Due Date : March 22th 23 :00, 2024

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),* ***unless specified****, you do not need to submit code – a report will suffice.*
- *You cannot use ChatGPT for this assignment. You are encouraged to ask questions on the Piazza, or to email/Slack questions or book office hours with the TA (****Aleksei Efremov, Email :*** *`aleksei DOT efremov AT mila DOT quebec`).*

Summary : In this assignment, you will take your implemented modules and perform sentiment analysis on the Yelp Polarity dataset. You will compare the performance of the RNNs to various configurations of the Transformer. This dataset will be pre-processed by a BERT based Hugging Face tokenizer, which outputs the padded sequence with a corresponding mask denoting where the padding is, and sentiment label.

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

We provide you with the dataset and its pre-processing pipeline that result in the dataloaders which provide fixed-length sequences. Throughout this assignment, **all sequences will have length 256**, and we will use zero-padding to pad shorter sequences. You will work with mini-batches of data, each with batchsize `B` elements.

Embeddings : In both the LSTM-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)`. In this assignment, apart from the fine-tuning task, 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 incredible amount of 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 there 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`.

Question 1. Implementing a LSTM Encoder Decoder with soft attention (22 pts) In this problem, you will be using PyTorch's built-in modules in order to implement LSTM and various architectures that make use of LSTM.

1. (10 pts) Long-Short Term Memory Network (LSTM) learns how to forget unimportant and retain important information using input, target, cell, and forget gates, and a running memory cell.

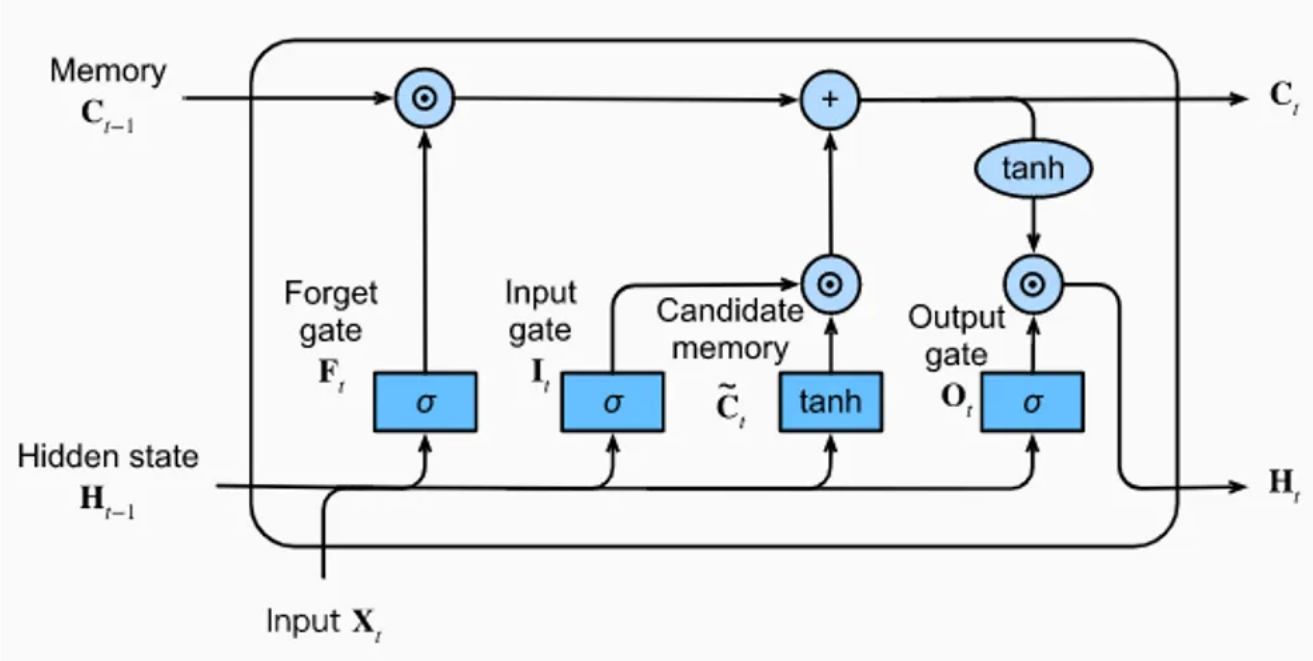


FIGURE 1 – Architecture of a LSTM Unit (Credits)

The following set of equations defines a portion of the forward pass for LSTM :

$$
\begin{aligned}
i_t &= \sigma(x_t W_{ii}^T + b_{ii} + h_{t-1} W_{hi}^T + b_{hi}) \\
f_t &= \sigma(x_t W_{if}^T + b_{if} + h_{t-1} W_{hf}^T + b_{hf}) \\
g_t &= \tanh(x_t W_{ig}^T + b_{ig} + h_{t-1} W_{hg}^T + b_{hg}) \\
o_t &= \sigma(x_t W_{io}^T + b_{io} + h_{t-1} W_{ho}^T + b_{ho}) \\
c_t &= f_t * c_{t-1} + i_t * g_t \\
h_t &= o_t * \tanh(c_t).
\end{aligned}
$$

Where $*$ indicates elementwise product.

