Goals. The main purpose of this assignment is to familiarize you with the implementation details underlying Transformers, and to show you how language models are constructed from them. Transformers are the foundation of almost all state-of-the-art language models today, including the proprietary ones like ChatGPT that you were probably already familiar with before taking this class. This assignment will also expose you to fine-tuning, and give you some idea of why it works so well in practice.

Resources

Dependencies. We will be using Google Colab for this assignment. GPUs will be basically required here, as training language models (even on the relatively small datasets we'll be using) would take way too long otherwise. I'll relink some helpful resources to get you started with Colab in the footnotes.[1][2]

We'll be using the `torch, torchvision, transformers, datasets, numpy` packages. If your Transformers implementation uses packages not in this list, it may cause the autograder to crash.

Follow the instructions at the top of the notebook to switch to a GPU runtime. You might run into GPU usage limits while working on this assignment; if this happens, feel free to massively subsample the data so that you can debug things on CPU, and then return to GPU once you regain access.

Data. For this assignment, we'll be using two datasets. In Task 1, you will use The Pile, a large-scale text dataset consisting of documents from many sources—but largely the internet. Download the .zip file linked here,[3] and upload it to your Colab runtime. You will need to re-upload it every time you restart your runtime.

In Part 2, you will use a Shakespeare corpus from HuggingFace. HuggingFace is a large and commonly used repository of datasets and Transformer-based language models.[4] Every example here consists of a line of dialogue from one of Shakespeare's plays.

Both datasets have been split into a training set, development set, and test set. The Pile has been subsampled such that you should have 10,000 train, 999 dev, and 200 test examples. We'll use the full Shakespeare corpus.

Code structure. For this assignment, we will be working in Google Colab. Everything is provided at the linked notebook here.[5]

Make a personal copy of this Google Colab notebook, and work from your copy. At a high level, here is what you will be implementing:

- You will implement the `Transformer` and `MultiHeadTransformer` classes. After doing so, you will copy and paste the contents of this cell into a Python script called `transformer.py`, which you will upload to Gradscope for autograding.
- You will implement temperature sampling so that you can generate textual outputs from your models.

Assignment

Download the Pile data, and make a personal copy of the Google Colab. Upload the `pile.zip` file to the Files tab.

[1] Google Colab guide
[2] Google Colab cheat sheet
[3] `https://drive.google.com/file/d/1Zuazv_I1bRpQWANsT5vBjU65qn1kKcAt/view?usp=sharing`
[4] `https://huggingface.co`
[5] `https://colab.research.google.com/drive/1Xxw0Q94PyMnIUHlmRnRMhg9Okqjm2XI1?usp=sharing`

Task 1: Transformers From Scratch (41 points)

Your first task is to implement a Transformer. You'll start by implementing a simplified version of the Transformer with just single-head attention. We'll slowly add complexity.

Recall that each layer of a Transformer decoder first projects the representations at all positions X into three separate matrices—namely, the **query**, **key**, and **value** matrices:

$$Q = XW_Q \tag{1}$$
$$K = XW_K \tag{2}$$
$$V = XW_V \tag{3}$$

Then, the query and key matrices are multiplied to yield **attention scores**. Note that the key matrix is transposed. The attention scores are normalized by the square root of the dimensionality of the key matrix $\sqrt{d_k}$, and then softmaxed to yield **attention weights**. The attention weights are multiplied by the value matrix. Finally, the output of this operation is projected back into the dimensionality of the hidden vectors using an output projection W_O:

$$A = \text{softmax}\Big(\frac{QK^\top}{\sqrt{d_k}}\Big)V \tag{4}$$
$$Z = AW_O \tag{5}$$

After this, there is a multi-layer perceptron (MLP):

$$H = \text{ReLU}(ZW_{\text{up}} + b_1)W_{\text{down}} + b_2, \tag{6}$$

where W_{up} is an up-projection into a higher-dimensional vector space, and W_{down} is a down-projection back into the model's hidden dimensionality. As-is, this model may have issues with unstable learning. We will address this by adding residual connections to the attention module and MLP:

$$Z' = Z + X \tag{7}$$
$$H' = H + Z \tag{8}$$

These connections add a direct path in the computation graph from the input of the attention module to its output, such that even if gradients become small in the attention, the gradients can just go around the attention module. Same for the MLP.

