Programming Assignment 3: Natural Language Processing and Multimodel Learning

Due Date: Fri, Mar. 18th, 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`, `bert.ipynb` and `clip.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 could not finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

Errata:

- v1.1 (Wednesday March 16th): There was a subtle typo in equation 9. h_{t-1} should be multiplied by r_t before being multiplied by the weights W_{hh}. Please update your implementation of `MyGRUCell` accordingly.

[1] `https://markus.teach.cs.toronto.edu/2022-01/courses/16`
[2] `https://uoft-csc413.github.io/2022/assets/misc/syllabus.pdf`

Introduction

In this assignment, you will explore common tasks and model architectures in Natural Language Processing (NLP). Along the way, you will gain experience with important concepts like *recurrent* neural networks and *sequence-to-sequence* architectures (Part 1), *attention* mechanisms (Part 2), *pretrained language models* (Part 3) and *multimodal* vision and language models (Part 4).

Setting Up

We recommend that you use **Colab**(`https://colab.research.google.com/`) for the assignment. To setup the Colab environment, just open the notebooks for each part of the assignment and **make a copy** in your own Google Drive account.

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: Neural machine translation (NMT) [2pt]

Neural machine translation (NMT) is a subfield of NLP that aims to translate between languages using neural networks. In this section, will we train a NMT model on the toy task of English → Pig Latin. Please read the following background section carefully before attempting the questions.

Background

The task

Pig Latin is a simple transformation of English based on the following rules:

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 sentence from English to Pig-Latin, we apply these rules to each word independently:

`i went shopping` → `iway entway oppingshay`

Our goal is to build a NMT model that can 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 (RNNs). Because English and Pig-Latin are 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.

The 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.[3] The dataset contains 3198 unique (English, Pig-Latin) pairs in total; the first few examples are:

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

[3]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.

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.[4] The second, larger dataset is obtained from Peter Norvig's natural language corpus.[5] 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.

The model

Translation is a *sequence-to-sequence* (seq2seq) problem. The goal is to train a model to transform one sequence into another. In our case, both the input and output are sequences of characters. A typical architecture used for seq2seq problems is the encoder-decoder model [8], composed of two RNNs. The *encoder* RNN compresses the input sequence into a fixed-length vector, h_T^{enc}. The *decoder* RNN conditions on this vector to produce the translation, character by character. Input characters are passed through an embedding layer before being fed into the encoder RNN. If 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 (i.e. by passing them through a softmax), 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. 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" and is 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.

A common choice for the encoder and decoder RNNs is the **L**ong **S**hort-**T**erm **M**emory (LSTM) architecture [3]. The forward pass of a LSTM unit is defined by the following equations:

[4]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)

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

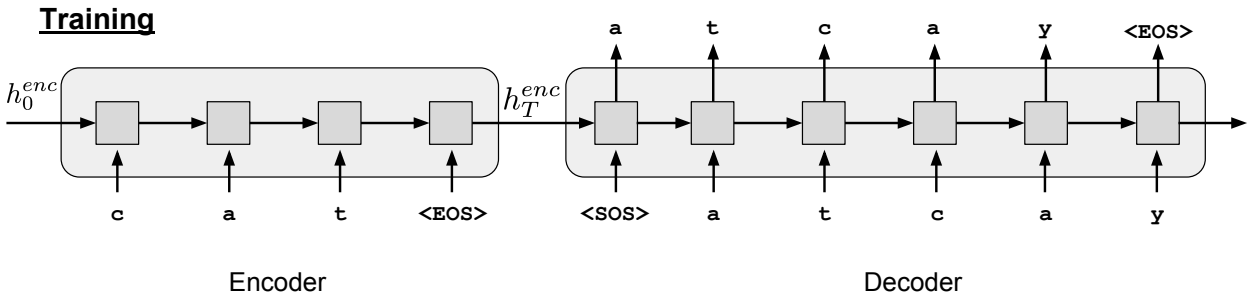

Figure 1: Training the NMT encoder-decoder architecture.

