Homework 3 Part 1

RNNs and GRUs and Search, Oh My!

11-785: Introduction to Deep Learning (Spring 2022)

OUT: **March 17, 2022**
DUE: **April 7, 2022, 11:59 PM**

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**
  - **Greedy Search and Beam Search**
  - **CTC**
- **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 your subsequent homeworks. For Homework 3, MyTorch will have the following structure:

Latest Autograder & Write-up Version: v1.1

- 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
    - **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]

The culmination of all of the Homework Part 1's will be your own custom deep learning library, which we are calling *MyTorch* ©. It will act similar to other deep learning libraries 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 your subsequent homeworks. For Homework 3, MyTorch will have the following structure:

---

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

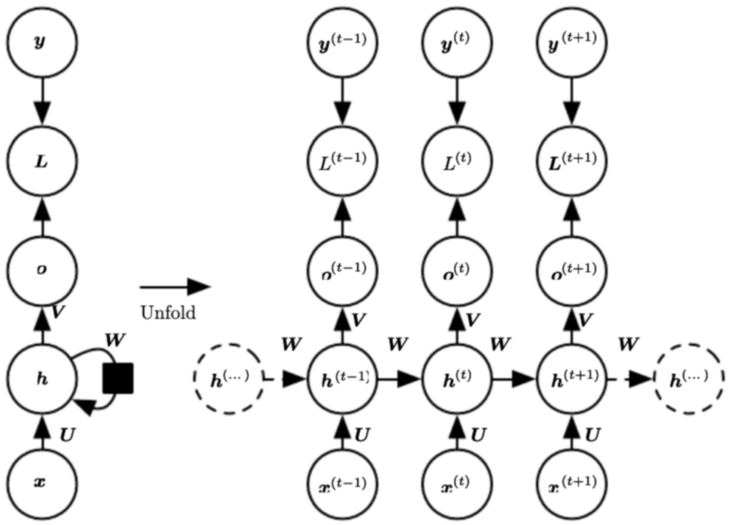


Figure 1: The high-level computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output values from http://www.deeplearningbook.org/contents/rnn.html. (Please note that this is just a general RNN, being shown as an 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 Cell

In `mytorch/rnn_cell.py` we will write an Elman RNN cell. This will help you grasp the concept of Backpropagation through time (BPTT).

2.1 RNN Cell Forward (5 points)

Follow the equation from the PyTorch documentation for computing the forward pass for an Elman RNN cell with a tanh activation found here: nn.RNNCell documentation

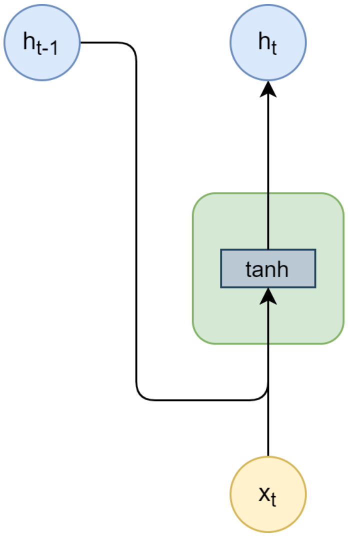


Figure 2: The computation flow for the RNN Cell forward.

$$h'_{t,l} = tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1,l} + b_{hh}) \tag{1}$$

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 at previous time step and current layer, $h_{t-1,l}$

Outputs

- h_prime: (batch_size, hidden_size)
    - New hidden state at the current time step and current layer, $h'_{t,l}$

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 with respect to the input at that layer, $\frac{\partial L}{\partial x_t}$
- dh: (batch_size, hidden_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.

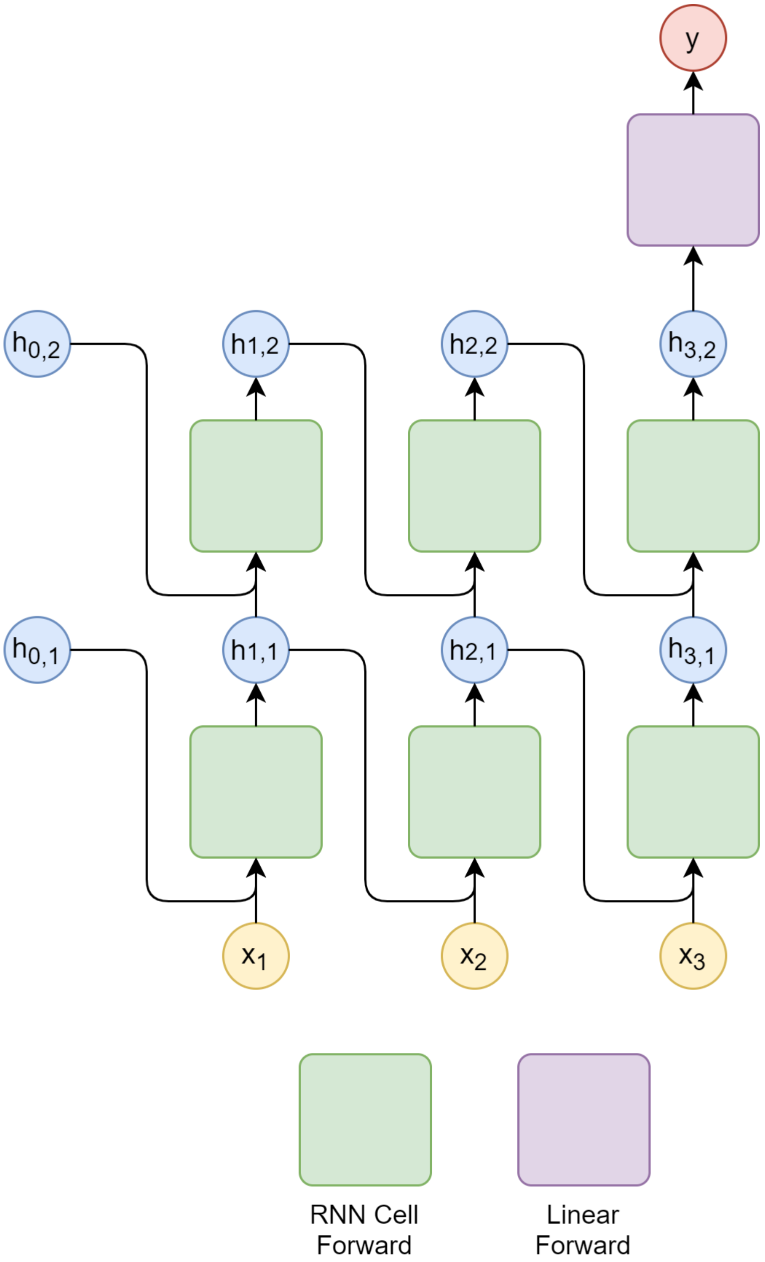


Figure 3: The forward computation flow for the RNN.

