Homework 3 Part 1

RNNs and GRUs and CTCs and Search, Oh My!!

11-785: Introduction to Deep Learning (Spring 2021)

OUT: **October 22 2021, 12:00 AM EST**
DUE: **November 11 2021, 11:59 PM EST**
(Last Updated: October 20 2021, 10:00PM EST)

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.
- **Overview:**
  - **MyTorch**
  - **Multiple Choice**
  - **RNN**
  - **GRU**
  - **CTC**
  - **Greedy Search and Beam Search**
- **Directions:**
  - You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.
  - We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.
  - If you haven't done so, use pdb to debug your code effectively.

MyTorch

The culmination of all of the Homework Part 1's will be your own custom deep learning library, which we are naming *mytorch* © just like any other deep learning library like PyTorch or Tensorflow. The files in your homework are structured in such a way that you can easily import and reuse modules of code for yoursubsequent homeworks. For Homework 3, MyTorch will have the following structure:

NOTE: We recommend you make a backup of your hw3 files before copying everything over, just in case you break code or want to revert back to an earlier version.

```
mytorch
├── rnn_cell.py
├── gru_cell.py
├── ctc.py
├── ctc_loss.py
├── search.py
├── linear.py
├── activation.py
└── loss.py
hw3
├── hw3.py
├── rnn_classifier.py
└── mc.py
autograder
└── hw3_autograder.py
    └── runner.py
create_tarball.sh
```

- **Install** Python3, NumPy and PyTorch in order to run the local autograder on your machine:
  - pip3 install numpy
  - pip3 install torch
- **Hand-in** your code by running the following command from the top level directory, then **SUBMIT** the created handin.tar file to autolab:
  - sh create_tarball.sh
- **Autograde** your code by running the following command from the top level directory:
  - **python3 autograder/hw3_autograder/runner.py**
- **DO NOT:**
  - Import any other external libraries other than numpy, as extra packages that do not exist in autolab will cause submission failures. Also do not add, move, or remove any files or change any file names.

1 Multiple Choice [5 points]

These questions are intended to give you major hints throughout the homework. Answer these questions by returning the correct letter as a string in the corresponding question function in **hw3/mc.py**. Each question has only a single correct answer. Verify your answer by running the **local autograder**. To get any credit, you must answer **all** questions correctly.

---

(1) **Question 1: Review the following chapter linked below to gain some stronger insights into RNNs. [2 points]**

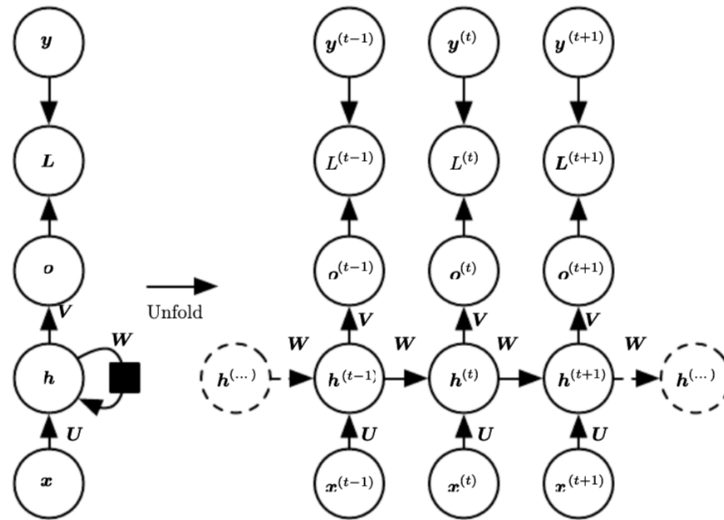

Figure 1: The high-level computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output values from http://www.deeplearningbook.org/contents/rnn.html. (Please note that this is just a general RNN, being shown as an example of loop unrolling, and the notation may not match the notation used later in the homework.)

(A) I have decided to forgo the reading of the aforementioned chapter on RNNs and have instead dedicated myself to rescuing wildlife in our polluted oceans.

(B) I have completed the optional reading of http://www.deeplearningbook.org/contents/rnn.html (Note the RNN they derive is different from the GRU later in the homework.)