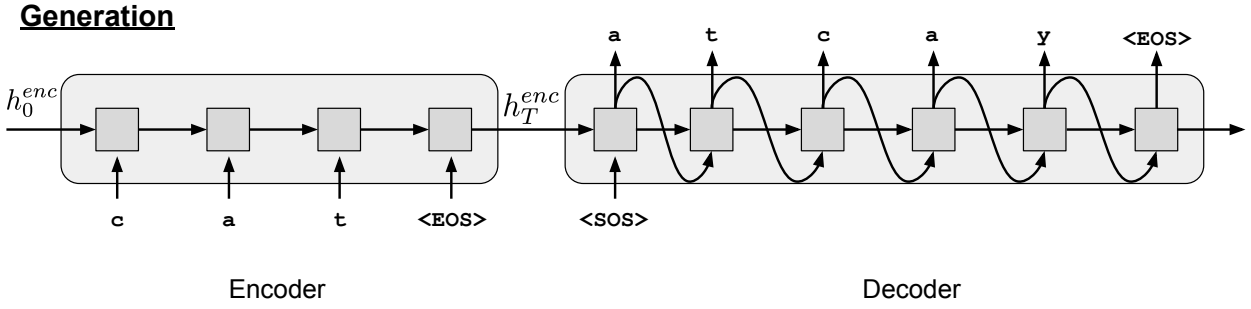

Figure 2: Generating text with the NMT encoder-decoder architecture.

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{1}$$
$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{2}$$
$$\tilde{C}_t = \tanh(W_{ic}x_t + b_{ic} + W_{hc}h_{t-1} + b_{hc}) \tag{3}$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{4}$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{5}$$
$$h_t = o_t * \tanh(C_t) \tag{6}$$

where $*$ is the element-wise multiplication, σ is the sigmoid activation function, x_t is the input for the timestep t, h_{t-1} is the hidden state from the previous timestep and W and b represent weight matrices and biases respectively. The **G**ated **R**ecurrent **U**nit (GRU) [1] can be seen as a simplification of the LSTM unit, as it combines the forget and input gates (f_t and i_t) into a single update gate (z_t) and merges the cell state (C_t) and hidden state (h_t)

$$z_t = \sigma(W_{iz}x_t + W_{hz}h_{t-1} + b_z) \tag{7}$$
$$r_t = \sigma(W_{ir}x_t + W_{hr}h_{t-1} + b_r) \tag{8}$$
$$\tilde{h}_t = \tanh(W_{ih}x_t + W_{hh}(r_t * h_{t-1}) + b_h) \tag{9}$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \tag{10}$$

See Understanding LSTM Networks for an excellent overview of LSTMs and GRUs. In this assignment, we will provide you with an implementation of an LSTM cell and ask you to implement a GRU cell. Please open `https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/nmt.ipynb` on Colab and answer the following questions.

1. [1pt] *Code a GRU Cell.* Although PyTorch has a built in GRU implementation (`nn.GRUCell`), we'll implement our own GRU cell from scratch, to better understand how it works. Complete the `__init__` and `forward` methods of the `MyGRUCell` class, by implementing the above equations.

   Train the RNN encoder/decoder model on both datasets. We've provided implementations for recurrent encoder/decoder models using your GRU 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_gru` method, which compares the loss curves of the two models. Then, answer the following questions in 3 sentences or less:

   - Does either model perform significantly better?
   - Why might this be the case?

2. [0.5pt] *Identify failure modes.* After training, pick the best model and use it to translate test sentences using the `translate_sentence` function. Try new words by changing the variable `TEST_SENTENCE`. Identify a distinct failure mode and describe it in 3 sentences or less.

3. [0.5pt] *Comparing complexity.* Consider a LSTM and GRU encoder, each with an H dimensional hidden state and an input sequence with V vocabulary size, D embedding features size, and K length.

