Programming Assignment 3: Attention-Based Neural Machine Translation

Due Date: Mon, Nov. 29, at 2:00 pm

Submission: You must submit 2 files through Gradescope:

1. A PDF file containing your write-up, titled a3-writeup.pdf,
2. Your code files nmt.ipynb. Your write-up must be typed.

The programming assignments are individual work.

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.

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)

[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)

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:

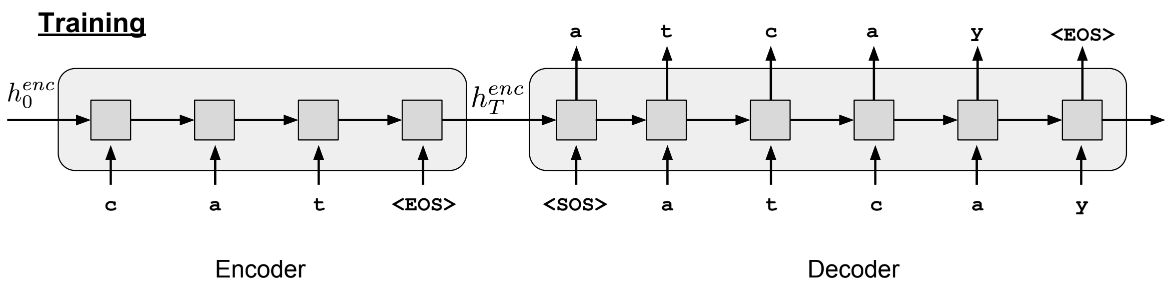

Figure 1: Training the NMT encoder-decoder architecture.

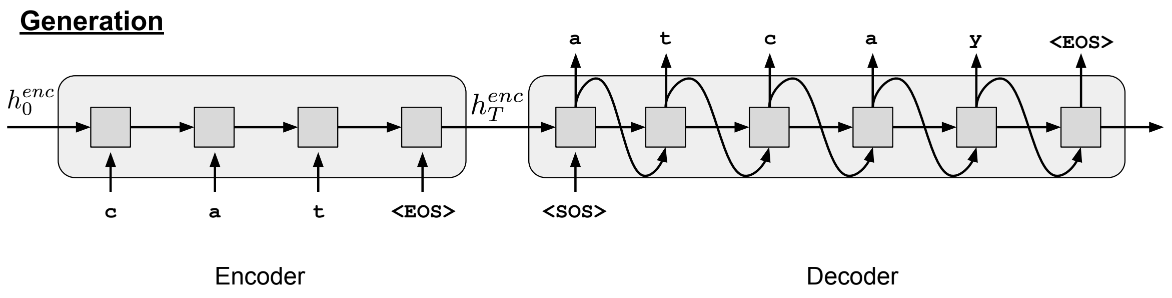

Figure 2: Generating text with the NMT encoder-decoder architecture.

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 your colab file and answer the following:

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

Text

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

   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. [redacted] 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. [redacted] 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 LSTM units and number of connections in this encoder model as a function of H, K, and D. For simplicity, you may ignore the bias units. You may also assume the input sequence has already been embedded, and ignore the embedding layer in your calculations.

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.
- 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.)
- Your answer for question 3.

Part 2: Additive Attention

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.

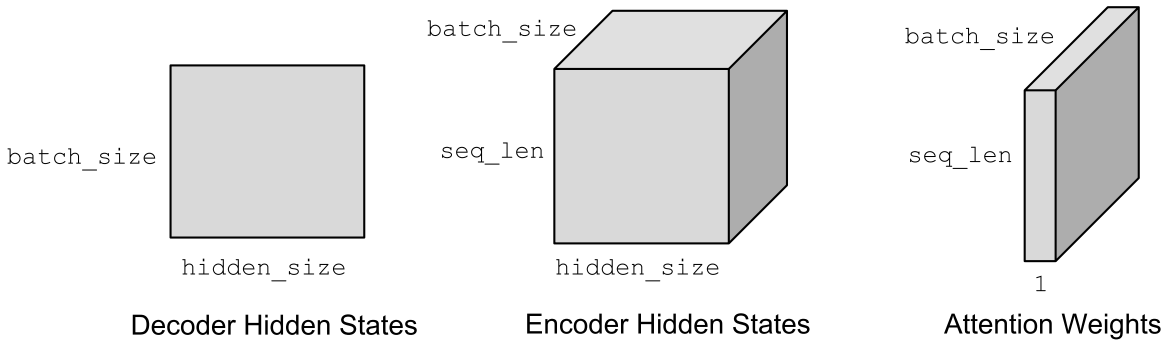

Figure 3: Dimensions of the inputs, Decoder Hidden States (*query*), Encoder Hidden States (*keys/values*) and the attention weights ($\alpha^{(t)}$).

