.

Also take a look at the *evaluation* code which has been implemented for you. It samples predictions from the trained model and calls `evaluate_places()` to get the total percentage of correct place predictions. You will run this code in part (d) to evaluate your trained models.

*Note that this is an intermediate step for later portions, including Part (e), which contains commands you can run to check your implementation. No written answer is required for this part.*

(e) **[6 points (Coding)] Make predictions (without pretraining).**
Train your model on `birth_places_train.tsv`, and evaluate on `birth_dev.tsv` and `birth_test.tsv`. Specifically, you should now be able to run the following three commands:

```
cd src

# Train on the names dataset
./run.sh vanilla_finetune_without_pretrain

# Evaluate on the dev set, writing out predictions
./run.sh vanilla_eval_dev_without_pretrain

# Evaluate on the test set, writing out predictions
./run.sh vanilla_eval_test_without_pretrain
```

Training will take less than 10 minutes (on Azure). Your grades will be based on the output files from the run.

Don't be surprised if the evaluation result is well below 10%; we will be digging into why in Part 2. As a reference point, we want to also calculate the accuracy the model would have achieved if it had just predicted "London" as the birth place for everyone in the dev set.

[3] https://arxiv.org/pdf/1910.10683.pdf

(f) **[20 points (Coding)] Pretrain, finetune, and make predictions. Budget 1 hour for training.** Now fill in the *pretrain* portion of `src/submission/helper.py`, which will pretrain a model on the span corruption task. Additionally, modify your *finetune* portion to handle finetuning in the case *with* pretraining. In particular, if a path to a pretrained model is provided in the bash command, load this model before finetuning it on the birth-place prediction task. Pretrain your model on `wiki.txt` (which should take approximately one hour), finetune it on `NameDataset` and evaluate it. Specifically, you should be able to run the following four commands:

```
cd src

# Pretrain the model
./run.sh vanilla_pretrain

# Finetune the model
./run.sh vanilla_finetune_with_pretrain

# Evaluate on the dev set; write to disk
./run.sh vanilla_eval_dev_with_pretrain

# Evaluate on the test set; write to disk
./run.sh vanilla_eval_test_with_pretrain
```

We expect the dev accuracy will be at least 10%, and will expect a similar accuracy on the held out test set.

(g) **[4 points (Written)] Different kind of position embeddings**

In the previous part, you used the vanilla Transformer model, which used learned positional embeddings. In Section 4, you also learned about the sinusoidal positional embeddings used in the original Transformer paper. In this part, you'll implement a different kind of positional embedding, called *RoPE* (Rotary Positional Embedding)[4].

RoPE is a fixed positional embedding that is designed to encode relative position rather than absolute position. The issue with absolute positions is that if the transformer won't perform well on context lengths (e.g. 1000) much larger than it was trained on (e.g. 128), because the distribution of the position embeddings will be very different from the ones it was trained on. Relative position embeddings like RoPE alleviate this issue.

Given a feature vector with two features $x_t^{(1)}$ and $x_t^{(2)}$ at position $t$ in the sequence, the RoPE positional embedding is defined as:

$$\text{RoPE}(x_t^{(1)}, x_t^{(2)}, t) = \begin{pmatrix} \cos t\theta & -\sin t\theta \\ \sin t\theta & \cos t\theta \end{pmatrix} \begin{pmatrix} x_t^{(1)} \\ x_t^{(2)} \end{pmatrix}$$

where $\theta$ is a fixed angle. For two features, the RoPE operation corresponds to a 2D rotation of the features by an angle $t\theta$. Note that the angle is a function of the position $t$.

For a $d$ dimensional feature, RoPE is applied to each pair of features with an angle $\theta_i$ defined as $\theta_i = 10000^{-2(i-1)/d},\ i \in \{1, 2, \ldots, d/2\}$.

$$\begin{pmatrix} \cos t\theta_1 & -\sin t\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin t\theta_1 & \cos t\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos t\theta_2 & -\sin t\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin t\theta_2 & \cos t\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos t\theta_{d/2} & -\sin t\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin t\theta_{d/2} & \cos t\theta_{d/2} \end{pmatrix} \begin{pmatrix} x_t^{(1)} \\ x_t^{(2)} \\ x_t^{(3)} \\ x_t^{(4)} \\ \vdots \\ x_t^{(d-1)} \\ x_t^{(d)} \end{pmatrix} \tag{1}$$

Finally, instead of adding the positional embeddings to the input embeddings, RoPE is applied to the key and query vectors for each head in the attention block for all the Transformer layers.

i. (2 points) Using the rotation interpretation, RoPE operation can be viewed as rotation of the complex number $x_t^{(1)} + ix_t^{(2)}$ by an angle $t\theta$. Recall that this corresponds to multiplication by $e^{it\theta} = \cos t\theta + i \sin t\theta$.

