Due Date: Monday, 15th November, 11pm ET

Instructions

- *This assignment is involved – please start well ahead of time.*
- *For all questions, show your work!*
- *Submit your report (PDF) and your code electronically via the course Gradescope page. Your report must contain answers to Problem 2 (question 5 only), and to Problem 3 (all questions).*
- *For open-ended experiments (i.e., experiments that do not have associated test-cases), you do not need to submit code – a report will suffice.*
- *TAs for this assignment are **Leo Feng** and **Milad Aghajohari**.*

Summary:

In this assignment, you will perform **sequential language modeling** on the Wikitext-2 dataset. The dataset contains sequences of *tokens* (i.e., words or subwords) extracted from contiguous sentences. Each unique token is denoted by an integer in the range $[1, N]$, where N is the number of all unique tokens (N is also often referred to as the *vocabulary size*). For this assignment, we use a vocabulary of size $N = 40,478$, as originally defined for OpenAI's GPT. Sequential language models do *next-word prediction*: they predict tokens in a sequence one at a time, with each prediction based on all the previous elements of the sequence. A trained sequential language model can then be used to generate new sequences of text, by making each prediction conditioned on the past *predictions*.

In this assignment, you will implement a sequential language model (an **LSTM**), and a masked language model (**Transformer** inspired by OpenAI GPT). In problem 1, you will use built-in PyTorch modules to implement an LSTM. In problem 2, you will implement various building blocks of a transformer, including **LayerNorm** (layer normalization) and the **Attention** mechanism.

The Wikitext-2 dataset comprises 2 million words extracted from the set of verified "Good" and "Featured" articles on Wikipedia. See this blog post for details about the Wikitext dataset and sample data. The dataset you get with the assignment has already been preprocessed using OpenAI's GPT vocabulary, and each file is a compressed numpy array containing two arrays: `tokens` containing a flattened list of (integer) tokens, and `sizes` containing the size of each document.

You are provided a PyTorch dataset class (`torch.utils.data.Dataset`) named `Wikitext2` in the `utils` folder. This class loads the Wikitext-2 dataset and generates fixed-length sequences from it. Throughout this assignment, **all sequences will have length 256**, and we will use zero-padding to pad shorter sequences. Each sample from the dataset is a dictionary containing 3 keys: `source` (the input sequence), `target` (the target sequence, which is just the input sequence shifted one position to the left)), and `mask` (a binary vector of the same shape as the input indicating whether

the token is valid (1), or is just a zero-padding to get the sequence length to 256). For example (the sequences have been shortened for presentation):

`source`	304	4731	8406	614	0	0	0	...	`torch.LongTensor`
`target`	4731	8406	614	304	0	0	0	...	`torch.LongTensor`
`mask`	1	1	1	1	0	0	0	...	`torch.FloatTensor`

In practice though, you will work with mini-batches of data, each with batchsize `B` elements. You can wrap this dataset object into a `torch.utils.data.DataLoader`, which will return a dictionary with keys `source`, `target`, and `mask`, each of shape `(B, 256)`.

Tests : For you to quickly verify that all the functions you implemented have valid input-output signatures (e.g. tensor types and shapes), we have provided public tests that will check if your output tensors have the correct shape, given random input data. Check the comments (docstrings) in each function to see what input/output shapes are expected. We recommend you to run these tests locally while completing the functions using the following command (to run inside your assignment folder)

```
python -m unittest
```

For students using Google Colab to complete their assignments, a cell with this command is available in the `main.ipynb` notebook. Note that these tests are not testing if the values returned by the functions are valid (only if the tensor shapes are correct), and will not be graded. You will be graded on a separate set of tests in Gradescope.

If the tests on Gradescope fail, as a rule of thumb x corresponds to the value in your assignment (e.g. the value returned by your function), and y is the expected value.

