Programming Assignment 3: Attention-Based Neural Machine Translation

Due Date: Sat, Mar. 20th, at 11:59pm

Submission: You must submit 3 files through MarkUs[1]: a PDF file containing your writeup, titled `a3-writeup.pdf`, and your code files `nmt.ipynb` and `bert_and_gpt.ipynb`. Your writeup must be typed.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

Errata:

- Part 1: Previously, the handout specified using a 10 dimensional embedding. Instead, you should use an embedding dimension equal to the RNN encoder hidden dimension. This requires no changes to the code notebook.

[1] `https://markus.teach.cs.toronto.edu/csc413-2021-01`

[2] `https://csc413-uoft.github.io/2021/assets/misc/syllabus.pdf`

Introduction

In this assignment, you will train a few attention-based neural machine translation models to translate words from English to Pig-Latin. Along the way, you'll gain experience with several important concepts in NMT, including *long short-term memory architectures* and *attention*.

Pig Latin

Pig Latin is a simple transformation of English based on the following rules (applied on a per-word basis):

1. If the first letter of a word is a *consonant*, then the letter is moved to the end of the word, and the letters "ay" are added to the end: `team` → `eamtay`.
2. If the first letter is a *vowel*, then the word is left unchanged and the letters "way" are added to the end: `impress` → `impressway`.
3. In addition, some consonant pairs, such as "sh", are treated as a block and are moved to the end of the string together: `shopping` → `oppingshay`.

To translate a whole sentence from English to Pig-Latin, we simply apply these rules to each word independently:

`i went shopping` → `iway entway oppingshay`

Goal: We would like a neural machine translation model to learn the rules of Pig-Latin *implicitly*, from (English, Pig-Latin) word pairs. Since the translation to Pig Latin involves moving characters around in a string, we will use *character-level* recurrent neural networks for our model.

Because English and Pig-Latin are so similar in structure, the translation task is almost a copy task; the model must remember each character in the input, and recall the characters in a specific order to produce the output. This makes it an ideal task for understanding the capacity of NMT models.

Setting Up

We recommend that you use **Colab**(`https://colab.research.google.com/`) for the assignment, as all the assignment notebooks have been tested on Colab. From the assignment zip file, you will find one python notebook file: `nmt.ipynb`. To setup the Colab environment, just upload this notebook file using the upload tab at `https://colab.research.google.com/`.

Data

The data for this task consists of pairs of words $\{(s^{(i)}, t^{(i)})\}_{i=1}^{N}$ where the *source* $s^{(i)}$ is an English word, and the *target* $t^{(i)}$ is its translation in Pig-Latin.

In this assignment, you will investigate the effect of dataset size on generalization ability. We provide a small and large dataset. The small dataset is composed of a subset of the unique words from the book "Sense and Sensibility," by Jane Austen. The vocabulary consists of 29 tokens: the 26 standard alphabet letters (all lowercase), the dash symbol `-`, and two special tokens `<SOS>` and `<EOS>` that denote the start and end of a sequence, respectively. [3] The dataset contains 3198 unique (English, Pig-Latin) pairs in total; the first few examples are:

```
{ (the, ethay), (family, amilyfay), (of, ofway), ... }
```

The second, larger dataset is obtained from Peter Norvig's natural langauge corpus[4]. It contains the top 20,000 most used English words, which is combined with the previous data set to obtain 22,402 unique words. This dataset contains the same vocabulary as the previous dataset.

In order to simplify the processing of *mini-batches* of words, the word pairs are grouped based on the lengths of the source and target. Thus, in each mini-batch the source words are all the same length, and the target words are all the same length. This simplifies the code, as we don't have to worry about batches of variable-length sequences.

Outline of Assignment

Throughout the rest of the assignment, you will implement some attention-based neural machine translation models, and finally train the models and examine the results. You will first implement three main building blocks: Long Short-Term Memory (LSTM), Additive attention and Scaled dot-product attention. Using these building blocks, you will implement two encoders (RNN and transformer encoders) and three decoders (RNN, RNN+additive attention and transformer decoders). Using these, you will train three final models:

- Part 1: (RNN encoder) + (RNN decoder)
- Part 2: (RNN encoder) + (RNN decoder with additive attention)
- Part 3: (Transformer encoder) + (Transformer decoder)
- Part 4: Fine-tuning pretrained transformers

[3] Note that for the English-to-Pig-Latin task, the input and output sequences share the same vocabulary; this is not always the case for other translation tasks (i.e., between languages that use different alphabets).

[4] https://norvig.com/ngrams/

Deliverables

Each section is followed by a checklist of deliverables to add in the assignment writeup. To also give a better sense of our expectations for the answers to the conceptual questions, we've put maximum sentence limits. You will not be graded for any additional sentences.

