Programming Assignment 3: Attention-Based Neural Machine Translation

Deadline: March 23, 2018, at 11:59pm
TAs: Matthew MacKay (`csc321staff@cs.toronto.edu`)
Assignment by Paul Vicol

Submission: You must submit two files through MarkUs[1]: a PDF file containing your writeup, titled `a3-writeup.pdf`, and your completed code file `models.py`. Your writeup must be typeset using LaTeX.

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 were unable to run the experiments, please discuss what outcomes you might hypothetically expect from the experiments. 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 an attention-based neural machine translation model to translate words from English to Pig-Latin. Along the way, you'll gain experience with several important concepts in NMT, including *attention* and *teacher forcing*.

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 *character-level* recurrent neural networks for our model.

[1] https://markus.teach.cs.toronto.edu/csc321-2018-01
[2] http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/syllabus.pdf

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. [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.

Part 1: Encoder-Decoder Models and Capacity [1 mark]

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:

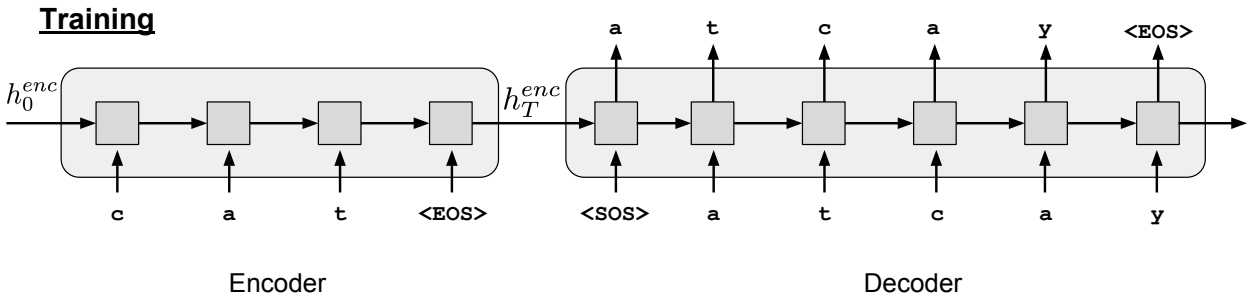

Figure 1: Training the NMT encoder-decoder architecture.

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

[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).

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.

Conceptual Questions

1. How do you think this architecture will perform on long sequences, and why? Consider the amount of information the decoder gets to see about the input sequence.

2. In the code folder, you will find a pre-trained model of the above architecture, using a hidden state of size 10. This model was trained to convergence. The script `translate_no_attn.py` uses this pre-trained model to translate words given in the list `words`. Run this script by calling:

   ```
   python translate_no_attn.py
   ```

   How do the results look, qualitatively? Does the model do better for certain types of words than others? Add a few of your own words to the `words` list at the top of the script, and run it again. Which failure modes can you identify?

Part 2: Teacher-Forcing [1 mark]

Teacher forcing works by using the actual or expected output from the training dataset at the current time step y(t) as input in the next time step X(t+1), rather than the output generated by the network.

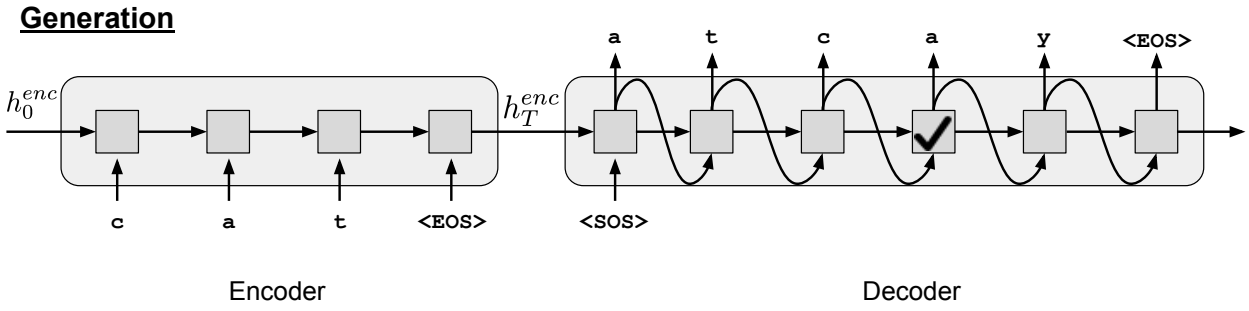

Figure 2: Generating text with the NMT encoder-decoder architecture.