In this problem, you will take the given `LSTM` class and implement it using the above set of equations. Be careful of indexing along sequences, and of appropriately storing each hidden state for the output. For this problem, **you are not allowed** to use the PyTorch `nn.LSTM` or `nn.LSTMCell` module. Your implementation will be compared against the outputs and backwards pass of Pytorch's `nn.LSTM` implementation.

2. (6pts) Bidirectional encoders make use of two RNN layers ; the 'forward' layer applies an RNN to an input sequence from start to end, while the 'backward' layer applies an RNN to a sequence end to start. You must implement a bidirectional LSTM encoder, with dropout on the embedding layer. The two directions of the LSTM network will be concatenated.

   The **PyTorch** implementation of the model (`nn.LSTM`) should be used for this and subsequent questions.

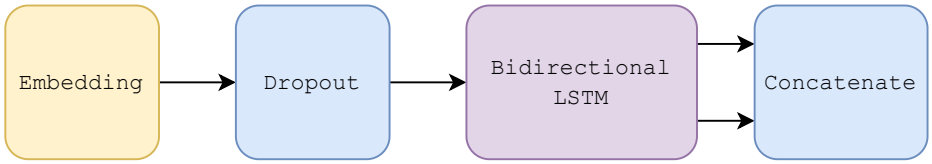


FIGURE 2 – Encoder Network.

3. (6pts) Attention mechanisms help with giving the network access to and concentrating on specific information in the sequence. Attention also helps with improving gradient flow. Now, you will implement a decoder with an *option* to include self-attention mechanism. The attention network itself will be the one you implement in the Problem 2, with just 1 head. The decoder must take in the encoder outputs as input and hidden state. The masked input must be fed into the attention mechanism and then the attended input to the decoder or the input will be fed directly to the decoder. The encoder hidden state should also be fed into the decoder LSTM layer.

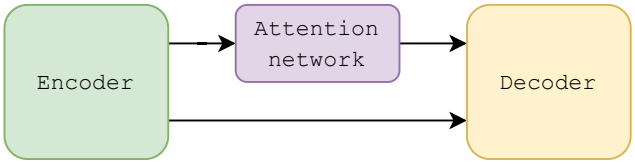


FIGURE 3 – Decoder Network.

In the file `lstm.py`, you are given an `Encoder`, and `DecoderAttn`, classes containing all the blocks necessary to create this model. In particular, `self.embedding`, located in `Encoder`, is a `nn.Embedding` module that converts sequences of token indices into embeddings, while `self.rnn`, located in both `Encoder` and `DecoderAttn` is a `nn.LSTM` module that runs a LSTM over a sequence of vectors.

Question 2. Implementing a Transformer (33pts) 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 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. The attention mechanism here must use padded masking in order to pass the unit tests.

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. (5pts) In the file `transformer_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 (22pts) : 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 :

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

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

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

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

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

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

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. (5pts) 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. (9pts) 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.Concretely, this means taking the softmax over the whole sequence. The softmax is then

$$[\text{softmax}(\boldsymbol{x})]_\tau = \frac{\exp(x_\tau)}{\sum_i \exp(x_i)}$$

   !Implement padded masking before the softmax !
4. (3pts) 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.

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

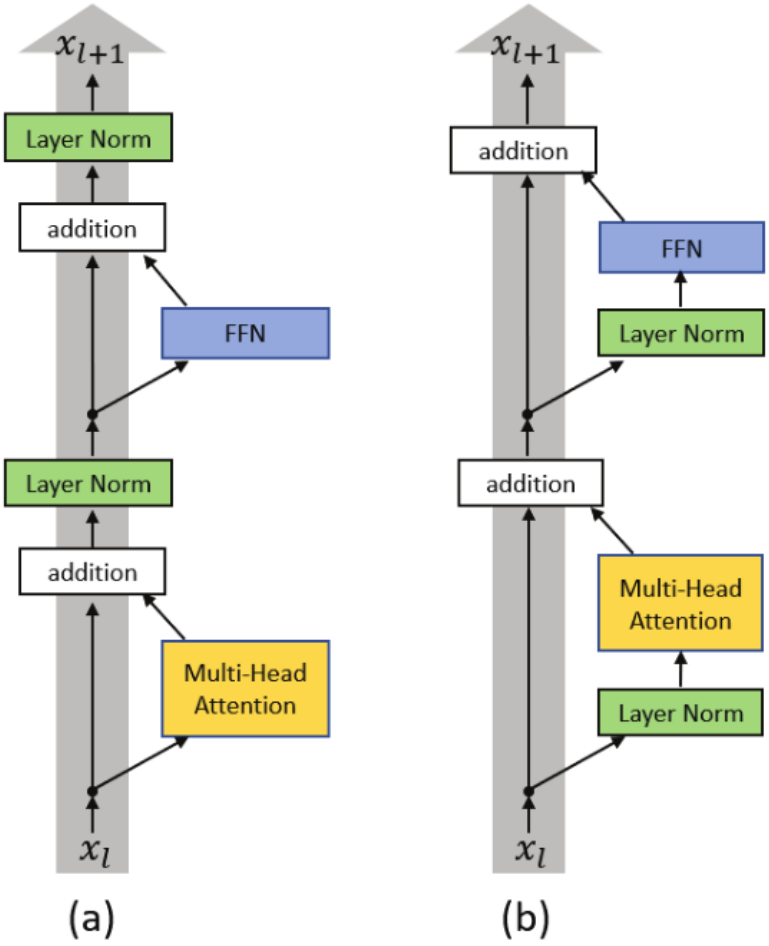
5. (5pts) Using the functions you have implemented, complete the function `forward()` in the class `MultiHeadedAttention`. You may implement the different affine projections however you want (do not forget the biases), and you can add modules to the `__init__()` function. How many learnable parameters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