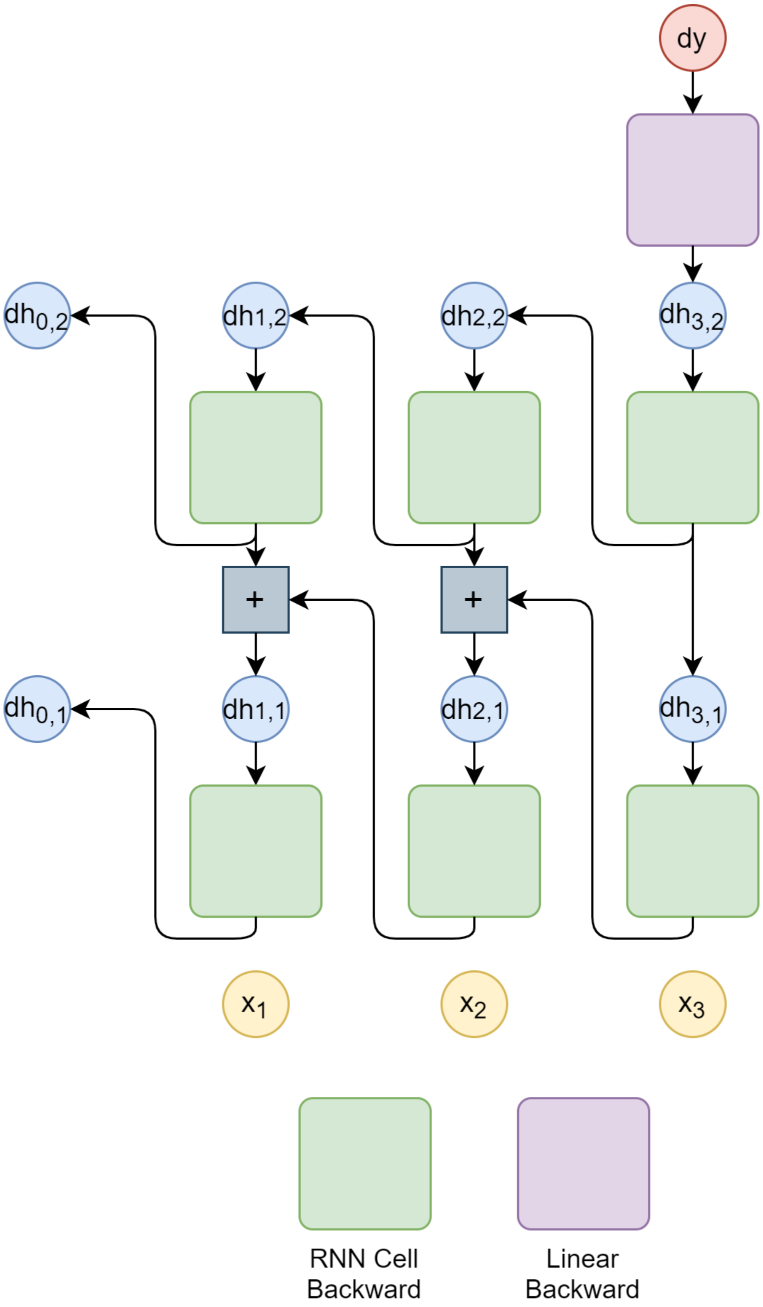


Figure 4: The backward computation flow for the RNN.

3 GRU Cell

In a standard RNN, a long product of matrices can cause the long-term gradients to vanish (i.e reduce to zero) or explode (i.e tend to infinity). One of the earliest methods that were proposed to solve this issue is LSTM (Long short-term memory network). GRU (Gated recurrent unit) is a variant of LSTM that has fewer parameters, offers comparable performance and is significantly faster to compute. 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 section, you are going to get a basic understanding of how the forward and backward pass of a GRU cell work.

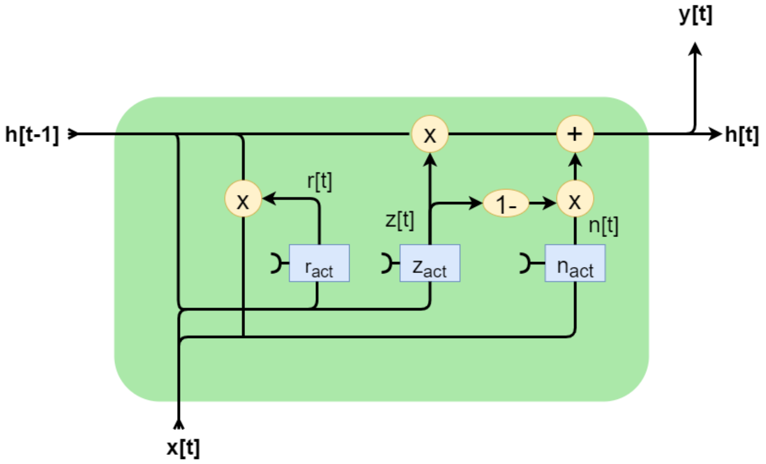


Figure 5: GRU Cell

Replicate a portion of the `torch.nn.GRUCell` interface.

```
class GRUCell:

    def forward(self, x, h):

        self.x = x
        self.hidden = h
        self.r = # TODO
        self.z = # TODO
        self.n = # TODO
        h_t = # TODO

        return h_t

    def backward(self, delta):

        self.dWrx = # TODO
        self.dWzx = # TODO
        self.dWnx = # TODO

        self.dWrh = # TODO
        self.dWzh = # TODO
        self.dWnh = # TODO

        self.dbrx = # TODO
        self.dbzx = # TODO
        self.dbnx = # TODO

        self.dbrh = # TODO
        self.dbzh = # TODO
        self.dbnh = # TODO

        return dx, dh
```

As you can see in the code given above, the GRUCell class has forward and backward attribute functions. In forward, we calculate `h_t`. We store x and h for use in backward. The attribute function `forward` includes multiple components.

- As an argument, forward expects input `x` and `h`.
- As an attribute, forward stores variables `x`, `hidden`, `r`, `z`, `n` and calculates `h_t`.
- As an output, forward returns variable `h_t`.

