Due Date: March 27th 23:00, 2023

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 (All Questions), and 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 (**Muawiz, Email:** `FIRST_NAME DOT LAST_NAME AT mila DOT quebec`).*
- *TAs for this assignment are **Muawiz Sajjad Chaudhary** (IFT6135B) and **Ghait Boukachab** (IFT6135A).*

Summary: In this assignment, you will implement a sequential language model (a **GRU**) with cross-attention and a self attention model (a **Transformer**). You will also investigate the limitations and biases of Large Language Models (LLMs).

In problem 1, you will use built-in PyTorch modules to implement a variety of RNN architectures (GRU, Bidirectional GRU Encoder, GRU Encoder Decoder with Soft Attention) along with implementing the Transformer architecture via it's various building blocks.

In problem 2, you will take your implemented modules and perform sentiment analysis on the Amazon Polarity dataset. You will compare the performance of the RNNs to various configurations of the Transformer.

In problem 3, you will use HuggingFace to critically analyze the limitations of LLMs and various sources of bias.

The Amazon Polarity dataset comprises 35 million Amazon reviews. See this HuggingFace Datasets page for details about the Amazon Polarity dataset and sample data. 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. Modern day datasets are so large that often models are trained on only one (or half!) pass through the entire dataset; emulating this approach you will train with one epoch over the entire train dataset with 3.6 million training examples, and evaluate on a subset of unseen datapoints.

You are provided a google colab allowing you to train your implemented RNN and Transformer models on the dataset. 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.

Coding instructions You will be required to use PyTorch to complete all questions. Moreover, completing this assignment in a timely manner **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 `IFT6135_2023_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. **If you are using Google Colab, it is heavily recommended that you test your RNN and Transformer implementations on CPU first.**

Problem 1

Implementing a GRU Encoder Decoder with soft attention (15 pts) In this problem, you will be using PyTorch's built-in modules in order to implement a GRU and various architectures that make use of a GRU

1. (6 pts) A Gated Recurrent Unit is a simplified version of a Long-Short Term Memory Network (LSTM). LSTMs learn how to forget and retain important information using input, target, cell, and forget gates, and a running memory cell. GRUs abandons the running memory cell and couples the notion of a input and forgetting gate with an update gate to help model long term dependencies, with reset gates modeling short term dependencies. The GRU is smaller in parameter size, but experimental evidence often shows GRU networks performing similarly to LSTM networks.

   The following set of equations defines a portion of the forward pass for a GRU.

$$\begin{aligned} r_t &= \sigma(x_t W_{ir}^T + b_{ir} + h_{t-1} W_{hr}^T + b_{hr}) \\ z_t &= \sigma(x_t W_{iz}^T + b_{iz} + h_{t-1} W_{hz}^T + b_{hz}) \\ n_t &= \tanh(x_t W_{in}^T + b_{in} + r_t * (h_{t-1} W_{hn}^T + b_{hn})) \\ h_t &= (1 - z_t) * n_t + z_t * h_{t-1}. \end{aligned}$$

   Where $*$ indicates elementwise product.

   In this problem, you will take the given `GRU` class and implement a GRU 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.GRU` or `nn.GRUCell` module. Your implementation will be compared against the outputs and backwards pass of Pytorch's `nn.GRU` implementation, on CPU using `torch.float64`. Note however that your implementation must also work with `torch.float32` inputs and on a CUDA enabled device.

2. (3pts) 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 GRU encoder, with dropout on the embedding layer. The two directions of the GRU network will be summed up together.

   Due to speed and numerical instability issues, you will use the Pytorch implementation of `nn.GRU` in developing your Encoder and Decoder networks.

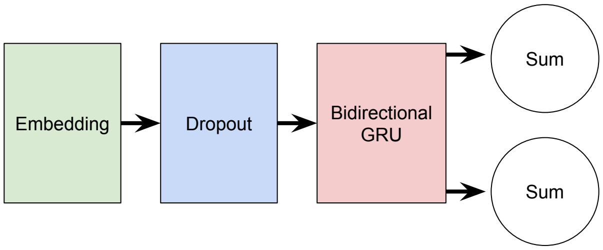


Figure 1: Encoder Network.

3. (4pts) 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 one layer MLP which implements a cross attention mechanism. You **must** make sure that this MLP attention model can perform masking on the attention vector.

