Programming Assignment 3: Attention-Based Neural Machine Translation

Due Date: Monday, Mar. 16th, at 11:59pm
Based on an assignment by Lisa Zhang

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.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.

[1] `https://markus.teach.cs.toronto.edu/csc413-2020-01`
[2] `https://csc413-2020.github.io/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 *gated recurrent neural networks* 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. The dataset is composed of 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 6387 unique (English, Pig-Latin) pairs in total; the first few examples are:

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

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: Gated Recurrent Unit (GRU), 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: BERT fine-tuning

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.

[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).

Part 1: Gated Recurrent Unit (GRU) [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 [2], composed of two RNNs, as follows:

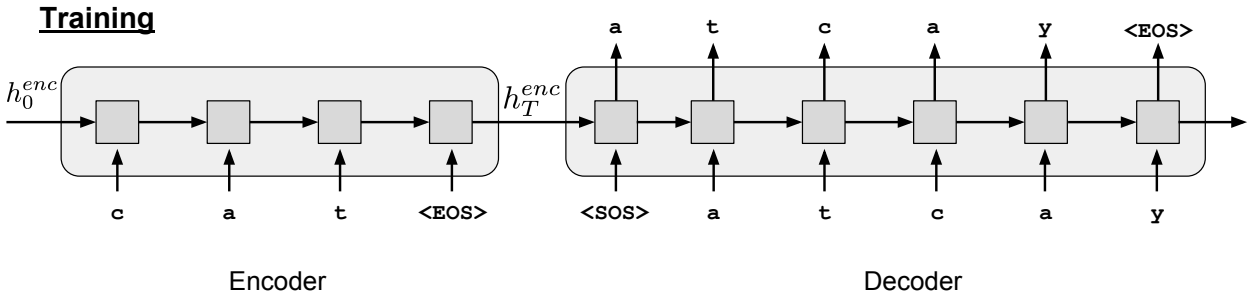

Figure 1: Training the NMT encoder-decoder architecture.

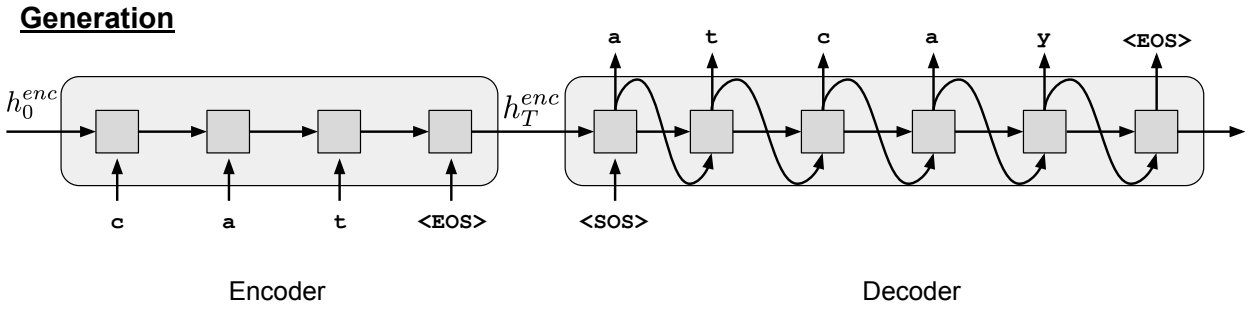

Figure 2: Generating text with the NMT encoder-decoder architecture.

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; in our model, we learn a 29×10 embedding matrix, where each of the 29 characters in the vocabulary is assigned a 10-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.

Lets begin with implementing common encoder models: the Gated Recurrent Unit and the transformer encoder.

Open `https://colab.research.google.com/drive/1rHYoCXb96INsxCSc1G4OmZhisnuabsIH` on Colab and answer the following questions.

1. [1pt] The forward pass of a Gated Recurrent Unit is defined by the following equations:

$$r_t = \sigma(W_{ir}x_t + W_{hr}h_{t-1} + b_r) \tag{1}$$
$$z_t = \sigma(W_{iz}x_t + W_{hz}h_{t-1} + b_z) \tag{2}$$
$$g_t = \tanh(W_{in}x_t + r_t \odot (W_{hn}h_{t-1} + b_g)) \tag{3}$$
$$h_t = (1 - z) \odot g_t + z \odot h_{t-1}, \tag{4}$$

   where $\odot$ is the element-wise multiplication. Although PyTorch has a GRU built in (`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, to implement the above equations. A template has been provided for the `forward` method.

