CS 224N: Assignment 5: Self-Attention, Transformers, and Pretraining

Anthony Weng

Due: Friday, February 17 @ 11:59 PM PST

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

- There are math questions again!
- 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 2 hours to perform on Azure, and you'll have to do it twice. Colab set-up notebook has been provided similar to Assignment 4. The 2 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 written (mathematical) part and a coding part, with its own written questions. Here's a quick summary:

1. **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.

2. **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.

This assignment was originally created by John Hewitt, CS 224N Head TA in Winter 2021.

1. Attention exploration (20 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) (5 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. (1 point) **Explain** why α can be interpreted as a categorical probability distribution.

Answer: α can be interpreted as a categorical probability distribution as it is generated from the softmax function (as seen in Equation (2)) which normalizes the input vector (here, $[k_1^\top q, \ldots, k_n^\top q]$) such that all the output vector components are bounded between (0, 1) and collectively sum to 1, enabling interpretation of the output vector (here, α) as a categorical probability distribution over the n different categories represented by the key vectors.

ii. (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 $\{k_1, \ldots, k_n\}$?

Answer: $\alpha_j \gg \sum_{i \neq j} \alpha_i$ when $k_i^\top q \approx 0$ for all $i \in \{1, \ldots n\}$ where $i \neq j$, meaning k_i and q are near-orthogonal for these i.

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

Answer: Since almost all the probability mass is placed on α_j, $\alpha_j \approx 1$ and therefore $c \approx v_j a_j \approx v_j$.

iv. (1 point) **Explain** (in two sentences or fewer) what your answer to (ii) and (iii) means intuitively.

Answer: From (ii): since k_i and q are orthogonal for almost all $i \in \{1, \ldots n\}$, this intuitively means that the query vector (representing the currently processed word) is largely unrelated to all of the key vectors (representing the tokens composing the considered context for the attention computation) except for one. From (iii): since the currently processed word is largely related to only one word in the surrounding context, this intuitively means that almost all of the attention is placed onto that context token, and accordingly the attention output c is almost a "copy" of the value vector v_j of said context token.

(b) (7 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.

i. (3 points) How should we combine two d-dimensional vectors v_a, v_b into one output vector c in a way that preserves information from both vectors? In machine learning, one common way to do so is to take the average: $c = \frac{1}{2}(v_a + v_b)$. It might seem hard to extract information about the original vectors v_a and v_b from the resulting c, but under certain conditions one can do so. In this problem, we'll see why this is the case.

Suppose that although we don't know v_a or v_b, we do know that v_a lies in a subspace A formed by the m basis vectors $\{a_1, a_2, \ldots, a_m\}$, while v_b lies in a subspace B formed by the p basis vectors $\{b_1, b_2, \ldots, b_p\}$. (This means that any v_a can be expressed as a linear combination of its basis vectors, as can v_b. All basis vectors have norm 1 and are orthogonal to each other.)

Additionally, suppose that the two subspaces are orthogonal; i.e. $a_j^\top b_k = 0$ for all j, k.

Using the basis vectors $\{a_1, a_2, \ldots, a_m\}$, construct a matrix M such that for arbitrary vectors $v_a \in A$ and $v_b \in B$, we can use M to extract v_a from the sum vector $s = v_a + v_b$. In other words, we want to construct M such that for any v_a, v_b, $Ms = v_a$. Show that $Ms = v_a$ holds for your M.

Hint: Given that the vectors $\{a_1, a_2, \ldots, a_m\}$ are both *orthogonal* and *form a basis* for v_a, we know that there exist some $c_1, c_2, \ldots, c_m$ such that $v_a = c_1 a_1 + c_2 a_2 + \cdots + c_m a_m$. Can you create a vector of these weights c?