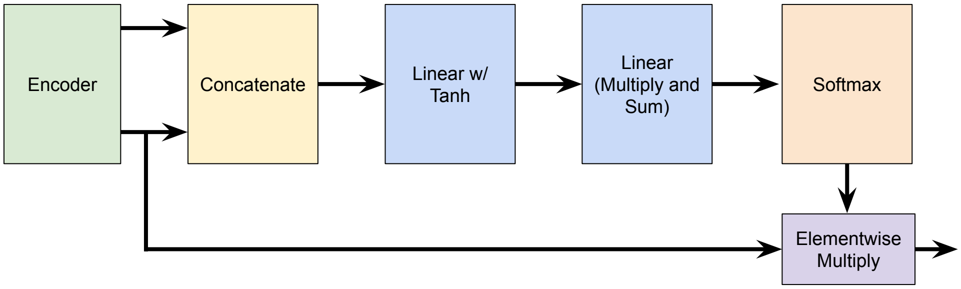


Figure 2: Attention Network.

4. (2pts) The decoder must take in the encoder outputs as input and hidden state, and these must be fed into the attention mechanism. The attended input and the encoder hidden state will then be fed into a GRU layer.

In the file `encoder_decoder_solution.py`, you are given an `Encoder`, `Attn`, and `DecoderAttn`, class 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.GRU` module that runs a GRU over a sequence of vectors.

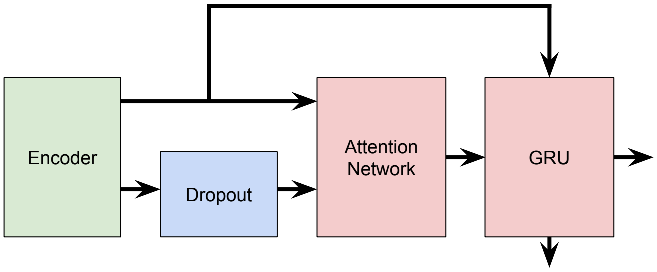


Figure 3: Decoder Network.

Implementing a Transformer (26pts) 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.

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

Implementing the attention mechanism (18pts): 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:

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

$$\begin{aligned} \boldsymbol{A}_i &= \text{softmax}\left(\frac{\boldsymbol{q}_i\boldsymbol{k}_i^\top}{\sqrt{d}}\right) \\ \boldsymbol{h}_i &= \boldsymbol{A}_i\boldsymbol{v}_i \\ A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m)\boldsymbol{W}_O + \boldsymbol{b}_O \end{aligned}$$

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. (4pts)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 or after the softmax.

4. (2pts)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.

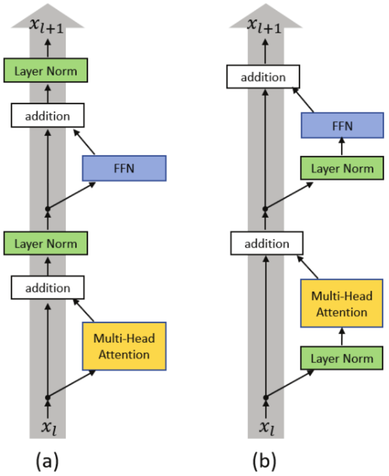
$$\texttt{apply\_attention}(\{\boldsymbol{q}_i\}_{i=1}^m, \{\boldsymbol{k}_i\}_{i=1}^m, \{\boldsymbol{v}_i\}_{i=1}^m) = \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m).$$

5. (3pts)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 (3pts): 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_solution_template.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. (1pts) 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. (2pts)In the class `Transformer`, complete the function `forward()` using the different modules described above.

Problem 2

Training Sequential models (22pts) You will train and evaluate each of the following architectures. For reference, we have provided a *feature-complete* training notebook (`IFT6135_2023_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.

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 6 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. (4pts)You are asked to run 6 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. (3pts)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 6 configurations, which would you use if you were most concerned with wall-clock time? With generalization performance?

4. (2pts) Compare experiments **1** and **2**. What was the impact of training the GRU network with attention?

6. (2pts)In experiment **3, 4, 5** and **6**, you trained or finetuned a Transformer. Given the recent high profile Transformer based models, are the results as you expected? Speculate as to why or why not.