In backward, we calculate the gradient changes needed for optimization. The attribute function `backward` includes multiple components.

- As arguments, backward expects input `delta`.
- As attributes, backward stores variables `dWrx`, `dWzx`, `dWnx`, `dWrh`, `dWzh`, `dWnh`, `dbrx`, `dbzx`, `dbnx`, `dbrh`, `dbzh`, `dbnh` and calculates `dz`, `dn`, `dr`, `dh` and `dx`.
- As an output, backward returns variables `dx` and `dh`.

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 using Numpy, analogous to the Pytorch equivalent nn.GRUCell. (Though we follow a slightly different naming convention than the Pytorch documentation.) The equations for a GRU cell are the following:

$$\mathbf{r}_t = \sigma(\mathbf{W}_{rx}\mathbf{x}_t + \mathbf{b}_{rx} + \mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{b}_{rh}) \tag{2}$$

$$\mathbf{z}_t = \sigma(\mathbf{W}_{zx}\mathbf{x}_t + \mathbf{b}_{zx} + \mathbf{W}_{zh}\mathbf{h}_{t-1} + \mathbf{b}_{zh}) \tag{3}$$

$$\mathbf{n}_t = tanh(\mathbf{W}_{nx}\mathbf{x}_t + \mathbf{b}_{nx} + \mathbf{r}_t \otimes (\mathbf{W}_{nh}\mathbf{h}_{t-1} + \mathbf{b}_{nh})) \tag{4}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \otimes \mathbf{n}_t + \mathbf{z}_t \otimes \mathbf{h}_{t-1} \tag{5}$$

Please refer to (and use) the GRUCell class attributes defined in the init method, and define any more attributes that you deem necessary for the backward pass. Store all relevant intermediary values in the forward pass.
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.

Equations given above can be represented by the following figures:

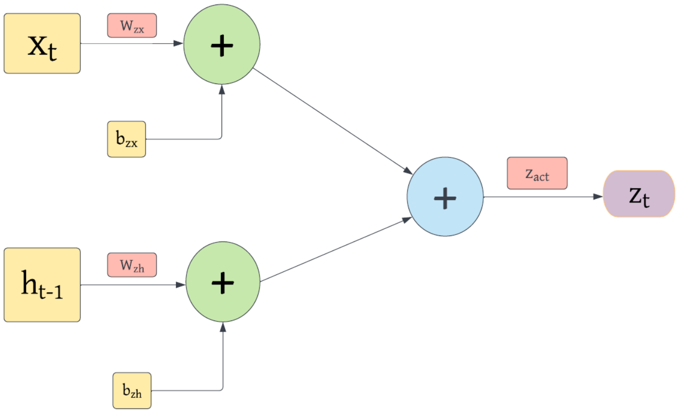


Figure 6: The computation for z_t

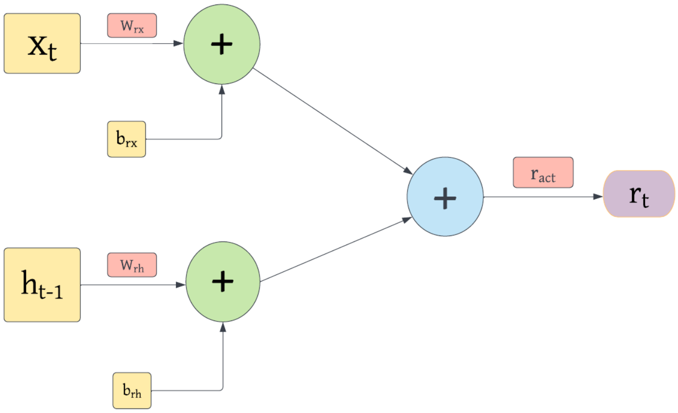


Figure 7: The computation for r_t

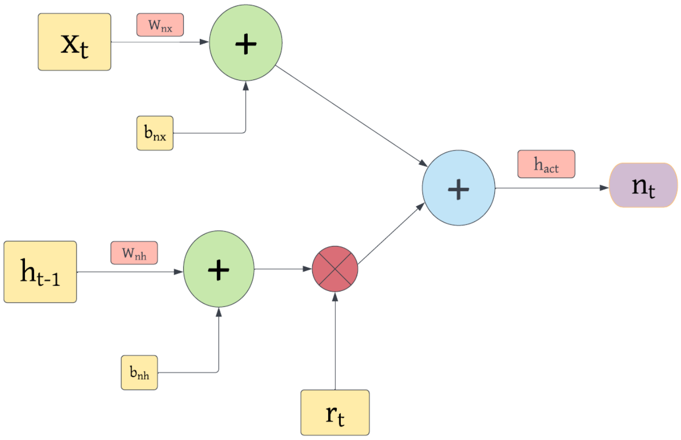


Figure 8: The computation for n_t

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 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*, is the summation of: the derivative of the loss w.r.t the input of the next layer $x_{l+1,t}$ and 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 eight gradients that need to be computed:

1. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWrx`)
2. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWzx`)
3. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWnx`)
4. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWrh`)
5. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWzh`)
6. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWnh`)
7. $\frac{\partial L}{\partial W_{rx}}$ (`self.dbrx`)
8. $\frac{\partial L}{\partial W_{rh}}$ (`self.dbzx`)
9. $\frac{\partial L}{\partial W_{zx}}$ (`self.dbnx`)
10. $\frac{\partial L}{\partial W_{zh}}$ (`self.dbrh`)
11. $\frac{\partial L}{\partial W_{x}}$ (`self.dbzh`)
12. $\frac{\partial L}{\partial W_{h}}$ (`self.dbnh`)
13. $\frac{\partial L}{\partial x_t}$ (returned by method)
14. $\frac{\partial L}{\partial h_t}$ (returned by method)

To be more specific, the input *delta* refers to the derivative with respect to the output of your forward pass.

$\frac{\partial L}{\partial h_t}$ (number 8 above) refers to the derivative with respect to the input `h` of your forward pass

How to start? Given below are the equations you need to compute the derivatives for backward pass. We also recommend refreshing yourself on the rules for gradients from Lecture 5.