For higher dimensional feature vectors, this interpretation allows us to compute Equation 1 more efficiently. Specifically, we can rewrite the RoPE operation as an element-wise multiplication (denoted by $\odot$) of two vectors as follows:

$$\begin{pmatrix} \cos t\theta_1 + i \sin t\theta_1 \\ \cos t\theta_2 + i \sin t\theta_2 \\ \vdots \\ \cos t\theta_{d/2} + i \sin t\theta_{d/2} \end{pmatrix} \odot \begin{pmatrix} x_t^{(1)} + ix_t^{(2)} \\ x_t^{(3)} + ix_t^{(4)} \\ \vdots \\ x_t^{(d-1)} + ix_t^{(d)} \end{pmatrix} \tag{2}$$

Show that the elements of the vector in Equation 1 can be obtained from Equation 2. Note that some additional operations like reshaping are necessary to make the two expressions equal but you do not need to provide a detailed derivation for full points.

ii. (2 point) **Relative Embeddings.** Now we will show that the dot product of the RoPE embeddings of two vectors at positions $t_1$ and $t_2$ depends on the relative position $t_1 - t_2$ only.

[4] https://arxiv.org/abs/2104.09864

For simiplicity, we will assume two dimensional feature vectors (eg. $[a, b]$) and work with their complex number representations (eg. $a + ib$).

Show that $\langle \text{RoPE}(z_1, t_1), \text{RoPE}(z_2, t_2)\rangle = \langle \text{RoPE}(z_1, t_1 - t_2), \text{RoPE}(z_2, 0)\rangle$ where $\langle \cdot, \cdot \rangle$ denotes the dot product and $\text{RoPE}(z, t)$ is the RoPE embedding of vector $z$ at position $t$.

(Hint: Dot product of vectors represented as complex numbers is given by $\langle z_1, z_2\rangle = \mathbb{R}(\overline{z_1} z_2)$. For a complex number $z = a + ib$ ($a, b \in \mathbb{R}$), $\mathbb{R}(z) = a$ indicates the real component of $z$ and $\bar{z} = a - ib$ is the complex conjugate of $z$.)

(h) **[10 points (Coding)] Write and try out a different kind of position embeddings (Budget about 1 hour for training)**

In the provided code, RoPE is implemented using the functions `precompute_rotary_emb` and `apply_rotary_emb` in `src/submission/attention.py`. You need to implement these functions and the parts of code marked `[part h]` in `src/submission/attention.py` and `src/run.py` to use RoPE in the model.

Train a model with RoPE on the span corruption task and finetune it on the birthplace prediction task. Specifically, you should be able to run the following four commands:

```
cd src

# Pretrain the model
./run.sh rope_pretrain

# Finetune the model
./run.sh rope_finetune_with_pretrain

# Evaluate on the dev set; write to disk
./run.sh rope_eval_dev_with_pretrain

# Evaluate on the test set; write to disk
./run.sh rope_eval_test_with_pretrain
```

We'll score your model as to whether it gets at least 20% accuracy on the dev set.

### Deliverables

For this assignment, please submit the following files within the src/submission directory. Update files **without directory structure**.

