Assignment 3 - RNNs

COMP-545 and LING-782/484

Note: please submit your code file as "code.py" to Gradescope.
Note 2: if you get a "missing properties" error, please make sure you've named the variables/layers/etc. exactly as specified in the instructions.

1. Character RNN (CharRNN) (60 pts)

In this part of the assignment, you will implement a character-based generative RNN-based language model. The model will accept a stream of characters and learn to generate a distribution of the next character based on the previous context, over the model's vocabulary (individual characters instead of words as you may be used to). Over the course of the training you'll observe the model learning English from scratch, how to form words, sentences, how to conjugate verbs, etc. By the end of this assignment you should be convinced about the **Unreasonable Effectiveness of RNNs** :)

Note for the whole assignment: do NOT set random seeds for PyTorch, this will potentially break our testing code.

1.1 Implement the CharSeqDataloader class (10 pts)

CharSeqDataloader.__init__ (3 pts)

Note:

- You need to load in the file in this method, and process the entire dataset into a list of characters
- You are **not meant to** use Pytorch's **DataLoader** class in this part of the assignment. DataLoader is meant to be used in Part 3. Everything should be doable with basic built-in operations that the Python runtime provides you (e.g. file.read(), etc.)
- You need to set up character-to-index and index-to-character dictionaries as instance variables using **generate_char_mappings** that you will subsequently write (**self.mappings**)
- You will need to process the entire dataset into indices using the **convert_seq_to_indices** method you will subsequently write
    - It would be useful to pack the dataset into a tensor and hand it off to the GPU using .to(device) at this point
- You can initialize **any instance variables you might need later** in __init__.
- Set up **unique_chars** as a **list** of unique characters in the dataset. You can use the **set** datatype, but make sure the variable is a list.

- **Initialize all the provided variables in __init__. If you're stuck as to what one might mean, think logically about what it might be needed for later. E.g. what could "vocab_size" be? What is our "vocabulary" in this case?**

Parameter	Type	Description
filepath	string	The filepath for the dataset
seq_len	int	The sequence length (set as an instance variable to be accessed later)
examples_per_epoch	int	How many examples should our class yield for a single epoch

CharSeqDataloader.generate_char_mappings (2 pts)

This returns a dictionary with two keys, one mapping characters to their corresponding sequential indices, and one mapping the indices back to characters.

Parameter	Type	Description
unique_chars	List[str]	The list of unique characters to create mappings for.

Key of Dict	Is (type)	Description
"char_to_idx"	Dict[str, int]	A dict mapping characters to their corresponding indices
"idx_to_char"	Dict[int, str]	A dict mapping indices to their corresponding characters

CharSeqDataloader.convert_seq_to_indices and convert_indices_to_seq (2 pts)

These two methods simply use your character mappings generated previously to map an entire sequence back and forth.

Note: do NOT modify the input sequence in-place. This may lead to weird behavior with the automated test.

convert_seq_to_indices

Parameter	Type	Description
seq	List[str]	The input sequence

Returns	Description
List[int]	The output sequence

convert_indices_to_seq

Parameter	Type	Description
seq	List[int]	The input sequence

Returns	Description
List[str]	The output sequence

CharSeqDataloader.get_example (3 pts)

This method is a generator function (if you're not familiar with generators in Python familiarize yourself now) that will return individual examples of size seq_len from the dataset. The input sequence will be a random consecutive sequence of characters (size seq_len) taken from our dataset, and the target_seq for the RNN to predict will be the input_seq shifted over by 1. You will be slicing into your list of character **indices** that represent the entire dataset that you generated in this class's __init__. You must **yield** these two values in a loop of **examples_per_epoch** times.

Name	Returns	Description
in_seq	Tensor of **ints**	The input sequence we will pass to the RNN
target_seq	Tensor of **ints**	Our target sequence for the RNN to predict (input sequence shifted by 1)

1.2 Implementing CharRNN (30 pts)

In this section you will be implementing the CharRNN itself, a single-layer basic RNN.