(C) Gravitational waves ate my homework.

(2) **Question 2: In an RNN with N layers, how many unique RNN Cells are there? [1 point]**

(A) 1, only one unique cell is used for the entire RNN

(B) N, 1 unique cell is used for each layer

(C) 3, 1 unique cell is used for the input, 1 unique cell is used for the transition between input and hidden, and 1 unique cell is used for any other transition between hidden and hidden

(3) **Question 3: Given a sequence of ten words and a vocabulary of five words, find the decoded sequence using greedy search. [1 point]**

```
probs = [[0.1, 0.2, 0.3, 0.4],
         [0.4, 0.3, 0.2, 0.1],
         [0.1, 0.2, 0.3, 0.4],
         [0.1, 0.4, 0.3, 0.2],
         [0.1, 0.2, 0.3, 0.4],
         [0.4, 0.3, 0.2, 0.1],
         [0.1, 0.4, 0.3, 0.2],
         [0.4, 0.3, 0.2, 0.1],
         [0.1, 0.2, 0.4, 0.3],
         [0.4, 0.3, 0.2, 0.1]]
```

Each row gives the probability of a symbol at that timestep, we have 10 time steps and 5 words for each time step. Each word is the index of the corresponding probability (ranging from 0 to 4).

(A) `[3,0,3,0,3,1,1,0,2,0]`

(B) `[3,0,3,1,3,0,1,0,2,0]`

(C) `[3,0,3,1,3,0,0,2,0,1]`

(4) **Question 4: I have watched the lectures for Beam Search and Greedy Search? Also, I understand that I need to complete each question for this homework in the order they are presented or else the local autograder won't work. Also, I understand that the local autograder and the autolab autograder are different and may test different things- passing the local autograder doesn't automatically mean I will pass autolab. [1 point]**

(A) I understand.

(B) I do not understand.

(C) Potato

2 RNN (Total 20 points)

In `mytorch/rnn_cell.py` you will implement a full-fledged Recurrent Neural Network module with the ability to handle variable length inputs in the same batch.

2.1 RNN Cell Forward (5 points)

Follow the starter code available in `mytorch/rnn_cell.py` to get a better sense of the various attributes of the RNNUnit.

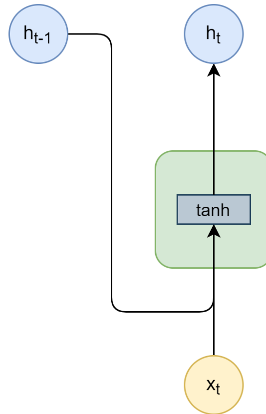

Figure 2: The computation flow for the RNN Unit forward.

$$h_t = tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \tag{1}$$

The equation you should follow is given in equation 1.

Use the "activation" attribute from the `init` method as well as all of the other weights and biases already defined in the `init` method. The inputs and outputs are defined in the starter code.

Also, note that this can be completed in one line of code.

Inputs

- x (batch_size, input_size)
  - Input at the current time step.
  - If this is the first layer, this will be the x_t if this is not the first layer, this will be the hidden output from the current time step and previous layer, h_{t,l_1}
- h (batch_size, hidden_size)
  - Hidden state generated by the previous time step, h_{t-1}

Outputs

- h_prime: (batch_size, hidden_size)
  - New hidden state generated at the current time step, h_t

2.2 RNN Cell Backward (5 points)

Calculate each of the gradients for the backward pass of the RNN Cell.

1. $\frac{\partial L}{\partial W_{ih}}$ (`self.dW_ih`)
2. $\frac{\partial L}{\partial W_{hh}}$ (`self.dW_hh`)
3. $\frac{\partial L}{\partial b_{ih}}$ (`self.db_ih`)
4. $\frac{\partial L}{\partial b_{hh}}$ (`self.db_hh`)
5. dx (returned by method, explained below)
6. dh (returned by method, explained below)

The way that we have chosen to implement the RNN Cell, you should add the calculated gradients to the current gradients. This follows from the idea that, given an RNN layer, the same cell is continuously being used. The first figure in the multiple choice shows this loop occurring for a single layer.

