XCS224N: Assignment #5

Welcome to the last assignment of the course! Assignment 5 is **coding-heavy** and the complexity of the code itself is similar to the complexity of the code you wrote in Assignment 4. What makes this assignment more difficult is that **we give you much less help** in writing and debugging that code. In particular, in this assignment:

- There is less scaffolding – instead of filling in functions, sometimes you will implement whole classes, without us telling you what the API should be.

- We do not tell you how many lines of code are needed to solve a problem.

- The local sanity-checks that we provide are almost all extremely basic (e.g. check output is correct type or shape). **More likely than not, the first time you write your code there will be bugs that the sanity check does not catch. It is up to you to check your own code, to maximize the chance that your model trains successfully.** You are advised to design and run your own tests, but you should be checking your code throughout.

- The final model (which you train at the end of Part 2) takes around **8-12** hours to train on the recommended Azure VM (time varies depending on your implementation, and when the training procedure hits the early stopping criterion). Keep this in mind when budgeting your time.

This assignment explores two key concepts – sub-word modeling and convolutional networks – and applies them to the NMT system we built in the previous assignment. The Assignment 4 NMT model can be thought of as four stages:

1. **Embedding layer:** Converts raw input text (for both the source and target sentences) to a sequence of dense word vectors via lookup.

2. **Encoder:** A RNN that encodes the source sentence as a sequence of encoder hidden states.

3. **Decoder:** A RNN that operates over the target sentence and attends to the encoder hidden states to produce a sequence of decoder hidden states.

4. **Output prediction layer:** A linear layer with softmax that produces a probability distribution for the next target word on each decoder timestep.

All four of these subparts model the NMT problem at a word level. In Section 1 of this assignment, we will replace it with a character-based convolutional encoder, and in Section 2 we will enhance it by adding a character-based LSTM decoder. This will hopefully improve our BLEU performance on the test set!

1. Character-based convolutional encoder for NMT (27 points)

In Assignment 4, we used a simple lookup method to get the representation of a word. If a word is not in our pre-defined vocabulary, then it is represented as the <UNK> token (which has its own embedding).
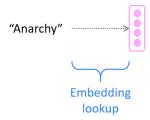

Figure 1: Lookup-based word embedding model from Assignment 4, which produces a word embedding of length e_{word}.

In this section, we will first describe a method based on Kim et al.'s work in *Character-Aware Neural Language Models*,[1] then we'll implement it. Specifically, we'll replace the 'Embedding lookup' stage in Figure 1 with a sequence of more involved stages, depicted in Figure 2.
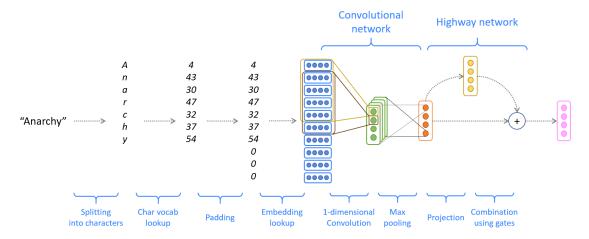

Figure 2: Character-based convolutional encoder, which ultimately produces a word embedding of length e_{word}.

Model description

The model in Figure 2 has four main stages, which we'll describe for a *single example* (not a batch):

1. **Convert word to character indices**. We have a word x (e.g. Anarchy in Figure 2) that we wish to represent. Assume we have a predefined 'vocabulary' of characters (for example, all lowercase letters, uppercase letters, numbers, and some punctuation). By looking up the index of each character, we can thus represent the length-l word x as a vector of integers:

$$\mathbf{x} = [c_1, c_2, \cdots, c_l] \in \mathbb{Z}^l \tag{1}$$

where each c_i is an integer index into the character vocabulary.