Part 1: Long Short-Term Memory (LSTM) [2pt]

Translation is a *sequence-to-sequence* problem: in our case, both the input and output are sequences of characters. A common architecture used for seq-to-seq problems is the encoder-decoder model [4], composed of two RNNs, as follows:

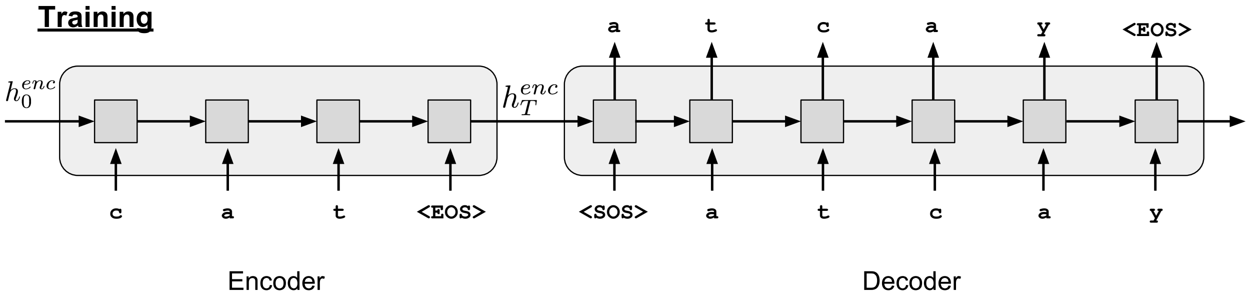

Figure 1: Training the NMT encoder-decoder architecture.

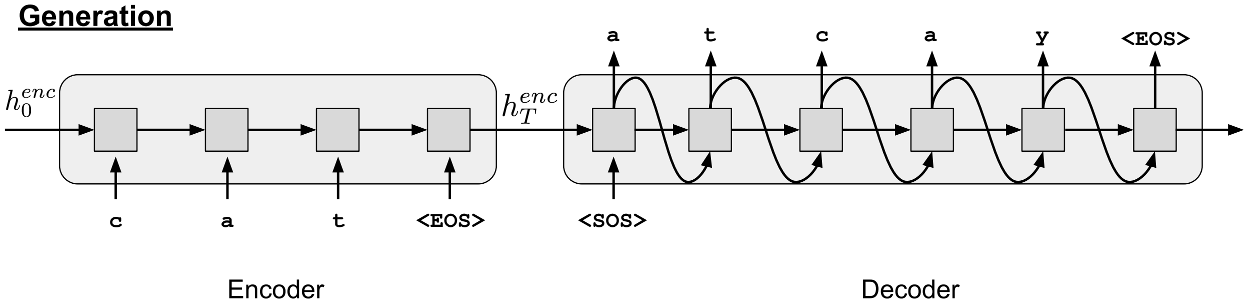

Figure 2: Generating text with the NMT encoder-decoder architecture.

The encoder RNN compresses the input sequence into a fixed-length vector, represented by the final hidden state h_T. The decoder RNN conditions on this vector to produce the translation, character by character.

Input characters are passed through an embedding layer before they are fed into the encoder RNN. Where H is the dimension of the encoder RNN hidden state, we learn a $29 \times H$ embedding matrix, where each of the 29 characters in the vocabulary is assigned a H-dimensional embedding. At each time step, the decoder RNN outputs a vector of *unnormalized log probabilities* given by a linear transformation of the decoder hidden state. When these probabilities are normalized, they define a distribution over the vocabulary, indicating the most probable characters for that time step. The model is trained via a cross-entropy loss between the decoder distribution and ground-truth at each time step.

The decoder produces a distribution over the output vocabulary conditioned on the previous hidden state and the output token in the previous timestep. A common practice used to train NMT models is to feed in the *ground-truth token* from the previous time step to condition the decoder output in the current step. This training procedure is known as "teacher-forcing" shown in Figure 1. At test time, we don't have access to the ground-truth output sequence, so the decoder must condition its output on the token it generated in the previous time step, as shown in Figure 2. Let's begin with implementing common encoder models: the LSTM and the transformer encoder.

Open `https://colab.research.google.com/github/csc413-uoft/2021/blob/master/assets/assignments/nmt.ipynb` on Colab and answer the following questions.

1. [0.5pt] The forward pass of a LSTM unit is defined by the following equations:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{1}$$
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{2}$$
$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{3}$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{4}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{5}$$
$$h_t = o_t \odot \tanh(c_t) \tag{6}$$

where $\odot$ is the element-wise multiplication. Although PyTorch has a built in LSTM implementation (`nn.LSTMCell`), we'll implement our own LSTM cell from scratch, to better understand how it works. Complete the `__init__` and `forward` methods of the `MyLSTMCell` class, to implement the above equations. A template has been provided for the `forward` method.

Train the RNN encoder/decoder model on both datasets. We've provided implementations for recurrent encoder/decoder models using the LSTM cell. (Make sure you have run all the relevant previous cells to load the training and utility functions.)

At the end of each epoch, the script prints training and validation losses, and the Pig-Latin translation of a fixed sentence, "the air conditioning is working", so that you can see how the model improves qualitatively over time. The script also saves several items:

- The best encoder and decoder model parameters, based on the validation loss.
- A plot of the training and validation losses.