Also, note that the gradients for the weights and biases should be averaged (i.e. divided by the batch size, but the gradients for dx and dh should not).

Also, note that you should only be writing six lines of code in the backward method. Meaning, each gradient can be computed in one line of code.

Inputs

- delta: (batch_size, hidden_size)
  - Gradient w.r.t the current hidden layer $\frac{\partial L}{\partial h_{t,l}}$
  - The gradient from current time step and the next layer + the gradient from next time step and current layer, $\frac{\partial L}{\partial h_{t,l+1}} + \frac{\partial L}{\partial h_{t+1,l}}$
- h (batch_size, hidden_size)
  - Hidden state of the current time step and the current layer $h_{t,l}$
- h_prev_l (batch_size, input_size)
  - Hidden state at the current time step and previous layer $h_{t,l-1}$
  - If this is the first layer, it will be the input at time t, x_t
- h_prev_t (batch_size, hidden_size)
  - Hidden state at previous time step and current layer $h_{t-1,l}$

Outputs

- dx: (batch_size, input_size)
  - Derivative w.r.t. the current time step and previous layer, $\frac{\partial L}{\partial h_{t,l-1}}$
  - If this is the first layer, it will be w.r.t. the input at that layer, $\frac{\partial L}{\partial x_t}$
- dh: (batch_size, hideen_size)
  - Derivative w.r.t. the previous time step and current layer, $\frac{\partial L}{\partial h_{t-1,l}}$

How to start? We recommend drawing a computational graph.

2.3 RNN Phoneme Classifier (10 points)

In `hw3/rnn_classifier.py` implement the forward and backward methods for the `RNN_Phoneme_Classifier`.

Read over the init method and uncomment the `self.rnn` and `self.output_layer` after understanding their initialization.

Making sure to understand the code given to you, implement an RNN as described in the images below. You will be writing the forward and backward loops. Both methods should require no more than 10 lines of code (on top of the code already given).

Below are visualizations of the forward and backward computation flows. Your RNN Classifier is expected to execute given with an arbitrary number of layers and time sequences.