CharRNN.__init__ (3 pts)

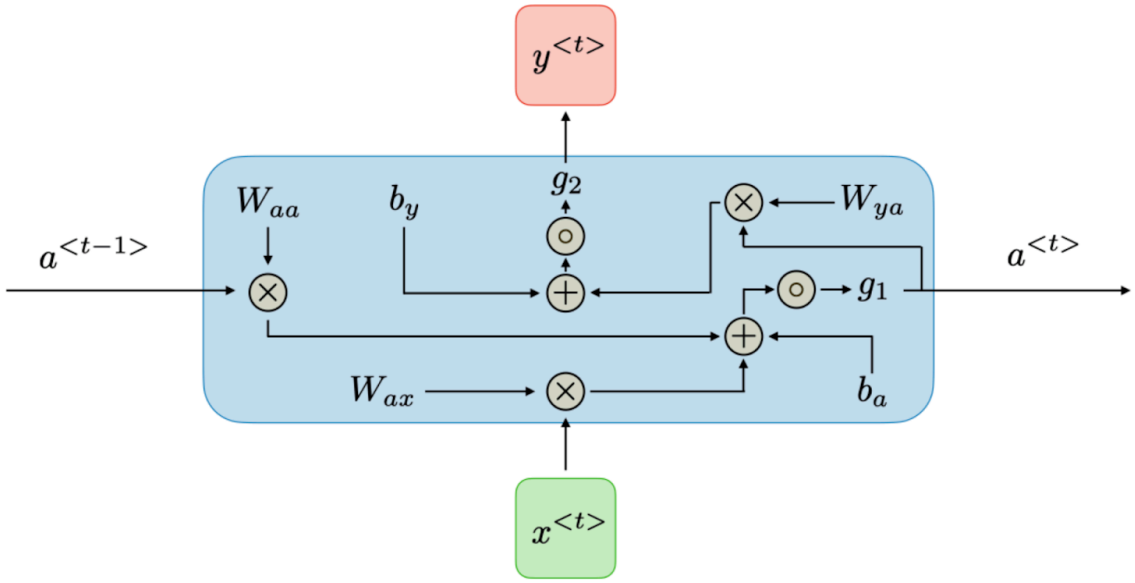
In the init method, you will be initializing the layers necessary for the RNN.
Note:

- Use **nn.Embedding** to map your character indices to a learned embedding representation.
- Use **nn.Linear** layers for the different weight matrices you need for the RNN cell. Set **bias=False** as necessary to mimic the diagram of the RNN cell given below. You should be able to map the weight matrices shown in the diagram to PyTorch Linear layers pretty cleanly. If you are stuck, use the equations. Remember: **each linear layer corresponds to one (1) weight matrix** (and **potentially** a bias term).
- More details on this equivalence:
  - $$W_{\text{linear}}x + b_{\text{linear}} = \text{linear}(x)$$
  - A Pytorch Linear layer bundles the weight matrix and the bias term into a single unit. Passing “bias=False” removes the bias part of the above equation. You shouldn’t need to use “matmul” or “add” at all in your implementation, nor to have the bias as a separate parameter. **All you should need are regular nn.Linear layers. Expressed differently:** `torch.matmul(W, x) + b = self.linear(x)`
  - We have a single equation with three additive terms. Think about how you can decompose this into regular layer application operations using the above information.
- **Use “wax”, “wya” and “waa” for the layers corresponding to these parts in the diagram as variable names. Use “embedding_layer” as the embedding layer name.**
- Your output should be a **distribution over the set of characters in the CharRNN’s vocabulary**. This thought should help you with the dimensions of **wya**.

RNN cell diagram and equations:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$



Parameter	Type	Description
n_chars	int	The number of unique characters the model has. This is important for the **nn.Embedding layer.**
hidden_size	int	The size of the hidden state for the RNN cell. Important for initializing the **nn.Linear layers.**

CharRNN.rnn_cell (4 pts)

Implement the RNN cell directly. Use the RNN cell diagram above to guide you, applying the **nn.Linear layers** that you initialized before to the right inputs and returning the right output(s). If you are doing this correctly, it should be relatively few lines of code. Make sure you are using **torch.tanh** for your **activation function** for the hidden state (**g_1**). Do **not softmax your output (g_2 in the diagram above)**, as we will be using a loss that includes the softmax operation. This means your cell should have **only a single activation function**.

Parameter	Type	Description
i	Tensor[emb_size] of floats	The tensor with the input for this timestep (should be a tensor representing a single character)
h	Tensor[hidden_size] of floats	The tensor representing the previous hidden state.

Returns	Type	Description
o	Tensor[n_char	The output of the RNN cell, in our case a distribution over

	s] of floats	our character vocabulary.
h_new	Tensor[hidden_size] of floats	Our new hidden state.