After the models have been trained on both datasets, `pig_latin_small` and `pig_latin_large`, run the `save_loss_comparison_lstm` method, which compares the loss curves of the two models. Answer the following two questions in less than 3 sentences: Does either model perform significantly better? Why might this be the case?

2. [0.5pt] After the training is complete, pick the best model and use it to translate test sentences using the `translate_sentence` function. Try a few of your own words by changing the variable `TEST_SENTENCE`. Identify a distinct failure mode and briefly describe it.

3. [1pt] Consider an LSTM encoder with an H dimensional hidden state, and an input sequence with V vocabulary size, D embedding features size, and K length. Write down the number of neurons and connections of this encoder model as a function of H, K, and D. For simplicity, you may ignore the bias units.

Deliverables

Create a section in your report called **LSTMs**. Add the following in this section:

- A screenshot of your full `MyLSTMCell` implementation, the loss plots output by `save_loss_comparison_lstm`, and your analysis. [0.5pt]

- Your answer for the question in step 2. Make sure to include the input-output pair for the failure case you identify. Your answer should not exceed **three** sentences in total (excluding the failure cases you've identified.) [0.5pt]

- Your answer for question 3. [1pts]

Part 2: Additive Attention [2pt]

Attention allows a model to look back over the input sequence, and focus on relevant input tokens when producing the corresponding output tokens. For our simple task, attention can help the model remember tokens from the input, e.g., focusing on the input letter `c` to produce the output letter `c`.

The hidden states produced by the encoder while reading the input sequence, $h_1^{enc}, \dots, h_T^{enc}$ can be viewed as *annotations* of the input; each encoder hidden state h_i^{enc} captures information about the i^{th} input token, along with some contextual information. At each time step, an attention-based decoder computes a *weighting* over the annotations, where the weight given to each one indicates its relevance in determining the current output token.

In particular, at time step t, the decoder computes an attention weight $\alpha_i^{(t)}$ for each of the encoder hidden states h_i^{enc}. The attention weights are defined such that $0 \leq \alpha_i^{(t)} \leq 1$ and $\sum_i \alpha_i^{(t)} = 1$. $\alpha_i^{(t)}$ is a function of an encoder hidden state and the previous decoder hidden state, $f(h_{t-1}^{dec}, h_i^{enc})$, where i ranges over the length of the input sequence.

There are a few engineering choices for the possible function f. In this assignment, we will investigate two different attention models: 1) the additive attention using a two-layer MLP and 2) the scaled dot product attention, which measures the similarity between the two hidden states.

To unify the interface across different attention modules, we consider attention as a function whose inputs are triple (queries, keys, values), denoted as (Q, K, V).

In the additive attention, we will *learn* the function f, parameterized as a two-layer fully-connected network with a ReLU activation. This network produces unnormalized weights $\tilde{\alpha}_i^{(t)}$ that are used to compute the final context vector.

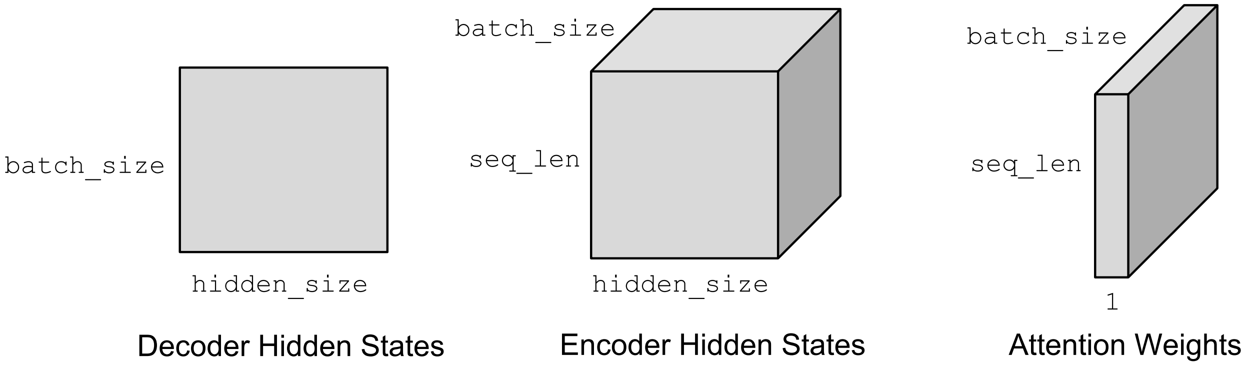

Figure 3: Dimensions of the inputs, Decoder Hidden States (*query*), Encoder Hidden States (*keys/values*) and the attention weights ($\alpha^{(t)}$).