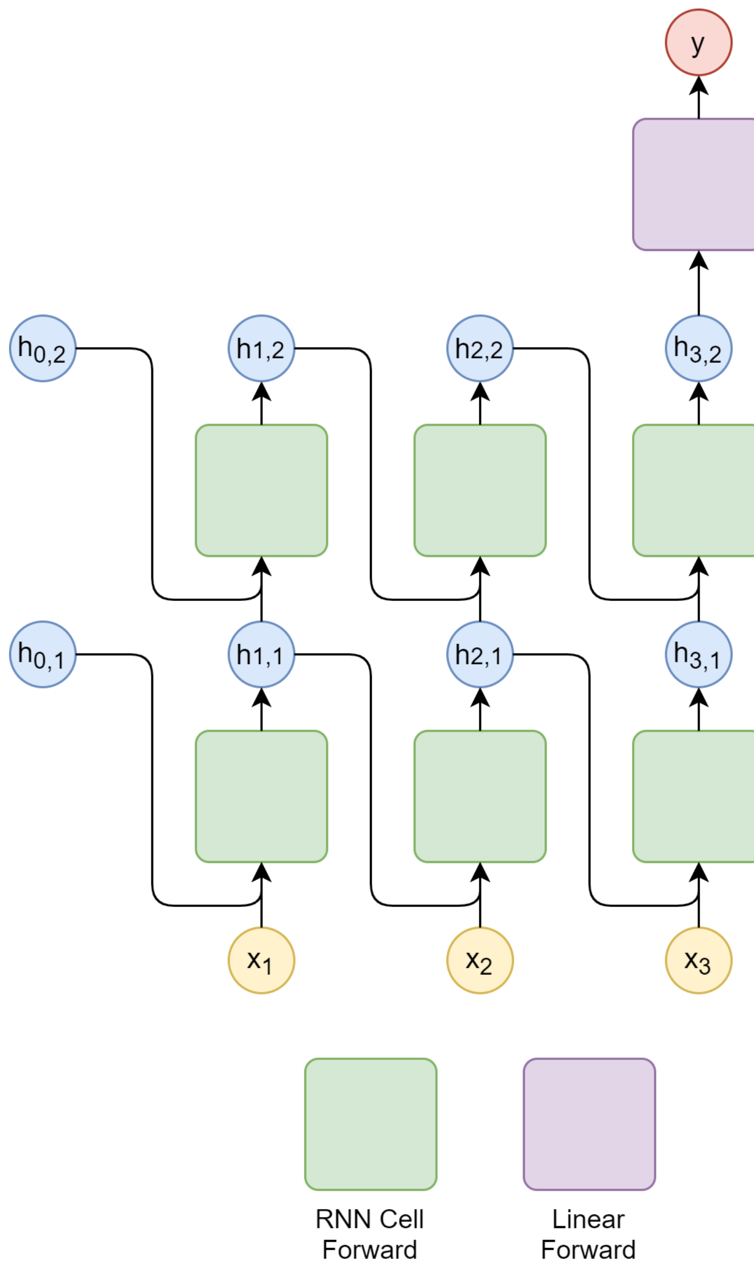

Figure 3: The forward computation flow for the RNN.

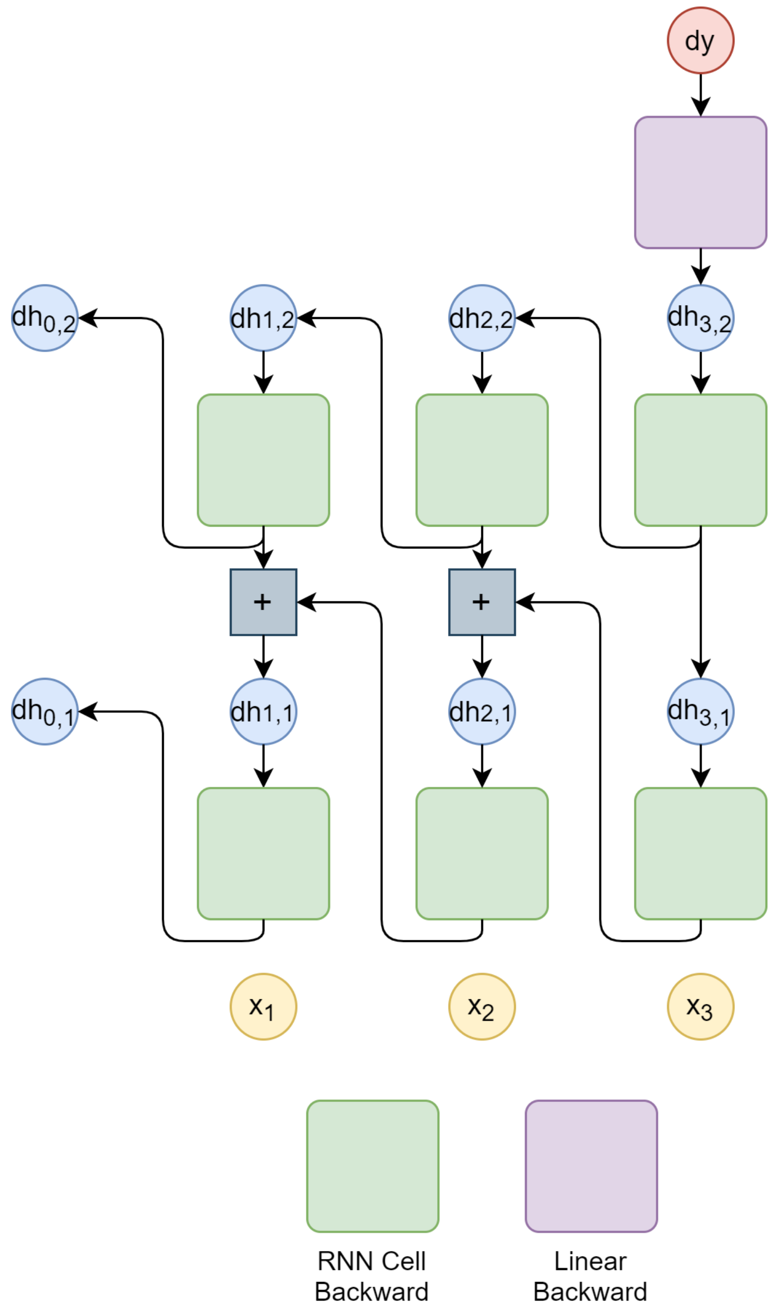

Figure 4: The backward computation flow for the RNN.

