XCS224N Problem Set 5 Self-attention, Transformers, Pretraining

---

Due Sunday, November 19 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 2 hours to perform on Azure, and you'll have to do it twice.

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.

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

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

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 c]` 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 (d), which contains commands you can run to check your implementation. No written answer is required for this part.

(d) **[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:

```
# 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.

(e) **[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 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.

(f) **[20 points (Coding)] Pretrain, finetune, and make predictions. Budget 2 hours for training.**
Now fill in the *pretrain* portion of `src/submission/helper.py`, which will pretrain a model on the span

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

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 two hours), finetune it on `NameDataset` and evaluate it. Specifically, you should be able to run the following four commands:

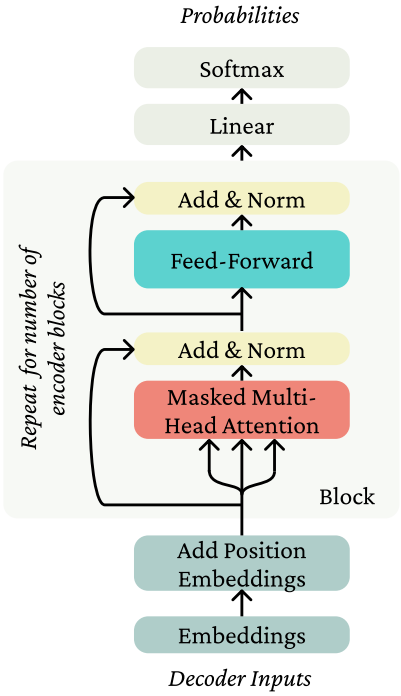
```
# Pretrain the model
./run.sh vanilla_pretrain

# 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) **[14 points (Coding)] 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 [4] 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.



Transformer Decoder

Figure 1: 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.).[5]

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

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

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

[4] https://arxiv.org/abs/2202.07765

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

the heads:

$$Y = [Y_1; \ldots; Y_h]A \tag{2}$$

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 1. 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 1 becomes:

$$Y_i^{(1)} = \text{softmax}\left(\frac{(CQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{3}$$

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 3, 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:

$$Y_i^{(L)} = \text{softmax}\left(\frac{(XQ_i)(Y^{(L-1)}K_i)^\top}{\sqrt{d/h}}\right)(Y^{(L-1)}V_i) \tag{4}$$

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 `src/submission/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`.

In the rest of the code in the `src/submission/helper.py`, modify your model to support using either `CausalSelfAttention` or `CausalCrossAttention`. Add the ability to switch between these attention variants depending on whether "vanilla" (for causal self-attention) or "perceiver" (for the perceiver variant) is selected in the command line arguments (see the section marked `[part g]` in `src/submission/helper.py`).

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.

Your model should get at least 6% accuracy on the dev set.

```
# Pretrain the model
./run.sh perceiver_pretrain

# Finetune the model
```

```
./run.sh perceiver_finetune_with_pretrain

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

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

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/perceiver.pretrain.params`
16. `src/submission/perceiver.finetune.params`
17. `src/submission/perceiver.pretrain.dev.predictions`
18. `src/submission/perceiver.pretrain.test.predictions`