This is the last assignment before you begin working on your projects. It is designed to prepare you for implementing things by yourself. This assignment is coding-heavy and written-question-light. 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. In some questions we explicitly ask you to design and run your own tests, but you should be checking your code throughout.
- When you upload your code to Gradescope, it will be autograded with some less-basic tests, but the results of these autograder tests will be hidden until after grades are released.
- 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 (1) with a character-based convolutional encoder, and in Section 2 we will enhance (4) by adding a character-based LSTM decoder.¹ This will hopefully improve our BLEU performance on the test set! Lastly, in Section 3, we will inspect the word embeddings produced by our character-level encoder, and analyze some errors from our new NMT system.

¹We could also modify parts (2) and (3) of the NMT model to use subword information. However, to keep things simple for this assignment, we just make changes to the embedding and output prediction layers.
1. Character-based convolutional encoder for NMT (36 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 $e_{\text{word}}$.](image)

In this section, we will first describe a method based on Kim et al.’s work in *Character-Aware Neural Language Models*, 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 $e_{\text{word}}$.](image)

Model description and written questions

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 represent a length-$l$ word $x$ as a vector of integers:

   $$x = [c_1, c_2, \cdots, c_l] \in \mathbb{Z}^l$$  \hspace{1cm} (1)

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

---

2. **Padding and embedding lookup.** Using a special `<PAD>` ‘character’, we pad (or truncate) every word so that it has length \( m_{\text{word}} \) (defined as the length of longest word in the batch):

\[
x_{\text{padded}} = [c_1, c_2, \ldots, c_{m_{\text{word}}}] \in \mathbb{Z}^{m_{\text{word}}}
\]

(2)

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

\[
x_{\text{emb}} = \text{CharEmbedding}(x_{\text{padded}}) \in \mathbb{R}^{m_{\text{word}} \times e_{\text{char}}}
\]

(3)

We’ll reshape \( x_{\text{emb}} \) to obtain \( x_{\text{reshaped}} \in \mathbb{R}^{e_{\text{char}} \times m_{\text{word}}} \) before feeding into the convolutional network.

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 \( W \in \mathbb{R}^{f \times e_{\text{char}} \times k} \) and a bias vector \( b \in \mathbb{R}^f \).

To compute the \( i \)th output feature (where \( i \in \{1, \ldots, f\} \)) for the \( t \)th window of the input, the convolution operation is performed between the input window\(^5\) \( (x_{\text{reshaped}})[i, t:t+k-1] \in \mathbb{R}^{e_{\text{char}} \times k} \) and the weights \( W[i,:,:] \in \mathbb{R}^{e_{\text{char}} \times k} \), and the bias term \( b_i \in \mathbb{R} \) is added:

\[
(x_{\text{conv}})[i,t] = \sum (W[i,:,:] \odot (x_{\text{reshaped}})[i, t:t+k-1]) + b_i \in \mathbb{R}
\]

(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 \( x_{\text{conv}} \):

\[
x_{\text{conv}} = \text{Conv1D}(x_{\text{reshaped}}) \in \mathbb{R}^{f \times (m_{\text{word}} - k + 1)}
\]

(5)

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

\[
x_{\text{conv}} \in \mathbb{R}^{e_{\text{word}} \times (m_{\text{word}} - k + 1)}
\]

(6)

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

\[
x_{\text{conv,out}} = \text{MaxPool}(\text{ReLU}(x_{\text{conv}})) \in \mathbb{R}^{e_{\text{word}}}
\]

(7)

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

4. **Highway layer and dropout.** As mentioned in Lectures 7 and 11, Highway Networks\(^6\) have a skip-connection controlled by a dynamic gate. Given the input \( x_{\text{conv,out}} \in \mathbb{R}^{e_{\text{word}}} \), we compute:

\[
x_{\text{proj}} = \text{ReLU}(W_{\text{proj}} x_{\text{conv,out}} + b_{\text{proj}}) \in \mathbb{R}^{e_{\text{word}}}
\]

(8)

\[
x_{\text{gate}} = \sigma(W_{\text{gate}} x_{\text{conv,out}} + b_{\text{gate}}) \in \mathbb{R}^{e_{\text{word}}}
\]