---

[1] Character-Aware Neural Language Models, Kim et al., 2016. https://arxiv.org/abs/1508.06615

2. **Padding and embedding lookup**. Using a special `<PAD>` 'character', we pad (or truncate) every word so that it has length m_{word} (this is some predefined hyperparameter representing maximum word length):

$$\mathbf{x}_{\text{padded}} = [c_1, c_2, \cdots, c_{m_{\text{word}}}] \in \mathbb{Z}^{m_{\text{word}}} \tag{2}$$

For each of these characters c_i, we lookup a dense character embedding (which has shape e_{char}). This yields a tensor $\mathbf{x}_{\text{emb}}$:

$$\mathbf{x}_{\text{emb}} = \text{CharEmbedding}(\mathbf{x}_{\text{padded}}) \in \mathbb{R}^{m_{\text{word}} \times e_{\text{char}}} \tag{3}$$

We'll reshape $\mathbf{x}_{\text{emb}}$ to obtain $\mathbf{x}_{\text{reshaped}} \in \mathbb{R}^{e_{\text{char}} \times m_{\text{word}}}$ before feeding into the convolutional network.[2]

3. **Convolutional network**. To combine these character embeddings, we'll use 1-dimensional convolutions. The convolutional layer has two hyperparameters:[3] the kernel size k (also called window size), which dictates the size of the window used to compute features, and the number of filters f (also called number of output features or number of output channels). The convolutional layer has a weight matrix $\mathbf{W} \in \mathbb{R}^{f \times e_{\text{char}} \times k}$ and a bias vector $\mathbf{b} \in \mathbb{R}^f$.

To compute the ith output feature (where $i \in \{1, \ldots, f\}$) for the tth window of the input, the convolution operation is performed between the input window[4] $(\mathbf{x}_{\text{reshaped}})_{[:, \, t:t+k-1]} \in \mathbb{R}^{e_{\text{char}} \times k}$ and the weights $\mathbf{W}_{[i,:,:]} \in \mathbb{R}^{e_{\text{char}} \times k}$, and the bias term $\mathbf{b}_i \in \mathbb{R}$ is added:

$$(\mathbf{x}_{\text{conv}})_{i,t} = \text{sum}\left(\mathbf{W}_{[i,:,:]} \odot (\mathbf{x}_{\text{reshaped}})_{[:, \, t:t+k-1]}\right) + \mathbf{b}_i \quad \in \mathbb{R} \tag{4}$$

where $\odot$ is elementwise multiplication of two matrices with the same shape and `sum` is the sum of all the elements in the matrix. This operation is performed for every feature i and every window t, where $t \in \{1, \ldots, m_{\text{word}} - k + 1\}$. Overall this produces output $\mathbf{x}_{\text{conv}}$:

$$\mathbf{x}_{\text{conv}} = \text{Conv1D}(\mathbf{x}_{\text{reshaped}}) \in \mathbb{R}^{f \times (m_{\text{word}} - k + 1)} \tag{5}$$

For our application, we'll set f to be equal to e_{word}, the size of the final word embedding for word x (the rightmost vector in Figure 2). Therefore,

$$\mathbf{x}_{\text{conv}} \in \mathbb{R}^{e_{\text{word}} \times (m_{\text{word}} - k + 1)} \tag{6}$$

Finally, we apply the ReLU function to $\mathbf{x}_{\text{conv}}$, then use max-pooling to reduce this to a single vector $\mathbf{x}_{\text{conv\_out}} \in \mathbb{R}^{e_{\text{word}}}$, which is the final output of the Convolutional Network:

$$\mathbf{x}_{\text{conv\_out}} = \text{MaxPool}(\text{ReLU}(\mathbf{x}_{\text{conv}})) \in \mathbb{R}^{e_{\text{word}}} \tag{7}$$

Here, MaxPool simply takes the maximum across the second dimension. Given a matrix $M \in \mathbb{R}^{a \times b}$, then $\text{MaxPool}(M) \in \mathbb{R}^a$ with $\text{MaxPool}(M)_i = \max_{1 \leq j \leq b} M_{ij}$ for $i \in \{1, \ldots, a\}$.