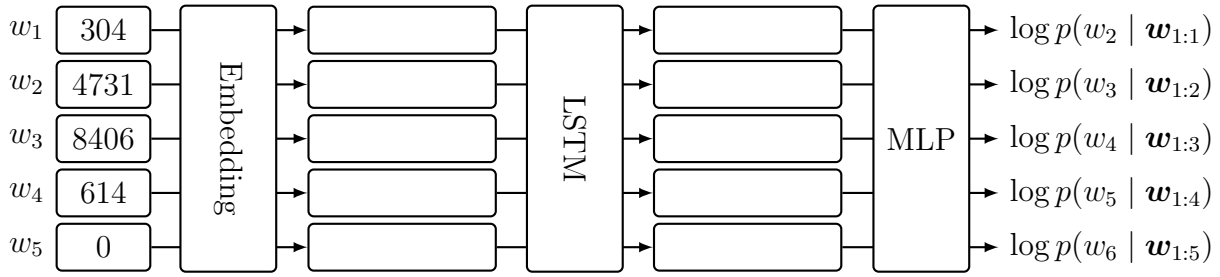
Emebeddings & parameter sharing In both the LSTM and the Transformer, the embedding layer is among the layers that contain the most parameters. Indeed, it consists of a matrix of size `(vocabulary_size, embedding_size)` (note that here `vocabulary_size` is equal to $N + 1$, to account for zero-padding), which represents about 31M parameters. Similarly, in both models, we also have a classifier of about the same size (i.e. the output has size `vocabulary_size` as well, to represent the probability of the next word). In order to speed up training, and for simplification in this assignment, we make two assumptions for both models:

- Following the architecture from OpenAI's GPT, the weights of the classifier layer and the embedding layer will be shared.
- You will use pretrained embeddings from OpenAI's GPT (they are provided in the `run_exp.py` script, described in Problem 3), which will remain frozen over the course of training.

Coding instructions You will be required to use PyTorch to complete all questions. Moreover, this assignment **requires running the models on GPU** (otherwise it will take an incredibly long time); if you don't have access to your own resources (e.g. your own machine, a cluster), please use Google Colab (the notebook `main.ipynb` is here to help you). For some questions, you will be asked to not use certain functions in PyTorch and implement these yourself using primitive functions from `torch`; in that case, the functions in question are explicitly disabled in the tests on Gradescope.

Problem 1

Implementing an LSTM (9pts) In this problem, you will be using PyTorch's built-in modules in order to implement an LSTM. The architecture you will be asked to implement is the following:



In the file `lstm_solution.py`, you are given an `LSTM` class containing all the blocks necessary to create this model. In particular, `self.embedding` is a `nn.Embedding` module that converts sequences of token indices into embeddings, `self.lstm` is a `nn.LSTM` module that runs an LSTM over a sequence of vectors, and `self.classifier` is a 2-layer MLP responsible for classification.

1. Using the different modules described above, complete the `forward()` function. This function must return the log-probabilities (not the logits) of the next words in the sequence, as well as the final hidden state of the LSTM.

2. Complete the `loss()` function, that returns the mean negative log-likelihood of the entire sequences in the minibatch (and also averaged over the mini-batch dimension). More precisely, for a single sequence in the mini-batch

$$\mathcal{L}(\boldsymbol{\theta}; \boldsymbol{w}_{1:T+1}) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{i=0}^{N} \log p(\widetilde{w}_{t+1} = i \mid \boldsymbol{w}_{1:t}; \boldsymbol{\theta}) \mathbf{1}(i = w_{t+1}),$$

where $\widetilde{w}$ are the predictions made by the model, and $\mathbf{1}(i = w_{t+1})$ is the indicator function which equals 1 if $i = w_{t+1}$, and 0 otherwise. Note that here T might be smaller than 256 (called `sequence_length` in the code), because the sequence might be zero-padded; you may use `mask` for this. The `loss` function directly takes the log-probabilities as input (e.g. returned by the `forward` function).

Problem 2

Implementing a GPT (Generative-Pretrained Transformer) (27pts) While typical RNNs "remember" past information by taking their previous hidden state as input at each step, recent years have seen a profusion of methodologies for making use of past information in different ways. The transformer[1] is one such fairly new architecture which uses several self-attention networks ("heads") in parallel, among other architectural specifics. Implementing a transformer is a fairly involved process – so we provide most of the boilerplate code and your task is only to implement the multi-head scaled dot-product attention mechanism, as well as the layernorm operation.

Implementing Layer Normalization (5pts): You will first implement the layer normalization (LayerNorm) technique that we have seen in class. For this assignment, **you are not allowed** to use the PyTorch `nn.LayerNorm` module (nor any function calling `torch.layer_norm`).