- What are the total number of parameters of the LSTM encoder?
- What are the total number of parameters of the GRU encoder?

Provide your answer in terms of H, V, D and K (you may not need all terms). For simplicity, assume the input to the encoders has already been embedded and ignore the embedding layer in your parameter count. You should also ignore the bias units. You may find it useful to read through the PyTorch documentation for `LSTMCell` and `GRUCell` before answering.

Deliverables

Create a section in your report called **Part 1: Neural machine translation (NMT)**. Add the following in this section:

1. Answer to question 1. Include the completed `MyGRUCell`. Either the code or a screenshot of the code. Make sure both the `__init__` and `forward` methods are visible. Also include your answer to the two questions in **three** sentences or less. [1pt]
2. Answer to question 2. Make sure to include input-output pair examples for the failure case you identify. Your answer should not exceed **three** sentences in total (excluding the input-output pair examples.) [0.5pt]
3. Answer to question 3. Your answer should not exceed **one** sentence in length. [0.5pt]

Part 2.1: Additive Attention [1pt]

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, $h_1^{enc}, \ldots, h_T^{enc}$ 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.

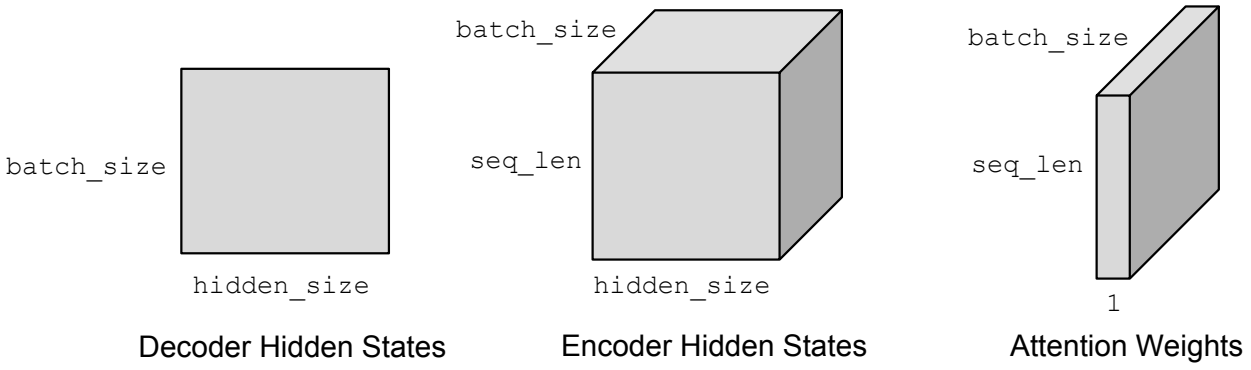

Figure 3: Dimensions of the inputs, Decoder Hidden States (*query*), Encoder Hidden States (*keys*/*values*) and the attention weights ($\alpha^{(t)}$).

In the `forward` pass, we are given a batch of queries, Q of the current time step, which has dimensions `batch_size x hidden_size`, and a batch of keys K and values V for each time step of the input sequence, both have dimensions `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. [0pt] Read how the provided `forward` methods of the `AdditiveAttention` class computes $\tilde{\alpha}_i^{(t)}, \alpha_i^{(t)}, c_t$.

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. [1pt] How does the training speed compare? Why? Explain your answer in no more than three sentences.

4. [0pt] Given an input sequence of length K and D embedding features size, assume the `RNNAttentionDecoder` uses this input to generate an output sequence of length K, which has V vocabulary size. Write down the number of LSTM units in `RNNAttentionDecoder` and the number of connections in the above computation, as a function of hidden state size H, V, D, and K. Assume the attention network is parameterized as in `AdditiveAttention`. For simplicity, you may ignore the bias units. You may also ignore the embedding process in your computations. However, do include the connections associated with the output layer.

Deliverables

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

- Answer to question 3. [1pt]

Part 2.2: Scaled Dot Product Attention [4pt]

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:

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

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.

