Homework 3 Part 1

RNNs and GRUs and Search, Oh My!

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

OUT: **October 27, 2022, 11:59 PM**
ESB: **November 3, 2022, 11:59 PM**
DUE: **November 17, 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**
  - **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 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
  - CTCDecoding.py
  - linear.py
  - activation.py
  - loss.py
- hw3
  - hw3.py
  - rnn_classifier.py
  - mc.py
- autograder
  - hw3_autograder
    - **runner.py**

- **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 directory containing the handout, then **SUBMIT** the created *handin.tar* file to autolab:

  ```
  tar -c -f handin.tar handout
  ```

- **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]

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

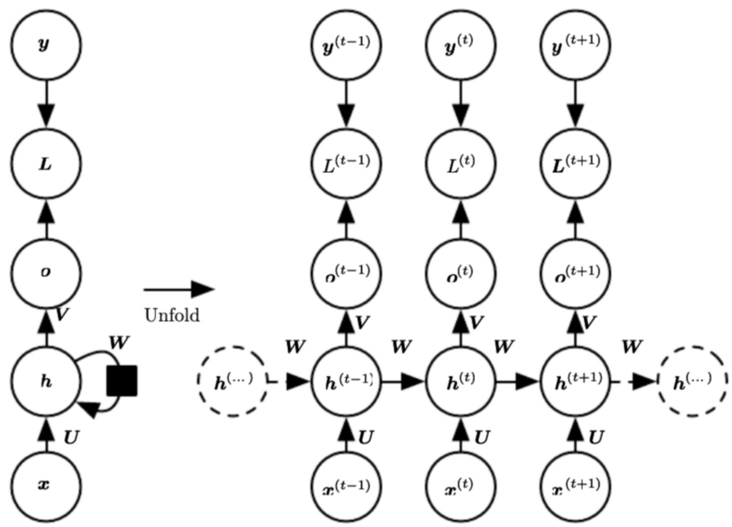

Figure 1: The high-level computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output values from http://www.deeplearningbook.org/contents/rnn.html. (Please note that this is just a general RNN, being shown as an 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 four 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 4 words for each time step. Each word is the index of the corresponding probability (ranging from 0 to 3).

(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

The RNN Cell can be thought of as the smallest unit of a Recurrent Neural Network (RNN). As you already know, RNNs deal with time-dependent and/or sequence-dependent problems. They are "recurrent" because they have memory that can be reused to predict future states. Typically, an RNN architecture will be spread over time and multiple layers. This might be overwhelming to look at, however, this homework will help you overcome that! Figure 2 shows what a typical RNN model may look like. As the model extends, previous hidden states will be utilized by newer time steps. Similarly, previous outputs will be utilized by newer layers. However, our focus for this section and the next will be to look closely at the red box highlighting a single RNN cell.
In `mytorch/rnn_cell.py` we will write an Elman RNN cell.