Answer: From the hint, we note that $\exists c = [c_1, \ldots, c_m]$ such that $v_a = Ac$ where $A = \{a_1, \ldots, a_m\}$. Analogously, for encoding v_bj, we note that $\exists d = [d_1, \ldots, d_p]$ such that $v_b = Bd$ where $B = \{b_1, \ldots, b_p\}$. Then, we seek to construct a matrix M such that $Ms = M(Ac+Bd) = MAc + MBd = v_a$.

To satisfy this condition, $M = A(A^\top A)^{-1}A^\top$. To see why:

$$\begin{aligned}
Ms &= MAc + MBd \\
&= A(A^\top A)^{-1}A^\top Ac + A(A^\top A)^{-1}A^\top Bd \\
&= A(A^\top A)^{-1}(A^\top A)c + A(A^\top A)^{-1}(A^\top B)d \\
&= AIc + A(A^\top A)^{-1} \times 0 \times d \\
&= Ac = v_a
\end{aligned}$$

where $A^\top B = 0$ follows from the two subspaces being orthogonal (i.e., $a_j^\top b_k = 0$ for all j, k).

ii. (4 points) As before, let v_a and v_b be two value vectors corresponding to key vectors k_a and k_b, respectively. 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.[1] **Find an expression** for a query vector q such that $c \approx \frac{1}{2}(v_a + v_b)$, and justify your answer. [2]

Answer: Set $q = 1000k_a + 1000k_b$. Then, to see that $c \approx \frac{1}{2}(v_a + v_b)$ under these conditions, observe the following:

$$\begin{aligned}
\alpha_i &= \frac{\exp(k_i^\top q)}{\sum_{j=1}^n \exp(k_j^\top q)} \\
&= \frac{\exp(k_i^\top (1000k_a + 1000k_b))}{\sum_{j=1}^n \exp(k_j^\top (1000k_a + 1000k_b))} \\
&= \frac{\exp(1000k_a k_i^\top + 1000k_b k_i^\top)}{\sum_{j=1}^n \exp(1000k_a k_j^\top + 1000k_b k_j^\top)}
\end{aligned}$$

Since $k_i^\top k_j = 0$ for all $i \neq j$ and $k_i^\top k_i = 1$ as all key vectors have norm 1, the final expression in the derivation prior can be simplified as follows:

$$\begin{aligned}
\alpha_a &= \frac{\exp(1000k_a k_i^\top + 1000k_b k_i^\top)}{\sum_{j=1}^n \exp(1000k_a k_j^\top + 1000k_b k_j^\top)} \approx \frac{\exp(1000)}{\exp(1000) + \exp(1000)} \approx \frac{1}{2} \\
\alpha_b &\approx \frac{1}{2} \text{ (analogous derivation to } \alpha_a) \\
\alpha_i &\approx \frac{\exp(0)}{\exp(1000) + \exp(1000)} = \frac{1}{2\exp(1000)} \approx 0 \text{ for } i \neq a, b
\end{aligned}$$

Then, as $c = \sum_{i=1}^n v_i \alpha_i$, it follows that $c \approx \frac{1}{2}v_a + \frac{1}{2}v_b + \sum_{\substack{i=1 \\ i \neq a,b}}^{n} v_i \alpha_i \approx \frac{1}{2}v_a + \frac{1}{2}v_b + 0$, and we observe $c \approx \frac{1}{2}(v_a + v_b)$ as desired.

[1] Recall that a vector x has norm 1 iff $x^\top x = 1$.

[2] 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.

(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. 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.

Answer: Set $q = K\mu_a + K\mu_b$ where K is some large scalar constant (i.e., $K >> 0$). Then, since $\Sigma_i = \alpha I, \forall i \in \{1, 2, \ldots, n\}$, for vanishingly small α, it follows that:

- $\Sigma_i = 0$ as $\alpha \to 0$; and
- $k_i \to \mu_i$ as $\alpha \to 0$.

