Instructor: Vishnu Boddeti
CSE 891-001: Deep Learning
Programming Homework 3
December, 07

Instructions:

- **Submission:** Only homeworks uploaded to Google Classroom will be graded. Submit solutions in PDF format.
- **Integrity and Collaboration:** You are expected to work on the homeworks by yourself. You are not permitted to discuss them with anyone except the instructor. The homework that you hand in should be entirely your own work. You may be asked to demonstrate how you got any results that you report.
- **Clarifications:** If you have any question, please look at Google Classroom first. Other students may have encountered the same problem, and is solved already. If not, post your question there. We will respond as soon as possible.

Introduction:
In this assignment, you will train two attention-based neural machine translation (NMT) model to translate words from English to Pig-Latin. Along the way, you will gain experience with several important concepts in NMT, including *gated recurrent neural networks* and *attention*.

Pig Latin: It 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. For instance, `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`.

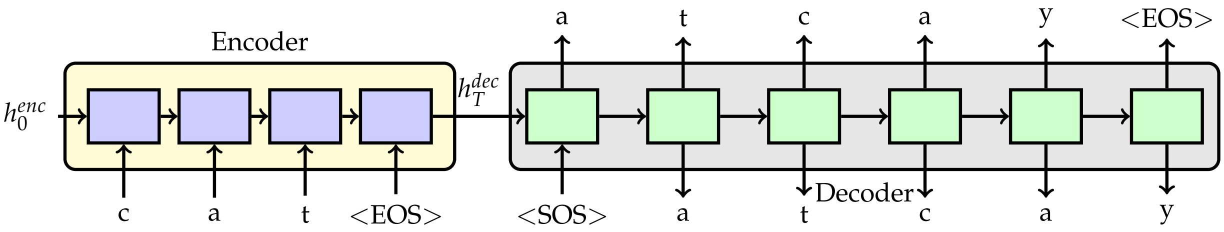
To translate a whole sentence from English to Pig-Latin, we simply apply these rules to each word independently: `i went shopping` → `iway entway oppingshay`

We would like to train 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. Since English and Pig-Latin are structurally very similar, 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, start of sentence <SOS> and end of sentence <EOS>. 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.

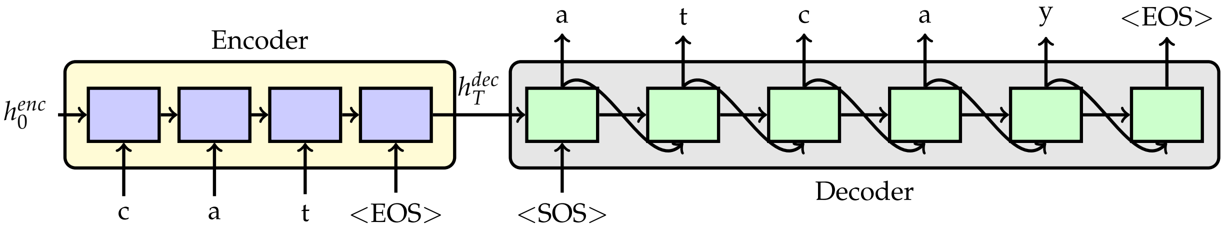
Encoder-Decoder NMT Setup: Translation is a sequence-to-sequence problem: in our case, both the input and output are sequences of characters. A common architecture used for seq-to-seq problems is the encoder-decoder model [1], composed of two RNNs, as follows:



The encoder RNN compresses the input sequence into a fixed-length vector, represented by the final hidden state 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 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 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, as shown in the previous question. 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 below.



1 Gated Recurrent Unit (2pts): The forward pass of a Gated Recurrent Unit [2] is defined by the following equations:

$$\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) && (1)\\
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) && (2)\\
g_t &= tanh(W_{xg}x_t + r_t \odot (W_{hg}h_{t-1} + b_g)) && (3)\\
h_t &= (1 - z_t) \odot g_t + z_t \odot h_{t-1} && (4)
\end{aligned}$$

1. (1pt) Although PyTorch has a GRU built in (`nn.GRUCell`), we will implement our own GRU cell from scratch, to better understand how it works. Fill in the `__init__` and `forward` methods of the `MyGRUCell` class in `models.py`, to implement the above equations. A template has been provided for the forward method.

2. (1pt) Train the RNN encoder/decoder model. We provided implementations for recurrent encoder/decoder models using the GRU cell. 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 parameters, 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 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:

- A python file with code: `models.py`
- Create a report with a section called **Gated Recurrent Units**. Include the training/validation loss plots.
- Make sure to add at least one input-output pair for each failure case you identify.

2 Additive Attention (3pts): 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 α_i for each of the encoder hidden states h_i^{enc}. The weights are defined such that $0 \leq \alpha_i \leq 1$ and $\sum_i \alpha_i = 1$. α_i 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 different choices for the possible function f. In this assignment we will explore 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.

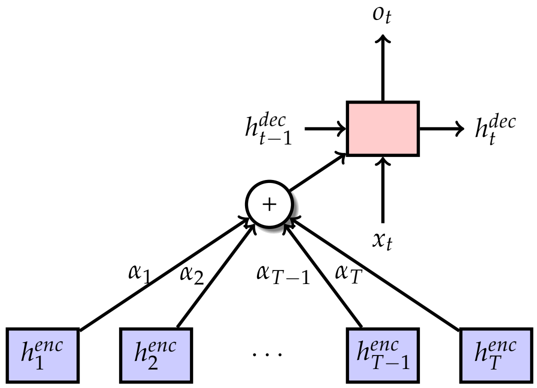
For the `forward` pass, you are given a batch of query of the current time step, which has dimension `batch_size x hidden_size`, and a batch of keys and values for each time step of the input sequence, both have dimension `batch_size x seq_len x hidden_size`. The goal is to obtain the context vector. We first compute the function $f(Qt, K)$ 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 $f(Qt, Ki)$ is a scalar, the resulting tensor of attention weights should have dimension `batch_size x seq_len x 1`. 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`.