2. [0pt] Run the cells including GRU-based encoder/decoder models.

3. [1pt] Train the RNN encoder/decoder model. We've provided implementations for recurrent encoder/decoder models using the GRU cell. (Make sure you have run all the relevant previous cells to load the training and utility functions.)

   By default, the script runs for 100 epochs. 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 to the directory `h20-bs64-rnn`:

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

   After the training is complete, we will now use this model to translate the words in the next notebook cell using `translate_sentence` function. Try a few of your own words by changing the variable `TEST_SENTENCE`. Identify two distinct failure modes and briefly describe them.

Deliverables

Create a section in your report called **Gated Recurrent Units**. Add the following in this section:

- A screenshot of your full `MyGRUCell` implementation. [1pt]
- The training/validation loss plots. [0pts]
- Your answer for the question in step 3. Make sure to add at least one input-output pair for each failure case you identify. Your answer should not exceed **three** sentences in total (excluding the failure cases you've identified.) [1pt]

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

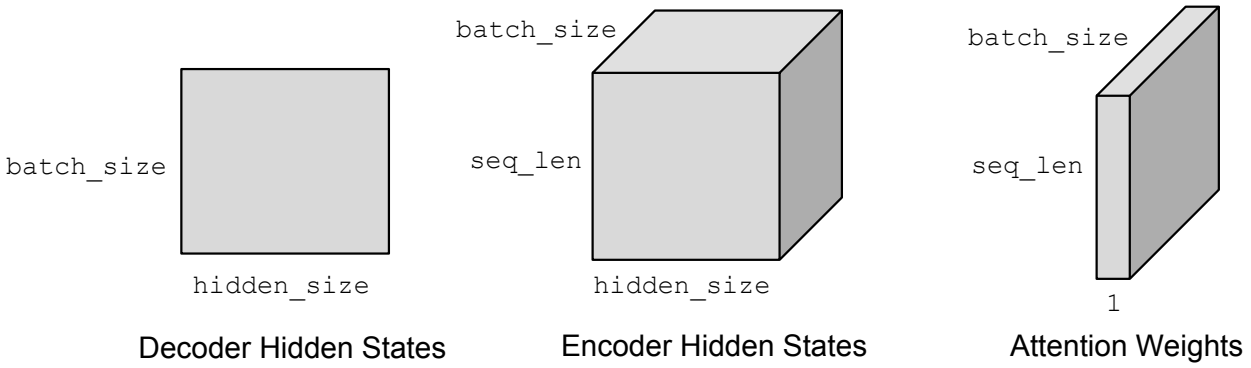

Figure 3: Dimensions of the inputs, Decoder Hidden States (*query*), Encoder Hidden States (*keys*/*values*) and the attention weights ($\alpha^{(t)}$).

For the `forward` pass, you 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. You must do this in a vectorized fashion. Since $f(Q_t, K_i)$ is a scalar, the resulting tensor of attention weights should have 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 should return 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.)

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

   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. [1pt] We will now apply the `AdditiveAttention` module to the RNN decoder. You are given a batch of decoder hidden states as the query, h_{t-1}^{dec}, for time $t-1$, which has dimension `batch_size x hidden_size`, and a batch of encoder hidden states as the keys and values, $h^{enc} = [h_1^{enc}, \ldots, h_i^{enc}, \ldots]$ (*annotations*), for each timestep in the input sequence, which has dimension `batch_size x seq_len x hidden_size`.

   $$Q_t \leftarrow h_{t-1}^{dec}, \quad K \leftarrow h^{enc}, \quad V \leftarrow h^{enc}$$

   We will use these as the inputs to the `self.attention` to obtain the context. The output context vector is concatenated with the input vector and passed into the decoder GRU cell at each time step, as shown in Figure 4.