As such, $k_i^\top k_j \approx 0$ for $i \neq j$ and $k_i^\top k_i \approx 1$ since $\mu_i^\top \mu_j = 0$ if $i \neq j$ and $\|\mu_i\| = 1$. Also from $k_i \to \mu_i$ as $\alpha \to 0$ once more, it follows that $k_i^\top \mu_j \approx 0$ for $i \neq j$, and $k_i^\top \mu_i \approx 1$ as $\|\mu_i\| = 1$. Then, this value for q and this set of conditions for the key vectors is approximating those of question 1(b)(ii), meaning the resulting computations under the present conditions will approximate those of 1(b)(ii), leading to $c \approx \frac{1}{2}(v_a + v_b)$ as before, as desired.

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. 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 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 $i \neq a$.

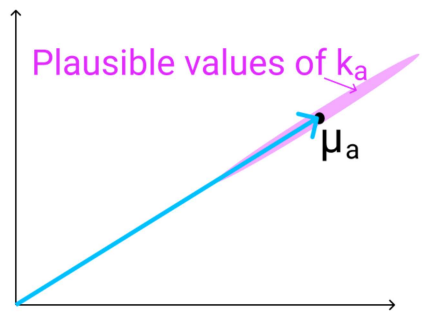

Figure 1: The vector μ_a (shown here in 2D as an example), with the range of possible values of k_a shown in red. As mentioned previously, k_a points in roughly the same direction as μ_a, but may have larger or smaller magnitude.

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.

Answer: For different samples, I expect the vector c to be heavily biased toward or away from the vector k_a (and correspondingly away or toward k_b) depending on whether k_a is greater or lesser than μ_a in magnitude, respectively. This is because k_a is involved in the softmax computation for each α_i, and changes in its magnitude will alter how much c is biased toward or away from it.

(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 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(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 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.

Answer: $c \approx \frac{1}{2}(v_a + v_b)$ can be achieved when $q_1 = K\mu_a$ and $q_2 = K\mu_b$ for some large scalar constant $K >> 0$.

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 $k_a^\top q_i < 0$.

Answer: For different samples, I expect the output vector c to be weakly biased toward or away from the vector k_a depending on whether k_a is greater or lesser than μ_a in magnitude, respectively. This variance in the output c will be similar in direction to the combined variance of c_1 and c_2, although it will be weaker in magnitude; to see why, note the following:

$$\begin{aligned} c &= \frac{1}{2}(c_1 + c_2) \\ Var(c) &= Var(\frac{1}{2}(c_1 + c_2)) \\ &= \frac{1}{4} Var(c_1 + c_2) \\ &= \frac{1}{4}(Var(c_1) + Var(c_2)) \text{ (assuming samples are drawn independently)} \end{aligned}$$

Thus, due to the averaging of c_1 and c_2 to produce c, c will experience variance and bias toward/away from k_a across the different samples, but it will be a lesser magnitude of variance, especially as compared to 1(c)(ii).

2. Pretrained Transformer models and knowledge access (35 points)

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 [2].

As in previous assignments, you will want to develop on your machine locally, then run training on Azure/Colab. You can use the same conda environment from previous assignments for local development, and the same process for training on a GPU.[3] You'll need around 5 hours for training, so budget your time accordingly! We have provided a sample Colab with the the commands that require GPU training. **Note that dataset multi-processing can fail on local machines without GPU, so to debug locally, you might have to change `num_workers` to 0.**

[3] See CS224n Azure Guide for a refresher on Azure.

Your work with this codebase is as follows:

(a) (0 points) **Check out the demo.**
In the `mingpt-demo/` folder 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) to get somewhat familiar with how it defines and trains models. Some of the code you're writing below will be inspired by what you see in this notebook.

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

(b) (0 points) **Read through `NameDataset` in `src/dataset.py`, our dataset for reading name-birthplace 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/` 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.

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

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

(c) (0 points) **Implement finetuning (without pretraining).**
Take a look at `run.py`. It has some skeleton code specifying flags you'll eventually need to handle as command line arguments. In particular, you might want to *pretrain*, *finetune*, or *evaluate* a model with this code. For now, we'll focus on the finetuning function, in the case without pretraining.