1. $\frac{\partial L}{\partial z_t} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial z_t}$
2. $\frac{\partial L}{\partial n_t} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial n_t}$
3. $\frac{\partial L}{\partial h_{t-1}} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial h_{t-1}} + \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial h_{t-1}} + \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial h_{t-1}} + \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial h_{t-1}}$
4. $\frac{\partial L}{\partial W_{nx}} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial W_{nx}}$
5. $\frac{\partial L}{\partial b_{nx}} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial b_{nx}}$
6. $\frac{\partial L}{\partial x_t} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial x_t} + \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial x_t} + \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial x_t}$

7. $\frac{\partial L}{\partial r_t} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial r_t}$
8. $\frac{\partial l}{\partial W_{nh}} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial W_{nh}}$
9. $\frac{\partial L}{\partial b_{nh}} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial b_{nh}}$
10. $\frac{\partial L}{\partial W_{zx}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial W_{zx}}$
11. $\frac{\partial L}{\partial b_{zx}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial b_{zx}}$
12. $\frac{\partial L}{\partial W_{zh}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial W_{zh}}$
13. $\frac{\partial L}{\partial b_{zh}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial b_{zh}}$
14. $\frac{\partial L}{\partial W_{rx}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial W_{rx}}$
15. $\frac{\partial L}{\partial b_{rx}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial b_{rx}}$
16. $\frac{\partial L}{\partial W_{rh}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial W_{rh}}$
17. $\frac{\partial L}{\partial b_{rh}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial b_{rh}}$

To facilitate understanding, we have organized a table describing all relevant variables.

Table 1: GRUCell Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`hd`	hd	scalar	-	Hidden Dimension
`id`	id	scalar	-	Input Dimension
`x`	x_t	vector	id	observation at the current time-step
`h`	h_{t-1}	vector	hd	hidden state at previous time-step
`Wrx`	W_{rx}	matrix	$hd \times id$	Weight matrix for input (for reset gate)
`Wzx`	W_{zx}	matrix	$hd \times id$	Weight matrix for input (for update gate)
`Wnx`	W_{nx}	matrix	$hd \times id$	Weight matrix for input (for candidate hidden state)
`Wrh`	W_{rh}	matrix	$hd \times hd$	Weight matrix for hidden state (for reset gate)
`Wzh`	W_{zh}	matrix	$hd \times hd$	Weight matrix for hidden state (for update gate)
`Wnh`	W_{nh}	matrix	$hd \times hd$	Weight matrix for hidden state (for candidate hidden state)
`brx`	b_{rx}	vector	hd	bias vector for input (for reset gate)
`bzx`	b_{zx}	vector	hd	bias vector for input (for update gate)
`bnx`	b_{nx}	vector	hd	bias vector for input (for candidate hidden state)
`brh`	b_{rh}	vector	hd	bias vector for hidden state (for reset gate)
`bzh`	b_{zh}	vector	hd	bias vector for hidden state (for update gate)
`bnh`	b_{nh}	vector	hd	bias vector for hidden state (for candidate hidden state)
`dWrx`	dW_{rx}	matrix	$hd \times id$	Derivative of loss w.r.t W_{rx}
`dWzx`	dW_{zx}	matrix	$hd \times id$	Derivative of loss w.r.t W_{zx}
`dWnx`	dW_{nx}	matrix	$hd \times id$	Derivative of loss w.r.t W_{nx}
`dWrh`	dW_{rh}	matrix	$hd \times hd$	Derivative of loss w.r.t W_{rh}
`dWzh`	dW_{zh}	matrix	$hd \times hd$	Derivative of loss w.r.t W_{zh}
`dWnh`	dW_{nh}	matrix	$hd \times hd$	Derivative of loss w.r.t W_{nh}
`dbrx`	db_{rx}	vector	hd	Derivative of loss w.r.t b_{rx}
`dbzx`	db_{zx}	vector	hd	Derivative of loss w.r.t b_{zx}
`dbnx`	db_{nx}	vector	hd	Derivative of loss w.r.t b_{nx}
`dbrh`	db_{rh}	vector	hd	Derivative of loss w.r.t b_{rh}
`dbzh`	db_{zh}	vector	hd	Derivative of loss w.r.t b_{zh}
`dbnh`	db_{nh}	vector	hd	Derivative of loss w.r.t b_{nh}
`dx`	dx_t	vector	hd	Derivative of loss w.r.t x_t
`dh`	db_{nh}	vector	hd	Derivative of loss w.r.t h_{t-1}
`dAdZ`	$\partial A/\partial Z$	matrix	$N \times C$	how changes in pre-activation features affect post-activation values

3.3 GRU Inference (10 points)

In `hw3/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 **ForwardBackward 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 $Pr(s_t = S_r|S, X) = \gamma(t, r)$

Use the `(mytorch/CTC.py)` file to complete this section.

```
class CTC(object):

    def __init__(self, BLANK=0)
        self.blank = BLANK

    def extend_target_with_blank(self, target):
        extSymbols = # TODO
        skipConnect = # TODO
        return extSymbols, skipConnect

    def get_forward_probs(self, logits, extSymbols, skipConnect):
        alpha = # TODO
        return alpha

    def get_backward_probs(self, logits, extSymbols, skipConnect):
        beta = # TODO
        return beta

    def get_posterior_probs(self, alpha, beta):
        gamma = # TODO
        return gamma
```

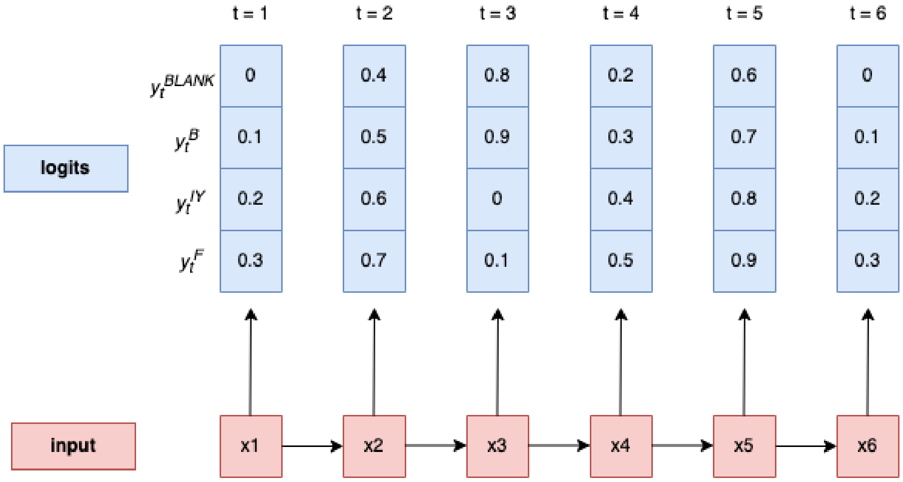
Table 2: CTC Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`target`	-	matrix	(target_len,)	Target sequence
`logits`	-	matrix	(input_len, len(Symbols))	Predicted (log) probabilities
`extSymbols`	-	vector	(2 * target_len + 1,)	Output from extending the target with blanks
`skipConnect`	-	vector	(2 * target_len + 1,)	Boolean array containing skip connections
`alpha`	α	vector	(input_len, 2 * target_len + 1)	Forward probabilities
`beta`	β	vector	(input_len, 2 * target_len + 1)	Backward probabilities
`gamma`	γ	vector	(input_len, 2 * target_len + 1)	Posterior probabilities

As you can see, the CTC class has initialization, get_forward_probs, and get_backward_probs attribute functions. Immediately once the class is instantiated, the code in init is run. The initialization phase using init includes just the `BLANK` as an argument.