This includes:

1. `src/submission/__init__.py`
2. `src/submission/attention.py`
3. `src/submission/dataset.py`
4. `src/submission/helper.py`
5. `src/submission/model.py`
6. `src/submission/trainer.py`
7. `src/submission/utils.py`
8. `src/submission/vanilla.model.params`
9. `src/submission/vanilla.nopretrain.dev.predictions`
10. `src/submission/vanilla.nopretrain.test.predictions`
11. `src/submission/vanilla.pretrain.params`
12. `src/submission/vanilla.finetune.params`
13. `src/submission/vanilla.pretrain.dev.predictions`
14. `src/submission/vanilla.pretrain.test.predictions`
15. `src/submission/rope.pretrain.params`
16. `src/submission/rope.finetune.params`
17. `src/submission/rope.pretrain.dev.predictions`
18. `src/submission/rope.pretrain.test.predictions`

We provide a script `src/collect_submission.sh` to collect these files and create a zip file for submission. You can run the script on Linux/Mac systems or use the `collect_submission.bat` on Windows, then submit the zip file to the assignment.

```
cd src
bash ./collect_submission.sh
```

# 2 Considerations in pretrained knowledge

In this section, we are giving you the intuitions and considerations from the pretrained Transformer coding in the previous section.

These are not graded and we encourage you to read the following questions and answers.

(a) Succinctly explain why the pretrained (vanilla) model was able to achieve a higher accuracy than the accuracy of the non-pretrained.

Pretraining, with some probability, masks out the name of a person while providing the birth place, or masks out the birth place while providing the name – this teaches the model to associate the names with the birthplaces. At finetuning time, this information can be accessed, since it has been encoded in the parameters (the initialization.) Without pretraining, there's no way for the model to have any knowledge of the birth places of people that weren't in the finetuning training set, so it can't get above a simple heuristic baseline (like the London baseline.)

(b) Take a look at some of the correct predictions of the pretrain+finetuned vanilla model, as well as some of the errors. We think you'll find that it's impossible to tell, just looking at the output, whether the model *retrieved* the correct birth place, or *made up* an incorrect birth place. Consider the implications of this for user-facing systems that involve pretrained NLP components. Come up with two reasons why this indeterminacy of model behavior may cause concern for such applications.

There is a large space of possible reasons indeterminacy could cause concern for user-facing applications. Here are some possible answers:

(a) Users will always get outputs that look valid (if the user doesn't know the real answer) and so won't be able to perform quality estimation themselves (like one sometimes can when, e.g., a translation seems nonsensical). System designers also don't have a way of filtering outputs for low-confidence predictions. Users may believe invalid answers and make incorrect decisions (or inadvertently spread disinformation) as a result.

(b) Once users realize the system can output plausible but incorrect answers, they may stop trusting the system, therefore making it useless.

(c) Models will not indicate that they simply do not know the birth place of a person (unlike a relational database or similar, which will return that the knowledge is not contained in it). This means the system cannot indicate a question is unanswerable.

(d) Made up answers could be biased or offensive.

(e) There is little avenue for recourse if users believe an answer is wrong, as it's impossible to determine the reasoning of the model is retrieving some gold standard knowledge (in which case the user's request to change the knowledge should be rejected), or just making up something (in which case the user's request to change the knowledge should be granted).

(c) If your model didn't see a person's name at pretraining time, and that person was not seen at fine-tuning time either, it is not possible for it to have "learned" where they lived. Yet, your model will produce *something* as a predicted birth place for that person's name if asked. Concisely describe a strategy your model might take for predicting a birth place for that person's name, and one reason why this should cause concern for the use of such applications.

1. The model could use character-level phonetic-like (sound-like) information to make judgments about where a person was born based on how their name "sounds", likely leading to racist outputs.

2. The model could learn that certain names or types of names tend to be of people born in richer cities, leading to classist outputs that predict a birth place simply based on whether the names are like that of rich people or poorer people.

# 3 Attention exploration

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.

Recall that attention can be viewed as an operation on a query $q \in \mathbb{R}^d$, a set of value vectors $\{v_1, \ldots, v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, \ldots, k_n\}, k_i \in \mathbb{R}^d$, specified as follows:

$$c = \sum_{i=1}^{n} v_i \alpha_i \tag{3}$$

$$\alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)}. \tag{4}$$

with $\alpha = \{\alpha_1, \ldots, \alpha_n\}$ termed the "attention weights". Observe that the output $c \in \mathbb{R}^d$ is an average over the value vectors weighted with respect to $\alpha$.