CharRNN.forward (5 pts)

Implement the forward function using your rnn_cell function you just wrote. The forward function should, in the following order:

- Pass the whole input sequence through the **nn.Embedding** layer
- Iterate the number of times we want to generate a new character (for loop is fine)
    - "Number of times" => length of input. We want to return a sequence of the same length.
    - Apply the rnn_cell this number of times, passing each corresponding element of the input, as well as the **previous/current hidden state**, with our function returning the **next hidden state** for the next step of the for loop.
    - Collect the output of the rnn_cell at each timestep
    - Stack the output into a single tensor of the length of our sequence **at the end** in a single **torch.stack operation**. This is the simplest way to generate our new sequence. Make sure the dimensions of the output are correct (based on the below tables).
- Finally, return the aggregated output (complete sequence) and final hidden state after our loop

Parameter	**Type**	**Description**
input_seq	Tensor[input_len] of indices (ints)	The tensor containing the complete input sequence (we want to be able to pass the output of get_example directly to this)
hidden	Tensor[hidden_size] of floats or None	The tensor representing the hidden state. **If None, feel free to initialize it in this method, or make an init_hidden helper method for the class. The hidden state should be initialized to all zeros (use torch.zeros) in the beginning.**

Returns	**Type**	**Description**
out	Tensor[seq_len, vocab_size]	The full output sequence of our single-layer RNN, consisting of a distribution over our character vocabulary * seq_len.
hidden_last	Tensor[hidden_size] of floats	Our last hidden state after having iterated over the whole sequence.

top_k_filtering (5 pt)

Implement the concept of top-K filtering as specified in the course. Set the values in the tensor that don't meet the criteria to **negative infinity (float('-inf'))**. My recommended approach is to use a **tensor mask.** Make sure to **return the filtered logits.**

Parameters:

Logits: tensor of [batch_size, vocab_size]. Make sure your implementation is operating along the right dimensions!

top_p_filtering (5 pt)

Implement the concept of top-P filtering as specified in the course. Set the values in the tensor that don't meet the criteria to negative infinity. Keep the first token above the threshold. **Hint: use torch.cumsum**.

CharRNN.sample_sequence (6 pts)

In this method, you will repeatedly random sample from your RNN, feeding back to itself its own output, to generate a whole natural language sequence. You will also implement a temperature parameter, which if you recall is simply a division operation on the logits of our network.

$$q_i = \frac{\exp\left(z_i/T\right)}{\sum_j \exp\left(z_j/T\right)}$$

Notes:

- There are **two ways** to implement this method. You can use your **forward** method, or you alternatively can use your **rnn_cell directly**.
    - If you choose to use forward, make sure you **squeeze** and **unsqueeze** appropriately since **forward** expects a **sequence of characters, not a single character.**
- You will need to use **torch.nn.functional.softmax (abbreviated as F.softmax if you import as F).** As temperature is an operation on the **logits** of the network, you need to do the division operation **before calling softmax. It's simpler than you think, don't overthink this part**. Softmax also requires a **dim** parameter, make sure you are applying softmax on the right dimension (depending on using **forward or rnn_cell**)
- You will use **torch.distributions.Categorical** for sampling randomly based on your RNN's output distribution. Read more here if you need to.
- Make sure you are **including the starting_char in your output**, as this is what the test expects.

Parameter	Type	Description

starting_char	A char **index**	The character we will start generating from, as a character index. This is necessary as we have not trained our RNN with a start of sequence token, thus we must pick the first character ourselves.
seq_len	int	A parameter representing how long our sampled sequence should be. Essentially, how many iterations of our sampling **for** loop.
temp	float, default 0.5	Our temperature parameter which controls the diversity of our generation.
top_p	Float, default to None	Our top_p parameter. Use the function you implemented previously only if not None.
top_k	Int, default to None	Our top_k parameter. Use the function you implemented previously only if not None.

Returns	**Type**	**Description**
generated_seq	List[int]	The full generated output sequence from our RNN. This should be a sequence of character indices.

CharRNN.get_loss_function (1 pt)

Return the correct loss function for this task. Make sure you use the loss function provided by PyTorch that **incorporates softmax already**, for numerical stability. This is **not** NLLLoss (it's CrossEntropyLoss).

CharRNN.get_optimizer (1 pt)

Return the Adam optimizer for this task, initialized correctly with your RNN's parameters. Accept an "lr" parameter for the learning rate.