1. Extend sequence with blank Given an output sequence from an RNN/GRU, we want to **extend** the target sequence with blanks, where blank has been defined in the initialization of CTC.

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



	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
y_t^{BLANK}	0	0.4	0.8	0.2	0.6	0
y_t^{B}	0.1	0.5	0.9	0.3	0.7	0.1
y_t^{IY}	0.2	0.6	0	0.4	0.8	0.2
y_t^{F}	0.3	0.7	0.1	0.5	0.9	0.3

Figure 9: An overall CTC setup example

True/False or 1/0. This will be used in the forward and backward algorithms.

The `extend target with blank` attribute function includes:

- As an argument, it expects target as input.
- As an attribute, forward stores no attributes.
- As an output, forward returns variable `extSymbols` and `skipConnect`.

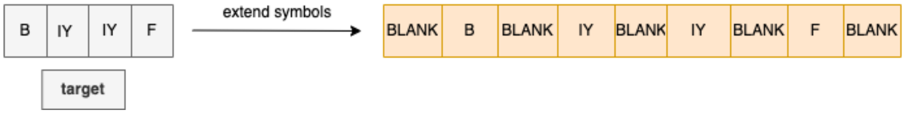


Figure 10: Extend symbols

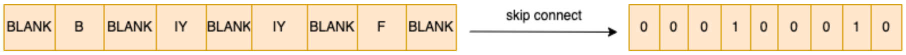


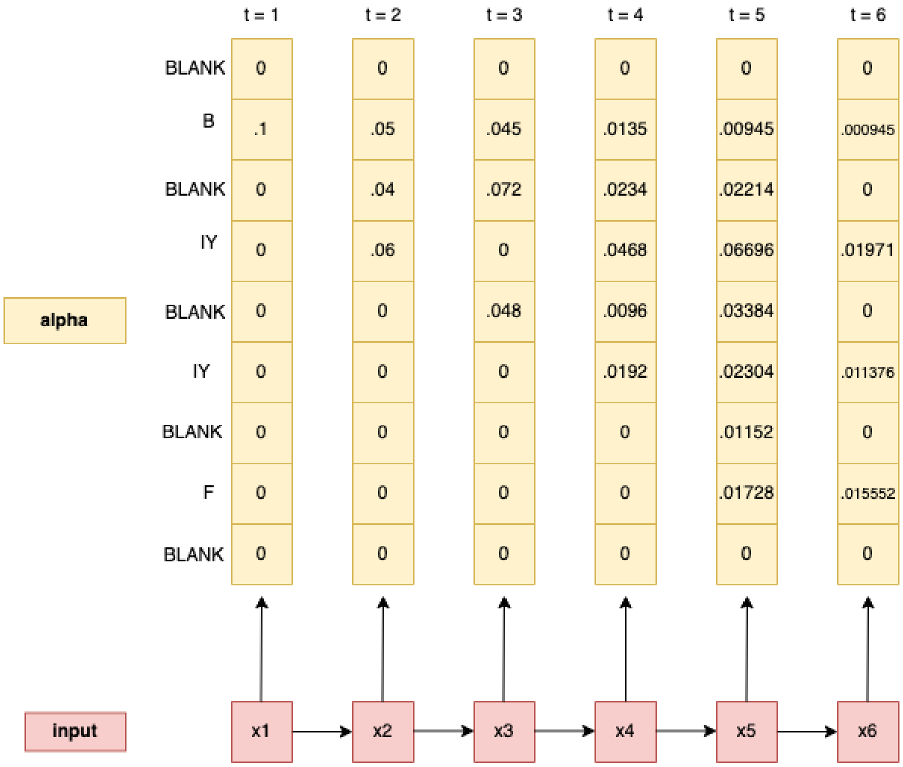
Figure 11: Skip connections

2. Forward Algorithm In forward, we calculate `alpha` $\alpha(t, r)$ (Fig.12). The attribute function forward include:

- As an argument, forward expects `logits`, `extSymbols`, `skipConnect` as input.
- As an attribute, forward stores no attributes.
- As an output, forward returns variable `alpha`

3. Backward Algorithm In backward, we calculate `beta` $\beta(t, r)$ (Fig. 13). The attribute function backward include:

- As an argument, forward expects `logits`, `extSymbols`, `skipConnect` as input.
- As an attribute, forward stores no attributes.
- As an output, forward returns variable `beta`