3 GRU (Total 30 points)

Replicate a portion of the `torch.nn.GRUCell` interface. GRUs are used for a number of tasks such as Optical Character Recognition and Speech Recognition on spectograms using transcripts of the dialog. In this homework, you will develop a basic understanding of completing a forward and backward pass through a GRUCell.

NOTE: Your GRU Cell will have a fundamentally different implementation in comparison to the RNN Cell (mainly in the backward method). This is a pedagogical decision to introduce you to a variety of different possible implementations, and we leave it as an exercise to you to gauge the effectiveness of each implementation.

3.1 GRU Cell Forward (5 points)

In `mytorch/gru.py` implement the forward pass for a GRUCell (though we follow a slightly different naming convention than the Pytorch documentation.) The equations for a GRU cell are the following:

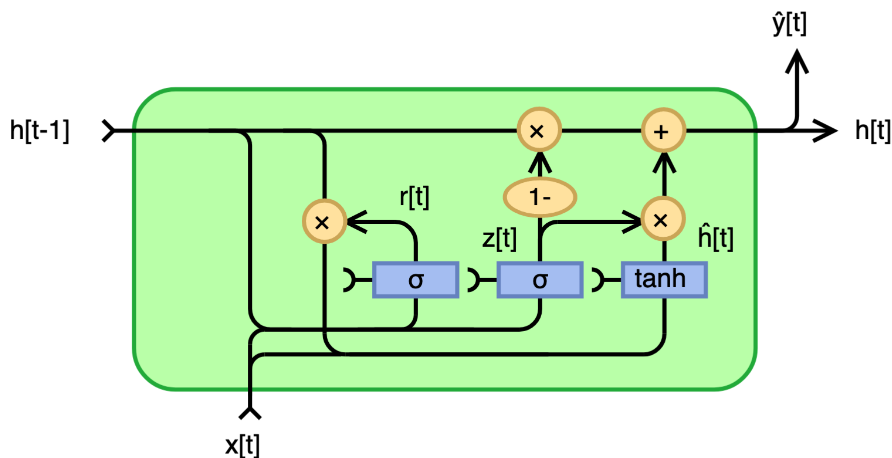

Figure 5: The computation flow for GRU.

$$\mathbf{r}_t = \sigma(\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{b}_{ir} + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr}) \tag{2}$$
$$\mathbf{z}_t = \sigma(\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{b}_{iz} + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz}) \tag{3}$$
$$\mathbf{n}_t = \tanh(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in} + \mathbf{r}_t \otimes (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn})) \tag{4}$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \otimes \mathbf{n}_t + \mathbf{z}_t \otimes \mathbf{h}_{t-1} \tag{5}$$

Please refer to (and use) the GRUCell class attributes defined in the init method. You are not expected to define any extra attributes for a working implementation of this class.

The inputs to the GRUCell forward method are x and h represented as x_t and h_{t-1} in the equations above. These are the inputs at time t.

The output of the forward method is h_t in the equations above.

There are other possible implementations for the GRU, but you need to follow the equations above for the forward pass. If you do not, you might end up with a working GRU and zero points on autolab. Do not modify the init method, if you do, it might result in lost points.

Your final implementation should be less than 10 lines of code (not including assert statements).

3.2 GRU Cell Backward (15 points)

In `mytorch/gru.py` implement the backward pass for the GRUCell specified before. The backward method of the GRUCell is one of the most time-consuming task in this homework.

This method takes as input *delta*, and you must calculate the gradients w.r.t the parameters and return the derivative w.r.t the inputs, x_t and h_t, to the cell.

The partial derivative input you are given, *delta*, refers to the derivative w.r.t to the output of your forward pass, i.e. the summation of: 1) the derivative of the loss w.r.t the input of the next layer $x_{l+1,t}$ and 2) the derivative of the loss w.r.t the input hidden-state at the next time-step $h_{l,t+1}$.