For the `forward` pass, we are given a batch of query of the current time step, which has dimension `batch_size x hidden_size`, and a batch of keys and values for each time step of the input sequence, both have dimension `batch_size x seq_len x hidden_size`. The goal is to obtain the context vector. We first compute the function $f(Q_t, K)$ for each query in the batch and *all* corresponding keys K_i, where i ranges over `seq_len` different values. Since $f(Q_t, K_i)$ is a scalar, the resulting tensor of attention weights has dimension `batch_size x seq_len x 1`. Some of the important tensor dimensions in the `AdditiveAttention` module are visualized in Figure 3. The `AdditiveAttention` module returns both the context vector `batch_size x 1 x hidden_size` and the attention weights `batch_size x seq_len x 1`.

1. [1pt] Read how the provided `forward` methods of the `AdditiveAttention` class computes $\tilde{\alpha}_i^{(t)}, \alpha_i^{(t)}, c_t$. Write down the mathematical expression for these quantity as a function of $W_1, W_2, b_1, b_2, Q_t, K_i$.

   (Hint: Take a look at the equations in Part 4.1 for the scaled dot product attention model.)

   $$\begin{aligned}\tilde{\alpha}_i^{(t)} &= f(Q_t, K_i) = \\ \alpha_i^{(t)} &= \\ c_t &=\end{aligned}$$

   Here, $\tilde{\alpha}_i^{(t)}$ is the unnormalized attention weights; $\alpha_i^{(t)}$ is the attention weights in between 0 and 1; c_t is the final context vector.

2. [0pt] The notebook provides all required code to run the additive attention model. Run the notebook to train a language model that has additive attention in its decoder. Find one training example where the decoder with attention performs better than the decoder without attention. Show the input/outputs of the model with attention, and the model without attention that you've trained in the previous section.

3. [0pt] How does the training speed compare? Why?

4. [1pt] Given an input sequence of length K with V vocabulary size and D embedding features size, write down the number of neurons and the number of connections in `RNNAttentionDecoder` as a function of hidden state size H, D, and K. Assume the attention network is parameterized as in `AdditiveAttention`. For simplicity, you may ignore the bias units.

Deliverables

Create a section called **Additive Attention**. Add the following in this section:

- Three equations for question 1. [1pt]

- Training/validation plots you've obtained in this section. [0 pts]
- Answers to question 2. [0 pts]
- Answer to question 3. [0 pts]
- Answer to question 4. [1pt]

Part 3: Scaled Dot Product Attention [3pt]

1. [0.5pt] In lecture, we learnt about Scaled Dot-product Attention used in the transformer models. The function f is a dot product between the linearly transformed query and keys using weight matrices W_q and W_k:

$$\begin{aligned}\tilde{\alpha}_i^{(t)} &= f(Q_t, K_i) = \frac{(W_q Q_t)^T (W_k K_i)}{\sqrt{d}},\\ \alpha_i^{(t)} &= \text{softmax}(\tilde{\alpha}^{(t)})_i,\\ c_t &= \sum_{i=1}^{T} \alpha_i^{(t)} W_v V_i,\end{aligned}$$

where, d is the dimension of the query and the W_v denotes weight matrix project the value to produce the final context vectors.

Implement the scaled dot-product attention mechanism. Fill in the `forward` methods of the `ScaledDotAttention` class. Use the PyTorch torch.bmm (or @) to compute the dot product between the batched queries and the batched keys in the forward pass of the `ScaledDotAttention` class for the unnormalized attention weights.

The following functions are useful in implementing models like this. You might find it useful to get familiar with how they work. (click to jump to the PyTorch documentation):

- squeeze
- unsqueeze
- expand_as
- cat
- view
- bmm (or @)

Your forward pass **needs to** work with both 2D query tensor (`batch_size x (1) x hidden_size`) **and** 3D query tensor (`batch_size x k x hidden_size`).

2. [0.5pt] **Implement the causal scaled dot-product attention mechanism.** Fill in the `forward` method in the `CausalScaledDotAttention` class. It will be mostly the same as the `ScaledDotAttention` class. The additional computation is to mask out the attention to the future time steps. You will need to add `self.neg_inf` to some of the entries in the unnormalized attention weights. You may find torch.tril handy for this part.