(9)

where the weight matrices \( W_{\text{proj}}, W_{\text{gate}} \in \mathbb{R}^{e_{\text{word}} \times e_{\text{word}}} \), bias vectors \( b_{\text{proj}}, b_{\text{gate}} \in \mathbb{R}^{e_{\text{word}}} \) and \( \sigma \) is the sigmoid function. Next, we obtain the output \( x_{\text{highway}} \) by using the gate to combine the projection with the skip-connection:

\[
x_{\text{highway}} = x_{\text{gate}} \odot x_{\text{proj}} + (1 - x_{\text{gate}}) \odot x_{\text{conv,out}} \in \mathbb{R}^{e_{\text{word}}}
\]

(10)

---

3Necessary because the PyTorch Conv1D function performs the convolution only on the last dimension of the input.

4We assume no padding is applied and the stride is 1.

5In the notation \( (x_{\text{reshaped}})[i, t:t+k-1] \), the range \( t:t+k-1 \) is inclusive, i.e. the width-\( k \) window \( \{t, t+1, \ldots, t+k-1\} \).

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

$$x_{\text{word,emb}} = \text{Dropout}(x_{\text{highway}}) \in \mathbb{R}^{e_{\text{word}}}$$  \hspace{1cm} (11)$$

We’re done! $x_{\text{word,emb}}$ is the embedding we will use to represent word $x$ – this will replace the lookup-based word embedding we used in Assignment 4.

(a) (1 point) (written) We learned in class that recurrent neural architectures can operate over variable length input (i.e., the shape of the model parameters is independent of the length of the input sentence). Is the same true of convolutional architectures? Write one sentence to explain why or why not.

(b) (2 points) (written) In step 2 of the character-based embedding model, for each word in a sentence, we pad it to length $m_{\text{word}}$ (the length of longest word in the batch). In fact, in our implementation, we also add two special characters to this sequence: one character at the beginning standing for the start of word token, and another character at the end standing for the end of word token. Therefore, $x_{\text{padded}}^j = [c_0, c_1, c_2, \cdots, c_{m_{\text{word}}}, c_{m_{\text{word}}+1}] \in \mathbb{Z}^{m_{\text{word}}+2}$, where $c_0 = \text{start of word}$, and $c_{m_{\text{word}}+1} = \text{end of word}$.

In 1D convolutions, we do padding, i.e. we add some zeros to both sides of our input, so that the kernel sliding over the input can be applied to at least one complete window.

In this case, if we use the kernel size $k = 5$, what will be the size of the padding (i.e. the additional number of zeros on each side) we need for the 1-dimensional convolution, such that there exists at least one window for all possible values of $m_{\text{word}}$ in our dataset? Explain your reasoning.

Hints:

- What is the smallest possible value that $m_{\text{word}}$ can take?
- After padding extra zeros to the input of 1-d convolution layer ($x_{\text{reshaped}}$), the updated $x'_{\text{reshaped}}$ will have size $x'_{\text{reshaped}} \in \mathbb{R}^{e_{\text{char}} \times (m_{\text{word}}+2+2 \times \text{padding})}$.

(c) (3 points) (written) In step 4, we introduce a Highway Network with $x_{\text{highway}} = x_{\text{gate}} \odot x_{\text{proj}} + (1 - x_{\text{gate}}) \odot x_{\text{conv \_out}}$. Since $x_{\text{gate}}$ is the result of the sigmoid function, it has the range $(0, 1)$. Consider the two extreme cases. If $x_{\text{gate}} \rightarrow 0$, then $x_{\text{highway}} \rightarrow x_{\text{conv \_out}}$. When $x_{\text{gate}} \rightarrow 1$, then $x_{\text{highway}} \rightarrow x_{\text{proj}}$. This means the Highway layer is smoothly varying its behavior between that of normal linear layer ($x_{\text{proj}}$) and that of a layer which simply passes its inputs ($x_{\text{conv \_out}}$) through. Use one or two sentences to explain why this behavior is useful in character embeddings.

Based on the definition of $x_{\text{gate}} = \sigma(W_{\text{gate}}x_{\text{conv \_out}} + b_{\text{gate}})$, do you think it is better to initialize $b_{\text{gate}}$ to be negative or positive? Explain your reason briefly.

(d) (2 points) (written) In Lecture 10, we briefly introduced Transformers, a non-recurrent sequence (or sequence-to-sequence) model with a sequence of attention-based transformer blocks. Describe 2 advantages of a Transformer encoder over the LSTM-with-attention encoder in our NMT model (which we used in both Assignment 4 and Assignment 5).

