XCS224N Problem Set 5 Character-based Neural Machine Translation with LSTMs

---

Due Sunday, April 25 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs224n-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. Your code will be autograded online using `src/grader.py`, which is provided for you in the `src/` subdirectory. You can also run this autograder on your local computer, although some of the tests will be skipped (since they require the instructor solution code for comparison).

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These unit tests will verify only that your code runs without errors on obvious test cases. These tests do not require the instructor solution code and can therefore be run on your local computer.
- `hidden`: These unit tests will verify that your code produces correct results on complex inputs and tricky corner cases. Since these tests require the instructor solution code to verify results, only the setup and inputs are provided. When you run the autograder locally, these test cases will run, but the results will not be verified by the autograder. When you run the autograder online, these tests will run and you will receive feedback on any errors that might occur.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

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 basic tests that we provide are almost all extremely simple (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 basic test 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

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).

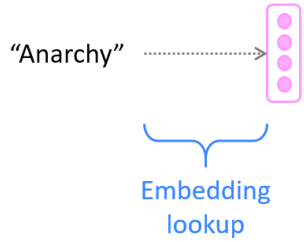

Figure 1: Lookup-based word embedding model from Assignment 4, which produces a word embedding of length e_{word}.

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.

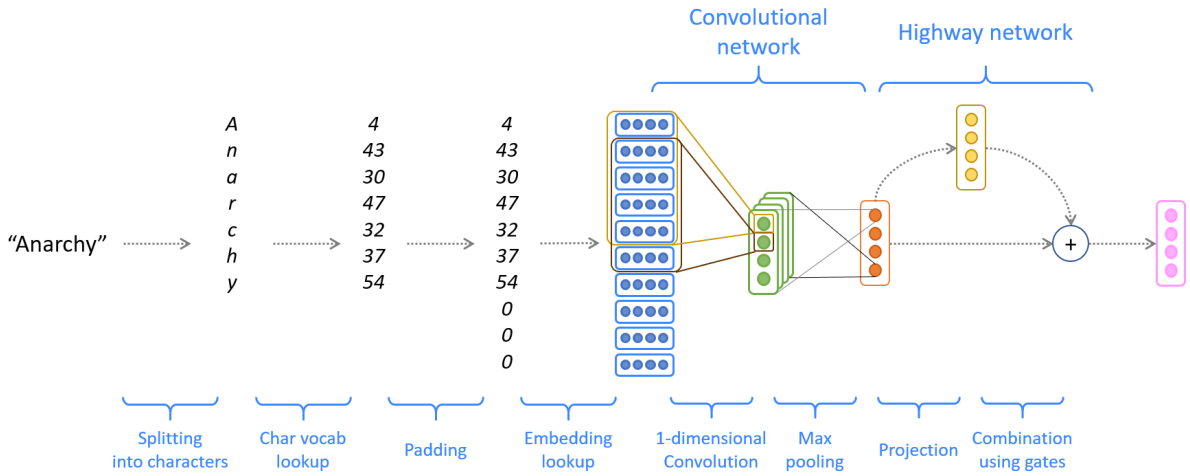

Figure 2: Character-based convolutional encoder, which ultimately produces a word embedding of length e_{word}.

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

Model description

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

(a) **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.

(b) **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]

(c) **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 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[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:

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

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 $t \in \{1, \ldots, m_{\text{word}} - k + 1\}$. 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 $\text{MaxPool}(M)_i = \max_{1 \le j \le b} M_{ij}$ for $i \in \{1, \ldots, a\}$.

[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 $\{t, t + 1, \ldots, t + k - 1\}$.

(d) **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:

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

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

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:

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

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.

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

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.

Run the following to create the correct vocab files:

```
(XCS224N)$ sh run.sh vocab
```

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

(a) **[2 points (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) **[5 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 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) 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 $\mathbf{x}_{\text{conv\_out}}$ 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.

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.

[6] available on Stanford Box; requires Stanford login

[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.

- 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) 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 $k = 5$.

(f) **[13 points (Coding)]** Write the `__init__()` and `forward()` functions for the `ModelEmbeddings` class in `model_embeddings.py`.[8]