Q1 (20 points; AUTOGRADED). Implement single-headed attention as specified above. Do this in the provided Google Colab. You must implement a **causal mask**, meaning that when the representation of a token is computed, the self-attention will only be allowed to attend to tokens before that token's position. Implement this as a mask on the attention score matrix. There are some code hints to help you with this.

Once you've implemented this, use the following cells to build a bare-bones decoder-only language model based on your `Transformer`. By default, this will initialize a two-layer model with a relatively small hidden dimensionality. Use this command to train, evaluate, and save your language model: Report the final dev perplexity of your trained language model in your written report.

Q2 (5 points). Now, you will add some fancier features to improve your Transformer language model. First, implement layer norm. This standardizes the range of each element in the output of the MLP by

computing the mean μ and variance σ of the elements of $\mathbf{h}$ (a single token's representation vector), and normalizing it as follows:

$$\hat{\mathbf{h}} = \gamma \frac{\mathbf{h} - \mu}{\sigma} + \beta, \tag{9}$$

where γ and β are learnable parameters. Apply a layer norm to the output of the attention A, as well as to the output of the MLP H. Retrain your language model, and report your model's dev perplexity in your written report.

Q3 (10 points; AUTOGRADED). Extend your Transformer to support multi-headed attention. This will require you to reshape your key, query, and value matrices. Complete the provided `MultiHeadTransformer` class, which extends the `Transformer` class you've been writing. We recommend copying your forward function into this new class and working from there going forward. By default, this will initialize a 2-layer model with 4 attention heads.

You must concatenate each head's output $A^{(i)}$ into a single vector, and then multiply this large vector by the output projection W_O to produce a hidden representation H of the same dimensionality as the embeddings. You might think that the output projection would need to be larger to handle the additional attention heads, but actually, each attention head will output a representation of size $d_{\mathbf{h}}/n$ for each token, where $d_{\mathbf{h}}$ is the hidden dimensionality (i.e., the width of a token's vector representation before getting processed by self-attention) and n is the number of attention heads. See `self.head_dim` in the class's initialization function.

Once you've implemented this, train, evaluate, and save your multi-head attention Transformer language model using the following command.

Report this model's final dev perplexity in your write-up. How does the perplexity of your model with multi-head attention compare to the perplexity of your model with single-head attention?

Once you've done all of this, copy the entire contents of the cell containing your `Transformer` and `MultiHeadTransformer` classes to a python script called `transformers.py`.

Q4 (6 points). Add support for generating multi-token outputs from your language model given an input prompt. Modify the `generate` function to do so. We'll implement **temperature sampling**, which involves sampling a token from a renormalized distribution as follows:

$$y \sim \text{softmax}(\mathbf{e}/\tau) \tag{10}$$

Here, $\mathbf{e}$ is the logit vector, and τ is the temperature hyperparameter.

This generation method doesn't always yield the most coherent results, but they are certainly more creative and diverse than in greedy decoding.

Report the outputs of your language model in response to each of these prefixes (generate no more than 30 tokens each, please):

1. The quick brown fox
2. In the beginning
3. What is 5 + 5?
4. The meaning of life is

In 1–3 sentences, what trends do you notice in the model's outputs? For example, in prompt (3), did it try to answer the question, or did it do something else?

Task 2: Transfer Learning via Fine-tuning (13 points)

In this section, you will train and fine-tune models on a new distribution of text. Specifically, we'll be using a small corpus of Shakespearean dialogue available online. You'll compare each of these models in terms of their perplexities on Shakespeare data:

- A multi-head Transformer model trained from scratch on Shakespeare
- Your model from Task 1, with no adaptation on the Shakespeare corpus
- Your model from Task 1 fine-tuned on the Shakespeare corpus
- GPT-2, fine-tuned on the Shakespeare corpus

"Fine-tuning" in this case really means doing a second round of language modeling after having already done language modeling on some other dataset first.

The interesting thing about the approach we'll be implementing is that you won't need to change much at all about your language modeling code! Instead, we'll just load some new data and run much of the same code. That said, you will need to modify and play around with the hyperparameters; relative to pre-training, we usually reduce the learning rate and number of epochs when fine-tuning.