1.3 Training (not autograded, needed for REPORT) (10 pts)

Some of the training code is provided. Fill in the blanks, specifically converting the output of our sample_sequence method back to natural language and printing it.

Notes:

- **Make sure any tensors you make are moved to the GPU with .to(device) where device is "torch.device("cuda" if torch.cuda.is_available() else "cpu")"**
- **Make sure you are passing a *new* hidden state for each training iteration, passing the same one will cause computation graph errors with Pytorch as it tries to accumulate the losses from all previous batches.**

The main loop of your training code should do the following, in this sequence:

1. Apply the RNN to the incoming sequence
2. Use the loss function to calculate the loss on the model's output
3. Zero the gradients of the optimizer
4. Perform a backward pass (calling .backward())
5. Step the weights of the model via the optimizer (.step())
6. Add the current loss to the running loss

After every epoch, you should:

1. Sample a character with some randomness (this is up to you, you could sample from the dataset or you could sample uniformly from unique characters, or only capital letters, or any other variation).
2. Print the result of the sampling to the output, so you can monitor the process of the training. **Over time your generated text should become more and more "believable"**. If not, something may be wrong with your implementation.

For the REPORT, please do the following:

1. Train your CharRNN on the Sherlock Holmes dataset provided. Include 3-5 samples generated from the model once you are reasonably confident in your model's modelling abilities. Show how the temperature parameter affects the output. (Give some samples with low, medium, high temperatures). **Try to get the best results possible from the training.**
    a. Additionally, give samples for at least 2 top_k values and at least 2 top_p values. Try to demonstrate the difference between the two as best as you can.
2. Train your CharRNN on the Shakespeare dataset provided. Do the same as above.

2. Character LSTM (CharLSTM) (30 pts)

2.1 Implementing CharLSTM (15 pts)

In this section you will be implementing the CharLSTM, a single-layer basic LSTM.

CharLSTM.__init__ (5 pts)

In the init method, you will be initializing the layers necessary for the LSTM. **Most things should remain constant from your previous code, except LSTM specific layers.**

Note:

- Use **nn.Embedding** to map your character indices to a learned embedding representation.

- Use **nn.Linear** layers for the different weight matrices and **gates** present in the LSTM cell. In total, you should have **5 different nn.Linear layers**. **Three of these will correspond to learned gate weights. Use "fc_output" for the layer that gives you the final output as a name.**
    - **The output of the cell should be a function of the *hidden state,* just as it was in the simple RNN cell (see the relevant equation).**
    - **This is the purple "h_t" coming out from the top of the cell. This is what is transformed into our output.**
- **The names of each of the gate layers should be {forget/input/output}_gate. Use "cell_state_layer" as the name for the remaining layer that has not been mentioned previously.**
- Each "_gate" instance variable should be an instance of **nn.Linear**. You will apply the activation in your **lstm_cell method**.
- **Note regarding the tests: the tests expect you to concatenate input with hidden state, not the other way around. In real-life contexts these should be functionally equivalent.**
- Try not to be overwhelmed by the number of equations. Each equation should map 1-to-1 or almost-1-to-1 to a single line of PyTorch code. Feel free to save repeated operations into a variable (eg. the result of the concatenation **[h_{t-1}, x_t]** is used in every single **gate**. Feel free to **torch.concat** it into a variable to reuse it more easily)
- As with before, do **not** apply the final activation (softmax), as it is part of the loss function.

LSTM cell diagram and equations:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

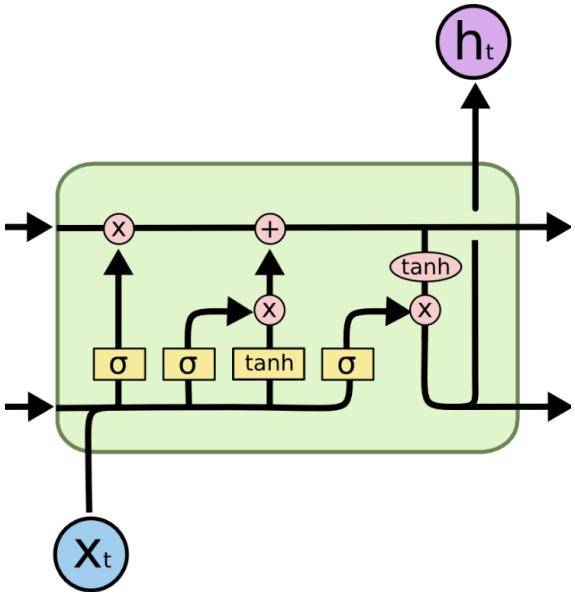
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

$$y_t = g(W_y h_t + b_y)$$



CharLSTM.lstm_cell (6 pts)