Using these partials, compute the partial derivative of the loss w.r.t each of the six weight matrices, and the partial derivative of the loss w.r.t the input x_t, and the hidden state h_t.

Specifically, there are **fourteen** gradients that need to be computed **(in no particular order)**:

1. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWrx`)
2. $\frac{\partial L}{\partial W_{rh}}$ (`self.dWrh`)
3. $\frac{\partial L}{\partial b_{ir}}$ (`self.db_ir`)
4. $\frac{\partial L}{\partial b_{hr}}$ (`self.db_hr`)
5. $\frac{\partial L}{\partial W_{zx}}$ (`self.dWzx`)
6. $\frac{\partial L}{\partial W_{zh}}$ (`self.dWzh`)
7. $\frac{\partial L}{\partial b_{iz}}$ (`self.db_iz`)
8. $\frac{\partial L}{\partial b_{hz}}$ (`self.db_hz`)
9. $\frac{\partial L}{\partial W_{nx}}$ (`self.dWnx`)
10. $\frac{\partial L}{\partial W_{nh}}$ (`self.dWnh`)
11. $\frac{\partial L}{\partial b_{in}}$ (`self.db_in`)
12. $\frac{\partial L}{\partial b_{hn}}$ (`self.db_hn`)
13. $\frac{\partial L}{\partial x_t}$ (returned by method)
14. $\frac{\partial L}{\partial h_t}$ (returned by method)

Note: $\frac{\partial L}{\partial h_t}$ (number 14 above) refers to the derivative with respect to the input `h` of your forward pass

How to start? You will need to derive the formulae for the back-propagation in order to complete this section of the assignment. We recommend creating your own computational graph and refreshing yourself on the rules for gradients from earlier recitations. See *pptx* file title ***How to compute a derivative*** attached to this handout. You may also refers to the resources below:
Back-propagation Tutorial

Note: If you find yourself writing > 50 lines of code for GRU backwards, you might be following a complex approach. Consider following the suggested approaches above.

3.3 GRU Inference (10 points)

In `mytorch/hw3.py`, use the GRUCell implemented in the previous section and a linear layer to compose a neural net. This neural net will unroll over the span of inputs to provide a set of logits per time step of input. You must initialize the GRU Cell, and then the Linear layer in that order within the init method.

Big differences between this problem and the RNN Phoneme Classifier are 1) we are only doing inference (a forward pass) on this network and 2) there is only 1 layer. This means that the forward method in the `CharacterPredictor` can be just 2 or 3 lines of code and the inference function can be completed in less than 10 lines of code.

First complete the `CharacterPredictor` class by initializing the GRU Cell and Linear layer. Then complete the forward pass for the class and the return what is necessary. The `input_dim` is the input dimension for the GRU Cell, the `hidden_dim` is the hidden dimension that should be outputted from the GRU Cell, and inputted into the Linear layer. And `num_classes` is the number of classes being predicted from the Linear layer.

Then complete the `inference` function which takes the following inputs and outputs.

- Input
  - net: An instance of CharacterPredictor
  - inputs (seq_len, feature_dim): a sequence of inputs
- Output
  - logits (seq_len, num_classes): Unwrap the net seq_len time steps and return the logits (with the correct shape)

You will compose the neural network with the CharacterPredictor class in `hw3/hw3.py` and use the inference function (also in `hw3/hw3.py`) to use the neural network that you have created to get the outputs.

4 CTC (25 points)

In Homework 3 Part 2, for the utterance to phoneme mapping task, you utilized **CTC Loss** to train a seq-to-seq model. In this part, `mytorch/ctc.py`, you will implement the CTC Loss based on the **Forward-Backward Algorithm** as shown in class.

For the input, you are given the output sequence from an RNN/GRU. This will be a probability distribution over all input symbols at each timestep. Your goal is to use the CTC algorithm to compute a new probability distribution over the symbols, **including the blank symbol**, and over all alignments. This is known as the posterior $\Pr(s_t = S_r|S, X)$ or $\gamma(t, r)$.

There are four parts to modify in this section (`mytorch/ctc.py`).