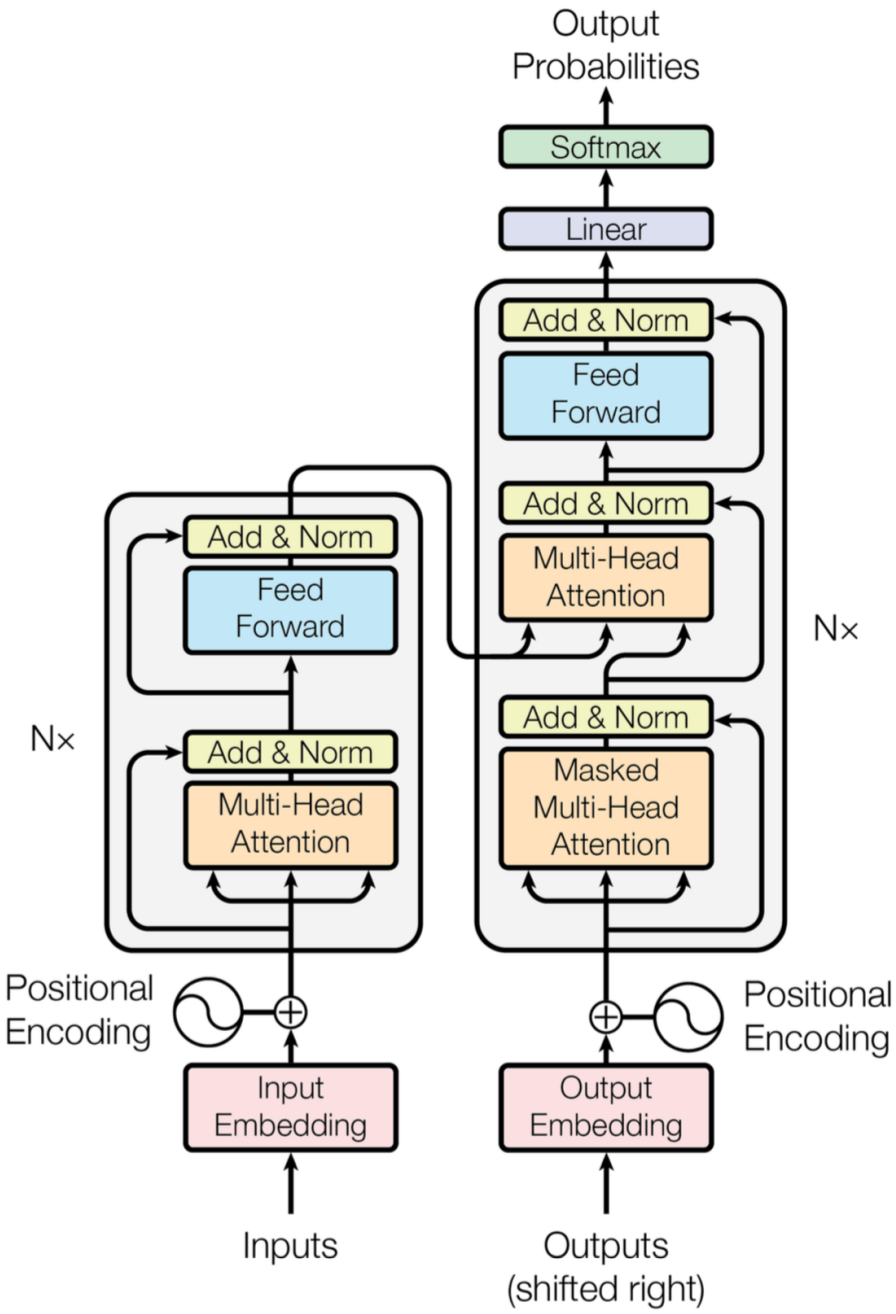

Figure 4: The transformer architecture. [9]

3. [0.5pt] We will train a model using the `ScaledDotAttention` mechanism as an encoder and decoder. Run the section `AttentionEncoder` and `AttentionDecoder` classes as well as the training block. Comment on the how performance of the network compares to the `RNNAttention` model. Why do you think the performance is better or worse?