4. **Highway layer and dropout.** As mentioned in Lectures 7 and 11, Highway Networks[5] have a skip-connection controlled by a dynamic gate. Given the input $\mathbf{x}_{\text{conv\_out}} \in \mathbb{R}^{e_{\text{word}}}$, we compute:

$$\mathbf{x}_{\text{proj}} = \text{ReLU}(\mathbf{W}_{\text{proj}}\mathbf{x}_{\text{conv\_out}} + \mathbf{b}_{\text{proj}}) \qquad \in \mathbb{R}^{e_{\text{word}}} \tag{8}$$

$$\mathbf{x}_{\text{gate}} = \sigma(\mathbf{W}_{\text{gate}}\mathbf{x}_{\text{conv\_out}} + \mathbf{b}_{\text{gate}}) \qquad \in \mathbb{R}^{e_{\text{word}}} \tag{9}$$

where the weight matrices $\mathbf{W}_{\text{proj}}, \mathbf{W}_{\text{gate}} \in \mathbb{R}^{e_{\text{word}} \times e_{word}}$ and bias vectors $\mathbf{b}_{\text{proj}}, \mathbf{b}_{\text{gate}} \in \mathbb{R}^{e_{\text{word}}}$. Next, we obtain the output $\mathbf{x}_{\text{highway}}$ by using the gate to combine the projection with the skip-connection:

$$\mathbf{x}_{\text{highway}} = \mathbf{x}_{\text{gate}} \odot \mathbf{x}_{\text{proj}} + (1 - \mathbf{x}_{\text{gate}}) \odot \mathbf{x}_{\text{conv\_out}} \in \mathbb{R}^{e_{\text{word}}} \tag{10}$$

---

[2]Necessary because the PyTorch Conv1D function performs the convolution only on the *last* dimension of the input.

[3]We assume no padding is applied and the stride is 1.

[4]In the notation $(\mathbf{x}_{\text{reshaped}})_{[:, \, t:t+k-1]}$, the range $t : t + k - 1$ is *inclusive*, i.e. the width-k window $\{t, t + 1, \ldots, t + k - 1\}$.

[5]Highway Networks, Srivastava et al., 2015. https://arxiv.org/abs/1505.00387

where $\odot$ denotes element-wise multiplication. Finally, we apply dropout to $\mathbf{x}_{\text{highway}}$:

$$\mathbf{x}_{\text{word\_emb}} = \text{Dropout}(\mathbf{x}_{\text{highway}}) \in \mathbb{R}^{e_{\text{word}}} \tag{11}$$

We're done! $\mathbf{x}_{\text{word\_emb}}$ is the embedding we will use to represent word x – this will replace the lookup-based word embedding we used in Assignment 4.

Implementation

In the remainder of Section 1, we will be implementing the character-based encoder in our NMT system. Though you could implement this on top of your own Assignment 4 solution code, for simplicity and fairness we have supplied you[6] with a full implementation of the Assignment 4 word-based NMT model (with some modifications); this is what you will use as a basis for your Assignment 5 code.

You will not need to use your VM until Section 2 – the rest of this section can be done on your local machine. In order to run the model code on your **local** machine, please run the following command to create the proper virtual environment:

```
conda env create --file local_env.yml
```

Run the following to create the correct vocab files:

```
sh run.sh vocab
```

*Note: **Please** do not use external python modules which are not specified in the gpu_requirements.txt or local_env.yml file. This will cause the autograder to fail.*

---

[6]available on Stanford Box; requires Stanford login

Let's implement the entire network specified in Figure 2, from left to right.