Implement the LSTM cell directly. Use the LSTM cell diagram above to guide you, applying the **nn.Linear layers** that you initialized before to the right inputs and returning the right output(s). If you are doing this correctly, it should be relatively few lines of code. Make sure you are using **torch.tanh** for your activation function for the candidate cell state, and make sure you are **torch.sigmoid**-ing **all the gate weight matrix product output (output of passing the concatenation of previous state and input through the gate's nn.Linear layer)**.

Parameter	Type	Description
i	Tensor[emb_size] of floats	The tensor with the input for this timestep (should be a tensor representing a single character)
h	Tensor[hidden_size] of floats	The tensor representing the previous hidden state.
c	Tensor[hidden_size] of floats	The tensor representing the previous cell state

Returns	Type	Description
o	Tensor[n_chars] of floats	The output of the LSTM cell, in our case a distribution over our character vocabulary.
h_new	Tensor[hidden_size] of floats	Our new hidden state.
c_new	Tensor[hidden	Our new cell state.

	_size] of floats	

CharLSTM.forward (2 pts)

Implement the forward function using your **lstm_cell method** you just wrote. The forward function for the LSTM should involve **relatively minor tweaks from your CharRNN forward method,** primarily involving **handling and passing a third parameter to the cell: the cell state.** Otherwise the code should be practically identical.

Parameter	**Type**	**Description**
input_seq	Tensor[input_len] of indices (ints)	The tensor containing the complete input sequence (we want to be able to pass the output of get_example directly to this)
hidden	Tensor[hidden_size] of floats or None	The tensor representing the hidden state. **If None, feel free to initialize it in this method, or make an init_hidden_and_cell helper method for the class. The hidden state should be initialized to all zeros (use torch.zeros) in the beginning.**
cell	Tensor[hidden_size] of floats or None	The tensor representing the cell state. **If None, feel free to initialize it in this method, or make an init_hidden_and_cell helper method for the class. The cell state should be initialized to all zeros (use torch.zeros) in the beginning.**

Returns	**Type**	**Description**
out_seq	Tensor[seq_len, vocab_size]	The full output sequence of our single-layer RNN, consisting of a distribution over our character vocabulary * seq_len. Depending on how you've structured your code, you might need to **squeeze()** the result
hidden_last	Tensor[hidden_size] of floats	Our last hidden state after having iterated over the whole sequence.
cell_last	Tensor[hidden_size] of floats	Our last cell state.

CharLSTM.sample_sequence (2 pts)

Tweak your CharRNN.sample_sequence slightly to account for the new parameter being passed to the forward method/LSTM cell (the cell_state). This should be **changing a few lines of code only**. Your new sample_sequence should also accommodate for top_p and top_k, you should be able to use the functions as-is with the LSTM as well.

Get_optimizer and get_loss_function (ungraded)

- Necessary for your model. Make sure to transfer them over from Part 1. Should not need to be modified.

2.2 Training (not autograded, needed for REPORT) (15 pts)

For the training part of the report, you should be able to reuse the training harness you wrote for part 1.

For the REPORT, please do the following:

1. Train your CharLSTM on the Sherlock Holmes dataset provided. Include 3-5 samples generated from the model once you are reasonably confident in your model's modelling abilities.
2. Train your CharLSTM on the Shakespeare dataset provided. Do the same as above.
3. Note some observations regarding training your CharRNN vs. CharLSTM. Is training faster or slower? How does the training loss compare? **Graph the loss**. Is the final model better or worse at language modeling, and in what way? Any specific strengths or weaknesses you can observe for each model?

3. Shallow BiLSTM for SNLI dataset (20 pts)

In this section of the assignment you will be working with the SNLI (Stanford Natural Language Inference) dataset. This dataset is composed of "premises" and "hypotheses", and the model needs to determine if the one sentence **entails, contradicts, or is neutral towards** the second. This makes it a 3-way classification task.

Note: I recommend reading all the instructions for Part 3 initially to get a sense of what we're trying to do. Parts of 3.1 assume knowledge of the architecture, so familiarize yourself with it first before starting the section.