4. [0.5pt] We will now use `ScaledDotAttention` as the building blocks for a simplified transformer [9] 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 [9]. That is we use sine and cosine functions:

     $$\text{PE}(\text{pos}, 2i) = \sin \frac{\text{pos}}{10000^{2i/d_{model}}} \tag{11}$$

     $$\text{PE}(\text{pos}, 2i+1) = \cos \frac{\text{pos}}{10000^{2i/d_{model}}} \tag{12}$$

     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.

5. [1pt] 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 `RNNAttention` and single-block `Attention` decoders? Write a short, qualitative analysis. Your answer should not exceed **three** sentences for each decoder (**six** total).

6. [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.

7. [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?

8. [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 answer to question 4. [0.5pt]
- Your response to question 5. Your analysis should not exceed **six** sentences. [0.5pt]
- The two loss curves plots output by the experimental setup in question 6, and the lowest validation loss for each run. [1pt]
- Your response to the written component of question 6. [0.5pt]

Part 3: Fine-tuning Pretrained Language Models (LMs) [2pt]

The previous sections had you train models *from scratch*. However, similar to computer vision (CV), it is now very common in natural language processing (NLP) to *fine-tune* pretrained models. Indeed, this has been described as "NLP's ImageNet moment."[6] In this section, we will learn how to fine-tune pretrained *language models* (LMs) on a new task. We will use a simple classification task, where the goal is to determine whether a verbal numerical expression is *negative* (label 0), *zero* (label 1), or *positive* (label 2). For example, "eight minus ten" is negative, so our classifier should output label index 0. As our pretrained LM, we will use the popular BERT model, which uses a transformer encoder architecture similar to the `TransformerEncoder` from Part 3. More specifically, we will explore two versions of BERT: **MathBERT** [7], which has been pretrained on a large mathematical corpus ranging from pre-kindergarten to college graduate level mathematical content and **BERTweet** [4], which has been pretrained on 100s of millions of tweets.

Most of the code is given to you in the notebook `https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/bert.ipynb`. The starter code uses the *HuggingFace Transformers* library[7], which has more than 50k stars on GitHub due to its ease of use, and will be very useful for your NLP research or projects in the future. Your task is to adapt BERT so that it can be fine-tuned on our downstream task. Before starting this section, please carefully review the background for BERT and the verbal arithmetic dataset (below).

Background

BERT

Bidirectional **E**ncoder **R**epresentations from **T**ransformers (BERT) [2] is a LM based on the Transformer [9] encoder architecture that has been pretrained on a large dataset of unlabeled sentences from Wikipedia and BookCorpus [10]. Given a sequence of tokens, BERT outputs a "contextualized representation" vector for each token. Because BERT is pretrained on a large amount of text, these contextualized representations encode useful properties of the syntax and semantics of language.

BERT has 2 pretraining objectives: (1) Masked Language Modeling (MLM), 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 MLM, some percentage of the input tokens are randomly "masked" by replacing them with the `[MASK]` token, and the objective is to use the final layer representation for that masked token to predict the correct word that was masked out[8]. In NSP, the task is to use the contextualized representation of the `[CLS]` token to

[6] `https://ruder.io/nlp-imagenet/`

[7] `https://huggingface.co/docs/transformers`

[8] The actual training setup is slightly more complicated but conceptually similar. Notice, this is similar to one of the models in Programming Assignment 1!