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\(^7\) 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:

\(^7\)available on Stanford Box; requires Stanford login
conda env create --file local_env.yml

Note that this virtual environment **will not** be needed on the VM.

Run the following to create the correct vocab files:

```bash
sh run.sh vocab
```
Let’s implement the entire network specified in Figure 2, from left to right.

(e) (4 points) (coding) Implement to_input_tensor_char() in vocab.py by using 2 methods:
   • Use words2charindices() in vocab.py to convert each character in all words to its corresponding index in the character-vocabulary.
   • Use pad_sents_char() in utils.py to pad all words to max_word_length of all words in the batch, and pad all sentences to max_sentence_length of all sentences in the batch.

Then convert the resulting padded sentences to a torch tensor. This corresponds to the first three steps of Figure 2 (splitting, vocab lookup and padding). Ensure you reshape the dimensions so that the output has shape: (max_sentence_length, batch_size, max_word_length). Run the following for a non-exhaustive sanity check:

   python sanity_check.py 1e

(f) (4 points) (coding and written) In the empty file highway.py, implement the highway network as a nn.Module class called Highway.\(^8\)
   • 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 x\(_{\text{conv}}\) \(_{\text{out}}\) to x\(_{\text{highway}}\).
   • 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 (this is important for the autograder).

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.

Once you’ve finished testing your module, write a short description of the tests you carried out, and why you believe they are sufficient. The 4 points for this question are awarded based on your written description of the tests only.

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

(g) (4 points) (coding and written) In the empty file cnn.py, implement the convolutional network as a nn.Module class called CNN.
   • 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 x\(_{\text{reshaped}}\) to x\(_{\text{conv}}\) \(_{\text{out}}\).

\(^8\)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.
• Note that although the model description above is not batched, your forward() function should operate on batches of words.
• Make sure that your module has an instance variable of type nn.Conv1d (this is important for the autograder).
• Use a kernel size of $k = 5$ and padding = 1.

As in (f), write code to thoroughly test your implementation. Once you’ve finished testing your module, write a short description of the tests you carried out, and why you believe they are sufficient. As in (f), the 4 points are for your written description only.

(h) (10 points) (coding) Write the __init__() and forward() functions for the ModelEmbeddings class in model_embeddings.py.\(^9\)

- The forward() function must map from $x_{\text{padded}}$ to $x_{\text{word,emb}}$—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). It should also contain a field named self.word_embed_size.
- Remember that we are using $e_{\text{char}} = 50$.
- 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().
- Make sure that you use permute() or transpose() instead of using view() or reshape() to swap dimensions in forward() (this is important for the autograder).

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

```
python sanity_check.py 1h
```

The majority of the 10 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 (f) and (g), before you do the ‘small training run’ check in (j).

(i) (4 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!

(j) (2 points) (coding) On your local machine, confirm that you're in the proper conda environment 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. (You may need to .contiguous().view() instead of .view() when reshaping).

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

Running these should take around 5 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_q1.txt) generated by the code – you need this to be included when you run the submission script.

---

\(^9\)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.\(^{10}\) 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. This graph is just for reference and in our implementation we will use only one layer of LSTM as stated in details below.](image)

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 \(x_1, \ldots, x_n \in \mathbb{R}^{e_{\text{char}}}\) and pass these as input into the (unidirectional) LSTM, obtaining hidden states \(h_1, \ldots, h_n\) and cell states \(c_1, \ldots, c_n\):

\[
h_t, c_t = \text{CharDecoderLSTM}(x_t, h_{t-1}, c_{t-1}) \quad \text{where} \quad h_t, c_t \in \mathbb{R}^h
\]

where \(h\) is the hidden size of the CharDecoderLSTM. The initial hidden and cell states \(h_0\) and \(c_0\) 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) \(s_t \in \mathbb{R}^{V_{\text{char}}}:\)

\[
s_t = W_{\text{dec}} h_t + b_{\text{dec}} \in \mathbb{R}^{V_{\text{char}}} \quad \text{(13)}
\]

where the weight matrix \(W_{\text{dec}} \in \mathbb{R}^{V_{\text{char}} \times h}\) and the bias vector \(b_{\text{dec}} \in \mathbb{R}^{V_{\text{char}}}\). If we passed \(s_t\) through a softmax function, we would have the probability distribution for the next character in the sequence.