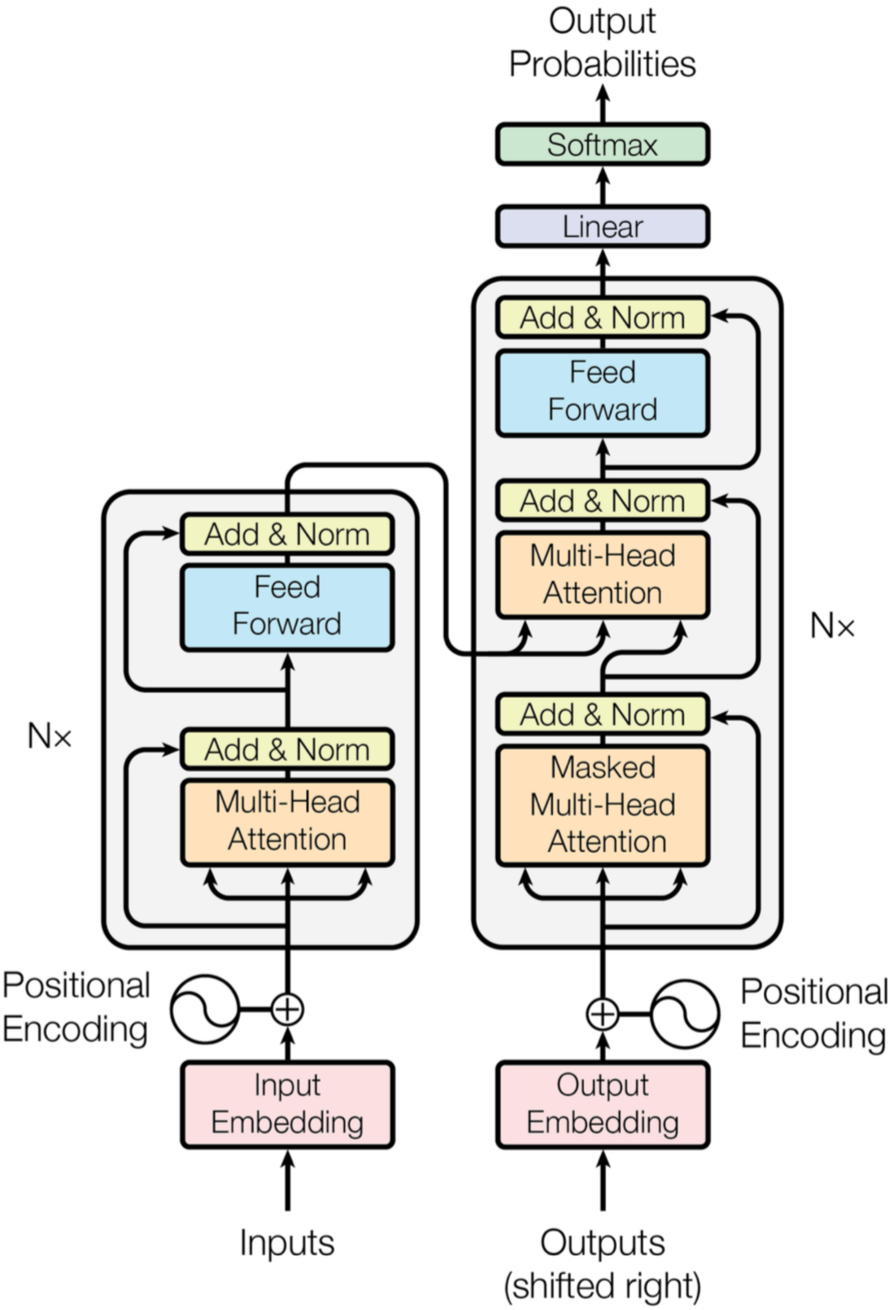

Figure 4: The transformer architecture. [5]

3. [0.5pt] We will now use `ScaledDotAttention` as the building blocks for a simplified transformer [5] encoder.

   The encoder looks like the left half of Figure 4. The encoder consists of three components:

   - Positional encoding: To encode the position of each word, we add to its embedding a constant vector that depends on its position:

     $$\text{pth word embedding} = \text{input embedding} + \text{positional encoding}(p)$$

     We follow the same positional encoding methodology described in [5]. That is we use sine and cosine functions:

     $$\text{PE}(\text{pos}, 2i) = \sin \frac{\text{pos}}{10000^{2i/d_{model}}} \tag{7}$$

     $$\text{PE}(\text{pos}, 2i+1) = \cos \frac{\text{pos}}{10000^{2i/d_{model}}} \tag{8}$$

     Since we always use the same positional encodings throughout the training, we pregenerate all those we'll need while constructing this class (before training) and keep reusing them throughout the training.
   - A `ScaledDotAttention` operation.
   - A following MLP.

   For this question, describe why we need to represent the position of each word through this positional encoding in one or two sentences. Additionally, describe the advantages of using this positional encoding method, as opposed to other positional encoding methods such as a one hot encoding in one or two sentences.

4. [0.5pt] The `TransformerEncoder` and `TransformerDecoder` modules have been completed for you. Train the language model with transformer based encoder/decoder using the first configuration (hidden size 32, small dataset). How do the translation results compare to the previous decoders? Write a short, qualitative analysis.

5. [1pt] In the code notebook, we have provided an experimental setup to evaluate the performance of the Transformer as a function of hidden size and data set size. Run the Transformer model using hidden size 32 versus 64, and using the small versus large dataset (in total, 4 runs). We suggest using the provided hyper-parameters for this experiment.

   Run these experiments, and report the effects of increasing model capacity via the hidden size, and the effects of increasing dataset size. In particular, report your observations on how loss as a function of gradient descent iterations is affected, and how changing model/dataset size affects the generalization of the model. Are these results what you would expect?

In your report, include the two loss curves output by `save_loss_comparison_by_hidden` and `save_loss_comparison_by_dataset`, the lowest attained validation loss for each run, and your response to the above questions.