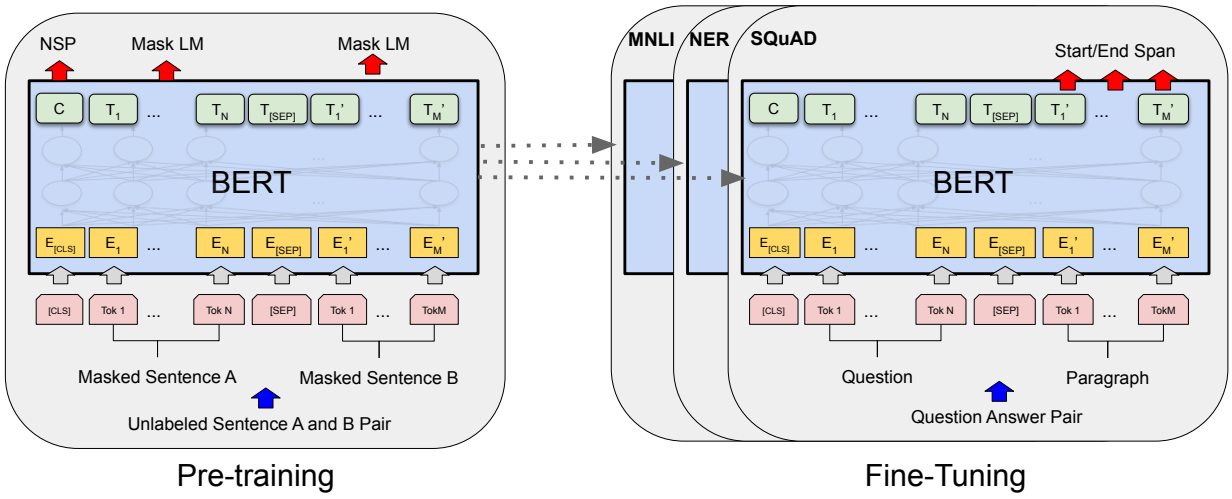

Figure 5: Overall pretraining and fine-tuning for BERT. Reproduced from BERT paper [2]

predict whether sentence A and sentence B are consecutive sentences in the unlabeled dataset. See Figure 5 for the conceptual picture of BERT pretraining and fine-tuning.

Once pretrained, we can fine-tune BERT on a downstream task of interest, such as sentiment analysis or question-answering, benefiting from its learned contextual representations. Typically, this is done by adding a simple classifier, which maps BERTs outputs to the class labels for our downstream task. Often, this classifier is a single linear layer + softmax. We can choose to train only the parameters of the classifier, or we can fine-tune both the classifier and BERT model jointly. Because BERT has been pretrained on a large amount of data, we can get good performance by fine-tuning for a few epochs with only a small amount of labelled data.

In this assignment, you will **fine-tune BERT** on a **single sentence classification task**. Figure 6 illustrates the basic setup 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 as input to a simple classifier, which will learn to predict the probabilities for each of the possible output classes of our task. We will use the pretrained weights of MathBERT, which uses the same architecture as BERT, but has been pretrained on a large mathematical corpus, which more closely matches our task data (see below).

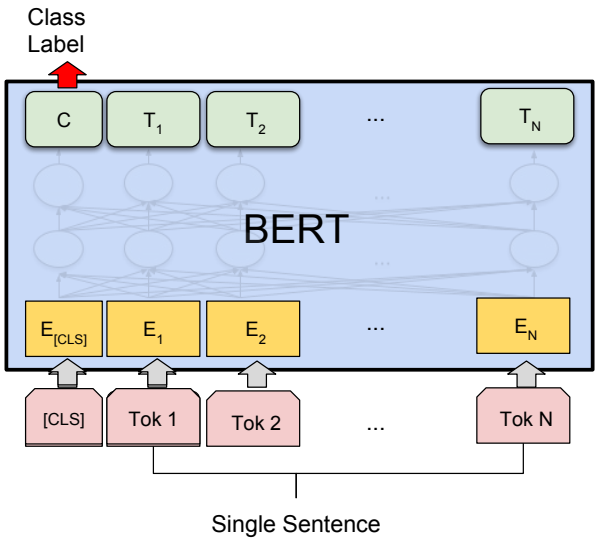

Figure 6: Fine-tuning BERT for single sentence classification by adding a layer on top of the contextualized `[CLS]` token representation. Reproduced from BERT paper [2]