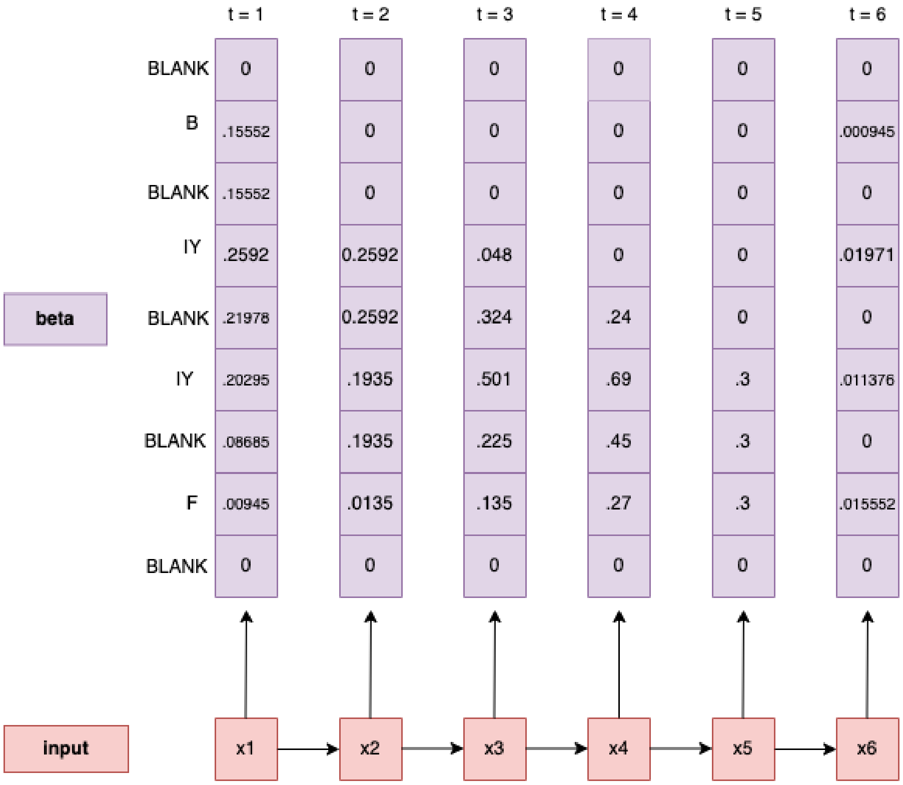


	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
BLANK	0	0	0	0	0	0
B	.1	.05	.045	.0135	.00945	.000945
BLANK	0	.04	.072	.0234	.02214	0
IY	0	.06	0	.0468	.06696	.01971
BLANK	0	0	.048	.0096	.03384	0
IY	0	0	0	.0192	.02304	.011376
BLANK	0	0	0	0	.01152	0
F	0	0	0	0	.01728	.015552
BLANK	0	0	0	0	0	0

Figure 12: Forward Algorithm

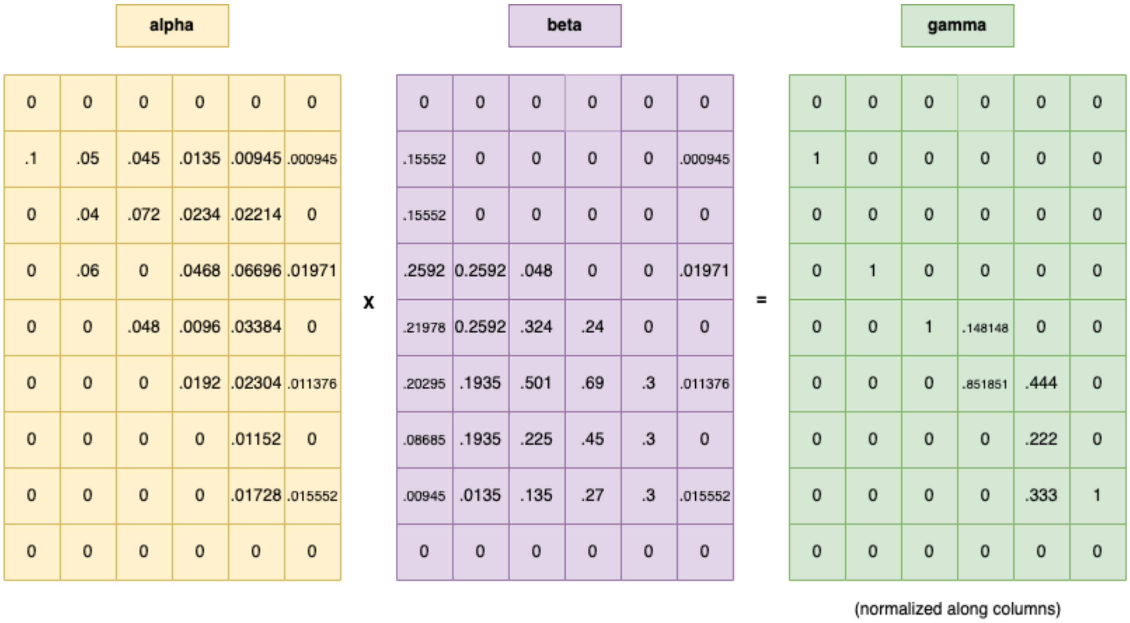
4. CTC Posterior Probability In posterior probability, we calculate `gamma` $\gamma(t, r)$ (Fig. ??). The attribute function backward include:

- As an argument, forward expects `alpha`, `beta` as input.
- As an attribute, forward stores no attributes.
- As an output, forward returns variable `gamma`



	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
BLANK	0	0	0	0	0	0
B	.15552	0	0	0	0	.000945
BLANK	.15552	0	0	0	0	0
IY	.2592	0.2592	.048	0	0	.01971
BLANK	.21978	0.2592	.324	.24	0	0
IY	.20295	.1935	.501	.69	.3	.011376
BLANK	.08685	.1935	.225	.45	.3	0
F	.00945	.0135	.135	.27	.3	.015552
BLANK	0	0	0	0	0	0

Figure 13: Backward Algorithm



alpha

0	0	0	0	0	0
.1	.05	.045	.0135	.00945	.000945
0	.04	.072	.0234	.02214	0
0	.06	0	.0468	.06696	.01971
0	0	.048	.0096	.03384	0
0	0	0	.0192	.02304	.011376
0	0	0	0	.01152	0
0	0	0	0	.01728	.015552
0	0	0	0	0	0

beta

0	0	0	0	0	0
.15552	0	0	0	0	.000945
.15552	0	0	0	0	0
.2592	0.2592	.048	0	0	.01971
.21978	0.2592	.324	.24	0	0
.20295	.1935	.501	.69	.3	.011376
.08685	.1935	.225	.45	.3	0
.00945	.0135	.135	.27	.3	.015552
0	0	0	0	0	0

gamma

0	0	0	0	0	0
1	0	0	0	0	0
0	0	0	0	0	0
0	1	0	0	0	0
0	0	1	.148148	0	0
0	0	0	.851851	.444	0
0	0	0	0	.222	0
0	0	0	0	.333	1
0	0	0	0	0	0

(normalized along columns)

Figure 14: Posterior Probability

4.1 CTC Loss

```
class CTCLoss(object):

    def __init__(self, BLANK=0)
        super(CTCLoss, self).__init__()
        self.BLANK = BLANK
        self.gammas = []
        self.ctc = CTC()


    def forward(self, logits, target, input_lengths, target_lengths):
        for b in range(B):
            # TODO

        total_loss = np.sum(total_loss) / B
        return total_loss


     def backward(self):
        dY = # TODO
        return dY
```

Table 3: CTC Loss Components

Code Name	Math	Type	Shape	Meaning
`target`	-	matrix	(batch_size, paddedtargetlen)	Target sequences
`logits`	-	matrix	(seqlength, batch_size, len(Symbols))	Predicted (log) probabilities
`input_lengths`	-	vector	(batch_size,)	Lengths of the inputs
`target_lengths`	-	vector	(batch_size,)	Lengths of the target
`loss`	-	scalar	-	Avg. divergence between posterior. probability $\gamma(t, r)$ and the input symbols ${y_t}^r$
`dY`	dY	matrix	(seqlength, batch_size, len(Symbols))	Derivative of divergence wrt the input symbols at each time.