3.1 Utility functions (10 pts)

fix_padding (3 pts)

This method should pad all the inputs so that they all have the same size **for that input only**, based on the largest sized single input point **for that collection of inputs (e.g. batch_premises)**. This means that you do **not need to pad the premises to have the same dimensions as the hypotheses, as long as the tensors are internally padded to the largest datapoint for each.** This is to allow us to feed both premises and hypotheses easily to the ShallowBiLSTM network. This method should also return **reversed and padded versions of its two inputs**. This is because we need our input reversed for our **backward LSTM**, but **also padded properly**. The normal and reversed matrices for a batch with the three sequences [1,2], [3], and [4,5,6] should look like this (rows are the data points):

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 0 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & 0 & 0 \\ 6 & 5 & 4 \end{bmatrix}$$

And you need to do this once for the first parameter (the premises), and once for the second (the hypotheses).

Luckily, Pytorch provides us a function for doing this for a single set of datapoints, which is “torch.nn.utils.rnn.pad_sequence”. You need to make sure you run this 4 times (once on the regular premises and hypotheses, once for the reversed premises and hypotheses). This function takes as input a **List of tensors (where each tensor is a single example),** and gives you back the **stacked matrix**, so make sure to **convert to tensors** as you go. Make sure to pass the correct value of **the batch_first parameter.**

Parameter	**Type**	**Description**
batch_premises	List[batch_size, num_words]	Batch of premises
batch_hypotheses	List[batch_size, num_words]	Batch of hypotheses

Returns	**Type**	**Description**
batch_premises	Tensor[batch_size, num_words] of indices	Batch of premises
batch_hypotheses	Tensor[batch_size, num_words] of indices	Batch of hypotheses
batch_premises_reversed	Tensor[batch_size, num_words] of indices	Reversed premises
batch_hypotheses_reversed	Tensor[batch_size, num_words] of indices	Reversed hypotheses

Create_embedding_matrix (3 pts)

This method should copy over the necessary vectors (if loaded with pandas, np.ndarrays) from the total set of Glove vectors that correspond only to the words present in our corpus. This will then be used to initialize the embedding layer for our LSTM networks. **From_numpy** will convert the incoming values to tensors for you.

Parameter	**Type**	**Description**
index_map	Dict[str, int]	The word_index produced by the given function.
emb_dict	Dict[str, np.ndarray]	The dict of all embeddings
emb_dim	int	The number of dimensions for the embedding.

Returns	**Description**
Tensor[word_indices, emb_dim]	The Tensor containing the embeddings for only the words present in our SNLI corpus (a subset of the total words present in the Glove embedding file).

evaluate (4 pts)

This method should iterate over all the data contained within the dataloader, accumulating the # of correct predictions (passing the data to the model **batch by batch**, not in one go), and should ultimately return a **float (between 0 and 1)** representing the percent accuracy of the model on the given data (i.e. 0.83 for 83%).

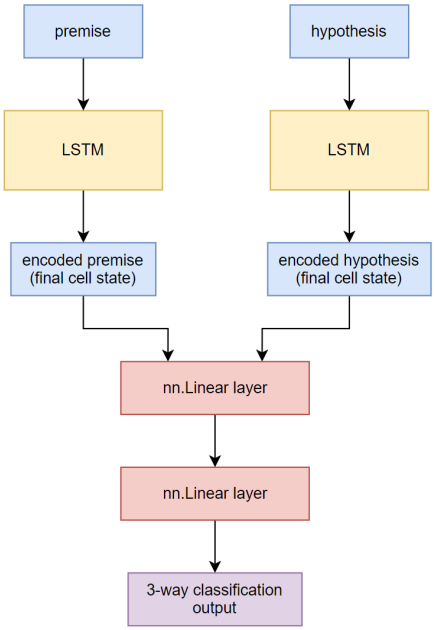
Parameter	**Type**	**Description**
model	model	A model
dataloader	dataloader	A dataloader
index_map	Dict[str, int]	The index_map you generated earlier with the given function. This is so that you can call tokens_to_ix in your evaluation loop.

Returns	**Description**
float	The percent accuracy when evaluating the model on the dataloader.

3.2 Implement the UniLSTM and ShallowBiLSTM classes (15 pts)

UniLSTM.__init__ (3 pts)

In the init method, you will be initializing the layers necessary for the UniLSTM architecture.The architecture is shown below:



The two LSTM blocks shown are actually the **same nn.LSTM module with the same parameters**.

