CS 224N Winter 2026 Assignment 3
Self-Attention and Transformers

Due date: February 5th, Thursday, 4:30 PM PST
This assignment is an investigation into Transformers, the prevailing architecture used for frontier LLMs. The pset has three questions:

- In the first, you will gain intuition about how the self-attention mechanism in transformers works
- In part two, you will derive some properties of positional encodings.
- In the third part, you will code a transformer (almost) from scratch, and start training it on your laptop.

Please tag the questions correctly on Gradescope, otherwise the TAs will take points off if you don't tag questions.

For code submission, run `bash create_submission.sh`, which will zip your `model_solution.py`, `train.py`, and `utils.py`. Then directly upload the resultant `submission.zip` to Gradescope.

1. Attention Exploration (14 points)

Multi-head 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* vector $q \in \mathbb{R}^d$, 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:

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

$$\alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)} \tag{2}$$

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 α.

(a) (3 points) **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. (2 points) The distribution α is typically relatively "diffuse"; the probability mass is spread out between many different α_i. However, this is not always the case. **Describe** (in one sentence) under what conditions the categorical distribution α puts almost all of its weight on some α_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) **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**.[*] (Recall what you learned in part (a).)

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

  i. (2 points) Assume that the covariance matrices are $\Sigma_i = \alpha I, \forall i \in \{1, 2, \ldots, n\}$, for vanishingly small α. Design a query q in terms of the μ_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. (3 points) Though single-headed attention is resistant to small perturbations in the keys, some types of larger perturbations may pose a bigger issue. 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 μ_a.[†]
  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 α (as shown in figure 1). This causes k_a to point in roughly the same direction as μ_a, but with large variances in magnitude. Further, let $\Sigma_i = \alpha I$ for all $i \neq a$.

[*]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.

[†]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.

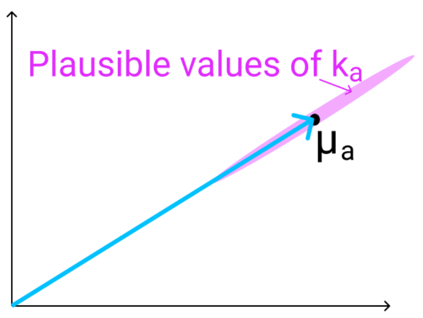

Figure 1: The vector μ_a (shown here in 2D as an example), with the range of possible values of k_a shown in red. As mentioned previously, k_a points in roughly the same direction as μ_a, but may have larger or smaller magnitude.

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) (3 points) **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, 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(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 μ_i are known to you, but the covariances Σ_i are unknown. Also as before, assume that the means μ_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 α. Design q_1 and q_2 in terms of μ_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. (2 points) Assume that the covariance matrices are $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^\top)$ for vanishingly small α, 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? Explain briefly in terms of variance in c_1 and c_2. You can ignore cases in which $k_a^\top q_i < 0$.

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

2. Position Embeddings Exploration (6 points)

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[‡]; 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) (4 points) **Permuting the input.**

i. (3 points) 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) **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 $t \in \{0, 1, \dots T-1\}$ and $i \in \{0, 1, \dots d/2-1\}$[§].

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.

[‡]Outer product with **1** represents broadcasting operation and makes feed forward network notations mathematically sound.
[§]Here d is assumed even which is typically the case for most models.

3. Coding a transformer from scratch (30 points)

In this question you will fill in code to implement a decoder only, GPT-2 style transformer, and a simple training loop.

For part (a), we have included unit tests for each sub problem that can run locally on your laptop. You will be awarded full points for the subproblem if you pass the unit test. Do not edit the unit test file as we will separately be running the tests when you submit your code.

The following tips might be useful during this part of the assignment:

- Add assert statements to check the shape of tensors matches what you think it should be.
- Consider the Jaxtyping package to type hint the shape of tensors.
- Consider the einops package for manipulating tensors (`einops.rearrange` is particularly useful).

This will help you not only write less buggy code, but also make your code far more readable.

(a) (20 points) In this part of the question we will implement a transformer in the `model_solution.py` file. The file contains a number of different classes that you will implement. In the end, you will have an implementation of the `Transformer` class with functioning `forward` and `generate` methods. In part (b), we will (start to) train your implementation of `Transformer`.

i. (0 points) Familiarize yourself with the classes in the `model_solution.py` file. We will ask you to implement them in the order `MLP`, `CausalAttention`, `DecoderBlock`, and finally `Transformer`. We will get you to implement the classes in this order because it is the order of dependence. `Transformer` depends on `DecoderBlock`, that in turn depends on `CausalAttention` and `MLP`.

ii. (1 point) Implement `MLP.forward`. Check you pass the corresponding test.

iii. (6 points) Implement `CausalAttention.forward`. Check you pass the corresponding test.

iv. (2 points) Implement `DecoderBlock.forward`. Check you pass the corresponding test.

v. (6 points) Implement `Transformer.forward`. Check you pass the corresponding test.

vi. (5 points) Implement `Transformer.generate`. Check you pass the corresponding test. Note: you should implement greedy decoding for this function..

(b) (10 points) After finishing part (a), you now have a functioning Transformer model. If you look at `Transformer.__innit__` we can see that when you create an instance of the `Transformer` class, we initialize the model with random weights according to the `Transformer._init_weights` method. In this part of the question, you will implement a training loop, and start training a small model locally on your laptop.

i. (0 points) Look in the `train.py` file and familiarize yourself with the training loop. We will run this code to train the model.

ii. (7 points) First, implement `Transformer.get_loss_on_batch`. This function maps a batch of tokens to a single loss value. We use this function in `train.py` to get the loss over a batch. Check you pass the corresponding test.

iii. (3 points) Run `train.py`. This will train the model, using your `Transformer.get_loss_on_batch` on 100 batches of data. At the end of training it will save a graph of the training loss and gradient norm over training to `losses_and_grad_norms.png`, include an image of this below.
If everything is correct, you should see a decreasing loss curve.

(c) (9 points) **(Bonus)** In this optional bonus question, your goal is to speed up the learning process. We will keep the number of gradient steps fixed to 100, however you can change anything else about `train.py` or `model.py` to speed up training.

We will consider a change to have succeeded if the final loss after 100 steps is lower than the baseline curve you reported in part biii).

We will award 3 points for each different change you make that leads to a speedup. Thus for full points, you will need to make three different changes to the training file, each of which leads to a speedup. These changes should compound. For example, you may begin by changing the learning rate, leading to a lower loss. You then might keep this better learning rate, and combine it with a second change (e.g., changing the optimizer or model architecture), that leads to an even lower loss. Note that changing the learning rate to three different values counts as one idea: we are looking for three different types of ideas.

When you are done, submit:

- A description of each change that you made.
- Up to three new learning curves, one for each change you made; and additionally include the baseline from biii).
- The lowest loss you achieved after 100 steps.