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, as 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.

Questions

1. What problem may arise when training with teacher forcing? Consider the differences that arise when we switch from training to testing.

2. Can you think of any way to address this issue? Read the abstract and introduction of the paper "*Scheduled sampling for sequence prediction with recurrent neural networks*" [1], and answer this question in your own words.

Teacher-Forcing Ratio (Optional)

In the starter code, teacher-forcing is used 50% of the time, and the model's own predictions are used 50% of the time when training (see [1]). If you want to observe the effects of using teacher-forcing more or less of the time, you can provide your own *teacher-forcing ratio* to train the model; for example, `python attention_nmt.py --teacher_forcing_ratio=1` trains purely with teacher-forcing. This is optional, and not required for this assignment.

Part 3: Gated Recurrent Unit (GRU) [2 marks]

Throughout the rest of the assignment, you will implement an attention-based neural machine translation model, and finally train the model and examine the results.

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

z update gate
r reset gate

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

   Although PyTorch has a GRU built in (`nn.GRUCell`), we'll 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.

Part 4: Implementing Attention [4 marks]

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 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. One possible function f is the dot product, which measures the similarity between the two hidden states.

1. For our model, we will *learn* the function f, parameterized as a two-layer fully-connected network with a ReLU activation. This network produces unnormalized weights $\tilde{\alpha}_i^{(t)}$ as:

$$\tilde{\alpha}_i^{(t)} = f(h_{t-1}^{dec}, h_i^{enc}) = W_2(\max(0, W_1[h_{t-1}^{dec}; h_i^{enc}] + b_1)) + b_2$$

   Here, the notation $[h_{t-1}^{dec}; h_i^{enc}]$ denotes the concatenation of vectors h_{t-1}^{dec} and h_i^{enc}. Because the attention weights must be normalized, we need to apply the softmax function over the output of the two-layer network: $\alpha_i^{(t)} = \text{softmax}(\tilde{\alpha}^{(t)})_i$.

Implement this two-layer attention mechanism. Fill in the `__init__` and `forward` methods of the `Attention` class in `models.py`. Use the PyTorch `nn.Sequential` class to define the attention network, and use the `self.softmax` function in the forward pass of the `Attention` class to normalize the weights.

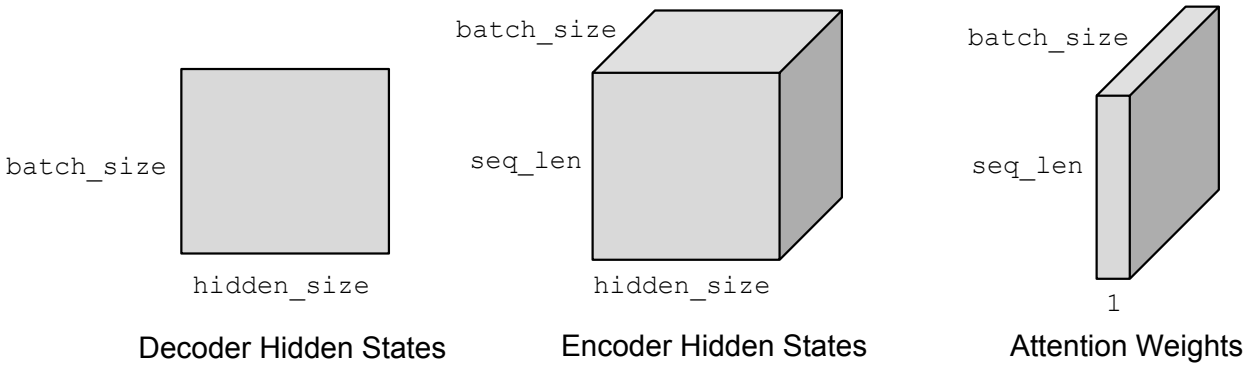

Figure 3: Dimensions of the input and output tensors of the Attention module.