4.1.1 CTC Forward

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.

In forward function, we calculate `avgLoss`. The attribute function forward include:

- As an argument, forward expects `target`, `input_lengths`, `target_lengths` as input.
- As an attribute, forward stores `gammas` and `extSymbols` as attributes.
- As an output, forward returns variable `avgLoss`.

4.1.2 CTC Backward

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

$$\nabla_{Y_t}\text{DIV} = \begin{bmatrix} \frac{d\text{DIV}}{dy_t^0} & \frac{d\text{DIV}}{dy_t^1} \cdots \frac{d\text{DIV}}{dy_t^{L-1}} \end{bmatrix}$$

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

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

In backward function, we calculate `dY`. The attribute function backward include:

- As an argument, backward expects no inputs.
- As an attribute, backward stores no attributes.
- As an output, backward returns variable `dY`

5 Greedy Search and Beam Search (20 points)

- In `mytorch/search.py`, you will implement greedy search and beam search.
- For both the 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.

After training a model with CTC, the next step is to use it for inference. During inference, given an input sequence X, we want to infer the most likely output sequence Y. We can find an approximate, sub-optimal solution Y^* using:

$$Y^* =_Y p(Y|X)$$

We will cover two approaches for the inference step:

- Greedy Search
- Beam Search

Use the `mytorch/CTCDecoding.py` file to complete this section.

A utility function that performs basic text cleaning on decoded paths has been provided. There is no need to modify this method, but you must use this method appropriately to clean the decoded paths before returning them.

```
def clean_path(path):

    path = str(path).replace("’","")
    path = path.replace(",","")
    path = path.replace(" ","")
    path = path.replace("[","")
    path = path.replace("]","")
```

```
        return path
```

5.1 Greedy Search

One possible way to decode at inference time is to simply take the most probable output at each time-step, which will give us the alignment A^* with the highest probability as:

$$A^* =_A \prod_{t=1}^{T} p_t(a_t|X)$$

where $p_t(a_t|X)$ is the probability for a single alignment a_t at time-step t. Repeated tokens and ϵ (the blank symbol) can then be collapsed in A^* to get the output sequence Y.

```
class GreedySearchDecoder(object):

    def __init__(self, symbol_set):

        self.symbol_set = symbol_set


    def decode(self, y_probs):

        decoded_path = []
        blank = 0
        path_prob = 1

        # TODO:
        # 1. Iterate over sequence length - len(y_probs[0])
        # 2. Iterate over symbol probabilities
        # 3. update path probability, by multiplying with the current max probability
        # 4. Select most probable symbol and append to decoded_path

        decoded_path = clean_path(decoded_path)

        return decoded_path, path_prob
```

Note: Detailed pseudo-code for Greedy Search can be found in the lecture slides, which is to be implemented in the `decode` method of the `GreedySearchDecoder` class

5.2 Beam Search

Although Greedy Search is easy to implement, it misses out on alignments that can lead to outputs with higher probability because it simply selects the most probable output at each time-step.

To deal with this short-coming, we can use Beam Search, a more effective decoding technique that obtains 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. Hence at each time-step we would maintain a list of possible outputs after collapsing repeating characters and blank symbols. The score for each possible output at current time-step will be the accumulated score of all alignments that map to it. Based on this score, the top-k beams will be selected to expand for the next time-step.

(Explanations and examples have been provided by referring: `https://distill.pub/2017/ctc/`)

```
class BeamSearchDecoder(object):

    def __init__(self, symbol_set):

        self.symbol_set = symbol_set
        self.beam_width = beam_width


    def decode(self, y_probs):

        decoded_path = []
        sequences = [[list(), 1.0]]
        ordered = None

        best_path, merged_path_scores = None, None

        # TODO:
        # 1. Iterate over sequence length - len(y_probs[0])
        #    - initialize a list to store all candidates
        # 2. Iterate over 'sequences'
        # 3. Iterate over symbol probabilities
        #    - Update all candidates by appropriately compressing sequences
        #    - Handle cases when current sequence is empty vs. when not empty
        # 4. Sort all candidates based on score (descending), and rewrite 'ordered'
        # 5. Update 'sequences' with first self.beam_width candidates from 'ordered'
        #    - Remember to prune the beams correctly according to lecture slides pseudo-code
        # 6. Merge paths in 'ordered', and get merged path scores
        # 7. Select best path based on merged path scores, and return

        return best_path, merged_path_scores
```

Note: Detailed pseudo-code for Beam Search can be found in the lecture slides, which is to be implemented in the `decode` method of the `BeamSearchDecoder` class. You can implement additional methods within this class, as long as you return the expected variables from the decode method (which is called during training).

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 ]]
```

For the RNN Classifier, you should expect the following values in your forward and backward calculation. You are encouraged to print out the intermediate values in **rnn_classifier.py** to check the correctness. Note that the variable naming follows Figure 3 and Figure 4.

```
*** time step 0 ***
input:
[[-0.5174464  -0.72699493]
 [ 0.13379902  0.7873791 ]
 [ 0.2546231   0.5622532 ]]

h_1,1:
[[-0.32596806 -0.66885584 -0.04958976]]
h_2,1:
[[ 0.0457021  -0.38009422  0.22511855]]
h_3,1:
[[-0.08056512 -0.3035707   0.03326178]]
h_1,2:
[[-0.57588165 -0.05876583  0.07493359]]
h_2,2:
[[-0.39792368 -0.50475268 -0.18843713]]
h_3,2:
[[-0.39261185 -0.16278453  0.06340214]]

dy:
[[-0.81569054  0.15619404  0.14065858  0.08515406  0.12171953  0.09829506  0.11741257  0.09625671]]
dh_3,2:
[[-0.10989283 -0.33949198 -0.13078328]]
dh_2,2:
[[-0.19552927  0.10362767  0.10584534]]
```

```
dh_1,2:
[[ 0.07086602  0.02721845 -0.10503672]]
dh_3,1:
[[ 0.10678867 -0.08892407  0.17659623]]
dh_2,1:
[[ 0.02254178 -0.10607887 -0.2609735 ]]
dh_1,1:
[[-0.00454101  0.00640496  0.14489316]]
```

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.