(a) (1 point) (coding) In `vocab.py`, implement the method `words2charindices()` (hint: copy the structure of `words2indices()`) to convert each character to its corresponding index in the character-vocabulary. This corresponds to the first two steps of Figure 2 (splitting and vocab lookup). Run the following for a non-exhaustive sanity check:

    python sanity_check.py 1a

(b) (4 points) (coding) Implement `pad_sents_char()` in `utils.py`, similar to the version for words. This method should pad at the character and word level. All words should be padded/truncated to max word length $m_{\mathrm{word}} = 21$, and all sentences should be padded to the length of the longest sentence in the batch. A padding word is represented by m_{word} `<PAD>`-characters. Run the following for a non-exhaustive sanity check:

    python sanity_check.py 1b

(c) (3 points) (coding) Implement `to_input_tensor_char()` in `vocab.py` to connect the previous two parts: use both methods created in the previous steps and convert the resulting padded sentences to a torch tensor. **Ensure you reshape the dimensions** so that the output has shape: (`max_sentence_length`, `batch_size`, `max_word_length`) so that we don't have any shape errors downstream!

(d) (coding) In the empty file `highway.py`, implement the highway network as a `nn.Module` class called `Highway`.[7]

- Your module will need a `__init__()` and a `forward()` function (whose inputs and outputs you decide for yourself).
- The `forward()` function will need to map from $\mathbf{x}_{\mathrm{conv\_out}}$ to $\mathbf{x}_{\mathrm{highway}}$.
- Note that although the model description above is not batched, your `forward()` function should operate on batches of words.
- Make sure that your module uses two `nn.Linear` layers.

There is no provided sanity check for your `Highway` implementation – instead, you will now write your own code to thoroughly test your implementation. You should do whatever you think is sufficient to convince yourself that your module computes what it's supposed to compute. Possible ideas include (and you should do multiple):

- Write code to check that the input and output have the expected shapes and types. Before you do this, make sure you've written docstrings for `__init__()` and `forward()` – you can't test the expected output if you haven't clearly laid out what you expect it to be!
- Print out the shape of every intermediate value; verify all the shapes are correct.
- Create a small instance of your highway network (with small, manageable dimensions), manually define some input, manually define the weights and biases, manually calculate what the output should be, and verify that your module does indeed return that value.
- Similar to previous, but you could instead print all intermediate values and check each is correct.
- If you can think of any 'edge case' or 'unusual' inputs, create test cases based on those.

Important: *to avoid crashing the autograder, make sure that any print statements are commented out when you submit your code.*

(e) (coding) In the empty file `cnn.py`, implement the convolutional network as a `nn.Module` class called `CNN`.

---

[7]If you're unsure how to structure a `nn.Module`, you can start here: https://pytorch.org/tutorials/beginner/examples_nn/two_layer_net_module.html. After that, you could look at the many examples of `nn.Modules` in Assignments 3 and 4.

- Your module will need a __init__() and a forward() function (whose inputs and outputs you decide for yourself).
- The forward() function will need to map from $\mathbf{x}_{reshaped}$ to $\mathbf{x}_{conv\_out}$.
- Note that although the model description above is not batched, your forward() function should operate on batches of words.
- Make sure that your module uses one nn.Conv1d layer (this is important for the autograder).
- Use a kernel size of $k = 5$.

(f) (12 points) (coding) Write the __init__() and forward() functions for the ModelEmbeddings class in model_embeddings.py.[8]

- The forward() function must map from $\mathbf{x}_{padded}$ to $\mathbf{x}_{word\_emb}$ – note this is for whole batches of sentences, rather than batches of words.
- You will need to use the CNN and Highway modules you created.
- Don't forget the dropout layer! Use 0.3 dropout probability.
- Your ModelEmbeddings class should contain one nn.Embedding object (this is important for the autograder).
- Remember that we are using $e_{char} = 50$.
- Depending on your implementation of CNN and Highway, it's likely that you will need to reshape tensors to get them into the shape required by CNN and Highway, and then reshape again to get the final output of ModelEmbeddings.forward().

Sanity check if the output from your model has the correct shape by running the following:

```
python sanity_check.py 1f
```

The majority of the 12 points are awarded based on a hidden autograder. Though we don't require it, you should check your implementation using techniques similar to what you did in (d) and (e), before you do the 'small training run' check in (h).

