CS 224n: Assignment #5

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.
- We also have **new policies** on how TAs can help students in Office Hours and on Piazza – see Piazza announcement.

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.

[1] We 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).

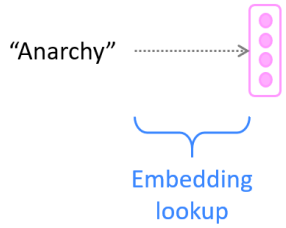

Figure 1: Lookup-based word embedding model from Assignment 4, which produces a word embedding of length e_{word}.

In this section, we will first describe a method based on Kim et al.'s work in *Character-Aware Neural Language Models*,[2] 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.

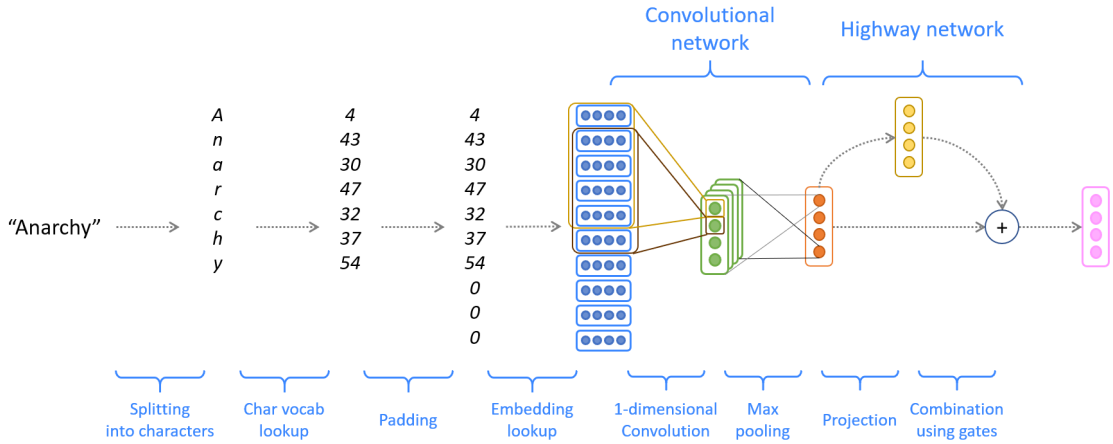

Figure 2: Character-based convolutional encoder, which ultimately produces a word embedding of length e_{word}.

Model description 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 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.

[2] 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.[3]

3. **Convolutional network**. To combine these character embeddings, we'll use 1-dimensional convolutions. The convolutional layer has two hyperparameters:[4] 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 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] $(\mathbf{x}_{\text{reshaped}})_{[:,\, t:t+k-1]} \in \mathbb{R}^{e_{\text{char}} \times k}$ 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:

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

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 $t \in \{1, \ldots, m_{\text{char}} - k + 1\}$. 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 $\text{MaxPool}(M)_i = \max_{1 \le j \le b} M_{ij}$ 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 $\mathbf{x}_{\text{conv\_out}} \in \mathbb{R}^{e_{\text{word}}}$, we compute:

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

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

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:

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

[3] Necessary because the PyTorch Conv1D function performs the convolution only on the *last* dimension of the input.
[4] We assume no padding is applied and the stride is 1.
[5] 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 $\{t, t + 1, \ldots, t + k - 1\}$.
[6] 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.

(a) (1 point) (written) In Assignment 4 we used 256-dimensional word embeddings ($e_{\text{word}} = 256$), while in this assignment, it turns out that a character embedding size of 50 suffices ($e_{\text{char}} = 50$). In 1-2 sentences, explain one reason why the embedding size used for character-level embeddings is typically lower than that used for word embeddings.

(b) (1 point) (written) Write down the total number of parameters in the character-based embedding model (Figure 2), then do the same for the word-based lookup embedding model (Figure 1). Write each answer as a single expression (though you may show working) in terms of e_{char}, k, e_{word}, V_{word} (the size of the word-vocabulary in the lookup embedding model) and V_{char} (the size of the character-vocabulary in the character-based embedding model).
Given that in our code, $k = 5$, $V_{\text{word}} \approx 50,000$ and $V_{\text{char}} = 96$, state which model has more parameters, and by what factor (e.g. twice as many? a thousand times as many?).

(c) (2 points) (written) In step 3 of the character-based embedding model, instead of using a 1D convnet, we could have used a RNN instead (e.g. feed the sequence of characters into a bidirectional LSTM and combine the hidden states using max-pooling). Explain one advantage of using a convolutional architecture rather than a recurrent architecture for this purpose, making it clear how the two contrast. Below is an example answer; you should give a similar level of detail and choose a different advantage.
When a 1D convnet computes features for a given window of the input, those features depend on the window only – not any other inputs to the left or right. By contrast, a RNN needs to compute the hidden states sequentially, from left to right (and also right to left, if the RNN is bidirectional). Therefore, unlike a RNN, a convnet's features can be computed in parallel, which means that convnets are generally faster, especially for long sequences.