As defined in the layer normalization paper, the layernorm operation over a minibatch of inputs x is defined as

$$\text{layernorm}(x) = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \texttt{weight} + \texttt{bias}$$

where $\mathbb{E}[x]$ denotes the expectation over x, $\text{Var}[x]$ denotes the variance of x, both of which are only taken over the last dimension of the tensor x here. `weight` and `bias` are learnable affine parameters.

1. In the file `gpt1_solution_template.py`, implement the `forward()` function of the `LayerNorm` class. Pay extra attention to the lecture slides on the exact details of how $\mathbb{E}[x]$ and $\text{Var}[x]$ are computed. In particular, PyTorch's function `torch.var` uses an unbiased estimate of the variance by default, defined as the formula on the left-hand side

$$\overline{\text{Var}}(X)_{\text{unbiased}} = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2 \qquad \overline{\text{Var}}(X)_{\text{biased}} = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2$$

   whereas LayerNorm uses the biased estimate on the right-hand size (where $\overline{X}$ here is the mean estimate). Please refer to the docstrings of this function for more information on input/output signatures.

Implementing the attention mechanism (17pts): You will now implement the core module of the transformer architecture – the multi-head attention mechanism. Assuming there are m attention heads, the attention vector for the head at index i is given by:

$$[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m] = \boldsymbol{Q}\boldsymbol{W}_Q + \boldsymbol{b}_Q \qquad [\boldsymbol{k}_1, \ldots, \boldsymbol{k}_m] = \boldsymbol{K}\boldsymbol{W}_K + \boldsymbol{b}_K \qquad [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m] = \boldsymbol{V}\boldsymbol{W}_V + \boldsymbol{b}_V$$

$$\boldsymbol{A}_i = \text{softmax}\left(\frac{\boldsymbol{q}_i \boldsymbol{k}_i^\top}{\sqrt{d}}\right)$$

$$\boldsymbol{h}_i = \boldsymbol{A}_i \boldsymbol{v}_i$$

$$A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m)\boldsymbol{W}_O + \boldsymbol{b}_O$$

[1] See https://arxiv.org/abs/1706.03762 for more details.

Here $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are queries, keys, and values respectively, where all the heads have been concatenated into a single vector (e.g. here $\boldsymbol{K} \in \mathbb{R}^{T \times md}$, where d is the dimension of a single key vector, and T the length of the sequence). $\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V$ are the corresponding projection matrices (with biases $\boldsymbol{b}$), and $\boldsymbol{W}_O$ is the output projection (with bias $\boldsymbol{b}_O$). $\boldsymbol{Q}, \boldsymbol{K}$, and $\boldsymbol{V}$ are determined by the output of the previous layer in the main network. $\boldsymbol{A}_i$ are the attention values, which specify which elements of the input sequence each attention head attends to. We recommend you to check this tutorial for implementation details of GPT. In this question, **you are not allowed** to use the module `nn.MultiheadAttention` (or any function calling `torch.nn.functional.multi_head_attention_forward`). Please refer to the docstrings of each function for a precise description of what each function is expected to do, and the expected input/output tensors and their shapes.

2. The equations above require many vector manipulations in order to split and combine head vectors together. For example, the concatenated queries $\boldsymbol{Q}$ are split into m vectors $[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m]$ (one for each head) after an affine projection by $\boldsymbol{W}_Q$, and the h_i's are then concatenated back for the affine projection with $\boldsymbol{W}_O$. In the class `MultiHeadedAttention`, implement the utility functions `split_heads()` and `merge_heads()` to do both of these operations, as well as a transposition for convenience later. For example, for the 1st sequence in the mini-batch:

   ```
   y = split_heads(x)  →  y[0, 1, 2, 3] = x[0, 2, num_heads * 1 + 3]
   x = merge_heads(y)  →  x[0, 1, num_heads * 2 + 3] = y[0, 2, 1, 3]
   ```

   These two functions are exactly inverse from one another. Note that in the code, the number of heads m is called `self.num_heads`, and the head dimension d is `self.head_size`. Your functions must handle mini-batches of sequences of vectors, see the docstring for details about the input/output signatures.