(g) (2 points) (coding) In nmt_model.py, complete the forward() method to use character-level padded encodings instead of word-level encodings. This ties together your ModelEmbeddings code with the preprocessing code you wrote. Check your code!

(h) (5 points) (coding) **On your local machine**, confirm that you're in the proper conda enironment then execute the following command. This will train your model on a very small subset of the training data, then test it. Your model should overfit the training data.

```
sh run.sh train_local_q1
sh run.sh test_local_q1
```

Running these should take around 5 minutes (but this depends otn your local machine). You should observe the average loss go down to near 0 and average perplexity on train and dev set go to 1 during training. Once you run the test, you should observe BLEU score on the test set higher than 99.00. If you don't observe these things, you probably need to go back to debug!

Important: Make sure not to modify the output file (outputs/test_outputs_local_q1.txt) generated by the code – you need this to be included when you run the submission script.

---

[8]Note that in this assignment, the full NMT model defines two ModelEmbeddings objects (one for source and one for target), whereas in Assignment 4, there was a single ModelEmbeddings object that contained two nn.Embedding objects (one for source and one for target).

2. Character-based LSTM decoder for NMT (26 points)

We will now add a LSTM-based character-level decoder to our NMT system, based on Luong & Manning's work.[9] The main idea is that when our word-level decoder produces an `<UNK>` token, we run our character-level decoder (which you can think of as a character-level conditional language model) to instead generate the target word one character at a time, as shown in Figure 3. This will help us to produce rare and out-of-vocabulary target words.
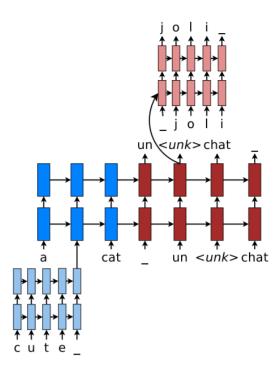

Figure 3: A character-based decoder which is triggered if the word-based decoder produces an UNK. Figure courtesy of Luong & Manning.

We now describe the model in three sections:

Forward computation of Character Decoder   Given a sequence of integers $x_1, \ldots, x_n \in \mathbb{Z}$ representing a sequence of characters, we lookup their character embeddings $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{e_{\text{char}}}$ and pass these as input into the (unidirectional) LSTM, obtaining hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_n$ and cell states $\mathbf{c}_1, \ldots, \mathbf{c}_n$:

$$\mathbf{h}_t, \mathbf{c}_t = \text{CharDecoderLSTM}(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}) \text{ where } \mathbf{h}_t, \mathbf{c}_t \in \mathbb{R}^h \tag{12}$$

where h is the hidden size of the CharDecoderLSTM. The initial hidden and cell states $\mathbf{h}_0$ and $\mathbf{c}_0$ are both set to the *combined output vector* (refer to Assignment 4) for the current timestep of the main word-level NMT decoder.

For every timestep $t \in \{1, \ldots, n\}$ we compute scores (also called logits) $\mathbf{s}_t \in \mathbb{R}^{V_{\text{char}}}$:

$$\mathbf{s}_t = \mathbf{W}_{\text{dec}} \mathbf{h}_t + \mathbf{b}_{\text{dec}} \in \mathbb{R}^{V_{char}} \tag{13}$$

where the weight matrix $\mathbf{W}_{\text{dec}} \in \mathbb{R}^{V_{\text{char}} \times h}$ and the bias vector $\mathbf{b}_{\text{dec}} \in \mathbb{R}^{V_{\text{char}}}$. If we passed $\mathbf{s}_t$ through a softmax function, we would have the probability distribution for the next character in the sequence.