Verbal Arithmetic Dataset

The verbal arithmetic dataset contains pairs of input sentences and labels. The input sentences express a simple addition or subtraction. Each input is labelled as 0, 1, or 2 if it evaluates to *negative*, *zero*, or *positive*, respectively. There are 640 examples in the train set and 160 in the test set. All inputs have only **three tokens** similar to the examples shown below:

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

Questions:

1. [1pt] *Add a classifier to BERT.* Open the notebook `https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/bert.ipynb` and complete Question 1 by filling in the missing lines of code in `BertForSentenceClassification`.

2. [0pt] *Fine-tune BERT.* Open the notebook and run the cells under Question 2 to fine-tune the BERT model on the verbal arithmetic dataset. If question 1 was completed correctly, the model should train, and a plot of train loss and validation accuracy will be displayed.

3. [0.5pt] *Freezing the pretrained weights.* Open the notebook and run the cells under Question 3 to fine-tune only the classifiers weights, leaving BERTs weights frozen. After training, answer the following questions (no more than **four** sentences total)

    - Compared to fine-tuning (see Question 2), what is the effect on train time when BERTs weights are frozen? Why? (1-2 sentences)
    - Compared to fine-tuning (see Question 2), what is the effect on performance (i.e. validation accuracy) when BERTs weights are frozen? Why? (1-2 sentences)

4. [0.5pt] *Effect of pretraining data.* Open the notebook and run the cells under Question 4 in order to repeat the fine-tuning process using the pretrained weights of BERTweet. After training, answer the following questions (no more than **three** sentences total).

    - Compared to fine-tuning BERT with the pretrained weights from MathBERT (see Question 2), what is the effect on performance (i.e. validation accuracy) when we fine-tune BERT with the pretrained weights from BERTweet? Why might this be the case? (2-3 sentences)

5. [0pt] *Inspect models predictions.* Open the notebook and run the cells under Question 5. We have provided a function that allows you to inspect a models predictions for a given input. Can you find examples where one model clearly outperforms the others? Can you find examples where all models perform poorly?

Deliverables:

- The completed `BertForSentenceClassification`. Either the code or a screenshot of the code. Make sure both the `__init__` and `forward` methods are clearly visible. [1pt]
- Answer to question 3. Your answer should not exceed **4 sentences**. [0.5pt]
- Answer to question 4. Your answer should not exceed **3 sentences**. [0.5pt]

Part 4: Connecting Text and Images with CLIP [1pt]

Throughout this course, we have seen powerful image models and expressive language models. In this section, we will connect the two modalities by exploring CLIP, a model trained to predict an image's caption to learn better image representations.