```
*** time step 0 ***
input data: [[ 0 -1]]
hidden: [ 0 -1  0]

Values needed to compute z_t for GRU Forward One Input:
W_zx :
[[ 0.33873054  0.32454306]
 [-0.04117032  0.15350085]
 [ 0.19508289 -0.31149986]]

b_zx: [ 0.3209093   0.48264325 -0.48868895]

W_zh:
[[ 5.2004099e-01 -3.2403603e-01 -2.4332339e-01]
 [ 2.0603785e-01 -3.4281990e-04  4.7853872e-01]
 [-2.5018784e-01  8.5339367e-02 -2.9516235e-01]]

b_rh: [ 0.05053747  0.27746138 -0.20656243]
z_act: Sigmoid activation

    Expected value of z_t using the above values:
    [0.58066287 0.62207662 0.55666673]

Values needed to compute r_t for GRU Forward One Input:
W_rx:
[[-0.12031382  0.48722494]
 [ 0.29883575 -0.13724688]
 [-0.54706806 -0.16238078]]

b_rx:
[-0.43146715  0.1538158  -0.01858002]

W_rh:
[[ 0.12764311 -0.4332353   0.37698156]
 [-0.3329033   0.41271853 -0.08287123]
 [-0.11965907 -0.4111069  -0.57348186]]

 b_rh:
 [ 0.05053747  0.2774614  -0.20656243]

 r_t: Sigmoid Activation
 Expected value of r_t using the above values:
 [0.39295226 0.53887278 0.58621625]

Values needed to compute n_t for GRU Forward One Input:
W_nx:
[[0.34669924 0.2716753 ]
 [0.2860521  0.06750154]
 [0.14151925 0.39595175]]

 b_nx:
 [ 0.54185045 -0.23604721  0.25992656]

 W_nh:
 [[-0.29145974 -0.4376279   0.21577674]
 [ 0.18676305  0.01938683  0.472116  ]
 [ 0.43863034  0.22506309 -0.04515916]]

 b_nh:
 [ 0.0648244   0.47537327 -0.05323243]

 n_t: Tanh Activation
 Expected value of n_t using the above values:
 [ 0.43627021 -0.05776569 -0.2905497 ]

Values needed to compute h_t for GRU Forward One Input:

z_t: [0.58066287 0.62207662 0.55666673]
n_t: [ 0.43627018 -0.05776571 -0.29054966]
h_(t-1): [ 0 -1  0]

    Expected values for h_t:
    [ 0.18294427 -0.64390767 -0.12881033]
```

6.3 Beam Search

Fig. 15 depict a toy problem for understanding Beam Search. Here, we are performing Beam Search over a vocabulary of {-, A, B}, where " - " is the BLANK character. Here, we are performing a beam search with **beam width = 3** and for three decoding steps. Table 4 shows the output probabilities for each token at each decoding step.

Vocabulary	P(symbol) @ T=1	P(symbol) @ T=2	P(symbol) @ T=3
-	**0.49**	**0.38**	**0.02**
A	**0.03**	**0.44**	**0.40**
B	**0.47**	**0.18**	**0.58**

Table 4: Probabilities of each symbol in the vocabulary over three consecutive decoding steps

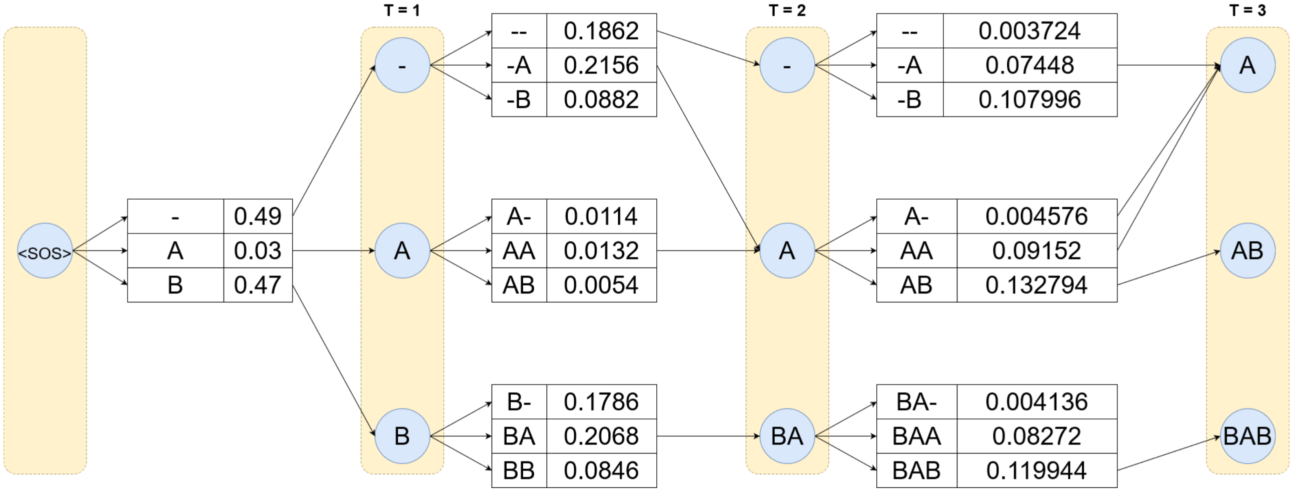


Figure 15: Beam Search over a vocabulary / symbols set = {-, A, B} with beam width (k) = 3. (" - " == BLANK). The blue shaded nodes indicate the compressed decoded sequence at given time step, and the expansion tables show the probability (right column) and symbols (left column) at each time step pre-pended with the current decoded sequence

To perform beam search with a beam width = 3, we select the top 3 most probable sequences at each time step, and expand them further in the next time step. It should be noted that the selection of top-k most probable sequences is made based on the probability of the entire sequence, or the conditional probability of a given symbol given a set of previously decoded symbols. This probability will also take into account all the sequences that can be reduced (collapsing blanks and repeats) to the given output. For example, the probability of observing a "A" output at the 2nd decoding step will be:

$$P(A) = P(A) * P(-|A) + P(A) * P(A|A) + P(-) * P(A|-)$$

This can also be observed in 15, where the sequence "A" at time step = 2 has three incoming connections. The decoded sequence with maximum probability at the final time step will be the required best output sequence, which is the sequence "A" in this toy problem.