3. In the class `MultiHeadedAttention`, implement the function `get_attention_weights()`, which is responsible for returning $\boldsymbol{A}_i$'s (for all the heads at the same time) from $\boldsymbol{q}_i$'s and $\boldsymbol{k}_i$'s. Remember that the language model here is *auto-regressive* (also sometimes called *causal*), meaning that the attention must be computed only on past inputs, and not the future.

   Concretely, this means that instead of taking the softmax over the whole sequence, we need to introduce a binary mask $\boldsymbol{s}_t$ (which is different from the `mask` key in the dataloader), where $s_t(\tau)$ is equal to 1 if the current element can attend position τ in the sequence (i.e. if $\tau \leq t$), and 0 otherwise. The softmax is then modified as

   $$[\text{softmax}(\boldsymbol{x}, \boldsymbol{s}_t)]_\tau = \frac{\exp(x_\tau) s_t(\tau)}{\sum_i \exp(x_i) s_t(i)}.$$

   In practice, in order to avoid potential numerical stability issues, we recommend to use a different implementation:

   $$[\text{softmax}(\boldsymbol{x}, \boldsymbol{s}_t)]_\tau = \frac{\exp(\widetilde{x}_\tau)}{\sum_i \exp(\widetilde{x}_i)} \qquad \text{where} \qquad \widetilde{x}_\tau = x_\tau s_t(\tau) - 10^4 (1 - s_t(\tau))$$

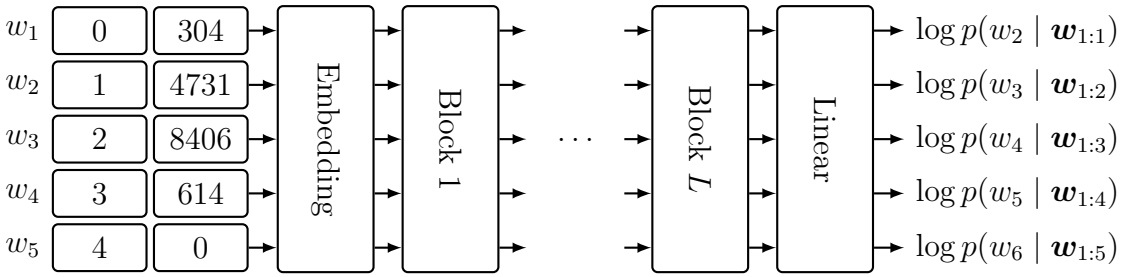
   The second version is almost equivalent to the first (up to numerical precision), as long as $x \gg -10^4$, which is the case in practice. You are strongly recommended to use vectorized operations as much as possible in order to speed-up training in Problem 3.

4. Using the functions you have implemented, complete the function `apply_attention()` in the class `MultiHeadedAttention`, which computes the vectors $\boldsymbol{h}_i$'s as a function of $\boldsymbol{q}_i$'s, $\boldsymbol{k}_i$'s and $\boldsymbol{v}_i$'s, and concatenates the head vectors.

$$\texttt{apply\_attention}(\{\boldsymbol{q}_i\}_{i=1}^m, \{\boldsymbol{k}_i\}_{i=1}^m, \{\boldsymbol{v}_i\}_{i=1}^m) \texttt{ = } \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m).$$

5. Using the functions you have implemented, complete the function `forward()` in the class `MultiHeadedAttention`. You may implement the different affine projections however you want (do not forget the biases), and you can add modules to the `__init__()` function. How many learnable parameters does your module have, as a function of `num_heads` and `head_size`?

The GPT forward pass (5pts): You now have all building blocks to implement the forward pass of a miniature GPT model. You are provided a module `Block` which corresponds to a full block with self-attention and a feed-forward neural network, with skip-connections, using the modules `LayerNorm` and `MultiHeadedAttention` you implemented before. The architecture of the GPT model is the following:



In this part of the exercise, you will fill in the `MiniGPT1` class in `gpt1_solution_template.py`. This module contains all the blocks necessary to create this model. In particular, `self.embedding` is a module responsible for converting sequences of token indices into embeddings (using input and positional embeddings), `self.layers` is a `nn.ModuleList` containing the different `Block` layers, and `self.classifier` is a linear layer responsible for classification.