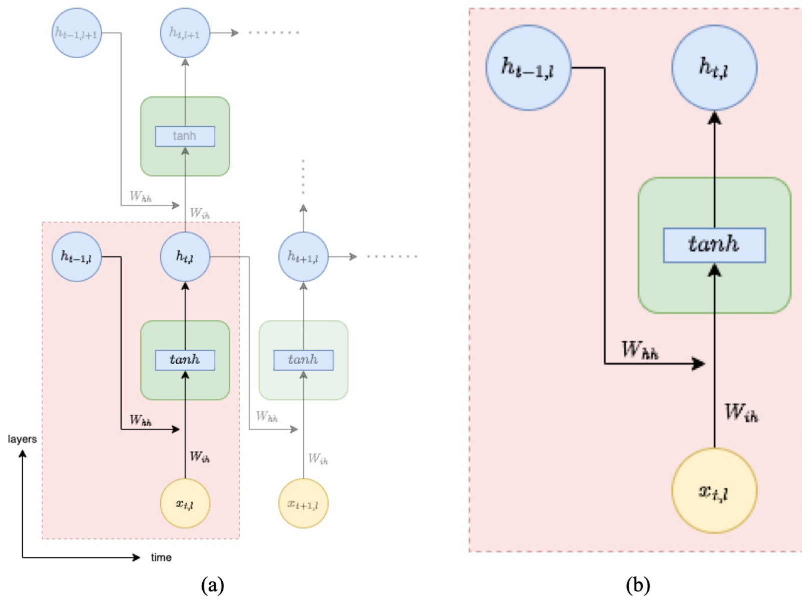

Figure 2: The red box shows one single RNN cell. RNNs can have multiple layers across multiple time steps. This is indicated by the two-axis in the bottom-left.

In this section, your task is to implement the forward and backward attribute functions of `RNNCell` class. Please consider the following class structure.

```
class RNNCell:

    def __init__(self, input_size, hidden_size):

        <Weight definitions>
        <Gradient Definitions>

    def init_weights(self, W_ih, W_hh, b_ih, b_hh):
        <Assignments>

    def zero_grad(self):
        <zeroing gradients>

    def forward(self, x, h_prev_t):
        h_t = # TODO
```

```
    def backward(self, delta, h, h_prev_l, h_prev_t):
        dz = None # TODO

        # 1) Compute the averaged gradients of the weights and biases
        self.dW_ih += None # TODO
        self.dW_hh += None # TODO
        self.db_ih += None # TODO
        self.db_hh += None # TODO

        # # 2) Compute dx, dh_prev_t
        dx = None # TODO
        dh_prev_t = None # TODO

        return dx, dh_prev_t
```

As you can see, the `RNNCell` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. In forward, we calculate `h_t`. The attribute function `forward` includes:

- As arguments, forward expects `x` and `h_prev_t` as input.
- As an attribute, forward stores no variables.
- As an output, forward returns variable `h_t`

In backward, we calculate gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As arguments, backward expects inputs `delta` (graident wrt current layer), `h_t`, `h_prev_l` and `h_prev_t`.
- As attributes, backward stores `dW_ih`, `dW_hh`, `db_hh` and `db_ih`.
- As an output, backward returns `dx` and `dh_prev_t`.

Table 1: RNNCell Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`x`	x_t	matrix	$B \times N_i$	Input at current time step
`h_prev_t`	$h_{t-1,l}$	matrix	$B \times N_h$	Previous time step hidden state and current layer
`W_ih`	W_{ih}	matrix	$N_h \times N_i$	Weight between input and hidden
`b_ih`	b_{ih}	vector	N_h	Bias between input and hidden
`W_hh`	W_{hh}	matrix	$N_h \times N_h$	Weight between hidden and hidden
`b_hh`	b_{hh}	vector	N_o	Bias between hidden and hidden
`delta`	$\partial L/\partial h$	matrix	$B \times N_h$	gradient wrt current hidden layer
`h_t`	$h_{t,l}$	matrix	$B \times N_h$	Hidden state at current time step and current layer
`h_prev_l`	$h_{t,l-1}$	matrix	$B \times N_i$	Hidden state at current time step and previous layer
`h_prev_t`	$h_{t-1,l}$	matrix	$B \times N_h$	Hidden state at previous time step and current layer
`dW_ih`	$\partial L/W_{ih}$	matrix	$N_h \times N_i$	Gradient between input and hidden
`db_ih`	$\partial L/b_{ih}$	vector	N_h	Gradient between input and hidden
`dW_hh`	$\partial L/\partial W_{hh}$	matrix	$N_h \times N_h$	Gradient between hidden and hidden
`db_hh`	$\partial L/\partial b_{hh}$	vector	N_o	Gradient between hidden and hidden

2.1 RNN Cell Forward (5 points)

The underlying principle of computing each cell's forward output is the same as any neural network you have seen before. There are weights and biases that are plugged into an affine function, and finally activated. But what does the forward pass look like for the RNN cell that has to also incorporate the previous state's memory?