Taking inspiration from the training code in the `play_char.ipynb` file, write code to finetune a Transformer model on the name/birthplace 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 birthplace prediction task from part (b)). You'll have to modify two sections, marked `[part c]` in the code: one to initialize the model, and one to finetune 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 `run.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.

This is an intermediate step for later portions, including Part d, which contains commands you can run to check your implementation. No written answer is required for this part.

(d) (5 points) **Make predictions (without pretraining).**
Train your model on `birth_places_train.tsv`, and evaluate on `birth_dev.tsv`. Specifically, you should now be able to run the following three commands:

```
# Train on the names dataset
python src/run.py finetune vanilla wiki.txt \
       --writing_params_path vanilla.model.params \
       --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set, writing out predictions
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.model.params \
       --eval_corpus_path birth_dev.tsv \
       --outputs_path vanilla.nopretrain.dev.predictions

# Evaluate on the test set, writing out predictions
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.model.params \
       --eval_corpus_path birth_test_inputs.tsv \
       --outputs_path vanilla.nopretrain.test.predictions
```

Training will take less than 10 minutes (on Azure). Report your model's accuracy on the dev set (as printed by the second command above). Similar to assignment 4, we also have Tensorboard logging in assignment 5 for debugging. It can be launched using `tensorboard --logdir expt/`. Don't be surprised if it is well below 10%; we will be digging into why in Part 3. 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. Fill in `london_baseline.py` to calculate the accuracy of that approach and report your result in your write-up. You should be able to leverage existing code such that the file is only a few lines long.

Answer:

- Accuracy of trained model on dev set: 1.6% (8.0 correct out of 500.0)
- Accuracy of `london_baseline.py`: 5.0% (25.0 correct out of 500.0)

(e) (10 points) **Define a *span corruption* function for pretraining.**

In the file `src/dataset.py`, implement the `__getitem__()` function for the dataset class `CharCorruptionDataset`. Follow the instructions provided in the comments in `dataset.py`. Span corruption 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 whether your 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.

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

No written answer is required for this part.

(f) (10 points) **Pretrain, finetune, and make predictions. Budget 2 hours for training.**

Now fill in the *pretrain* portion of `run.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 birthplace prediction task. Pretrain your model on `wiki.txt` (which should take approximately two hours), finetune it on `NameDataset` and evaluate it. Specifically, you should be able to run the following four commands: (Don't be concerned if the loss appears to plateau in the middle of pretraining; it will eventually go back down.)

```
# Pretrain the model
python src/run.py pretrain vanilla wiki.txt \
       --writing_params_path vanilla.pretrain.params

# Finetune the model
python src/run.py finetune vanilla wiki.txt \
       --reading_params_path vanilla.pretrain.params \
       --writing_params_path vanilla.finetune.params \
       --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set; write to disk
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.finetune.params \
       --eval_corpus_path birth_dev.tsv \
       --outputs_path vanilla.pretrain.dev.predictions

# Evaluate on the test set; write to disk
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.finetune.params \
       --eval_corpus_path birth_test_inputs.tsv \
       --outputs_path vanilla.pretrain.test.predictions
```

Report the accuracy on the dev set (printed by the third command above). We expect the dev accuracy will be at least 10%, and will expect a similar accuracy on the held out test set.

Answer:

- Accuracy on dev set: 25.4% (127.0 correct out of 500.0)

(g) (10 points) **Research! Write and try out a more efficient variant of Attention (Budget 2 hours for pretraining!)**

We'll now go to changing the Transformer architecture itself – specifically the first and last transformer blocks. The transformer model uses a self-attention scoring function based on dot products, this involves a rather intensive computation that's quadratic in the sequence length. This is because the dot product between ℓ^2 pairs of word vectors is computed in each computation, where ℓ is the sequence length. If we can reduce the length of the sequence passed on the self-attention module, we should observe significant reduction in compute. For example, if we develop a technique that can reduce the sequence length to half, we can save around 75% of the compute time!