---

\(^{10}\) Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models, Luong and Manning, 2016. 
**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 \([x_1, \ldots, x_n] = [\text{<START>}, m, u, s, i, c]\) and the target sequence for the CharDecoderLSTM is \([x_2, \ldots, x_{n+1}] = [m, u, s, i, c, \text{<END>}]\).

We pass the input sequence \(x_1, \ldots, x_n\), (along with the initial states \(h_0\) and \(c_0\) obtained from the combined output vector) into the CharDecoderLSTM, thus obtaining scores \(s_1, \ldots, s_n\) 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:

\[
p_t = \text{softmax}(s_t) \in \mathbb{R}^{V_{\text{char}}} \quad \forall t \in \{1, \ldots, n\}
\]

\[
\text{loss}_{\text{char,dec}} = -\sum_{t=1}^{n} \log p_t(x_{t+1}) \in \mathbb{R}
\]

Note that when we compute 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 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 \(h_0\) and \(c_0\), 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\(_{\text{length}}\). The algorithm is given below, for a single example (not batched).

**Algorithm 1 Greedy Decoding**

**Input:** Initial states \(h_0, c_0\)

**Output:** output\(_{\text{word}}\) generated by the character decoder (doesn’t contain <START> or <END>)

1: procedure \(\text{DECODE\_GREEDY}\)
2: output\(_{\text{word}}\) ← []
3: current\(_{\text{char}}\) ← <START>
4: for \(t = 0, 1, \ldots, \text{max\(_{\text{length}}\)} - 1\) do
5: \(h_{t+1}, c_{t+1} \leftarrow \text{CharDecoder}(\text{current\(_{\text{char}}\), } h_t, c_t)\) \(\triangleright\) use last predicted character as input
6: \(s_{t+1} \leftarrow W_{\text{dec}}h_{t+1} + b_{\text{dec}}\) \(\triangleright\) compute scores
7: \(p_{t+1} \leftarrow \text{softmax}(s_{t+1})\) \(\triangleright\) compute probabilities
8: current\(_{\text{char}}\) ← \(\text{argmax}_c \ p_{t+1}(c)\) \(\triangleright\) the most likely next character
9: if current\(_{\text{char}}\) = <END> then
10: break
11: output\(_{\text{word}}\) ← output\(_{\text{word}}\) + [current\(_{\text{char}}\)] \(\triangleright\) append this character to output word
12: return output\(_{\text{word}}\)

**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.
If your Azure subscription runs out of money your VM will be temporarily locked and inaccessible. If that happens, make a private post on Piazza with your name, email used for Azure, and SUNetID, to request more credits.

(a) (4 points) (coding) Write the `forward()` function in `char_decoder.py`. This function takes input \( x_1, \ldots, x_n \) and \((h_0, c_0)\) (as described in the Forward computation of the character decoder section) and returns \( s_1, \ldots, s_n \) and \((h_n, c_n)\).

Run the following for a non-exhaustive sanity check:

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

(b) (5 points) (coding) Write the `train_forward()` function in `char_decoder.py`. This function computes 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
python sanity_check.py 2b
```

(c) (8 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
python sanity_check.py 2c
```

(d) (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.

(e) (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 gpus_requirements.txt
```

Once your VM is configured and you are in a tmux session, 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:

```
sh run.sh test
```

---

11 Refer to [Practical tips for using VMs](#) for more information on tmux
and report your test set BLEU score in your assignment write-up. Also 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 verify your self-reported BLEU score.

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

<table>
<thead>
<tr>
<th>BLEU Score</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq b &lt; 35$</td>
<td>0</td>
</tr>
<tr>
<td>$35 \leq b &lt; 36$</td>
<td>2</td>
</tr>
<tr>
<td>$36 \leq b$</td>
<td>6</td>
</tr>
</tbody>
</table>

### 3. Analyzing NMT Systems (8 points)

(a) (2 points) (written) The following table shows some of the forms of the Spanish word *traducir*, which means ‘to translate’.

<table>
<thead>
<tr>
<th>Infinitive</th>
<th>traducir</th>
<th>to translate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Present</td>
<td>traduzco</td>
<td>I translate</td>
</tr>
<tr>
<td></td>
<td>traduces</td>
<td>you translate</td>
</tr>
<tr>
<td></td>
<td>traduce</td>
<td>he or she translates</td>
</tr>
<tr>
<td>Subjunctive</td>
<td>traduzca</td>
<td>that I translate</td>
</tr>
<tr>
<td></td>
<td>traduzcas</td>
<td>that you translate</td>
</tr>
</tbody>
</table>

Use `vocab.json` to find (e.g. using `grep`) which of these six forms are in the word-vocabulary, which consists of the 50,000 most frequent words in the training data for English and for Spanish. Superstrings don’t count (e.g. having *traducen* in the vocabulary is not a hit for *traduce*). State which of these six forms occur, and which do not. Explain in one sentence why this is a bad thing for word-based NMT from Spanish to English. Then explain in detail (approximately two sentences) how our new character-aware NMT model may overcome this problem.

(b) i. (0.5 points) (written) In Assignments 1 and 2, we investigated word embeddings created via algorithms such as Word2Vec, and found that for these embeddings, semantically similar words are close together in the embedding space. In this exercise, we’ll compare this with the word embeddings constructed using the CharCNN trained in our NMT system. Go to [https://projector.tensorflow.org/](https://projector.tensorflow.org/). The website by default shows data from Word2Vec. Look at the nearest neighbors of the following words with dataset Word2Vec All (in cosine distance).

- financial
- neuron
- Francisco
- naturally
- expectation

For each word, report the single closest neighbor. For your convenience, for each example take a screenshot of all the nearest words (so you can compare with the CharCNN embeddings).

ii. (0.5 points) (written) The TensorFlow embedding projector also allows you to upload your own data – you may find this useful in your projects!

Download the character-based word embeddings obtained from our implementation of the character-aware NMT model from this link. Navigate to [https://projector.tensorflow.org/](https://projector.tensorflow.org/), select *Load Data*, and upload the files `character-embeddings.txt` (the embeddings themselves) and `metadata.txt` (the words associated with the embeddings).

---

Now look at the nearest neighbors of the same words. Again, report the single closest neighbors with dataset **Word2Vec All** and take screenshots for yourself.

iii. (3 points) (written) Compare the closest neighbors found by the two methods: briefly describe what kind of similarity is modeled by Word2Vec, and what kind of similarity is modeled by the CharCNN. Explain in detail (2-3 sentences) how the differences in the methodology of Word2Vec and a CharCNN explain the differences you have found.

(c) (2 points) (written) As in Assignment 4, we’ll take a look at the outputs of the model that you have trained! The test set translations your model generated in 2(e) should be located in the outputs directory at: outputs/test_outputs.txt. We also provided translations from a word-based model from our Assignment 4 model in the file outputs/test_outputs_a4.txt.

Find places where the word-based model produced <UNK>, and compare to what the character-based decoder did. Find one example where the character-based decoder produced an acceptable translation in place of <UNK>, and one example where the character-based decoder produced an incorrect translation in place of <UNK>. As in Assignment 4, ‘acceptable’ and ‘incorrect’ doesn’t just mean ‘matches or doesn’t match the reference translation’ – use your own judgment (and Google Translate, if necessary). For each of the two examples, you should:

1. Write the source sentence in Spanish. The source sentences are in en_es_data/test.es.
2. Write the reference English translation of the sentence. The reference translations are in en_es_data/test.en.
3. Write the English translation generated by the model from Assignment 4. These translations are in outputs/test_outputs_a4.txt. Underline the <UNK> you are talking about.
4. Write your character-based model’s English translation. These translations are in outputs/test_outputs.txt. Underline the CharDecoder-generated word you are talking about.
5. Indicate whether this is an acceptable or incorrect example. Give a brief possible explanation (one sentence) for why the character-based model performed this way.

**Submission Instructions**

You will submit this assignment on GradeScope as two submissions – one for **Assignment 5 [coding]** and another for **Assignment 5 [written]**:

1. 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
2. Run the collect_submission.sh script to produce your assignment5.zip file.
3. Upload your assignment5.zip file to GradeScope to **Assignment 5 [coding]**.
4. Check that the public autograder tests passed correctly. In particular, check that the BLEU scores calculated for parts 1(j), 2(d) and 2(e) match what you expect, because points will be awarded based on these autograder-computed numbers.
5. Upload your written solutions to GradeScope to **Assignment 5 [written]**. Tag it properly!