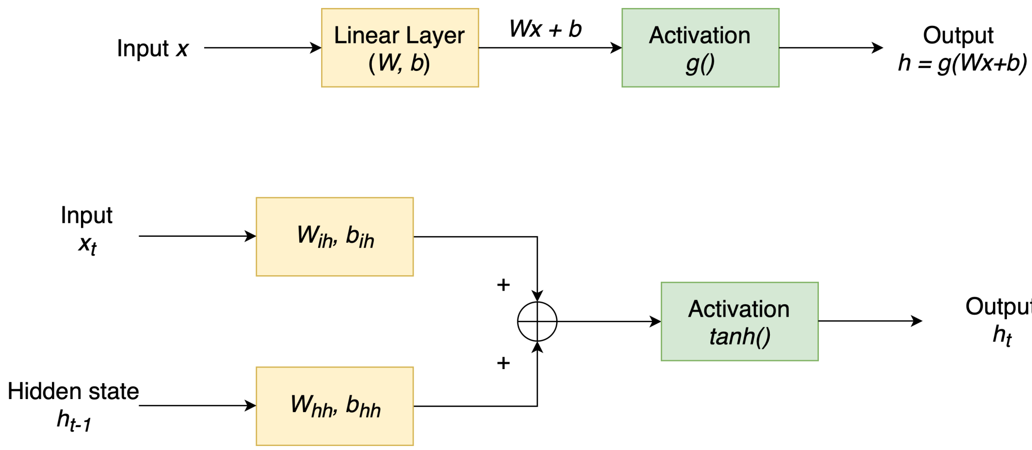

Figure 3: The figure compares a perceptron with a single layer, single time step RNN cell.

Each of the inputs have a weight and bias attached to their connections. First, we compute the affine function of both these inputs as follows.

Affine function for inputs

$$W_{ih}x_t + b_{ih} \tag{1}$$

Affine function for previous hidden state

$$W_{hh}h_{t-1,l} + b_{hh} \tag{2}$$

Now we add up these affines and pass it through the tanh activation function. The final equation can be written as follows.

$$h_{t,l} = tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1,l} + b_{hh}) \tag{3}$$

These equations are for a single element. **You may need to transpose in order to accommodate for the batch dimension in your code.**

You can also refer to 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. **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!

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. $\frac{\partial L}{\partial x}$ (`dx`) (returned by the method, explained below)
6. $\frac{\partial L}{\partial h_{t-1}}$ (`dh_prev_t`) (returned by the method, explained below)

With 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 used at each time step. The Figure 1 in the multiple choice shows this loop occurring for a single layer.

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_prev_t should not.

(Also, note that a clean implementation will only require 6 lines of code. In other words, you can calculate each gradient in one line, if you wish)

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. `self.rnn` consists of RNNCell and `self.output_layer` is a Linear layer that maps hidden states to the output.

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. A clean implementation will 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.

Table 2: RNNPhonemeClassifier Class Components

Code Name	Math	Type	Shape	Meaning
`x`	x	matrix	(batch size, seq_len, input size)	Input
`h_0`	h_0	matrix	(num layers, batch size, hidden size)	Initial hidden states
`delta`	$\partial L/\partial h$	matrix	(batch size, hidden size)	Gradient w.r.t. last time step output

2.3.1 RNN Classifier Forward

Follow the diagram given below to complete the forward pass of RNN Phoneme Classifier.