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

```
(XCS224N)$ sh run.sh train_local_q1
(XCS224N)$ 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

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.

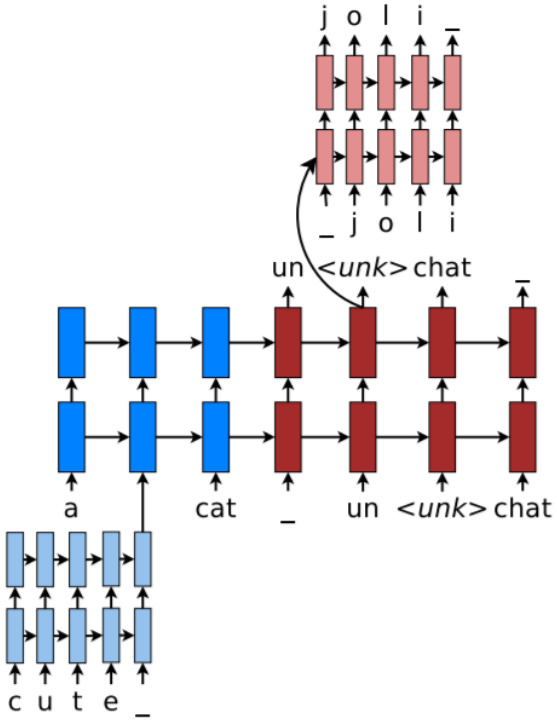

Figure 3: A character-based decoder which is triggered if the word-based decoder produces an UNK. Figure courtesy of Luong & Manning.

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.

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

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

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 $t = 0, 1, ...,$ `max_length` $- 1$ **do**
    $\mathbf{h}_{t+1}, \mathbf{c}_{t+1} \leftarrow \text{CharDecoder}(\texttt{current\_char}, \mathbf{h}_t, \mathbf{c}_t)$ ▷ use last predicted character as input
    $\mathbf{s}_{t+1} \leftarrow \mathbf{W}_{\text{dec}}\mathbf{h}_{t+1} + \mathbf{b}_{\text{dec}}$ ▷ 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
    end if
    `output_word` $\leftarrow$ `output_word` $+$ [`current_char`] ▷ append this character to output word
  end for
  return `output_word`
end procedure

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) **[3 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) **[4 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) **[6 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) **[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 sanity_check.py 2d
```

(e) **[3 points (Coding)]** (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.

```
(XCS224N)$ sh run.sh train_local_q2
(XCS224N)$ 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) **[4 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 activate the environment that you created for Assignment 4:

```
$ conda activate XCS224N
(XCS224N)$
```

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:

```
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
$0 \leq b < 10$	0
$10 \leq b < 16$	2
$16 \leq b$	6

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

3 Quiz

This remainder of this homework is a series of multiple choice questions related to the word2vec algorithm.

How to submit: Even though these are not coding questions, you will submit your response to each question in the `src-quiz/submission.py` file. This file will act as your 'bubble sheet' for multiple choice questions in this course. A sample response might look like this:

```
def multiple_choice_1a():
  """
  # Return a python collection with the option(s) that you believe are correct
  # like this:
  # `return ['a']`
  # or
  # `return ['a', 'd']`
  response = []
  ### START CODE HERE ###
  ### END CODE HERE ###
  return response
```

If you believe that `a` and `b` are the correct responses to this question, you will type `response = ['a', 'b']` between the indicated lines like this:

```
def multiple_choice_1a():
  """
  # Return a python collection with the option(s) that you believe are correct
  # like this:
  # `return ['a']`
  # or
  # `return ['a', 'd']`
  response = []
  ### START CODE HERE ###
  response = ['a', 'b']
  ### END CODE HERE ###
  return response
```

How to verify your submission: You can run the student version of the autograder locally like all coding problem sets. In the case of this problem set, the helper tests will verify that your responses are within the set of possible choices for each question (e.g. the helper functions will flag if you forget to answer a question of if you respond with `['a', 'd']` when the choices are `['a', 'b', 'c']`.) See the front pages of this assignment for instructions to run the autograder.

Note: *To answer Questions 1 and 2, review and refer to the notes and overview on Page 2-4 of the Assignment 5 PDF handout.*

1. **[2 points]**

   In Assignment 4 we used 256-dimensional word embeddings ($e_{\text{word}} = 256$). In the Assignment 5 handout you will see that a character embedding size of 50 suffices ($e_{\text{char}} = 50$).

   Select all of the options that explain why the embedding size used for character-level embeddings is typically lower than that used for word embeddings:

   (a) Words have richer semantic meaning than characters, thus they need higher dimensions to be represented.

   (b) There are far fewer unique characters than unique words, thus fewer character relationships to represent and fewer features required to capture the difference in meaning between characters.

   (c) Having a large embedding size of character-level embedding will result in an underfit model.

2. Compute the total number of parameters in the character-based embedding model presented below (first image), then do the same for the word-based lookup embedding model (second image), given that in our code, $k = 5$, $V_{\text{text}} = 50000$, $V_{\text{char}} = 96$, $e_{\text{word}} = 256$ and $e_{\text{char}} = 50$:

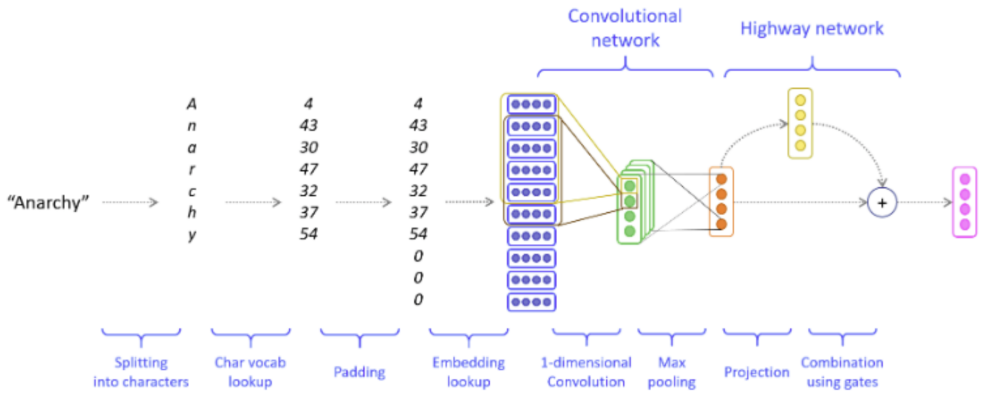

Figure 2: Character-based convolutional encoder, which ultimately produces a word embedding of length e_{word}.

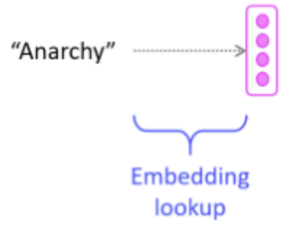

Figure 1: Lookup-based word embedding model from Assignment 4, which produces a word embedding of length e_{word}.

2a. **[2 points]**

Total parameter for character-based embedding model

(a) $n_{\text{paramschar}} = 200640$

(b) $n_{\text{paramschar}} = 12800000$

(c) $n_{\text{paramschar}} = 134336$

2b. **[2 points]**

Total parameter for word-based lookup embedding model

(a) $n_{\text{paramschar}} = 200640$

(b) $n_{\text{paramschar}} = 12800000$

(c) $n_{\text{paramschar}} = 134336$

3. **[2 points]**

   In lectures, we learned about both max-pooling and average-pooling. Choose all the options that are true regarding pooling (either max or average) operation:

   (a) **Max pooling** is able to pick up on sparse signals and retain their information, even over large window sizes. By contrast, in average pooling, information from sparse signals is diluted or lost, especially over large window sizes.

   (b) **Max pooling** is computationally quicker because the gradient only flows through the indices where max occurs, whereas in average pooling gradient flows through all indices.

   (c) **Average pooling** is good at representing the overall strength of a feature in the input – by contrast, max pooling is good at being a detector for whether a feature appears somewhere in the input.

   (d) **Max pooling** reduces the effect of extremes and outliers – by contrast, these are highlighted by average pooling.

4. **[2 points]**

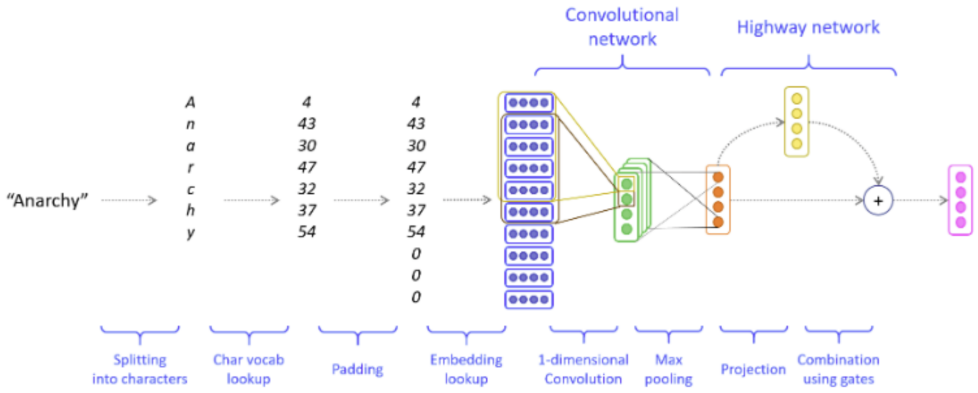

Figure 2: Character-based convolutional encoder, which ultimately produces a word embedding of length e_{word}.

In the character-based embedding model, instead of using a 1D convnet we could have used an RNN instead (e.g. feed the sequence of characters into a bidirectional LSTM and combine the hidden states using max-pooling). Choose all the options that explain how char-level CNNs and char-level RNNs differ from each other:

(a) The features computed by the CNNs are position-invariant – for example if the input word is 'couscous', and the kernel size is 4, a feature that's computed over the first 'cous' is the same as the same feature that's computed over the second 'cous'. By contrast, an RNN would have a different hidden state at the end of the first 'cous' and at the end of the second 'cous'.

(b) 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, an 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.

(c) Unlike CNNs with long window size, RNNs don't suffer from the vanishing gradient problem and, therefore, can better model the dependency between distant characters in the sequence.w

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

THERE IS NO WRITTEN SUBMISSION FOR THIS ASSIGNMENT.

YOU ARE NOT REQUIRED TO SUBMIT ANYTHING.