6. In the class `MiniGPT1`, implement the function `get_embeddings()` which computes the embedding vectors, based on the input sequences and the positions of the tokens. See the docstrings of `GPT1Embedding` (in `utils/embeddings.py`) for details about this module.

7. In the class `MiniGPT1`, complete the function `forward()` using the different modules described above. This function must return the log-probabilities (not the logits) of the next words in the sequence.

8. Complete the `loss()` function, that returns the mean negative log-likelihood of the entire sequences in the mini-batch (and also averaged over the mini-batch dimension). See the definition of the loss in Problem 1.

Problem 3

Training language models and model comparison (25pts) Unlike in classification problems, where the performance metric is typically accuracy, in language modelling, the performance metric is typically based directly on the cross-entropy loss, i.e. the negative log-likelihood (NLL) the model assigns to the tokens. For word-level language modelling it is standard to report **perplexity (PPL)**, which is the exponentiated average per-token NLL (over all tokens):

$$\exp\left(\frac{1}{TM}\sum_{t=1}^{T}\sum_{j=1}^{M} -\log p(\boldsymbol{w}_t^{(j)} \mid \boldsymbol{w}_1^{(j)}, ..., \boldsymbol{w}_{t-1}^{(j)}; \boldsymbol{\theta})\right),$$

where t is the index with the sequence, and j indexes different sequences. The purpose of this question is to perform model exploration, which is done using a validation set. As such, we do not require you to run your models on the test set.

You will train each of the following architectures using an optimization technique and scheduler of your choice. For reference, we have provided a *feature-complete* training script (`run_exp.py`) that uses the AdamW optimizer. You are free to modify this script as you deem fit. You do not need to submit code for this part of the assignment. However, you are required to create a report that presents the perplexities and training curve comparisions as specified in the following questions.

Note: For each experiment, closely observe the training curves, and report the lowest validation perplexity score across epochs (not necessarily the validation score for the last epoch)

Configurations to run: At the top of the runner file (`run_exp.py`), we have provided 12 experiment configurations for you to run. Together, these configurations span several choices of neural network architecture, optimizer, and weight-decay parameters. Perform the following analysis on the logs.

1. You are asked to run 12 experiments with different architectures, optimizers, and hyperparameters settings. These parameter settings are given to you at the top of the runner file (`run_exp.py`). For each of these 12 experiments, plot learning curves (train and validation) of perplexity over both **epochs** and **wall-clock-time**. Figures should have labeled axes and a legend and an explanatory caption.

2. Make a table of results summarizing the train and validation performance for each experiment, indicating the architecture and optimizer. Sort by architecture, then number of layers, then optimizer, and use the same experiment numbers as in the runner script for easy reference. Bold the best result for each architecture.[2] The table should have an explanatory caption, and appropriate column and/or row headers. Any shorthand or symbols in the table should be explained in the caption.

3. Which hyperparameters + optimizer would you use if you were most concerned with wall-clock time? With generalization performance?

[2]You can also make the table in LaTeX; for convenience you can use tools like LaTeX table generator to generate tables online and get the corresponding LaTeX code.

4. Between the experiment configurations 1-4 and 5-8 at the top of `run_exp.py`, only the optimizer changed. What difference did you notice about the four optimizers used? What was the impact of weight decay, momentum, and ADAM?

5. Compare experiments 1 and 5. Which model did you think performed better (LSTM or GPT)? Why?

6. In configurations 5-8 and in configurations 11 and 12, you trained a transformer with various hyper-parameter settings. Given the recent high profile transformer based language models, are the results as you expected? Speculate as to why or why not.

7. For each of the experiment configurations above, measure the average steady-state GPU memory usage (`nvidia-smi` is your friend!). Comment about the GPU memory footprints of each model, discussing reasons behind increased or decreased memory consumption where applicable.

8. Comment on the overfitting behavior of the various models you trained, under different hyper-parameter settings. Did a particular class of models overfit more easily than the others? Can you make an informed guess of the various steps a practitioner can take to prevent overfitting in this case? (You might want to refer to sets of experiments 2, 9, 10 for the LSTM and 6, 11, 12 for GPT – that evaluate models of increasing capacities).