For the `forward` pass, 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. [redacted] 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. [redacted] 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. [redacted] How does the training speed compare? Why?

4. [redacted] 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:

- Three equations for question 1.
- Training/validation plots you've obtained in this section.
- Answers to question 2.
- Answer to question 3.
- Answer to question 4.

Part 3: Scaled Dot Product Attention

1. [redacted] 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. [redacted] **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.

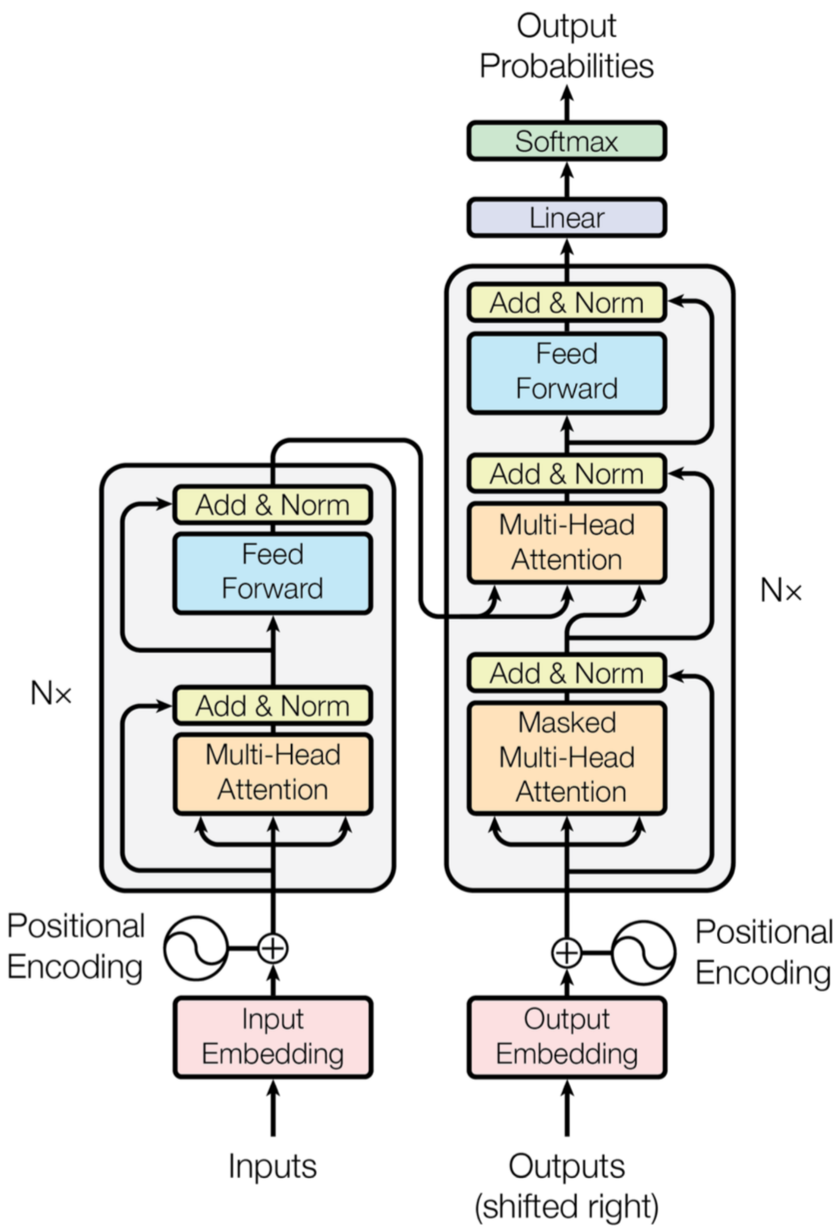

Figure 4: The transformer architecture. [5]

3. [redacted] 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. [redacted] 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. [redacted] 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. 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.
- Your answer to question 3.
- Your response to question 4. Your analysis should not exceed **three** sentences.
- The two loss curves plots output by the experimental setup in question 5, and the lowest validation loss for each run.
- Your response to the written component of question 5.
- Your response to question 6.