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`.

$$Q_t \leftarrow h_{t-1}^{dec} \quad K \leftarrow h^{enc} \quad V \leftarrow h^{enc} \tag{5}$$

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 the figure below. Fill in the forward method of the `RNNAttentionDecoder` class to implement the interface shown in the figure below. You will need to implement the following steps :

- Get the embedding corresponding to the time step. (provided)
- Compute the context vector and the attention weights using `self.attention`. (implement)
- Concatenate the context vector with the current decoder input. (implement)
- Feed the concatenation to the decoder GRU cell to obtain the new hidden state. (provided)



- Train the model with attention by running the following command:

  ```
  python main.py
  ```

  By default, the script runs for 100 epochs, which should be enough to get good results; this takes approximately 24 minutes on my desktop.

  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 fixed sentence is stored in the variable `TEST_SENTENCE` in `main.py`. You can change this variable to see how translation improves for your own sentence.

  The script also saves several items to the directory `checkpoints/h20-bs64-rnn_attention`:

  - The training and validation loss curves.
  - The best encoder and decoder model parameters, based on the validation loss.

Deliverables: Create a section called Additive Attention and include the following in this section:

- A python file with code: `models.py`
- Create a section called **Additive Attention**. Include the training/validation loss plots.
- Make sure to add at least one input-output pair for each failure case you identify.

3 Scaled Dot Product Attention (3pts): In lecture, we introduced 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 $\boldsymbol{W}_q$ and $\boldsymbol{W}_k$:

$$\begin{aligned} \tilde{\alpha}_i^{(t)} &= f(\boldsymbol{Q}_t, \boldsymbol{K}_i) = \frac{(\boldsymbol{W}_q\boldsymbol{Q}_t)^T(\boldsymbol{W}_k\boldsymbol{K}_i)}{\sqrt{d}} & (6) \\ \alpha_i^{(t)} &= softmax(\tilde{\alpha}^{(t)})_i & (7) \\ \boldsymbol{c}_t &= \sum_{i=1}^{T} \alpha_i^{(t)} \boldsymbol{W}_v \boldsymbol{v}_i & (8) \end{aligned}$$

where, d is the dimension of the query and $\boldsymbol{W}_v$ is the weight matrix corresponding to the values $\boldsymbol{v}_i$.

1. Fill in the forward methods of the `ScaledDotAttention` class. Use the `torch.bmm` command 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`, `cat`, `expand_as`, `view`, `bmm`. Your forward pass needs to work with both 2D query tensor `batch_size x hidden_size` and 3D query tensor `batch_size x k x hidden_size`.

2. 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.

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

     $$\text{pth word embedding} = \text{input embedding} + \text{positional encoding(p)} \tag{9}$$

     We follow the same positional encoding methodology described in [3]. That is we use sine and cosine functions:

     $$\begin{aligned} PE(pos, 2i) &= \sin \frac{pos}{10000^{2i/d_{model}}} & (10) \\ PE(pos, 2i+1) &= \cos \frac{pos}{10000^{2i/d_{model}}} & (11) \\ & & (12) \end{aligned}$$

   - A `ScaledDotAttention` operation
   - An MLP

   Now, complete the forward method of TransformerEncoder. Most of the code is provided, except for a few lines with . . . in them. Complete these lines.

4. The decoder, in addition to all the components the encoder has, also requires a `CausalScaledDotAttention` component. 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 `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 lines that have . . . .

5. 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. 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. 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. Include the following:

- Training/validation plots you generated. Your response to question 5.
- Your response to question 6.
- Your response to question 7.

4 Visualizing Attention (2pts): 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.

The script `visualize_attention.py` loads a pre-trained model and uses it to translate a given set of words: it prints the translations and saves heatmaps to show how attention is used. To call this script, you need to pass in the path to a checkpoint folder, as follows:

```
python visualize_attention.py --load checkpoints/h20-bs64-rnn_attention
```

```
python visualize_attention.py --load checkpoints/h20-bs64-transformer
```

The `visualize_attention.py` script produces visualizations for the strings in the words list, found at the top of the script. The visualizations are saved as PDF files in the same directory as the loaded model checkpoint, so they will be in `checkpoints/h20-bs64-rnn_attention` or `checkpoints/h20-bs64-transformer`. Add your own strings to words list in `visualize_attention.py` and run the script as shown above. 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 as creative as you would like to !! After running the script, 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:

1. Words that begin with a single consonant (e.g., fake).
2. Words that begin with two or more consonants (e.g., brink).
3. Words that have unusual/rare letter combinations (e.g., aardvark).
4. Compound words consisting of two words separated by a dash (e.g., long-term). 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: onglay-ermtay.
5. Made-up words or toy examples to show a particular behavior.

Include attention maps for both success and failure cases in your writeup, along with your hypothesis about why the model succeeds or fails.

Submission: Here is everything you need to submit.

- A python file with code: `models.py`
- A PDF document titled `programming-assignment-3-msunetid.pdf` containing your answers to the conceptual questions and the attention visualizations with explanations.

References

[1] 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.

[2] 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.

[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.