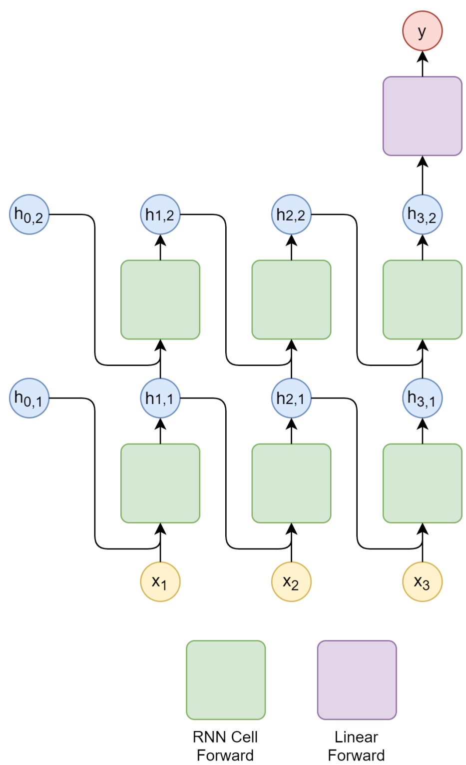

Figure 4: The forward computation flow for the RNN.

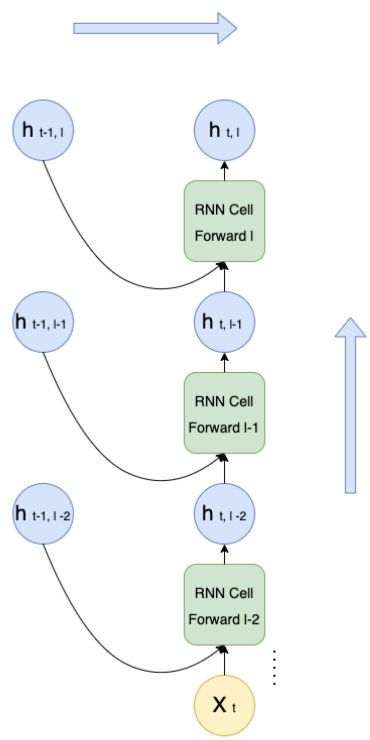

Figure 5: The forward computation flow for the RNN at time step t.

2.3.2 RNN Classifier Backward

This question might be the toughest question conceptually, in this homework. However, if you follow this pseudocode and try to understand whats going on, you can complete it without much hassle.

```
PSEUDOCODE:
    * Iterate in reverse order of time (from seq_len-1 to 0)
    * Iterate in reverse order of layers (from num_layers-1 to 0)
        * Get h_prev_l either from hiddens or x depending on the layer
            (Recall that hiddens has an extra initial hidden state)
        * Use dh and hiddens to get the other parameters for the backward method
            (Recall that hiddens has an extra initial hidden state)
        * Update dh with the new dh from the backward pass of the rnn cell
        * If you aren't at the first layer, you will want to add
          dx to the gradient from l-1th layer.

    * Normalize dh by batch_size since initial hidden states are also treated
    as parameters of the network (divide by batch size)
```

The exact same is given in your handout as well. You will be able to complete this question easily, if you understand the flow with the help of the figures 6, 7 and then follow the pseudocode.

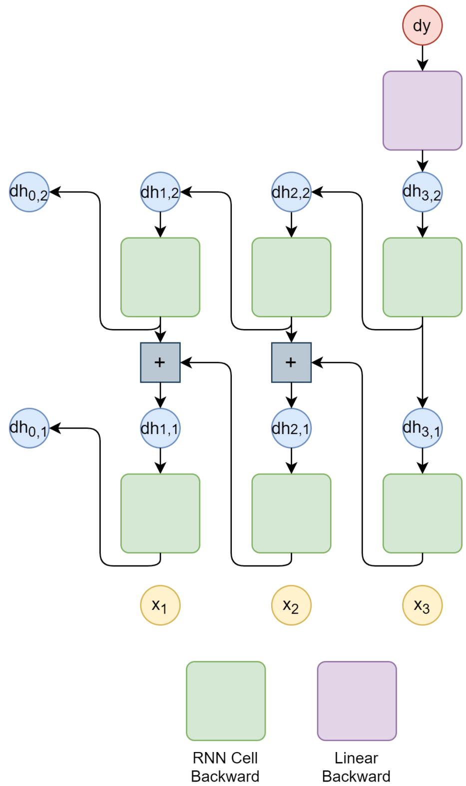

Figure 6: The backward computation flow for the RNN.