1. Extend sequence with blank (`targetWithBlank`)

Inputs

- target (target_len)
  - An output sequence from an RNN/GRU.

Outputs

- extSymbols (2 * target_len + 1)
  - The extended target sequence with blanks, where blank has been defined in the initialization of CTC.
- skipConnect (2 * target_len + 1)
  - An array with same length as extSymbols to keep track of whether an extended symbol Sext(j) is allowed to connect directly to Sext(j-2) (instead of only to Sext(j-1)) or not.
    The elements in the array can be True/False or 1/0. This will be used in the forward and backward algorithms.

2. Calculate forward probability $\alpha(t, r)$ (`forwardProb`)

Inputs

- logits (input_len, len(Symbols))
  - This represents the **log** probability (output sequence) from the RNN/GRU.
- extSymbols (2 * target_len + 1)
  - Output from extending the target with blanks.
- skipConnect (2 * target_len + 1)
  - Output from extending the target with blanks.

Outputs

- alpha (input_len, 2 * target_len + 1)
  - Forward probability for each symbol in the sequence. Notice the shape of alpha is transposed from the one in the lectures.

3. Calculate backward probability $\beta(t, r)$ (`backwardProb`)

Inputs the same as forward probability calculation.

Outputs

- beta (input_len, 2 * target_len + 1)
  - Backward probability for each symbol in the sequence. Notice the shape of beta is transposed from the one in the lectures.

4. Calculate joint probability $\gamma(t, r)$ (`postProb`)

Inputs

- alpha (input_len, 2 * target_len + 1)
  - Output from forward probability.
- beta (input_len, 2 * target_len + 1)
  - Output from backward probability.

Outputs:

- gamma (input_len, 2 * target_len + 1)
  - Posterior probability.

4.1 CTC Forward (20 points)

In the forward method, `mytorch/ctc_loss.py`, you will implement **CTC Loss** using your implementation from `mytorch/ctc.py`. Here for one batch, the CTC loss is calculated for each element in a loop and then meaned over the batch. Within the loop, follow the steps: 1) set up a `CTC` 2) truncate the target sequence and the logit with their lengths 3) extend the target sequence with blanks 4) calculate the forward probablities, backward probabilities and posteriors 5) compute the loss.

Inputs

- logits (seq_length, batch size, len(Symbols))
  - This represents the **log** probability (output sequence) from the RNN/GRU.
- target (batch size, padded_target_len)
  - This represents the target sequences.
- input_lengths (batch size,)
  - This represents the lengths of the inputs.
- target_lengths (batch size,)
  - This represents the lengths of the targets.

Outputs

- **loss**: scalar
  - Divergence between the posterior probability $\gamma(t, r)$ and the input symbols y_t^r

4.2 CTC Backward (5 points)

Using the posterior probability distribution you computed in the forward pass, you will now compute the **divergence** $\nabla_{Y_t}\text{DIV}$ of each Y_t.

$$\nabla_{Y_t}\text{DIV} = \left[\frac{d\text{DIV}}{dy_t^0} \quad \frac{d\text{DIV}}{dy_t^1} \cdots \frac{d\text{DIV}}{dy_t^{L-1}}\right]$$

$$\frac{d\text{DIV}}{dy_0^l} = -\sum_{r:S(r)=l} \frac{\gamma(t,r)}{y_t^l}$$

Similar to the CTC forward, loop over the items in the batch and fill in the divergence vector.

Outputs

- **dY**: (seq_length, batch size, len(Symbols))
  - Derivative of divergence w.r.t. the input symbols at each time y_t^r

5 Greedy Search and Beam Search (Total 20 points)

- In `mytorch/search.py`, you will implement greedy search and beam search.
- For both functions you will be provided with:
  - **SymbolSets**, a list of symbols that can be predicted, **except for the blank symbol.**
  - **y_probs**, an array of shape (len(SymbolSets) + 1, seq length, batch size) which is the probability distribution over all symbols **including the blank symbol** at each time step.
    - The probability of blank for all time steps is the first row of y probs (index 0).
    - The batch size is 1 for all test cases, but if you plan to use your implementation for part 2 you need to incorporate batch size.
  - An additional argument, **BeamWidth**, is provided for the Beam Search function. This is an integer value representing the width of the beam.

