Due Date: November 14th, 2022 at 11:00 pm

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 3 (all questions).*
- *For open-ended experiments (i.e., experiments that do not have associated test-cases – Problem 3), you do not need to submit code – a report will suffice.*
- *TAs for this assignment are* ***Arian Khorasani*** *and* ***Nanda Harishankar Krishna****.*

Link (Notebook): click here

In this assignment, you will perform **neural machine translation** on the Multi30k dataset, between English and French. The dataset contains pairs of sentences in English and French. 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*. You will develop encoder-decoder models that process the sentence from the source language and translate it to the target language.

You will implement a Seq2Seq (encoder-decoder) model using **GRUs**, and an encoder-decoder **Transformer** model. In problem 1, you will use built-in PyTorch modules to implement an encoder-decoder GRU-based model. In problem 2, you will implement various building blocks of a transformer, including **LayerNorm** (layer normalization) and the **Attention** mechanism.

We have already pre-processed the Multi30k dataset and provided it to you in the form of dataloaders which provide fixed-length sequences. Throughout this assignment, **all sequences will have length 60**, and we will use zero-padding to pad shorter sequences. Each sample from the dataset is a tuple of the `source` (the input sentence in English) and the `target` (the target sentence in French). These have been encoded already, so you do not need to perform any other pre-processing. Each minibatch from the dataloader contains 128 sequences for translation.

Tests: We have provided some public test cases for certain functions so that you can verify your code. For example, we have provided sample random inputs to check your `forward` methods. Note that these tests are not comprehensive and you will be graded on a separate set of tests on 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.

Embeddings: In both the Seq2Seq GRU-based model 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). In this assignment, we learn the embeddings, i.e., we do not use pre-trained embeddings.

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 `solution.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`.

Problem 1

Implementing a GRU-based Seq2Seq model (22 pts): In this problem, you will be using PyTorch's built-in modules in order to implement a GRU-based Seq2Seq model. The architecture you will be asked to implement is the following:

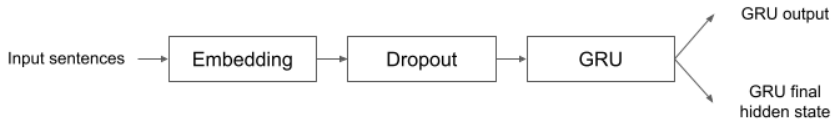

Figure 1: Encoder model

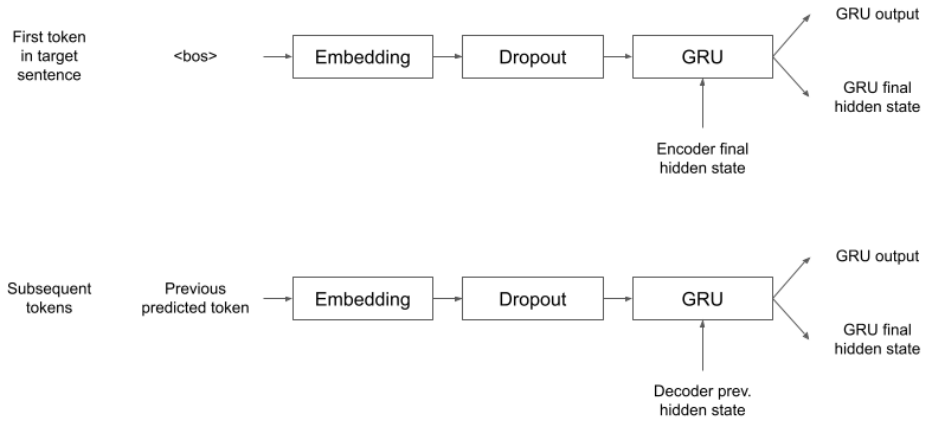

Figure 2: Decoder model

You are given:

- `Encoder` class containing all the blocks necessary to create the encoder model. In particular, `self.embedding` is a `nn.Embedding` module that converts sequences of token indices into embeddings, `self.dropout` is an instance of `nn.Dropout` and `self.gru` is a `nn.GRU` module that runs a GRU over a sequence of vectors.
- `Decoder` class containing all the blocks necessary to create the decoder model. In particular, `self.embedding` is a `nn.Embedding` module that converts a token index (the previous predicted token) into an embedding, `self.dropout` is an instance of `nn.Dropout` and `self.gru` is a `nn.GRU` module that runs a GRU that takes the embedded token and the previous hidden state (for the first target token, this is the encoder's final hidden state) as input, and returns the updated hidden state and output.
- `Seq2Seq` class which puts together the model. This consists of the encoder and decoder models and performs the entire process of encoding and decoding.