The model should include the following layers:

- 1 nn.LSTM (called “lstm”)
- 2 nn.Linear layers (one intermediate layer, one output layer) (called “int_layer” and “out_layer”).
    - **Note: think about what the int_layer’s incoming feature size should be, considering that we will be concatenating two cell states together (see spec for “forward”). It might be easier to complete the forward method first before filling in these dimensions.**
- 1 nn.Embedding layer (called “embedding_layer”). Make sure to set padding_idx correctly. **For simplification purposes, set the embedding dimension to be the same as the hidden dimension of the LSTM**.

Note: pass "batch_first" to your LSTM, as we are putting our batches as the first dimension.

Note 2: I *highly highly* recommend reading PyTorch's documentation on nn.LSTM to understand in depth the inputs and outputs to this module, their dimensions, etc., especially if you are getting stuck and things aren't making sense.

Parameter	Type	Description
vocab_size	int	An int representing the size of the vocabulary (necessary to initialize the nn.Embedding layer)
hidden_dim	int	The size of the hidden dimension of the LSTMs (both forward and backward)
num_layers	int	The number of layers for the LSTMs
num_classes	int	The number of classes for our final prediction (in our case, 3)

UniLSTM.forward (4 pts)

In the forward method, you will be applying the layers you created previously to create the forward pass of the UniLSTM model.

Notes:

- Use **fix_padding** to fix the padding of our inputs. Simply discard the reversed versions for UniLSTM because we don't need them. **Move tensors to GPU if necessary.**
- Use the final **cell state** as the representation of each sentence. The theory here is that the final cell state reflects the internal state of the LSTM after having processed *the entire sequence*, and as such contains information about every word in the sentence. In this architecture, we are using the internal nn.LSTM as an **encoder** to create a representation of our premise and hypothesis, and then passing that representation to our fully-connected Linear layers (acting as a **decoder**) to get our class prediction. Another way to do this would be to pool the word-level outputs from the LSTM, which would likely also be effective, but we will use the cell state instead.
- **Concatenate** all the cell states and pass through the two **Linear** layers to create your prediction (with a **ReLU activation** in between as our **nonlinearity**). Make sure you are concatenating **on the correct axis.** You should be concatenating **2 different states together**

Parameter	Type	Description
a	List[batch_size, num_words] of word indices	The tensor containing the padded premises (should be a tensor of indices).
b	List[batch_size,	The tensor containing the padded hypotheses (should

	num_words] of word indices	be a tensor of indices).

Returns	Description
Tensor[batch_size, num_classes]	The output prediction as a tensor, a distribution of probabilities over our three classes (unnormalized! Do not apply softmax.)

ShallowBiLSTM.__init__ (3 pts)

In the init method, you will be initializing the layers necessary for the **ShallowBiLSTM** architecture. The ShallowBiLSTM architecture will follow a similar architecture as the UniLSTM diagram shown previously, however now you will be dealing with two LSTMs instead of one, a forward LSTM and a backward LSTM. We are naming it ***Shallow*BiLSTM** because we are dealing with two **separate** LSTMs, a forward and backward one, where there are no **layer-level linkages between them,** and simply concatenating the representations (cell states) resulting from passing our inputs through each**.** A **true** BiLSTM would have inter-layer connections as well (i.e. between layers of each LSTM). See the class slides for more details. In the **training section we will also experiment with a true BiLSTM by passing bidirectional=True to our nn.LSTM layer.**

The model should include the following layers:

- 2 nn.LSTMs, one forward unidirectional LSTM and one backward unidirectional LSTM (initialized the same way, the backward LSTM is made backwards by reversing the input) (forward “lstm_forward”, backward “lstm_backward”)
- 2 nn.Linear layers (one intermediate layer, one output layer)
- 1 nn.Embedding layer

Parameter	Type	Description
vocab_size	int	An int representing the size of the vocabulary (necessary to initialize the nn.Embedding layer)
hidden_dim	int	The size of the hidden dimension of the LSTMs (both forward and backward)
num_layers	int	The number of layers for the LSTMs
num_classes	int	The number of classes for our final prediction (in our case, 3)

ShallowBiLSTM.forward (5 pts)

In the forward method, you will be applying the layers you created previously to create the forward pass of the ShallowBiLSTM model.