---

[9] Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models, Luong and Manning, 2016. https://arxiv.org/abs/1604.00788

Training of Character Decoder   When we train the NMT system, we train the character decoder on *every* word in the target sentence (not just the words represented by <UNK>). For example, on a particular step of the main NMT decoder, if the target word is music then the input sequence for the CharDecoderLSTM is $[x_1, \ldots, x_n] = [\texttt{<START>}, \texttt{m}, \texttt{u}, \texttt{s}, \texttt{i}, \texttt{c}]$ and the target sequence for the CharDecoderLSTM is $[x_2, \ldots, x_{n+1}] = [\texttt{m}, \texttt{u}, \texttt{s}, \texttt{i}, \texttt{c}, \texttt{<END>}]$.

We pass the input sequence $x_1, \ldots, x_n$, (along with the initial states $\mathbf{h}_0$ and $\mathbf{c}_0$ obtained from the combined output vector) into the CharDecoderLSTM, thus obtaining scores $\mathbf{s}_1, \ldots, \mathbf{s}_n$ which we will compare to the target sequence $x_2, \ldots, x_{n+1}$. We optimize with respect to sum of the cross-entropy loss:

$$\mathbf{p}_t = \text{softmax}(\mathbf{s}_t) \in \mathbb{R}^{V_{\text{char}}} \qquad \forall t \in \{1, \ldots, n\} \tag{14}$$

$$\text{loss}_{\text{char\_dec}} = -\sum_{t=1}^{n} \log \mathbf{p}_t(x_{t+1}) \in \mathbb{R} \tag{15}$$

Note that when we compute $\text{loss}_{\text{char\_dec}}$ for a batch of words, we take the sum (not average) across the entire batch. On each training iteration, we add $\text{loss}_{\text{char\_dec}}$ to the loss of the word-based decoder, so that we simultaneously train the word-based model and character-based decoder.

Decoding from the Character Decoder   At test time, first we produce a translation from our word-based NMT system in the usual way (e.g. a decoding algorithm like beam search). If the translation contains any <UNK> tokens, then for each of those positions, we use the word-based decoder's combined output vector to initialize the CharDecoderLSTM's initial $\mathbf{h}_0$ and $\mathbf{c}_0$, then use CharDecoderLSTM to generate a sequence of characters.

To generate the sequence of characters, we use the *greedy decoding* algorithm, which repeatedly chooses the most probable next character, until either the <END> token is produced or we reach a predetermined max_length. The algorithm is given below, for a single example (not batched).

Algorithm 1 Greedy Decoding

Input: Initial states $\mathbf{h}_0, \mathbf{c}_0$

Output: output_word generated by the character decoder (doesn't contain <START> or <END>)

1: **procedure** DECODE_GREEDY
2:    output_word ← []
3:    current_char ← <START>
4:    **for** $t = 0, 1, \ldots, \texttt{max\_length} - 1$ **do**
5:        $\mathbf{h}_{t+1}, \mathbf{c}_{t+1} \leftarrow \text{CharDecoder}(\texttt{current\_char}, \mathbf{h}_t, \mathbf{c}_t)$ ▷ use last predicted character as input
6:        $\mathbf{s}_{t+1} \leftarrow \mathbf{W}_{\text{dec}} \mathbf{h}_{t+1} + \mathbf{b}_{\text{dec}}$ ▷ compute scores
7:        $\mathbf{p}_{t+1} \leftarrow \text{softmax}(\mathbf{s}_{t+1})$ ▷ compute probabilities
8:        current_char ← $\text{argmax}_c \ \mathbf{p}_{t+1}(c)$ ▷ the most likely next character
9:        **if** current_char=<END> **then**
10:            **break**
11:        output_word ← output_word + [current_char]   ▷ append this character to output word
12:    **return** output_word

Implementation

At the end of this section, you will train the full NMT system (with character-encoder and character-decoder). As in the previous assignment, we strongly advise that you first develop the code locally and ensure it does not crash, before attempting to train it on your VM. Finally, **make sure that your VM is turned off whenever you are not using it.**

(a) (2 points) (coding) Write the `__init__` function for the `CharDecoder` module in `char_decoder.py`. Run the following for a non-exhaustive sanity check:

```
python sanity_check.py 2a
```

(b) (3 points) (coding) Write the `forward()` function in `char_decoder.py`. This function takes input $x_1, \ldots, x_n$ and $(\mathbf{h}_0, \mathbf{c}_0)$ (as described in the **Forward computation of the character decoder** section) and returns $\mathbf{s}_1, \ldots, \mathbf{s}_n$ and $(\mathbf{h}_n, \mathbf{c}_n)$.