(a) **[2 points (Written, Extra Credit)] Copying in attention:** One advantage of attention is that it's particularly easy to "copy" a value vector to the output $c$. In this problem, we'll motivate why this is the case.

i. (1 point) The distribution $\alpha$ is typically relatively "diffuse"; the probability mass is spread out between many different $\alpha_i$. However, this is not always the case. **Describe** (in one sentence) under what conditions the categorical distribution $\alpha$ puts almost all of its weight on some $\alpha_j$, where $j \in \{1, \ldots, n\}$ (i.e. $\alpha_j \gg \sum_{i \neq j} \alpha_i$). What must be true about the query $q$ and/or the keys $\{k_1, \ldots, k_n\}$?

ii. (1 point) Under the conditions you gave in (i), **describe** the output $c$.

(b) **[2 points (Written, Extra Credit)] An average of two:**

Instead of focusing on just one vector $v_j$, a Transformer model might want to incorporate information from *multiple* source vectors.

Consider the case where we instead want to incorporate information from **two** vectors $v_a$ and $v_b$, with corresponding key vectors $k_a$ and $k_b$. Assume that (1) all key vectors are orthogonal, so $k_i^\top k_j = 0$ for all $i \neq j$; and (2) all key vectors have norm 1. **Find an expression** for a query vector $q$ such that $c \approx \frac{1}{2}(v_a + v_b)$, and **justify your answer**.[5] (Recall what you learned in part (a).)

(c) **[3 points (Written, Extra Credit)] 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 $\{k_1, \ldots, k_n\}$ that are now randomly sampled, $k_i \sim \mathcal{N}(\mu_i, \Sigma_i)$, where the means $\mu_i$ are known to you, but the covariances $\Sigma_i$ are unknown (unless specified otherwise in the question). Further, assume that the means $\mu_i$ are all perpendicular; $\mu_i^\top \mu_j = 0$ if $i \neq j$, and unit norm, $\|\mu_i\| = 1$.

i. (1 point) Assume that the covariance matrices are $\Sigma_i = \alpha I, \forall i \in \{1, 2, \ldots, n\}$, 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.

ii. (2 point) 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 $\mu_a$ [6].

As an example, let us consider a covariance for item $a$ as $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^\top)$ for vanishingly small $\alpha$ (as shown in figure 1). Further, let $\Sigma_i = \alpha I$ for all $i \neq a$.

[5] 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.

[6] Unlike the original Transformer, some newer Transformer models apply layer normalization before attention. In these pre-layernorm models, norms of keys cannot be too different which makes the situation in this question less likely to occur.

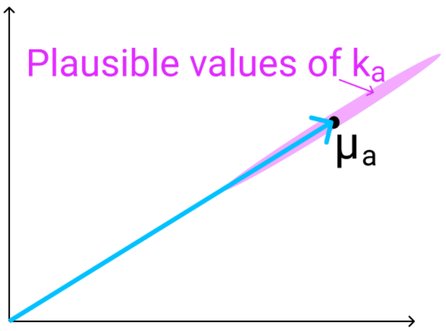

Figure 1: The vector $\mu_a$ (shown here in 2D as an example), with the range of possible values of $k_a$ shown in red. As mentioned previously, $k_a$ points in roughly the same direction as $\mu_a$, but may have larger or smaller magnitude.

When you sample $\{k_1, \ldots, k_n\}$ multiple times, and use the $q$ vector that you defined in part i., what do you expect the vector $c$ will look like qualitatively for different samples? Think about how it differs from part (i) and how $c$'s variance would be affected.

(d) **[2 points (Written, Extra Credit)] Benefits 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 3((c)), consider a set of key vectors $\{k_1, \ldots, k_n\}$ that are randomly sampled, $k_i \sim \mathcal{N}(\mu_i, \Sigma_i)$, 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; $\mu_i^\top \mu_j = 0$ if $i \neq j$, and unit norm, $\|\mu_i\| = 1$.

i. (1 point) Assume that the covariance matrices are $\Sigma_i = \alpha I$, for vanishingly small $\alpha$. Design $q_1$ and $q_2$ in terms of $\mu_i$ such that $c$ is approximately equal to $\frac{1}{2}(v_a + v_b)$. Note that $q_1$ and $q_2$ should have different expressions.

ii. (1 points) Assume that the covariance matrices are $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^\top)$ 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 part i. 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 $q_i^\top k_a < 0$.

(e) **[1 point (Written, Extra Credit)]** Based on part (d), briefly summarize how multi-headed attention overcomes the drawbacks of single-headed attention that you identified in part (c).