PerceiverAR [1] proposes a solution to make the model more efficient by reducing the sequence length of the input to self-attention for the intermediate layers. In the first layer, the input sequence is projected onto a lower-dimensional basis. Subsequently, all self-attention layers operate in this smaller subspace. The last layer projects the output back to the original input sequence length. In this assignment, we propose a simpler version of the PerceiverAR transformer model.

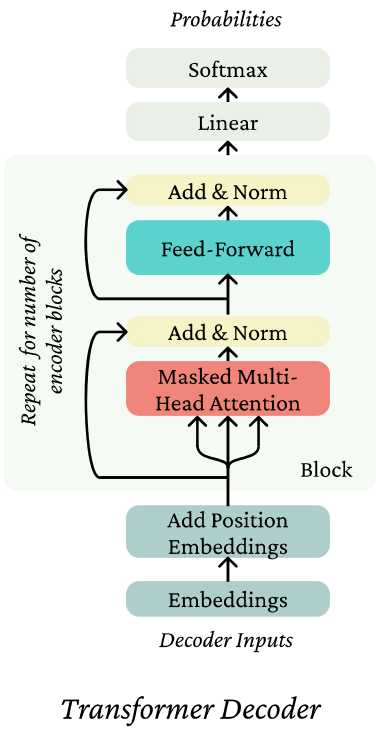

Figure 2: Illustration of the transformer block.

The provided `CausalSelfAttention` layer implements the following attention for each head of the multi-headed attention: Let $X \in \mathbb{R}^{\ell \times d}$ (where ℓ is the block size and d is the total dimensionality, d/h is the dimensionality per head.).[4]

Let $Q_i, K_i, V_i \in \mathbb{R}^{d \times d/h}$. Then the output of the self-attention head is

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{3}$$

[4]Note that these dimensionalities do not include the minibatch dimension.

where $Y_i \in \mathbb{R}^{\ell \times d/h}$. Then the output of the self-attention is a linear transformation of the concatenation of the heads:

$$Y = [Y_1; \ldots; Y_h]A \tag{4}$$

where $A \in \mathbb{R}^{d \times d}$ and $[Y_1; \ldots; Y_h] \in \mathbb{R}^{\ell \times d}$. The code also includes dropout layers which we haven't written here. We suggest looking at the provided code and noting how this equation is implemented in PyTorch.

Our model uses this self-attention layer in the transformer block as shown in Figure 2. As discussed in the lecture, the transformer block contains residual connections and layer normalization layers. If we compare this diagram with the `Block` code provided in `model.py`, we notice that the implementation does not perform layer normalization on the output of the MLP (Feed-Forward), but on the input of the `Block`. This can be considered equivalent since we have a series of transformer blocks on top of each other.

In the Perceiver model architecture, we replace the first transformer `Block` in the model with the `DownProjectBlock`. This block reduces the length of the sequence from ℓ to m. This is followed by a series of regular transformer blocks, which would now perform self-attention on the reduced sequence length of m. We replace the last block of the model with the `UpProjectBlock`, which takes in the m length output of the previous block, and projects it back to the original sequence length of ℓ.

You need to implement the `DownProjectBlock` in `model.py` that reduces the dimensionality of the sequence in the first block. To do this, perform cross-attention on the input sequence with a learnable basis $C \in \mathbb{R}^{m \times d}$ as the query, where $m < \ell$. Consequently, Equation 3 becomes:

$$Y_i^{(1)} = \text{softmax}\left(\frac{(CQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{5}$$

resulting in $Y_i^{(1)} \in \mathbb{R}^{m \times d}$, with $^{(1)}$ denoting that the output corresponds to the first layer. With this dimensionality reduction, the subsequent `CausalSelfAttention` layers operate on inputs $\in \mathbb{R}^{m \times d}$ instead of $\mathbb{R}^{l \times d}$. We refer to m as the `bottleneck_dim` in code. Note that for implementing Equation 5, we need to perform cross attention between the learnable basis C and the input sequence. This has been provided to you as the `CausalCrossAttention` layer. We recommend reading through `attention.py` to understand how to use the cross-attention layer, and map which arguments correspond to the key, value and query inputs. Initialize the basis vector matrix C using Xavier Uniform initialization.