Q5 (3 points). All code needed to answer this question has either been provided for you or has already been written in previous questions. Take your multi-head Transformer model from Task 1 and evaluate its perplexity on the dev split of the Shakespeare corpus.

Then, instantiate a `Transformer`, and train an LM from scratch on the Shakespeare data. Report this model's final dev perplexity. Note that you should not need to modify much compared to your previous training cell—maybe just the hyperparameters at most.

Finally, instead of training from scratch, load the language model you trained in Q5 and perform **fine-tuning** on the Shakespeare data.

For each of these models, report their perplexities in your written report. In 1–2 sentences, compare these perplexities, and describe whether the trends across models match what you would expect.

Q6 (2 points). Take the fine-tuned model from Q5 and re-evaluate its perplexity on the dev split of the Pile (the data you used for Task 1).

Compare this perplexity to the perplexity you obtained in Q3. In 1–3 sentences, what do you notice about the perplexity of the fine-tuned model on the data it was originally trained on, and why do you think it changed?

Q7. Now, you'll download a small language model that was trained on a *lot* more data than the one you trained. Specifically, we'll be using the HuggingFace library to load a model and fine-tune it on data from a different distribution. We'll use "GPT-2", a well-known decoder-only model from 2019.

The code you need has been provided for you. The provided code loads the GPT-2 tokenizer and language model, and also loads a dataset in a format that this model expects. We will use the HuggingFace `Trainer` to fine-tune this model.

a (3 points). Run fine-tuning on the Shakespeare dataset. You may need to do some hyperparameter fine-tuning; I recommend only modifying the initial learning rate and number of epochs, and leaving everything else untouched. Afterwards, evaluate its perplexity on the Shakespeare corpus; report the best score you were able to obtain after some hyperparameter tuning in your write-up. Also report the best hyperparameters.

b (5 points). You may find it difficult to get a perplexity lower than the models from Q6. However, try generating some text with this model using the provided 8 prompts (a mix of the prompts we used before, and some new more Shakespearean-sounding prompts). Compare these generations with generations from your other 3 models from Q6. If everything worked, GPT-2's outputs should be significantly better. Provide the generated texts from the fine-tuned GPT-2, as well as generated texts from any one of the models from Q6 (16 total outputs) in your written report. Make it clear which outputs were from which models.

In 1–2 sentences, why do you think GPT-2's outputs are better than the other models'? In 1–2 additional sentences, what do you think explains the discrepancy between GPT-2's worse perplexity but better textual outputs?

Task 3: Conceptual Questions

There is no coding in this part of the assignment.

Q8 (3 points). In 1–3 sentences, why do Transformers need positional embeddings while RNNs don't?

Q9 (2 points). You're training a Transformer on a small training set, and it's strongly overfitting on the training data. Your collaborator suggests removing the residual connections to simplify the model. Do you think this would be likely to improve performance? In 1–2 sentences, why or why not?

Q10. In this assignment, you implemented a decoder-only Transformer; this is similar to the architecture used by proprietary models like ChatGPT.

a. (4 points). Encoder-only models differ from decoder-only models in terms of (i) their attention mask and (ii) their training objective. In 1–2 sentences each, describe these two differences.

b. (2 points). Give an example of a task that a decoder-only model can do that an encoder-only model cannot do.

Extra Credit

Nucleus Sampling (8 points)

Optionally, for a bit of extra credit, you can implement nucleus sampling, and compare the outputs from this method to your temperature sampling outputs.

E1. (8 points) Implement nucleus sampling. You can do this in the `nucleus_sampling_generate` method, in the cell following the `generate` method's. If you did this, tell us in your written report and we will check your code.

Then, take the model and prompts from Q4 and use nucleus sampling with $p = 0.9$ to generate some textual outputs. Compare these outputs to those you obtained in Q4. In 2–3 sentences, what differences do you notice between the outputs?

You must report your generations with nucleus sampling to receive *any* credit on this question. That is, it is not sufficient to just implement nucleus sampling and tell us you did it; you must provide textual outputs and compare them to your outputs from Q4.

Submission

Upload the following to Gradescope:

1. Upload your written responses as a PDF to **HW2: Transformers [Written]** on Gradescope.
2. Upload your `transformers.py` script and your full `.ipynb` notebook to **HW2: Transformers [Code]**.