# 4 Position Embeddings Exploration

Position embeddings are an important component of the Transformer architecture, allowing the model to differentiate between tokens based on their position in the sequence. In this question, we'll explore the need for positional embeddings in Transformers and how they can be designed.

Recall that the crucial components of the Transformer architecture are the self-attention layer and the feed-forward neural network layer. Given an input tensor $\mathbf{X} \in \mathbb{R}^{T\times d}$, where $T$ is the sequence length and $d$ is the hidden dimension, the self-attention layer computes the following:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V$$
$$\mathbf{H} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}$$

where $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d\times d}$ are weight matrices, and $\mathbf{H} \in \mathbb{R}^{T\times d}$ is the output.

Next, the feed-forward layer applies the following transformation:

$$\mathbf{Z} = \text{ReLU}(\mathbf{H}\mathbf{W}_1 + \mathbf{1}\cdot\mathbf{b}_1)\mathbf{W}_2 + \mathbf{1}\cdot\mathbf{b}_2$$

where $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d\times d}$ and $\mathbf{b}_1, \mathbf{b}_2 \in \mathbb{R}^{1\times d}$ are weights and biases; $\mathbf{1} \in \mathbb{R}^{T\times 1}$ is a vector of ones[7]; and $\mathbf{Z} \in \mathbb{R}^{T\times d}$ is the final output.

(Note that we have omitted some details of the Transformer architecture for simplicity.)

(a) **[3 points (Written, Extra Credit)] Permuting the input.**

i. (2 point) Suppose we permute the input sequence $\mathbf{X}$ such that the tokens are shuffled randomly. This can be represented as multiplication by a permutation matrix $\mathbf{P} \in \mathbb{R}^{T\times T}$, i.e. $\mathbf{X}_{\text{perm}} = \mathbf{P}\mathbf{X}$. (See Wikipedia for a recap on permutation matrices.)

**Show** that the output $\mathbf{Z}_{\text{perm}}$ for the permuted input $\mathbf{X}_{\text{perm}}$ will be $\mathbf{Z}_{\text{perm}} = \mathbf{P}\mathbf{Z}$.

You are given that for any permutation matrix $\mathbf{P}$ and any matrix $\mathbf{A}$, the following hold: $\text{softmax}(\mathbf{P}\mathbf{A}\mathbf{P}^\top) = \mathbf{P}\ \text{softmax}(\mathbf{A})\ \mathbf{P}^\top$ and $\text{ReLU}(\mathbf{P}\mathbf{A}) = \mathbf{P}\ \text{ReLU}(\mathbf{A})$.

ii. (1 point) Think about the implications of the result you derived in part i. **Explain** why this property of the Transformer model could be problematic when processing text.

(b) **[2 points (Written, Extra Credit)] Position embeddings** are vectors that encode the position of each token in the sequence. They are added to the input word embeddings before feeding them into the Transformer.

One approach is to generate position embedding using a fixed function of the position and the dimension of the embedding. If the input word embeddings are $\mathbf{X} \in \mathbb{R}^{T\times d}$, the position embeddings $\Phi \in \mathbb{R}^{T\times d}$ are generated as follows:

$$\Phi_{(t,2i)} = \sin\left(t/10000^{2i/d}\right)$$
$$\Phi_{(t,2i+1)} = \cos\left(t/10000^{2i/d}\right)$$

where $t \in \{0, 1, \ldots T-1\}$ and $i \in \{0, 1, \ldots d/2-1\}$[8].

Specifically, the position embeddings are added to the input word embeddings:

$$\mathbf{X}_{\text{pos}} = \mathbf{X} + \Phi$$

i. (1 point) Do you think the position embeddings will help the issue you identified in part (a)? If yes, explain how and if not, explain why not.

ii. (1 point) Can the position embeddings for two different tokens in the input sequence be the same? If yes, provide an example. If not, explain why not.

[7] Outer product with **1** represents broadcasting operation and makes feed forward network notations mathematically sound.

[8] Here $d$ is assumed even which is typically the case for most models.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

---

# 1.g.i

## 1.g.ii

# 3.a.i

## 3.a.ii

# 3.b

# 3.c.i

## 3.c.ii

# 3.d.i

# 3.d.ii

# 3.e

# 4.a.i

## 4.a.ii

# 4.b.i

# 4.b.ii