1. (5 pts) For `Encoder`, complete the `__init__()` and `forward()` function. The `forward()` function must return the outputs and the final hidden state of the GRU.
2. (5 pts) For `Decoder`, complete the `__init__()` and `forward()` function. The `forward()` function takes in the previous token and previous hidden state as inputs and must return the outputs and the final hidden state of the GRU.
3. (5 pts) For `Seq2Seq`, complete the `__init__()` and `forward()` function. The `forward()` function takes in source and target sentences. You will also implement greedy decoding and random sampling with temperature for decoding sentences.
   - For greedy decoding, at each decoding timestep you must choose the token with the highest score greedily.
   - For random sampling with temperature, you will calculate the probabilities of tokens at each decoding timestep as

     $$p(x_i|x_{1:i-1}) = \frac{\exp\left(\frac{o_i}{t}\right)}{\sum_{j=0}^{n_{\text{tokens}}-1} \exp\left(\frac{o_j}{t}\right)}$$

     where x_i are tokens at timestep i, o_i represent the values predicted by the decoder at timestep i for every token in the vocabulary and t represents the temperature factor between 0 and 1.

     Using these probabilities, you will sample a token at every decoder timestep.
4. (2 pts) Complete the `get_criterion()` function, which returns an `nn.CrossEntropyLoss` object which ignores the padding index (as specified in the notebook).
5. (5 pts) Complete `train` and `validate`, to perform each training and validation epoch.

Problem 2

Implementing an Encoder-Decoder 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. 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. 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 $\boldsymbol{s}_t$, 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, $\boldsymbol{v}_i$'s and the mask, and concatenates the head vectors.

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

5. Using the functions you have implemented, complete the function `forward()` in the class `MultiHeadAttention`. 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`?

Answer 2. Below is the answer to (2.5):

Each linear layer contains a weight of shape (`num_heads` $\times$ `head_size`, `num_heads` $\times$ `head_size`) and a bias of shape (`num_heads` $\times$ `head_size`, 1). There are 4 such layers, so the total number of parameters is given by $4 \times (\texttt{num\_heads} \times \texttt{head\_size})^2 + 4 \times \texttt{num\_heads} \times \texttt{head\_size}$.

Forward pass and decoding (5pts): You now have all building blocks to implement the forward pass of an encoder-decoder Transformer model. You are provided modules `EncoderLayer` and `DecoderLayer` which correspond to a full block with self-attention (and encoder-decoder attention in `DecoderLayer`) using the modules `LayerNorm` and `MultiHeadAttention` you implemented before. You will then implement 2 decoding strategies in the `compute_bleu` function.

6. In this part of the exercise, you will fill in the `Encoder` and `Decoder` classes' `forward` method. This module contains all the blocks necessary to create the model. In particular, `self.embedding` is a module responsible for converting sequences of token indices into embeddings (using token and positional embeddings), `self.layers` is a `nn.ModuleList` containing the different `EncoderLayer` or `DecoderLayer` layers, and `self.classifier` in `Decoder` is a linear layer to give scores of next (predicted) tokens.

7. Implement greedy decoding and random sampling with temperature in the `compute_bleu` function. The implementation is almost identical to that of Problem 1.

Problem 3

Training language models and model comparison (25pts): For machine translation, it is standard to report the BLEU score as a measure of a model's performance. It is defined as "the weighted geometric mean of all the modified n-gram precisions, multiplied by the brevity penalty". You can read more about it here. We provide code to calculate the BLEU score (after you have completed the decoding strategies in the `compute_bleu` function). We ask you to report only BLEU-1 and BLEU-2, i.e., for 1-gram precision and the geometric mean of 1- and 2-gram precisions respectively. In research it is standard to report the geometric mean of 1- to 4-gram precisions.

During training, we will also look at perplexity, which is the exponential of the cross-entropy loss. While it is common to monitor BLEU during training, we do not do this because it greatly increases the compute time (as you have to run decoding).

The purpose of this question is to perform model exploration and evaluation. You will train each of the configurations specified and record the metrics we ask for in your report.

Note: For each experiment, closely observe the training curves, and report the lowest validation loss/perplexity across epochs (not necessarily the validation score for the last epoch). Also report the final test BLEU score.

Configurations to run:

For the GRU-based model and the Transformer, run the following configurations:

1. Adam optimizer, `use_dropout=False`, greedy decoding
2. SGD optimizer, `use_dropout=True`, greedy decoding
3. Adam optimizer, `use_dropout=True`, greedy decoding
4. Adam optimizer, `use_dropout=True`, random sampling with temperature

1. You are asked to run 8 experiments with different architectures, optimizers, and decoding strategies. These parameter settings are given to you as separate cells in the notebook. For each of these 8 experiments, plot learning curves (train and validation) of the loss and perplexity over both **epochs** and **time**. Figures should have labeled axes and a legend and an explanatory caption.
2. Make a table of results summarizing the train, validation and test performance for each experiment, indicating the architecture, optimizer, dropout and decoding strategy. Sort by architecture, then `use_dropout`, then optimizer and finally by decoding strategy. Bold the best result for each architecture. You will report loss and perplexity (best across epochs) and the final test BLEU score.[2]
3. Which model and configuration would you use if you were most concerned with time? With generalization performance? What was the impact of Dropout?
4. Compare the GRU and Transformer based on the experiment configuration: Adam optimizer, `use_dropout=False`, greedy decoding. Which model did you think performed better? Why?
5. You have trained a transformer with a few different configurations in this experiment. Given the recent high profile transformer-based language models, are the results as you expected? Speculate as to why or why not.

