# Attention-Based Neural Machine Translation

## Introduction

In this assignment, you will train an attention-based neural machine translation model to translate words from English to Pig Latin.

### 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

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 *characterlevel* 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.

### 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.[1] 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.

## Assignment

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-toseq problems is the encoder-decoder model.

The encoder produces hidden states while reading the input sequence, $h_1^{enc}, \ldots, h_T^{enc}$, which can be viewed as *annotations* of the input; each encoder hidden state $h^{enc}_i$ captures information about the $i^{th}$ input token, along with some contextual information. In this assignment, the encoder is a gated recurrent unit (GRU), a type of recurrent neural network (RNN), and has been implemented for you.

Attention allows a model to look back over the input sequence, and focus on relevant input tokens when producing the corresponding output tokens.

[1] 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).

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.

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 $h^{enc}{}_i$. 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 a decoder hidden state and an encoder hidden state, $f(h^{dec}{}_t, h^{enc}{}_i)$, 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 implement scaled dot-product attention, which measures the similarity between the two hidden states.

We consider attention as a function whose inputs are triples of (queries, keys, values), denoted as (*Q,K,V* ).

## Scaled Dot-Product Attention (30 points)

In scaled dot-product attention, the function *f* is a dot product between the linearly transformed query and keys using weight matrices $W_q$ and $W_k$, where *d* is the dimension of the query and $W_v$ denotes the weight matrix that projects the value to produce the final context matrix $c_t$:

$$\tilde{\alpha}_i^{(t)} = f(Q_t, K_i) = \frac{(Q_t W_q)(K_i W_k)^\top}{\sqrt{d}}$$

$$\alpha_i^{(t)} = \text{softmax}(\tilde{\alpha}_i^{(t)}),$$

$$c_t = \alpha_i^{(t)}(V_i W_v)$$

**Implement the scaled dot-product attention mechanism.** Fill in the forward method of the ScaledDotAttention class. Use the PyTorch function torch.bmm 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, as well as for the product between the normalized attention weights and the batched values.

## Causal Scaled Dot-Product Attention (10 points)

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.triu handy for this part.

## Transformer Decoder (30 points)

We will now use ScaledDotAttention and CausalScaledDotAttention as the building blocks for a simplified transformer decoder. You are given a batch of decoder input embeddings, $x^{dec}$ across all time steps, which has dimension batch size x decoder seq len x hidden size. and a batch of encoder hidden states, $h^{enc} = [h_1^{enc}, \ldots, h_i^{enc}, \ldots]$ (*annotations*), for each
time step in the input sequence, which has dimension batch size x encoder seq len x hidden size.

The transformer solves the translation problem using layers of attention modules. In each layer, we first apply the CausalScaledDotAttention selfattention to the decoder inputs followed by the ScaledDotAttention attention module to the encoder annotations. The output of the attention layers are fed into an hidden layer using ReLU activation. The final output of the last transformer layer is passed to self.out to compute the word prediction. To improve the optimization, we add residual connections between the attention layers and ReLU layers (in which the outputs of two different layers are added together). The simple transformer architecture is shown in Figure 1.

Fill in the forward method of the TransformerDecoder class, to implement the interface shown in Figure 1.

It took about **2 minutes** (using an AMD Ryzen 7 5800U) to train the model. Your mileage may vary, depending on your computer. If you have a CUDAcompatible GPU, you may choose to train the model on it by setting 'cuda':True in args dict. (Since the model is relatively small, you are unlikely to see much, if any, speedup on GPU.) Because there is some randomness when training, your final training and validation losses may vary; that being said, the training and validation loss curves should look somewhat similar to Figure 2.