To get back to the original dimensions, the last block in the model is replaced with the `UpProjectBlock`. This block will bring back the output sequence length to be the same as input sequence length by performing cross-attention on the previous layer's output Y^{L-1} with the original input vector X as the query:

$$Y_i^{(L)} = \text{softmax}\left(\frac{(XQ_i)(Y^{(L-1)}K_i)^\top}{\sqrt{d/h}}\right)(Y^{(L-1)}V_i) \tag{6}$$

where L is the total number of layers. This results in the final output vector having the same dimension as expected in the original `CausalSelfAttention` mechanism. Implement this functionality in the `UpProjectBlock` in `model.py`.

We provide the code to assemble the model using your implemented `DownProjectBlock` and `UpProjectBlock`. The model uses these blocks when the `variant` parameter is specified as `perceiver`.

Below are bash commands that your code should support in order to pretrain the model, finetune it, and make predictions on the dev and test sets. Note that the pretraining process will take approximately 2 hours.

```
# Pretrain the model
python src/run.py pretrain perceiver wiki.txt --bottleneck_dim 64 \
       --pretrain_lr 6e-4 --writing_params_path perceiver.pretrain.params

# Finetune the model
python src/run.py finetune perceiver wiki.txt --bottleneck_dim 64 \
       --reading_params_path perceiver.pretrain.params \
       --writing_params_path perceiver.finetune.params \
       --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set; write to disk
python src/run.py evaluate perceiver wiki.txt --bottleneck_dim 64 \
       --reading_params_path perceiver.finetune.params \
       --eval_corpus_path birth_dev.tsv \
       --outputs_path perceiver.pretrain.dev.predictions

# Evaluate on the test set; write to disk
python src/run.py evaluate perceiver wiki.txt --bottleneck_dim 64 \
       --reading_params_path perceiver.finetune.params \
       --eval_corpus_path birth_test_inputs.tsv \
       --outputs_path perceiver.pretrain.test.predictions
```

Report the accuracy of your perceiver attention model on birthplace prediction on `birth_dev.tsv` after pretraining and fine-tuning.

Save the predictions of the model on `birth_test_inputs.tsv` to `perceiver.pretrain.test.predictions`.

For this section, you'll submit: `perceiver.finetune.params`, `perceiver.pretrain.dev.predictions`, and `perceiver.pretrain.test.predictions`. Your model should get at least 6% accuracy on the dev set.

i. (8 points) We'll score your model as to whether it gets at least 5% accuracy on the test set, which has answers held out.

ii. (2 points) Provide an expression for the time complexity of the Perceiver model and the vanilla model, in terms of number of layers (L), input sequence length (ℓ) and basis bottleneck dimension (m).

Answer:

i. Accuracy of model on dev set: 6.4% (32.0 correct out of 500.0)

ii. Time complexity of:

- Perceiver model: $O(L\ell m)$ – the `DownProjectBlock` and `UpProjectBlock` both perform an attention computation of complexity ℓm and the intervening layers have complexity m^2 (quadratic in m owing to cross-attention), leading to a total of $\ell m + (L-2) \times m^2 + \ell m$ computations, which can be generalized as $O(L\ell m)$ since $\ell > m$.
- Vanilla model: $O(L\ell^2)$ – each of the L layers performs an attention computation involving ℓ^2 pairs of words.

3. Considerations in pretrained knowledge (5 points)

Please type the answers to these written questions (to make TA lives easier).