6. [0pt] The decoder includes the additional `CausalScaledDotAttention` component. Take a look at Figure 4. The transformer solves the translation problem using layers of attention modules. In each layer, we first apply the `CausalScaledDotAttention` self-attention to the decoder inputs followed by `ScaledDotAttention` attention module to the encoder annotations, similar to the attention decoder from the previous question. The output of the attention layers are fed into an hidden layer using ReLU activation. The final output of the last transformer layer are passed to the `self.out` to compute the word prediction. To improve the optimization, we add residual connections between the attention layers and ReLU layers.

   Modify the transformer decoder `__init__` to use non-causal attention for both self attention and encoder attention. What do you observe when training this modified transformer? How do the results compare with the causal model? Why?

7. [0pt] What are the advantages and disadvantages of using additive attention vs scaled dot-product attention? List one advantage and one disadvantage for each method.

Deliverables

Create a section in your report called **Scaled Dot Product Attention**. Add the following:

- Screenshots of your `ScaledDotProduct`, `CausalScaledDotProduct` implementations. Highlight the lines you've added. [1pt]
- Your answer to question 3. [0.5pt]
- Your response to question 4. Your analysis should not exceed **three** sentences. [0.5pt]
- The two loss curves plots output by the experimental setup in question 5, and the lowest validation loss for each run. [0.5pt]
- Your response to the written component of question 5. [0.5pt]
- Your response to question 6. Your response should not exceed **three** sentences. [0pt]
- Your response to question 7. [0pt]

Part 4: Fine-tuning for arithmetic sentiment analysis [2pt]

In this section, we will learn how to use pre-trained language models to determine whether an verbal numerical expression is negative (label 0), zero (label 1), or positive (label 2). For example, "eight minus ten" is negative so the output of our sentence classifier should output label index 0. We are going to use two transformer based models, GPT and BERT. We start by explaining what the two models are and how we can add a classifier on top of either pretrained model to perform sentiment analysis for verbal numerical expressions. Most code is given to you in the notebook `https://colab.research.google.com/github/csc413-uoft/2021/blob/master/assets/assignments/bert_and_gpt.ipynb`. Your task is to slightly modify the sentence classifier layer, make plots, report performances, and think about inference examples to test the model. Please carefully review the background for GPT and BERT before starting to answer the questions. The *Hugging Face transformers* library, used in this tutorial, has more than 40k stars on github due to its ease of use, and will be very useful for your research or projects in the future.

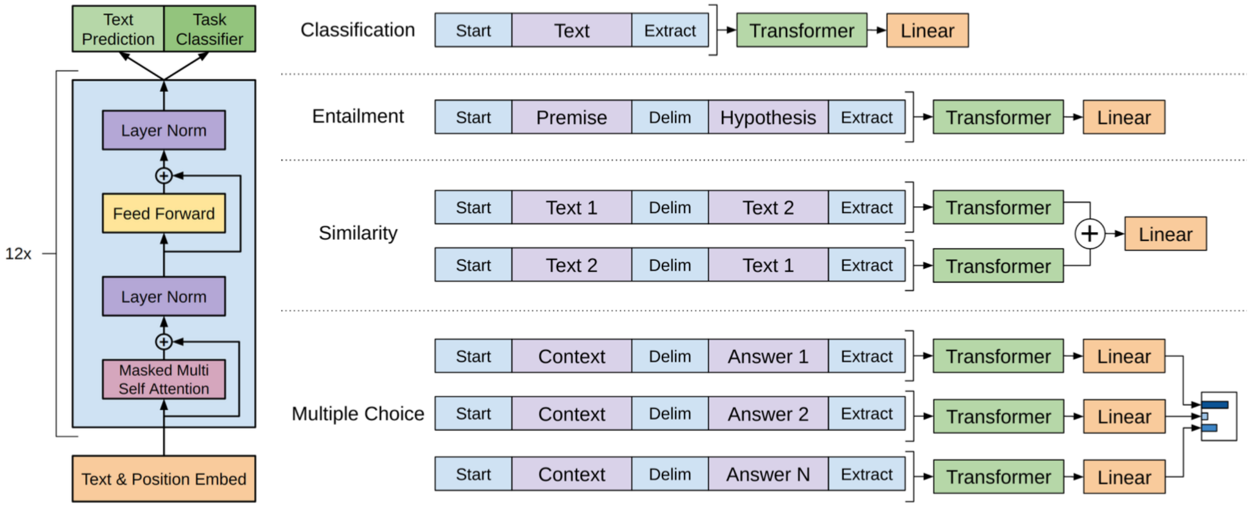

Figure 5: (left) Transformer architecture and training objectives used in GPT. (right) Input transformations for fine-tuning on different tasks. Reproduced from GPT paper [3]

Background for GPT:

In traditional language modeling task, the objective is to maximize the log likelihood of predicting the current word (or token) in the sentence, given the previous words (to the left of current work) as context. This is called the "autoregressive model". The GPT model [3] is an example of this kind which is pretrained on BookCorpus [6], a large dataset of unlabeled sentences and can be later fine-tuned on specific downstream tasks.