The Transformer forward pass (6pts) : You now have all building blocks to implement the forward pass of a miniature Transformer model. You are provided a module `PostNormAttentionBlock` 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.

In this part of the exercise, you will fill in the `Transformer` class in `transformer.py`. This module contains all the blocks necessary to create this model. In particular, an embedding layer using

input and positional embeddings, `self.transformer` is a `nn.ModuleList` containing the different `Attention Block` layers.

6. (2pts) By taking inspiration from the `PostNormAttentionBlock`, implement the PreNormAttentionBlock. You can look at the implementation of the forward function of the PostNorm block to complete the forward function in PreNormAttentionBlock. See the figure below for a comparison of the post-norm and pre-norm.



(a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

FIGURE 4 – Image from Ruibin Xiong et al On Layer Normalization in the Transformer Architecture

7. (4pts) In the class `Transformer`, complete the function `forward()` using the different modules described above.

Question 3. Training Sequential models (45pts)
Before starting this assignment make sure your code for the previous problems works correctly by uploading it to the Gradescope !

You will train and evaluate each of the following architectures. For reference, we have provided a *feature-complete* training notebook
(`main.ipynb`) that uses the AdamW optimizer. You are free to modify this notebook 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 accuracy and training curve comparisons as specified in the following questions. You will train each model for at least 3 epochs except for the fine-tuning of pre-trained BERT - in that case you need only 2 epochs - one for training the classifier, another for fine-tuning. You are free to train any model for longer if you think that will improve your report.

Note : For each experiment, closely observe the training curves, and report the best validation accuracy score across training iterations (not necessarily the validation score for the last training iteration).

Configurations to run : We have provided 8 experiment configurations for you to run. These configurations span over several neural network architectures. Each run creates a log. Perform the following analysis on the logs.

1. (6pts) You are asked to run 8 experiments. For each of these experiments, plot learning curves (train loss and validation loss/accuracy) over **training iterations**. Figures should have labeled axes and a legend and an explanatory caption.
2. (5pts) Make a table of results summarizing the train and validation performance for each experiment, indicating the architecture, and including total wall clock time it took to train the architecture, and how long it takes to evaluate the model. [2] Sort by experiment number, and make sure to refer to these experiment numbers in bold script for easy reference. Bold the best validation result. 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. (2pts) Among the 8 configurations, which would you use if you were most concerned with wall-clock time ? With generalization performance ?
4. (3pts) Compare experiments **1** and **2**. What was the impact of having dropout ?
5. (3pts) Compare experiments **2** and **3**. Are there benefits of having Encoder-Decoder network instead of just an Encoder ? Speculate as to why or why not.
6. (3pts) Compare experiments **3** and **4**. What was the impact of training the LSTM network with self-attention ?
7. (3pts) In experiments **5, 6, 7** and **8**, you trained or fine-tuned a Transformer. Given the recent high profile Transformer based models, are the results as you expected ? Speculate as to why or why not.
8. (3pts) For each of the experiment configurations above, measure the average steady-state GPU memory usage. Comment about the GPU memory footprints of each model, discussing reasons

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.

behind increased or decreased memory consumption where applicable. One way to measure GPU usage would be via monitoring `nvidia-smi`. If you are using Google Colab, use the resources tab to monitor GPU usage.

9. (7pts) Comment on the learning curve behavior of the various models you trained, under different hyper-parameter settings. Did a particular class of models over/underfit more easily than the others ? Exhibited instabilities in training ? Can you make an informed guess of the various steps a practitioner can take to prevent over/under-fitting or instability in this case ?
10. (10pts) Finally, we have prepared for you the code to perform different perturbations on the input text to see how robust your models are. You have a set of possible perturbations and a set of trained models. You are free to choose any perturbations to test the robustness of your models. Provide an analysis of your interventions and explanations of why a particular model was robust or not to your perturbations. Be sure to include the examples you got. You should test at least 3 models and at least 9 combinations of perturbations.