   Fill in the `forward` method of the `RNNAttentionDecoder` class to implement the interface shown in Figure 4. There are three steps we will need to implement:

   (a) Get the embedding corresponding to the time step. (given)

   (b) Compute the context vector and the attention weights using `self.attentionl`. (implement)

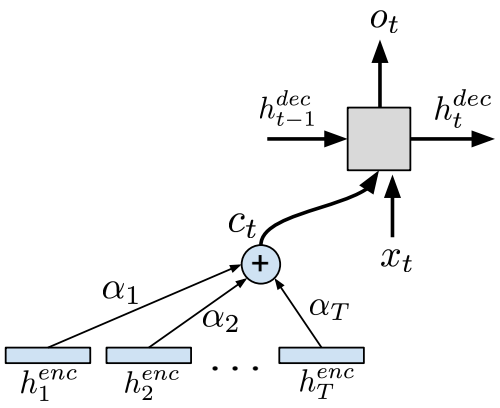

Figure 4: Computing a context vector with attention.

(c) Concatenate the context vector with the current decoder input. (implement)

(d) Feed the concatenation to the decoder GRU cell to obtain the new hidden state. (given)

3. [0pt] Now run the following cell 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.

4. [0pt] How does the training speed compare? Why?

Deliverables

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

- Three equations for question 1. [1pt]
- A screenshot of your `RNNAttentionDecoder` class implementation. [1pt]
- Training/validation plots you've obtained in this section. [0 pts]
- Answers to question 3. [0 pts]
- Answer to question 4.[0 pts]

Part 3: 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.