7. (2pts)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 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.

[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.

8. (2pts)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?

9. (5 pts) In this assignment you implemented Cross Attention in the form of Soft Attention, and then you implemented Self Attention. Give a high level overview of what an attention mechanism is, and go into each attention mechanism (Self, Cross), and their differences. Write two paragraphs total.

Problem 3

Critically Examining Limitations and Bias of Large Language Models (36 pts) In the previous problem, you finetuned a pretrained BERT model on the sentiment analysis task and should have obtained good performance while noting increased resource utilization. While finetuning a large pretrained BERT model and getting great performance might be impressive, sending these models to production requires considering several limitations and tradeoffs.

Aside from resource constraints, you will investigate the biases produced by these Large Language Models (LLMs) and critically think about several possible sources of bias, from the data to the model level.

You will use a notebook in the following questions, located here, to inspect biases encoded by a BERT model. You must submit this notebook (titled `IFT6135_2023_Problem_2_Notebook.ipynb`).

Each model on HuggingFace has it's own model card, a descriptive accounting of various metadata about the model, such as it's training data and in what applications the model is used for. Take note of the Limitations and bias section of the model card for the BERT model.

```
from transformers import pipeline

unmasker = pipeline('fill-mask', model='bert-base-uncased')

unmasker(‘‘The man works as a [MASK].”)
>>>['carpenter', 'waiter', 'barber', 'mechanic',  'salesman']

unmasker(‘‘The woman works as a [MASK].”)
>>>['nurse', 'waitress', 'maid', 'prostitute', 'cook']
```

As noted, the model produces biased predictions, and this bias will affect all fine-tuned versions of this model.

According to the model card for the ‘bert-base’ model, the model is trained on BooksCorpus, a dataset consisting of 11,038 unpublished books, and English Wikipedia (excluding lists, tables and headers).

You will look at the biases of a model trained on this dataset more in-depth, pondering data and model-level biases and investigating model-level interventions. Finally, you will weigh various considerations for sending LLMs into production.

1. (8 pts) Play around with the above set of two entries given to the unmasker. Make three **original sets**, of at least size two, of fill-mask prompts that induce the model to exhibit negative bias towards a traditionally minoritized population (such as women, people of color, queer, lower caste, etc.) and a positive bias towards a traditionally normative population (men, white, straight, upper caste, etc.). In your report, define your assumptions/context of what is "minoritized" and what is "normative". What biases are your examples showing?

2. (2 pts) Come up with one "switched" (anti-stereotype) example where the model exhibits positive-negative bias the other way around. Explain the bias being shown here, how you came up with this example, and include this example in the notebook submission.

3. (10 pts) Visualize and contrast your sets + switched example using the interactive scatter plot provided here, under the section "What's in a Name?". Provide these scatter plots and comparisons in your report with commentary. Do the correlations give evidence for the biases you show?

   Now look under the section "How Can We Fix This"? Read what is written in the section; is the type of bias you are showing in your prompts reduced by the mentioned gender bias mitigation? If the biases you show are not related to gender, should you expect these biases to be mitigated? Will your switched example still be a switched example? Write up your thoughts before moving on.

   Now validate your hypothesis; visualize and contrast your sets + switched example using the provided interactive scatter plot with **model set to Zari** and the same settings as your previous set of graphs/comparisons. Provide your analysis on the accuracy of your hypothesis.

4. (4 pts) Two models were trained on the same datasets but do not exhibit the same biases. How/Why?

5. (6 pts) Critically analyze the English Wikipedia dataset. Wikipedia takes note of it's own institutional gender biases here. How is English Wikipedia gender biased? What are the causes of this? While qualitative points and arguments can be discussed and brought up here, make sure you mention or use quantitative facts. You are encouraged to use additional sources as long as you properly cite them.

6. (6 pts) Much research has shown a whole host of biases encoded within Transformer models. However, the traditional evaluation performance of these models (when fine-tuned) is oftentimes good on a variety of tasks (such as speech recognition, sentiment analysis, named entity recognition, information retrieval, etc). Transformer models are often sent into production due to this utility, and there is a need to balance ethical and economic trade-offs.

   In what deployment contexts might it be OK to use a biased BERT model? In what contexts not? Come up with three (total) examples.