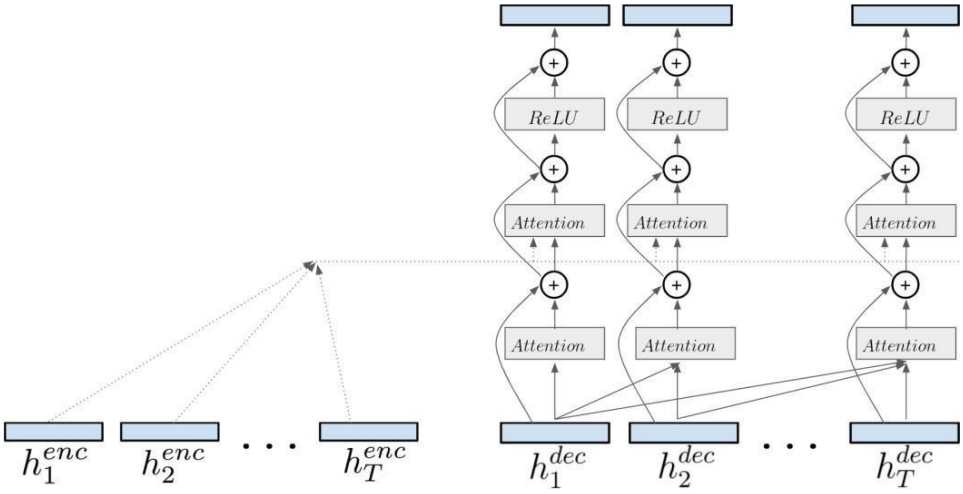


Figure 1: Computing the output of a transformer layer.

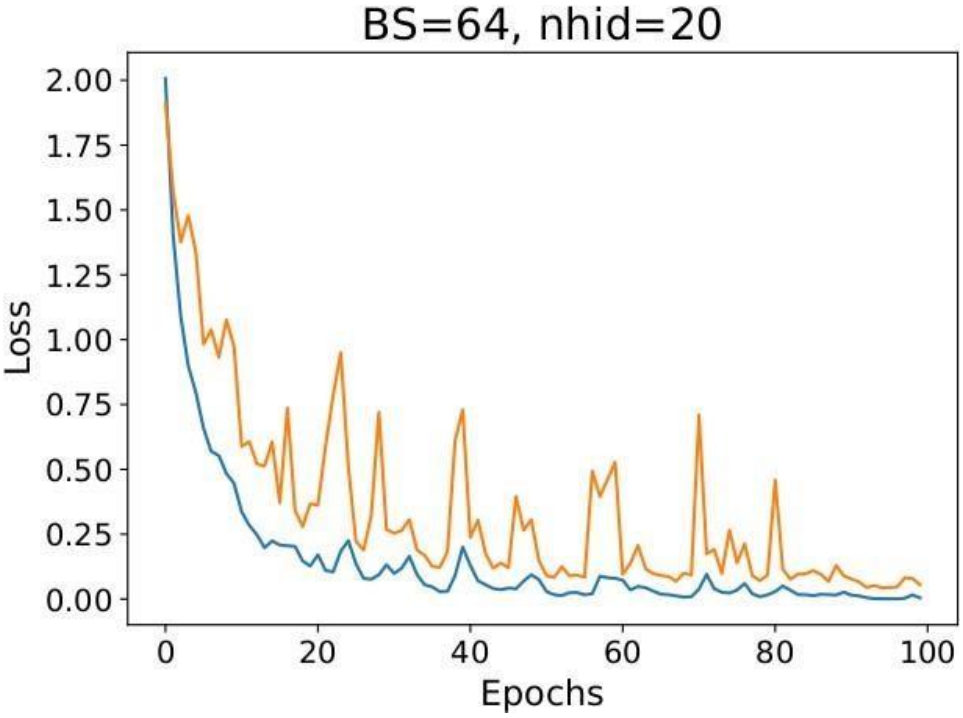


Figure 2: Sample training (blue) and validation (orange) loss curves.

### Attention Visualizations (30 points)

One of the benefits of using attention is that it allows us to gain insight into the inner workings of the model. By visualizing the attention weights generated for the input tokens in each decoder step, we can see where the model focuses while producing each output token. In this part of the assignment, you will visualize the attention learned by your model, and try to find interesting success and failure modes that illustrate its behavior.

You can load the model you trained from the previous section (by setting `TRAIN_MODEL = False`) and use it to translate a given set of words: it prints the translations and displays heatmaps to show how attention is used at each step.

Visualize different attention models using your own words by modifying `TEST_WORD_ATTN`. Since the model operates at the character level, the input doesn't even have to be a real word in the dictionary. You can be creative! You should examine the generated attention maps. Try to find failure cases, and hypothesize about why they occur. Some interesting classes of words you may want to try are:

> Words that begin with a single consonant (e.g., *cake*).
>
> Words that begin with two or more consonants (e.g., *drink*).
>
> Words that have unusual/rare letter combinations (e.g., *aardvark*).
>
> Compound words consisting of two words separated by a dash (e.g., *well-mannered*). These are the hardest class of words present in the training data, because they are long, and because the rules of Pig Latin dictate that each part of the word (e.g., *well* and *mannered*) must be translated separately, and stuck back together with a dash: *ellwayannerdmay*.
>
> Made-up words or toy examples to show a particular behavior.

## Write-up

You should also prepare a short write-up that includes attention maps for both success and failure cases, along with your hypotheses about why the model succeeds or fails.