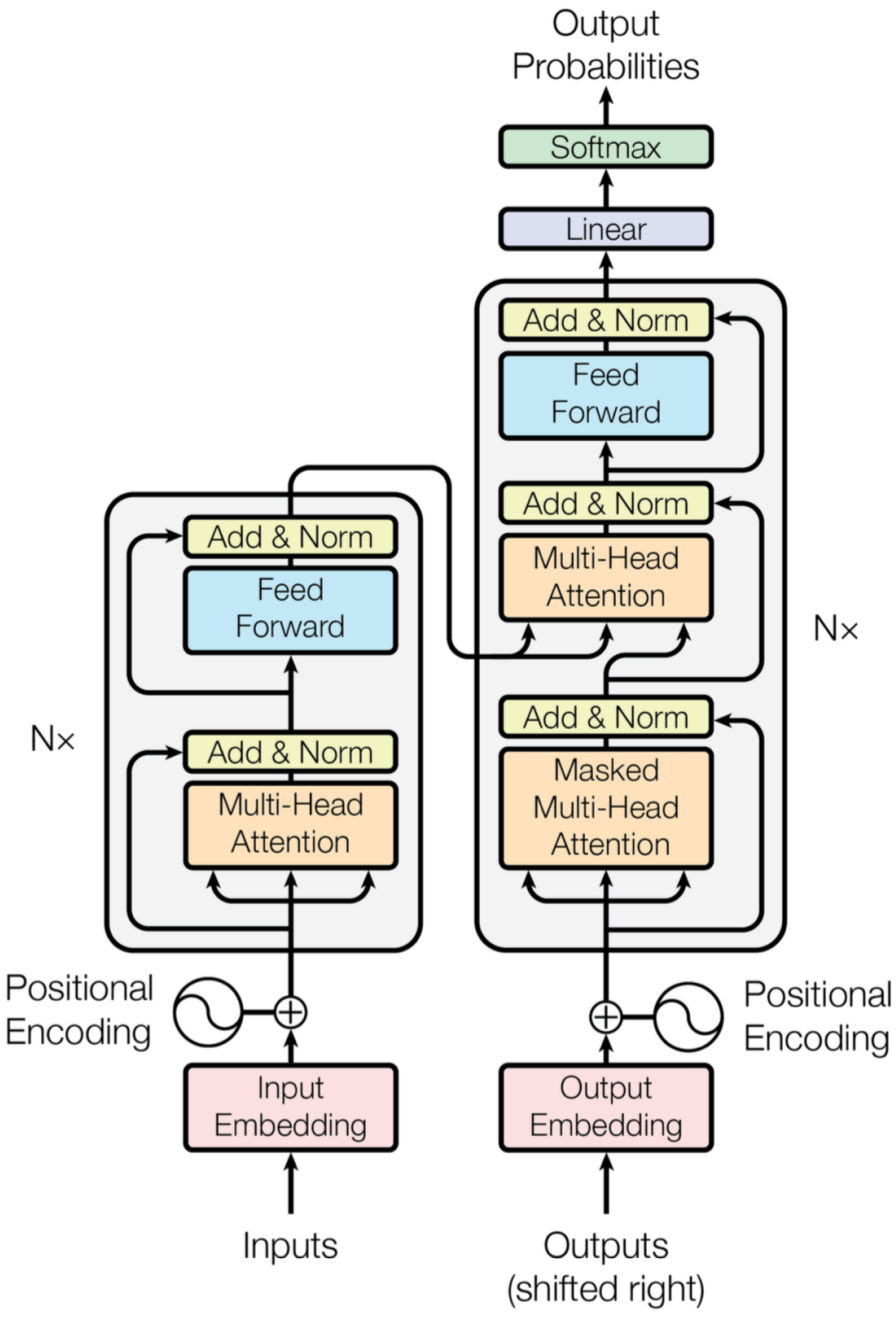

Figure 5: The transformer architecture. [3]

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

   The encoder looks like the left half of Figure 5. The encoder consists of three components (already provided):

   - Positional encoding: Without any additional modifications, self attention is permutation-equivariant. 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 [3]. That is we use sine and cosine functions:

     $$\text{PE}(\text{pos}, 2i) = \sin \frac{\text{pos}}{10000^{2i/d_{model}}} \tag{5}$$

     $$\text{PE}(\text{pos}, 2i+1) = \cos \frac{\text{pos}}{10000^{2i/d_{model}}} \tag{6}$$

     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.

   Now, complete the `forward` method of `TransformerEncoder`. Most of the code is given, except for two lines with ... in them. Complete these lines.

4. [0.5pt] The decoder, in addition to all the components the encoder has, also requires a `CausalScaledDotAttention` component. Take a look at Figure 5. 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.

   Now, complete the `forward` method of `TransformerDecoder`. Again, most of the code is given to you - fill in the two lines that have ....

5. [1pt] Now, train the language model with transformer based encoder/decoder. How do the translation results compare to the previous decoders? Write a short, qualitative analysis.

6. [1pt] 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`, `TransformerEncoder` and `TransformerDecoder` implementations. Highlight the lines you've added. [2pt]
- Training/validation plots you've generated. Your response to question 5. Your analysis should not exceed **three** sentences (excluding the failure cases you've identified). [1pt]
- Your response to question 6. Your response should not exceed **three** sentences. [1pt]
- Your response to question 7. [0pt]

Part 4: BERT for arithmetic sentiment analysis [2pt]

In this section, we will learn how to use a pre-trained BERT model 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 start by explaining what BERT is, and how we can add a classifier on top of the pre-trained BERT to perform sentiment analysis for verbal numerical expressions. Most code is given to you in the notebook `https://colab.research.google.com/drive/1QMGZsQ5u7JWuXiwvOhaH_OUd8Cn8E3aw` 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 BERT before starting to answer the questions. The *Hugging Face transformers* library, used in this tutorial, has more than 20k stars on github due to its ease of use, and will be very useful for your research or projects in the future.

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 [3] encoder architecture that has been pre-trained on a large dataset of unlabeled sentences from Wikipedia and BookCorpus [4]. 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 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". In BERT, however, we wish to predict 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[4]. For