GPT use staked Transformer decoders [2] for its language modeling. It applies a multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens. See figure 5 for the overall architecture. In the pretraining stage, the training objective for predicting next token is used,

$$\mathcal{L}(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta),$$

where $\mathcal{U}$ is the collection of text tokens, $\{u_1, \ldots, u_n\}$. GPT process the text sentence unidirectionally by masking out the tokens at future positions (i.e. $> i$).

Background for BERT:

Bidirectional **E**ncoder **R**epresentations from **T**ransformers (BERT) [1], as the name suggests, is a language model based on the Transformer [5] encoder architecture that has been pre-trained on a large dataset of unlabeled sentences from Wikipedia and BookCorpus [6]. Given a sequence of tokens representing sentence(s), BERT outputs a "contextualized representation" vector for each of the token. Now, suppose we are given some down-stream tasks, such as sentence classification or question-answering. We can take the BERT model, add a small layer on top of the BERT representation(s), and then fine-tune the added parameters *and* BERT parameters on the down-stream dataset, which is typically much smaller than the data used to pre-train BERT.

In contrast to a unidirectional autoregressive model, however, BERT predicts the current word given both the words before and after (i.e. to the left and to the right) of the sentence–hence "bidirectional". To be able to attend from both directions, BERT uses the encoder Transformer, which does not apply any attention masking unlike the decoder.

We briefly describe how BERT is pre-trained. BERT has 2 task objectives for pre-training: (1) Masked Language Modeling (Masked LM), and (2) Next Sentence Prediction (NSP). The input to the model is a sequence of tokens of the form:

`[CLS] Sentence A [SEP] Sentence B`,

where `[CLS]` ("class") and `[SEP]` ("separator") are special tokens. In Masked LM, some percentage of the input tokens are converted into `[MASK]` tokens, and the objective is to use the final layer representation for that masked token to predict the correct word that was masked out[5]. For NSP, the task is to use the contextualized representation for the `[CLS]` token to perform binary classification for whether sentence A and sentence B are consecutive sentences in the unlabeled dataset. See Figure 6 for the conceptual picture of BERT pre-training and fine-tuning.

[5] The full training detail is slightly more complicated, but conceptually similar.

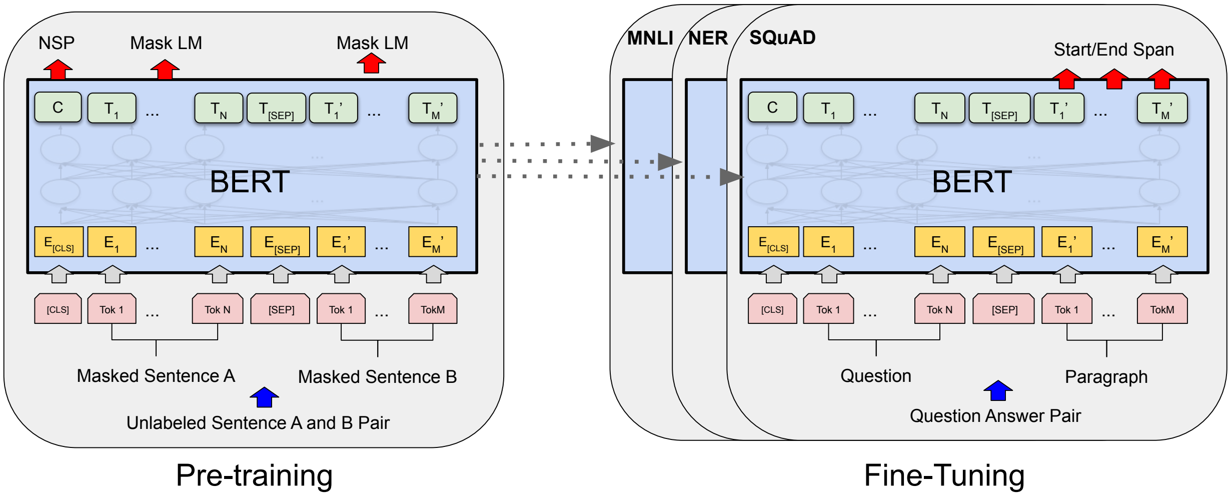

Figure 6: Overall pre-training and fine-tuning for BERT. Reproduced from BERT paper [1]

In this assignment, you will be **fine-tuning BERT and GPT on a single sentence classification task** (see below about the dataset).

Figure 7 illustrates the architecture for fine-tuning BERT on this task. We prepend the tokenized sentence with the `[CLS]` token, then feed the sequence into BERT. We then take the contextualized `[CLS]` token representation at the last layer of BERT and add either a softmax layer on top corresponding to the number of output classes in the task. Alternatively, we can have fully connected hidden layers before the softmax layer for more expressivity for harder tasks. Then, both the new layers and the entire BERT parameters are trained end to end on the task for a few epochs.

Similarly to fine-tune GPT on this task. We feed the tokenized sentence to the model and take the representation at the last transformer layer in GPT. We don't append special [CLS] tokens here in GPT, instead we take the representation at the position of last token in the sentence. Then we add either a softmax layer or additional fully connected layers, as in fine-tuning BERT. Finally, parameters in all layers are trained on the task for a few epochs.