For the `forward` pass, you will need to do some *reshaping* of tensors. You are given a batch of decoder hidden states for time $t-1$, which has dimension `batch_size x hidden_size`, and a batch of encoder hidden states (*annotations*) for each timestep in the input sequence, which has dimension `batch_size x seq_len x hidden_size`. The goal is to compute the function $f(h_{t-1}^{dec}, h_i^{enc})$ for each decoder hidden state in the batch and *all* corresponding encoder hidden states h_i^{enc}, where i ranges over `seq_len` different values. You must do this in a vectorized fashion. Since $f(h_{t-1}^{dec}, h_i^{enc})$ is a scalar, the resulting tensor of attention weights should have dimension `batch_size x seq_len x 1`. The input and output dimensions of the `Attention` module are visualized in Figure 3.

Depending on your implementation, you will need one or more of these functions (click to jump to the PyTorch documentation):

- squeeze
- unsqueeze
- expand_as
- cat
- view

The `self.attention_network` module takes as input a 2-dimensional tensor; you will need to `view` a 3D tensor as a 2D tensor to pass it through the attention network, and then `view` it as a 3D tensor again. We have provided a template for the `forward` method of the `Attention` class. You are free to use the template, or code it from scratch, as long as the output is correct.

2. Once we have the attention weights, a *context vector* c_t is computed as a linear combination of the encoder hidden states, with coefficients given by the weights:

$$c_t = \sum_{i=1}^{T} \alpha_i^{(t)} h_i^{enc}$$

input_size is size of x + size of h

This context vector is concatenated with the input vector and passed into the decoder GRU cell at each time step, as shown in Figure 4.

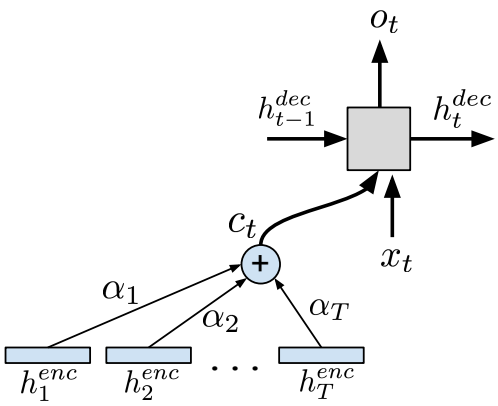

Figure 4: Computing a context vector with attention.

Fill in the `forward` method of the `AttentionDecoder` class, to implement the interface shown in Figure 4. You will need to:

(a) Compute the attention weights using `self.attention_network`

(b) Multiply these weights by the corresponding encoder hidden states and sum them to form the context vector.

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

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

(e) Compute the output using `self.out`.

3. Train the model with attention by running the following command:

```
python attention_nmt.py
```

By default, the script runs for 100 epochs, which should be enough to get good results; this takes approximately 24 minutes on the teaching lab machines. If necessary, you can train for fewer epochs by running `python attention_nmt.py --nepochs=50`, or you can exit training early with `Ctrl-C`.

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`, at the top of `attention_nmt.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/h10-bs16`:

- The best encoder and decoder model paramters, based on the validation loss.
- A plot of the training and validation losses.
- Attention maps generated during training for a fixed word, given by the variable `TEST_WORD_ATTN` in `attention_nmt.py`. These maps allow you to see how the attention improves over the course of training.

Part 5: Attention Visualizations [2 marks]

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 behaviour.

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 at each step. To call this script, you need to pass in the path to a checkpoint folder, as follows:

```
python visualize_attention.py --load checkpoints/h10-bs16
```

1. 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/h10-bs16`. 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 creative! 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:

   - 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: *ellway-annerdmay*.
   - Made-up words or toy examples to show a particular behaviour.

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

What you need to submit

- One code file: `models.py`.
- A PDF document titled `a3-writeup.pdf` containing your answers to the conceptual questions, and the attention visualizations, with explanations.

References

[1] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

[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.