(d) (4 points) (written) In lectures we learned about both max-pooling and average-pooling. For each pooling method, please explain one advantage in comparison to the other pooling method. For each advantage, make it clear how the two contrast, and write to a similar level of detail as in the example given in the previous question.

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:

```
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:

```
sh run.sh vocab
```

[7] available on Stanford Box; requires Stanford login

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

(e) (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 1e
```

(f) (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 $m_{\text{word}} = 21$, 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 1f
```

(g) (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!

(h) (3 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 $\mathbf{x}_{\text{conv\_out}}$ to $\mathbf{x}_{\text{highway}}$.
- 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 3 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.*

[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.

(i) (3 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 $\mathbf{x}_{\text{reshaped}}$ to $\mathbf{x}_{\text{conv\_out}}$.
- 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 $k = 5$.

As in (h), 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 (h), the 3 points are for your written description only.

(j) (9 points) (coding) Write the `__init__()` and `forward()` functions for the `ModelEmbeddings` class in `model_embeddings.py`.[9]

- The `forward()` function must map from $\mathbf{x}_{\text{padded}}$ to $\mathbf{x}_{\text{word\_emb}}$ – 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).
- 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()`.

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

```
python sanity_check.py 1j
```

The majority of the 9 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 (h) and (i), before you do the 'small training run' check in (l).

(k) (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!

(l) (3 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 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.

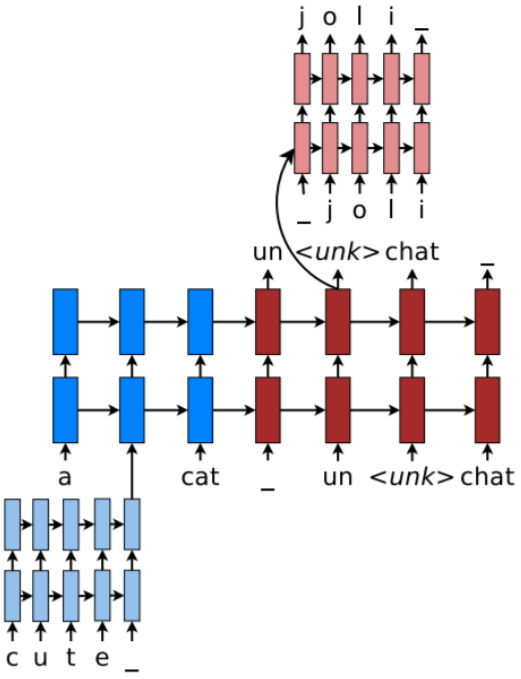

Figure 3: A character-based decoder which is triggered if the word-based decoder produces an UNK. Figure courtesy of Luong & Manning.

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.

[10] 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 $[x_1, \ldots, x_n] =$ [<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,<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>)

procedure DECODE_GREEDY
  `output_word` $\leftarrow$ []
  `current_char` $\leftarrow$ <START>
  for $t = 0, 1, ..., \texttt{max\_length} - 1$ **do**
    $\mathbf{h}_{t+1}, \mathbf{c}_{t+1} \leftarrow$ CharDecoder(`current_char`, $\mathbf{h}_t, \mathbf{c}_t$) ▷ use last predicted character as input
    $\mathbf{s}_{t+1} \leftarrow \mathbf{W}_{\text{dec}}\mathbf{h}_{t+1} + \mathbf{b}_{\text{dec}}$ ▷ compute scores
    $\mathbf{p}_{t+1} \leftarrow \text{softmax}(\mathbf{s}_{t+1})$ ▷ compute probabilities
    `current_char` $\leftarrow \text{argmax}_c \ \mathbf{p}_{t+1}(c)$ ▷ the most likely next character
    if `current_char`=<END> **then**
      break
    `output_word` $\leftarrow$ `output_word` + [`current_char`] ▷ append this character to output word
  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.**

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) (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,[11] execute:

```
sh run.sh train
```

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

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
```

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:

BLEU Score	Points
$0 \leq b < 21$	0
$21 \leq b < 22.5$	2
$22.5 \leq b$	6

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'.[12]

Infinitive	traducir	to translate
Present	traduzco	I translate
	traduces	you translate
	traduce	he or she translates
Subjunctive	traduzca	that I translate
	traduzcas	that you translate

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 a 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/. The website by default shows data from Word2Vec. Look at the nearest neighbors of the following words (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!

[12] You can check http://www.spanishdict.com/conjugate/traducir for a more complete table.

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/`, 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 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. Briefly describe 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(f) 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(l), 2(e) and 2(f) 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!