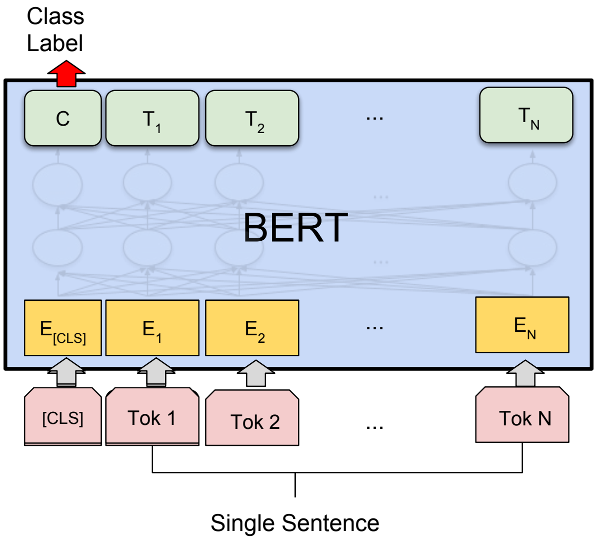

Figure 7: Fine-tuning BERT for single sentence classification by adding a layer on top of the contextualized `[CLS]` token representation. Reproduced from BERT paper [1]

Dataset Description

The verbal arithmetic dataset contains pairs of input sentence and label. The label is tertiary. Label 0, 1, 2 mean the input expressions are evaluated as "negative" , "zero", and "positive" respectively. Note that the size of training dataset is 640 and the size of test dataset is 160. In our dataset, we only have sentences with **three word tokens** as the input, similar to the examples shown below:

Input expression	Label	Label meaning
eighteen minus eighteen	1	"zero"
four plus seven	2	"positive"
four minus ten	0	"negative"

Questions:

1. [1pt] Classifier layer. Open the notebook `https://colab.research.google.com/github/csc413-uoft/2021/blob/master/assets/assignments/bert_and_gpt.ipynb`, we have provided two example BERT classes:

   `BertCSC413_Linear` and `BertCSC413_MLP` that both add a classifier for classification.

   In this part, you need to make your own `GPTCSC413 class` to add a classifier for the GPT model. You can follow the examples in BERT to similarly add either a linear layer or MLP layers.

2. [0pt] In the notebook, we instantiated two different BERT models from *BertCSC413_MLP* class, which are called *model_freeze_bert* and *model_finetune_bert* in the notebook. Run the training and evaluation functions to train both models.

   Comment on how these two models will differ during the training? Which one would lead to smaller training errors? Which one would generalize better? And briefly discuss why models are failing under certain target labels.

3. [0pt] We instantiated a GPT model from *GPTCSC413 class*, which is called *model_finetune_gpt* in the notebook. Run the training and evaluation functions for the model.

   Compare the performance of *model_finetune_bert* and *model_finetune_gpt*. Try a few unseen examples of arithmetic questions using both model, and find 10 interesting results. The interesting results can, for example, be both successful extrapolation/interpolation results or surprising failure cases.

4. [1pt] Come up with 1 scenario/application that GPT architecture is more preferred than BERT. The proposed scenario/appication is not limited to sentiment classification or Natual Language Processing (NLP).

5. [0pt] This is an open question, and we will give full marks as long as you show **an attempt to try one of the following tasks.** [1] Try data augmentation tricks to improve the performances for certain target labels that models were failing to predict. [2] Make a t-sne or PCA plot to visualize the embedding vectors of word tokens related to arithmetic expressions. [3] Try different hyperparameter tunings. E.g. learning rates, optimizer, architecture of the classifier, training epochs, and batch size. [4] Evaluate the Multi-class Matthews correlation score for our imbalanced test dataset. [5] Run a baseline model using MLP without pre-trained BERT or GPT. You can assume the sequence length of all the data is 3 in this case.

Deliverables:

- Description of how you build the sentence classifier and make sure it works in the training in question 3. Your answer should be **one sentence**. [1pt]
- Two training error curves with "freeze" and "fine-tuned" models. Two tables or lists that show the test performance with trained "freeze" and "fine-tuned" models. Your qualitative answer for question 2. Your answer should not exceed **4 sentences** [0pt]
- 10 inference results as well as brief comments on why they are interesting or representative results. Your answer should not exceed **3 sentences**, you don't need to describe all 10 inference results [0pt]
- Description of the scenario/application and the reason GPT is preferred. Your answer should not exceed **3 sentences**. [1pt]

- Explanation of what you did for the open question and some preliminary results. Your answer should not exceed **4 sentences**. [0pt]

What you need to submit

- Two code files: `nmt.ipynb`, `bert_and_gpt.ipynb`.
- A PDF document titled `a3-writeup.pdf` containing your answers to the conceptual questions.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[2] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*, 2018.

[3] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[4] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[6] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.