AI51701/CSE71001 Assignment 3

Transformers and Pretraining

Due: December 3 2022, 11:59:00pm KST

Submission Instructions: You shall submit this assignment on BlackBoard as two submission files – your code as `assignment3coding.zip` and write up for `assignment3.pdf`; Run the `collect_submission.sh` script to produce your `assignment3.zip` file. As in the previous assignment, we use Jupyter Notebook and Google Colab for using GPU. It is your responsibility to make sure your code is runnable at Google Colab. If your code is not runnable, you will get no point. Also do not write down your name/student ID in your submission.
This assignment involves a pretraining step that takes approximately 2 hours to perform on Golab GPU, and you'll have to do it twice. Colab set-up notebook has been provided similar to Assignment 2. 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.

Collaboration Policy: You are welcome to discuss assignments with others in the course, but solutions and code must be written individually.

1. Pretrained Transformer models and knowledge access (70 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 Colab. You can use the same conda environment from previous assignments for local development, and the same process for training on a GPU. 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.**

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) (10 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.

(e) (20 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) (20 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.

(g) (20 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.

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.).[1]

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 the heads:

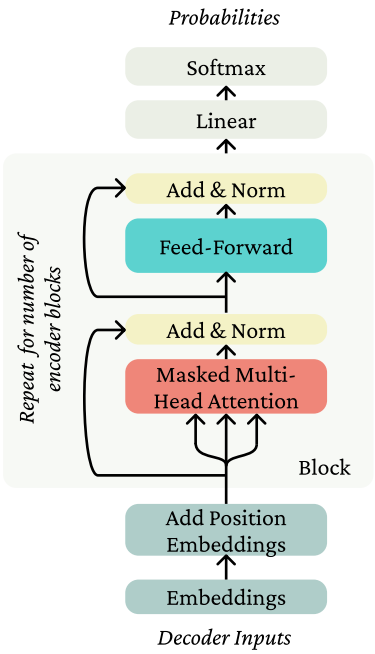
$$Y = [Y_1; \ldots; Y_h]A \tag{2}$$

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

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



Transformer Decoder

Figure 1: Illustration of the transformer block.

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 `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-3 --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. (16 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. (4 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).

2. Considerations in pretrained knowledge (10 points)

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

(a) (2 points) Succinctly explain why the pretrained (vanilla) model was able to achieve an accuracy of above 10%, whereas the non-pretrained model was not.

(b) (4 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.

(c) (4 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.)

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.