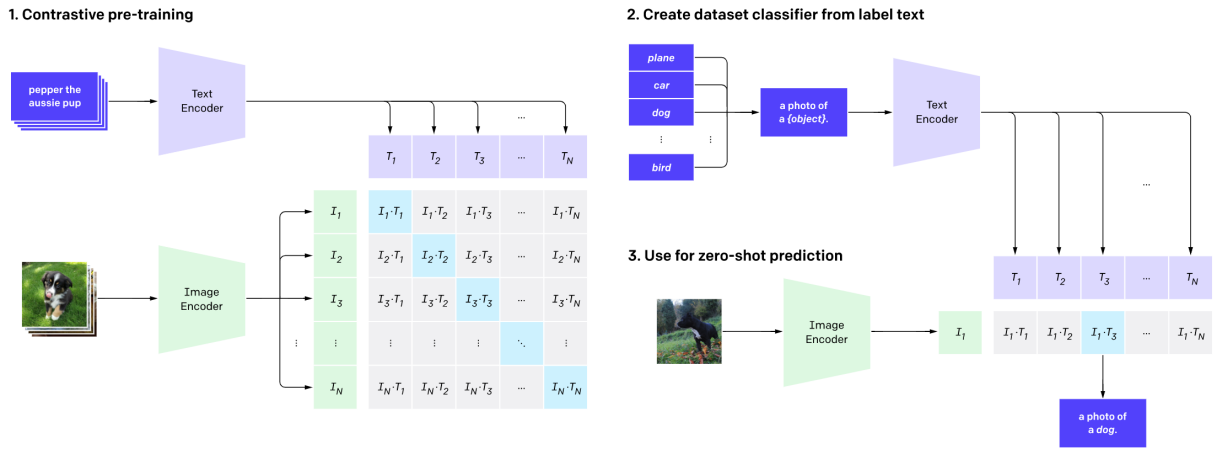

Figure 7: 1. Contrastive pre-training task that predicts the caption that corresponds to an image out of many possible captions. 2. At test time, each class is converted to a caption. This is used with 3. as a zero-shot classifier for a new image that predicts the best (image, caption) pair. Figure taken from [6]

Background for CLIP:

The motivation behind **C**ontrastive **L**anguage-**I**mage **P**re-training (CLIP) [5] was to leverage information from natural language to improve zero-shot classification of images. The model is pre-trained on 400 million (image, caption) pairs collected from the internet on the following task: given the image, predict which caption was paired with it out of 32,768 randomly sampled captions (Figure 7). This is done by first computing the feature embedding of the image and feature embeddings of possible captions. The cosine similarity of the embeddings is computed and converted into a probability distribution. The outcome is that the network learns many visual concepts and associates them with a name.

At test time, the model is turned into a zero-shot classifier: all possible classes are converted to a caption such as "a photo of a (class)" and CLIP estimates the best (image, caption) pair for a new image. Overall, CLIP offers many significant advantages: it does not require expensive

hand-labelling while achieving competitive results and offers greater flexibility and generalizability over existing ImageNet models.

Questions:

1. [0pt] *Interacting with CLIP*. Open the notebook `https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/clip.ipynb`. Read through Section I and run the code cells to get familiar with CLIP.

2. [1pt] *Prompting CLIP*. Complete Section II. Come up with a caption that will "prompt" CLIP to select the following target image:

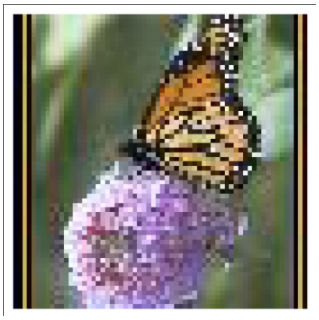

Figure 8: Image that should be selected by CLIP.

Comment on the process of finding the caption: was it easy, or were there any difficulties? (no more than **one** sentence)

Deliverables:

- The caption you wrote that causes CLIP to select the image in Figure 8, as well as a brief (1 sentence) comment on the search process. [1pt]

What you need to submit

- The completed notebook files: `nmt.ipynb`, `bert.ipynb` and `clip.ipynb`.
- A PDF document titled `a3-writeup.pdf` containing your answers to the conceptual questions.

References

[1] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[2] 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.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Dat Quoc Nguyen, Thanh Vu, and Anh Tuan Nguyen. Bertweet: A pre-trained language model for english tweets. *arXiv preprint arXiv:2005.10200*, 2020.

[5] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[6] Alec Radford, Ilya Sutskever, Jong Wook Kim, Gretchen Krueger, and Sandhini Agarwal. Clip: Connecting text and images, Jan 2021.

[7] Jia Tracy Shen, Michiharu Yamashita, Ethan Prihar, Neil Heffernan, Xintao Wu, Ben Graff, and Dongwon Lee. Mathbert: A pre-trained language model for general nlp tasks in mathematics education. *arXiv preprint arXiv:2106.07340*, 2021.

[8] 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.

[9] 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.

[10] 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.