Notes:

- Use **fix_padding** to get the input in padded form, as well as the reversed versions of it.
- Use the final **cell state** as the representation of each sentence.
- **Concatenate** all the cell states and pass through the two **Linear** layers to create your prediction (with our trusty **ReLU as our nonlinearity between them**). Make sure you are concatenating **on the correct axis.** You should be concatenating **4 different states together (2 cell states from the forward LSTM, and 2 states from the backward LSTM)**
    - **Concat in the order premises_forward, premises_backward, hypotheses_forward, hypotheses_backward**

Parameter	Type	Description
a	List[batch_size, num_words] of word indices	The tensor containing the padded premises (should be a tensor of indices).
b	List[batch_size, num_words] of word indices	The tensor containing the padded hypotheses (should be a tensor of indices).

Returns	Description
Tensor[batch_size, num_classes] of floats	The output prediction as a tensor, a distribution of probabilities over our three classes (unnormalized! Do not apply softmax.)

3.3 REPORT – Compare the ShallowBiLSTM class with the regular unidirectional LSTM class provided (5 pts)

- **Note: the training for these two models may take a while.**
- Load in the SNLI dataset using Pytorch's **DataLoader** class. Use batch_size=32 and shuffle=True.
- **Use create_embedding_matrix to create the embedding matrix initialized with Glove embeddings that you will use to initialize your embedding_layer. Use from_pretrained for this purpose (consult PyTorch nn.Embedding documentation if you need to). Overwrite the embedding layer of your model (outside the class, in the training code) after initializing it. Pass freeze as False and padding_idx as 0. We do not want the Glove embeddings to be frozen, as having them trainable can only boost our performance.** Due to our embedding_size == hidden_dim simplification, you need to modify your classes in the following way:

    - **Decouple the hidden_dim from the embedding_size in your model, by passing another parameter (e.g. embedding_size) to your model.__init__ method and setting it separately, to initialize your Embedding layer within __init__. This will require you to tweak the input features to the LSTM(s) as well.**
- **Make sure any tensors you make are moved to the GPU with .to(device) where device is "torch.device("cuda" if torch.cuda.is_available() else "cpu")"**
- **Use evaluate method to evaluate dataset performance on validation and test set.**
- To make a **fair comparison** between the two models, halve the hidden_dim parameter on the BiLSTM for your comparison (e.g. 100-dim for UniLSTM, 50-dim for ShallowBiLSTM, see note about Glove embeddings above). Start with the num_layers as 1. Feel free to play around with increasing it, but make sure that you're making a fair comparison always (approx same number of params overall). Increase it to beyond 1, making sure you're taking the **cell state from the final layer**.
- Compare 2 versions of each model: 1 **with Glove embeddings** and 1 **without**. What do you observe in these two cases?
- What is the difference in performance between the two models in the two cases for each? Does this correspond to what you expected? Provide a potential explanation for what you observe.
- Modify your UniLSTM (or alternatively your ShallowBiLSTM) by passing **bidirectional=True and num_layers=2** with a single LSTM initialization (modifying the incoming feature size to the Linear layers accordingly) to create a **true BiLSTM** to compare against. **Compare this model against the equivalent UniLSTM and ShallowBiLSTM**.
    - You will need to modify the concatenation as well, to concatenate **4** different cell states instead of **just 2**. The bidirectional LSTM with num_layers=2 will output a tensor of **[4, BATCH_SIZE, hidden_size]** for the final cell state. In this case, final_**cell_state[-1,:,:]** and final_**cell_state[-2,:,:]** are the now **2** outputs (forward and backward LSTMs) that we care about, similar to **ShallowBiLSTM**. The difference is that now we have a deeper network, with interconnections between layers – PyTorch takes care of this for us (**true** BiLSTM vs. **shallow** BiLSTM).
- Report final accuracy on the validation and test set for each model (of the same epoch), **with** and **without** Glove embeddings.
- Comment on the efficacy of the Glove embeddings.
- Plot accuracy over time on the validation and test set for each model, **with** and **without** Glove embeddings.
- Train the model until validation accuracy stops going up.
- Do you observe any other differences between the two models? Training time?
- You should have **5 different configurations** being compared in the end. UniLSTM with Glove, UniLSTM without Glove, ShallowBiLSTM with Glove, ShallowBiLSTM without Glove, **True BiLSTM (modified UniLSTM)**.