[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.

6. 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.

7. Comment on the overfitting behavior of the various models you trained. 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?

Answer 3. We provide solutions to most questions here (except the plots). Note that while we had fixed the random seed for every experiment, there may be some small differences in the values reported.

(3.1) Some pointers:

- You may plot multiple curves in a single plot as long as a legend and informative caption is included.
- There must be a total of 64 curves: 8 experiments, each with training and validation curves for both loss and perplexity, over both epochs and time.
- Note that you must use `np.cumsum()` to plot the loss and perplexity values over cumulative time on the x-axis.

(3.2) The results are reported in Table 1.

Architecture	use_dropout	Optimizer	Decoding	Train Loss	Train PPL	Val Loss	Val PPL	Test Loss	Test PPL	Test BLEU-1	Test BLEU-2
GRU	False	Adam	Greedy	4.942	140.080	5.017	151.013	5.005	149.101	24.054	14.339
GRU	True	SGD	Greedy	6.332	562.514	6.271	528.986	6.289	538.523	3.539	0.0
GRU	True	Adam	Greedy	4.890	132.992	4.930	138.386	4.909	135.438	27.179	16.109
GRU	**True**	**Adam**	**Random**	**4.512**	**91.073**	**4.561**	**95.679**	**4.532**	**92.990**	**33.805**	**16.871**
Transformer	**False**	**Adam**	**Greedy**	**2.744**	**15.549**	**2.754**	**15.704**	**2.693**	**14.774**	**45.301**	**59.457**
Transformer	True	SGD	Greedy	9.476	13039.658	9.378	11822.598	9.369	11713.893	0.0	0.026
Transformer	True	Adam	Greedy	3.846	46.827	3.722	41.330	3.706	40.690	22.978	36.617
Transformer	True	Adam	Random	3.846	46.827	3.722	41.330	3.706	40.690	22.105	38.329

Table 1: Table of results for all 8 experiments, where the models are trained for 10 epochs.

(3.3) If we are most concerned with time, we could use one of the GRU models, e.g. GRU-True-Adam-Greedy or GRU-True-Adam-Random, which has good performance and lower training time. Note however that in this particular case, the number of parameters is much higher for the Transformer and there is a very small difference in the training time. This is because Transformers take advantage of parallelism during training. Thus, in general, we would prefer Transformers (especially at scale). Also note that during inference in the Transformer, the encoder can be parallelized while the decoder is sequential just like the GRU models.

If we are most concerned with performance, we would go with the Transformer-False-Adam-Greedy model which has the highest BLEU score. Note that usually models with Dropout do not overfit and could generalize better, but the lower performance here could be due to training for less epochs or due to sub-optimal hyperparameter choices.

We notice that Dropout doesn't have much of an effect here – the models do not overfit much and so in the case of the Transformers, adding Dropout increases the time taken to converge. The poor performance could be due to training for less epochs or due to poor hyperparameter choices. For the GRU, we see slight improvement in performance on adding Dropout.

(3.4) The Transformer model performed much better, as indicated by its BLEU scores. The Transformer is capable of viewing all words in the sentence to make each prediction and thus can model long term dependencies better than the GRU. Furthermore, among the models considered, the Transformer has much greater capacity (higher number of parameters). These are a couple of reasons that explain the Transformer's good performance.

(3.5) Perhaps the results are as expected – the Transformer outshines the GRU almost across the board. While these models are not even close to the level of models like GPT, the results in this simple setting indicate the promise of these models at scale. With large scale pre-training and more data and parameters, we can expect these models to perform even better. One of the reasons the Transformer could be performing better than the GRU is that it models long term dependencies better.

Further inspection of the translations output by the Transformer would indicate that it actually doesn't perform very well, and this could indeed be because of the small size of the dataset or the lack of pre-training.

(3.6) We expect you to report the steady state GPU usage for all configurations here. These values may vary drastically based on the hardware you are using. On Google Colab, we generally expect that GPU usage with the GRU models is around 2-3 GB while for the Transformer models it is around 6-8 GB. Dropout tends to increase memory consumption as it would need to remember the neurons that were dropped out in order to selectively propagate gradients (the other weights would still have to be in memory).

(3.7) We do not notice much overfitting in these experiments. This is because we are training the models for just a few epochs. There are some indications that GRU models without Dropout may overfit, but adding Dropout could fix these. Methods to prevent overfitting would include regularization techniques such as Dropout or using more data. If we are training models for many epochs, early stopping based on improvement in validation performance could also prevent overfitting.