5.1 Greedy Search (5 points)

- Greedy search greedily picks the label with maximum probability at each time step to compose the output sequence.
- **Refer to the pseudocode from the lecture.**

5.2 Beam Search (15 points)

- Beam search is a more effective decoding technique to obtain a sub-optimal result out of sequential decisions, striking a balance between a greedy search and an exponential exhaustive search by keeping a beam of top-k scored sub-sequences at each time step (BeamWidth).
- In the context of CTC, you would also consider a blank symbol and repeated characters, and merge the scores for several equivalent sub-sequences.
- **Refer to the pseudocode from the lecture.**

6 Toy Examples

In this section, we will provide you with a detailed toy example for each section with intermediate numbers. You are not required but encouraged to run these tests before running the actual tests.

Run the following command to run the whole toy example tests

- **python3 autograder/hw3_autograder/toy_runner.py**

6.1 RNN

You can run tests for RNN only with the following command.

python3 autograder/hw3_autograder/toy_runner.py rnn

You can run the above command first to see what toy data you will be tested against. You should expect something like what is shown in the code block below. If your value and the expected does not match, the expected value will be printed. You are also encouraged to look at the **test_rnn_toy.py** file and print any intermediate values needed.

```
*** time step 0 ***
input:
[[-0.5174464  -0.72699493]
 [ 0.13379902  0.7873791 ]]
hidden:
[[-0.45319408  3.0532858   0.1966254 ]
 [ 0.19006363 -0.32204345  0.3842657 ]]
```

6.2 GRU

You can run tests for GRU only with the following command.

python3 autograder/hw3_autograder/toy_runner.py gru

Similarly to RNN toy examples, we provide two inputs for GRU, namely GRU Forward One Input (single input) and GRU Forward Three Input (a sequence of three input vectors). You should expect something like what is shown in the code block below.

```
Section 3.1 - GRU Forward One Input

*** time step 0 ***
input data: [[-1  0]]
hidden: [ 0 -1 -1]
*** expected h_t: [-0.31107613 -0.6337013  -0.77576846]
*** got result: [-0.31107614 -0.63370131 -0.77576846]
*** passed ***
GRU Forward One Input:  PASS
```

6.3 CTC

You can run toy tests for CTC and CTC loss only with the following command.

python3 autograder/hw3_autograder/toy_runner.py ctc

Each function in **ctc.py** and **ctc_loss.py** will be tested and you will receive scores if you pass or error messages if you fail. Example failure messages are shown below:

```
--------------------
Section 4 - Extend Sequence with Blank
Shape error, your shapes doesnt match the expected shape.
Wrong shape for extSymbols
Your shape:     (0,)
Expected shape: (5,)
Extend Sequence with Blank:  *** FAIL ***
--------------------

--------------------
Section 4 - Extend Sequence with Blank
Closeness error, your values dont match the expected values.
Wrong values for Skip_Connect
Your values:     [0 0 0 1 1]
Expected values: [0 0 0 1 0]
Extend Sequence with Blank:  *** FAIL ***
--------------------
```

6.4 BeamSearch

For our beam search toy example, we will consider a vocabulary of two symbols *'a'* and *'b'*. The (unnormalized) probability matrix output by the recurrent net is

$$y_probs = \begin{matrix}[[[1],[4],[2],[1]],\\ [[2],[1],[1],[4]],\\ [[3],[1],[3],[1]]]\end{matrix}$$

The top row corresponds to blank, the second row corresponds to 'a' and the third row corresponds to 'b'.

Find the most likely output sequence using:

(a) Greedy search (b) Beam search with beamwidth 1 (c) Beam search with beamwidth 3.

Greedy should give you output ***'bba'*** with (log) score 144

Beam search with beamWidth 1, should give you ***'bba'*** with 144 score.
Beam search with beamWidth 3 should give you ***'bba'*** with 144 score.
Beam search with beamWidth 4 should give you ***'ba'*** with 210 score.

The beam search function toy problem is not integrated to the local autograder. So, you would have to manually call your beam search implementation with the provided toy input in order to verify if your output corresponds with the above provided result.