(a) (1 point) Succinctly explain why the pretrained (vanilla) model was able to achieve an accuracy of above 10%, whereas the non-pretrained model was not.

Answer: Since the vanilla model was pretrained on the span corruption task, at evaluation time for the birthplace retrieval task, it is better able to process malformed inputs and/or inputs not seen during finetuning, leading to the higher accuracy as compared to the non-pretrained model.

(b) (2 points) 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 **distinct** reasons why this model behavior (i.e. unable to tell whether it's retrieved or made up) may cause concern for such applications, and an example for each reason.

Answer: Reasons why this model behavior may cause concern for user-facing applications include:

1. User belief in and use of factually incorrect information to potential human detriment – i.e., the model could generate factually incorrect information that sounds plausible but ultimately harms the user.
   - *Example:* A user could query a model pretrained on medical knowledge for an answer to "what should I do if my friend is having an allergic reaction?" and the model could make up real-sounding medical advice that, when applied, actually aggravates the friend's condition.
2. General lack of trust and adoption of the model/application – i.e., a model could provide an outsized benefit to its users collectively, but if the users do not believe in the reliability of the model, the model may go unused and this benefit unrealized.
   - *Example:* Suppose a model pretrained on personal finance books could provide wealth-increasing financial advice for 99.99% of user queries and recommends wealth-decreasing advice for the other 0.01%. Users may fear that their query will be one which elicits detrimental advice and may collectively eschew model adoption, despite that, if they all used it, the net benefit would be positive and significant for their collective financial health.

(c) (2 points) 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. (You do not need to submit the same answer for 3c as for 3b.)

Answer: The model may be employing a strategy where, for these unseen name queries, it identifies a name it has seen during pretraining and/or finetuning which is most similar to the unseen name (for some definition of similarity) and simply outputs the birthplace for the person with the most similar name.

This strategy should cause concern for the use of such applications because it could lead to privacy infringements – e.g., a celebrity may choose to keep their birthplace off of the Internet, but the application may be able to predict their birthplace based off the birthplaces of individuals with similar-sounding names. Bad actors could use this information to infringe upon the privacy of the celebrity and otherwise harm them if the prediction is correct – by harassing family members, vandalizing their childhood home, etc.

Submission Instructions

You will submit this assignment on GradeScope as two submissions – one for **Assignment 5 [coding]** and another for **Assignment 5 [written]**:

1. Verify that the following files exist at these specified paths within your assignment directory:
   - The no-pretraining model and predictions: `vanilla.model.params`, `vanilla.nopretrain.dev.predictions`, `vanilla.nopretrain.test.predictions`
   - The pretrain-finetune model and predictions: `vanilla.finetune.params`, `vanilla.pretrain.dev.predictions`, `vanilla.pretrain.test.predictions`
   - The Perceiver model and predictions: `perceiver.finetune.params`, `perceiver.pretrain.dev.predictions`, `perceiver.pretrain.test.predictions`
2. Run the `collect_submission.sh` script to produce your `assignment5.zip` file.
3. Upload your `assignment5.zip` file to GradeScope to **Assignment 5 [coding]**.
4. Check that the public autograder tests passed correctly.
5. Upload your written solutions, for questions 1, parts of 2, and 3, to GradeScope to **Assignment 5 [written]**. Tag it properly!

References

[1] Hawthorne, C., Jaegle, A., Cangea, C., Borgeaud, S., Nash, C., Malinowski, M., Dieleman, S., Vinyals, O., Botvinick, M. M., Simon, I., Sheahan, H., Zeghidour, N., Alayrac, J., Carreira, J., and Engel, J. H. General-purpose, long-context autoregressive modeling with perceiver AR. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA* (2022), vol. 162 of *Proceedings of Machine Learning Research*, pp. 8535–8558.

[2] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding with unsupervised learning. *Technical report, OpenAI* (2018).

[3] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research 21*, 140 (2020), 1–67.