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

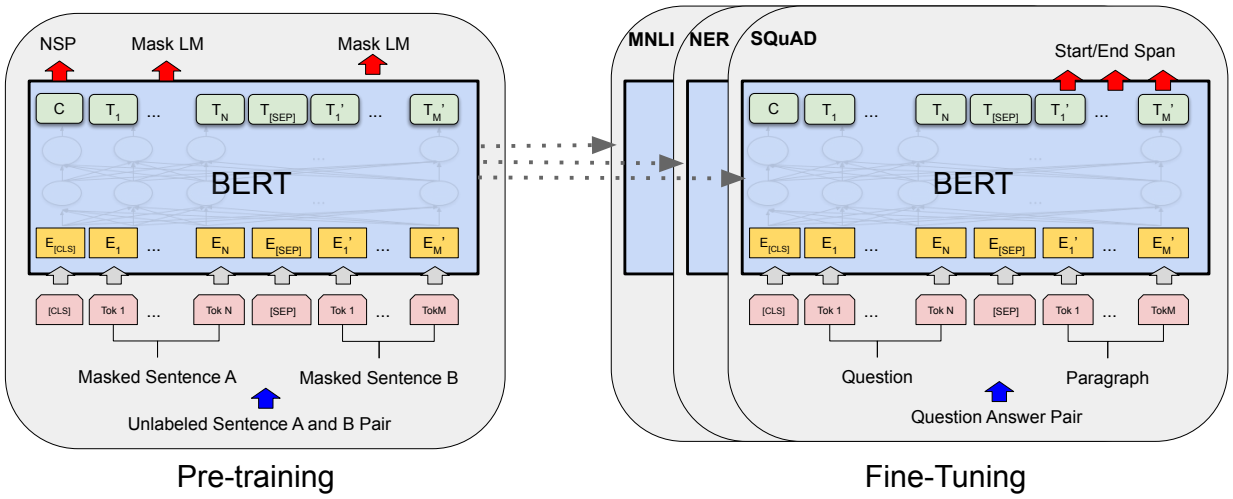

Figure 6: Overall pre-training and fine-tuning for BERT. Reproduced from BERT paper [1]

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.

In this assignment, you will be **fine-tuning BERT on a single sentence classification task** (see below about the dataset). Figure 7 illustrates the architecture for fine-tuning 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.

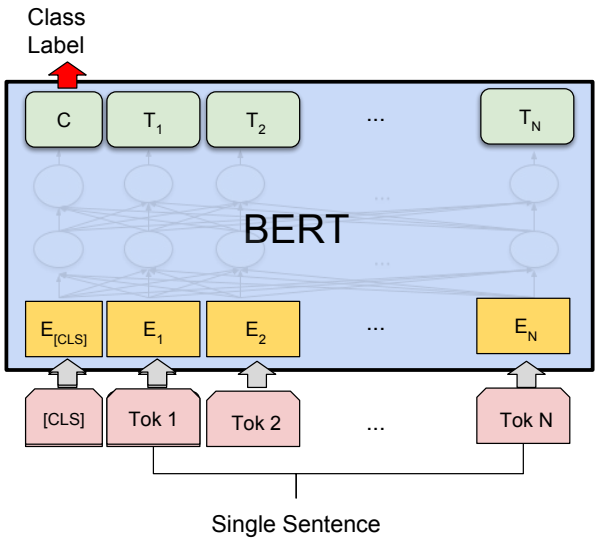

Figure 7: Fine-tuning BERT for single sentence classification by adding a layer on top of the contextualized `[CLS]` token representation. Reproduced from BERT paper [1]

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. [0pt] Classifier layer. Open the notebook `https://colab.research.google.com/drive/1QMGZsQ5u7JWuXiwvOhaH_OUd8Cn8E3aw`, we have provided two example BERT classes:

   `BertCSC413_Linear` and `BertCSC413_MLP_Example` that both add a classifier for classification.

   In this part, you need to make your own `BertCSC413_MLP class` by, for example, modifying the provided examples: change the number of layers; change the number of hidden neurons; or try a different activation.

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. [1pt] Try a few unseen examples of arithmetic questions using either *model_freeze_bert* or *model_finetune_bert* model, and find 10 interesting results. We will give full marks as long as you provide some comments for why you chose **some of the examples**. The interesting results can, for example, be both successful extrapolation/interpolation results or surprising failure cases. You can find some examples in our notebook.

4. [1pt] 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. You can assume the sequence length of all the data is 3 in this case.

Deliverables:

- Description of how your sentence classifier on top of BERT architecture is different from the one given. Your answer should be **one sentence**. [0pts]
- 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** [0pts]
- 10 inference results in question 5 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 [1pt]
- Explanation of what you did for the open question and some preliminary results. Your answer should not exceed **4 sentences**. [1pt]

What you need to submit

- Two code files: `nmt.ipynb`, `bert.ipynb`.

- A PDF document titled `a3-writeup.pdf` containing your answers to the conceptual questions, and the attention visualizations, with explanations.

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] 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.

[3] 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.

[4] 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.