Run the following for a non-exhaustive sanity check:

```
python sanity_check.py 2b
```

(c) (5 points) (coding) Write the `train_forward()` function in `char_decoder.py`. This function computes $\text{loss}_{\text{char\_dec}}$ summed across the whole batch. Hint: Carefully read the documentation for `nn.CrossEntropyLoss`.

Run the following for a non-exhaustive sanity check:

```
python sanity_check.py 2c
```

(d) (7 points) (coding) Write the `decode_greedy()` function in `char_decoder.py` to implement the algorithm DECODE_GREEDY. Note that although Algorithm 1 is described for a single example, your implementation must work over a batch. Algorithm 1 also indicates that you should break when you reach the `<END>` token, but in the batched case you might find it more convenient to complete all `max_length` steps of the for-loop, then truncate the `output_words` afterwards.

Run the following for a non-exhaustive sanity check:

```
python sanity_check.py 2d
```

(e) (3 points) (coding) Once you have thoroughly checked your implementation of the `CharDecoder` functions (checking much more than just the sanity checks!), it's time to do the 'small training run' check.

Confirm that you're in the proper conda environment and then execute the following command on your **local** machine to check if your model overfits to a small subset of the training data.

```
sh run.sh train_local_q2
sh run.sh test_local_q2
```

Running these should take around 10 minutes (but this depends on your local machine). You should observe the average loss go down to near 0 and average perplexity on train and dev set go to 1 during training. Once you run the test, you should observe BLEU score on the test set higher than 99.00. If you don't observe these things, you probably need to go back to debug!

Important: Make sure not to modify the output file (`outputs/test_outputs_local_q2.txt`) generated by the code – you need this to be included when you run the submission script.

(f) (6 points) (coding) Now that we've implemented both the character-based encoder and the character-based decoder, it's finally time to train the full system!

Connect to your VM and install necessary packages by running:

```
pip install -r gpu_requirements.txt
```

Once your VM is configured and you are in a tmux session,[10] execute:

```
sh run.sh train
```

This should take between **8-12 hours** to train on the VM, though time may vary depending on your implementation and whether or not the training procedure hits the early stopping criterion.

Run your model on the test set using:

---

[10]Refer to *Practical tips for using VMs* for more information on tmux

```
sh run.sh test
```

and **ensure that the output file `outputs/test_outputs.txt` is present and unmodified – this will be included in your submission, and we'll use it to evaluate the final BLEU score**

Given your BLEU score b, here's how your points will be determined for this sub-problem:

BLEU Score	Points
$0 \le b < 10$	0
$10 \le b < 16$	2
$16 \le b$	6

Submission Instructions

You will submit this assignment on GradeScope:

1. **Please** do not use external python modules which are not specified in the `gpu_requirements.txt` or `local_env.yml` file. This will cause the `autograder` to fail.

2. Verify that the following files exist at these specified paths within your assignment directory:

   - `outputs/test_outputs.txt`
   - `outputs/test_outputs_local_q1.txt`
   - `outputs/test_outputs_local_q2.txt`

3. Run the `collect_submission.sh` script to produce your `assignment5.zip` file.

4. Upload your `assignment5.zip` file to GradeScope to **Assignment 5 [coding]**.

5. Check that the public autograder tests passed correctly. In particular, check that the BLEU scores calculated for parts 1(h), 2(e) and 2(f) match what you expect, because points will be awarded based on these autograder-computed numbers.