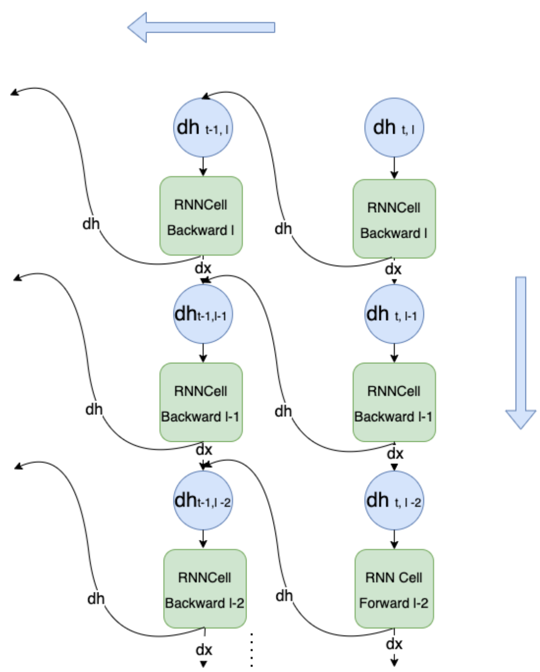

Figure 7: The backward computation flow for the RNN at time step t.

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.

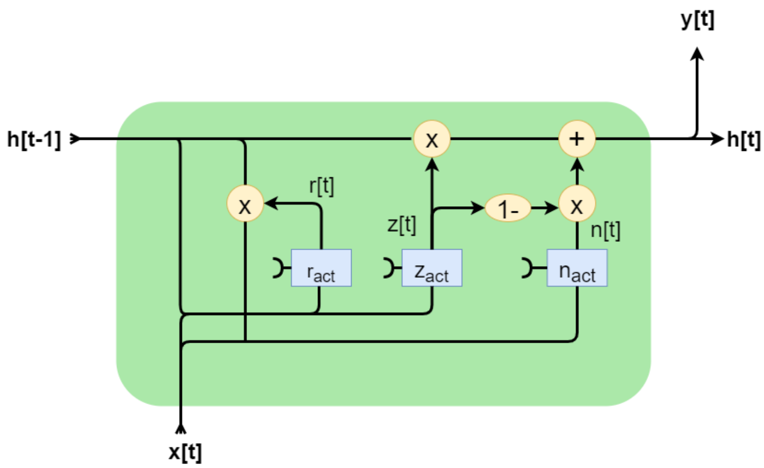

Figure 8: GRU Cell

Replicate a portion of the `torch.nn.GRUCell` interface. Consider the following class definition.

```
class GRUCell:

    def forward(self, x, h_prev_t):

        self.x = x
        self.hidden = h_prev_t
        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`. The attribute function `forward` includes multiple components:

- As an argument, forward expects input `x` and `h_prev_t`.
- As an attribute, forward stores variables `x`, `hidden`, `r`, `z`, and `n`.
- 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_prev_t` and `dx`.
- As an output, backward returns variables `dx` and `dh_prev_t`.

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:

$$\mathbf{r}_t = \sigma(\mathbf{W}_{rx}\mathbf{x}_t + \mathbf{b}_{rx} + \mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{b}_{rh}) \tag{4}$$
$$\mathbf{z}_t = \sigma(\mathbf{W}_{zx}\mathbf{x}_t + \mathbf{b}_{zx} + \mathbf{W}_{zh}\mathbf{h}_{t-1} + \mathbf{b}_{zh}) \tag{5}$$
$$\mathbf{n}_t = tanh(\mathbf{W}_{nx}\mathbf{x}_t + \mathbf{b}_{nx} + \mathbf{r}_t \otimes (\mathbf{W}_{nh}\mathbf{h}_{t-1} + \mathbf{b}_{nh})) \tag{6}$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \otimes \mathbf{n}_t + \mathbf{z}_t \otimes \mathbf{h}_{t-1} \tag{7}$$

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_prev_t` 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:

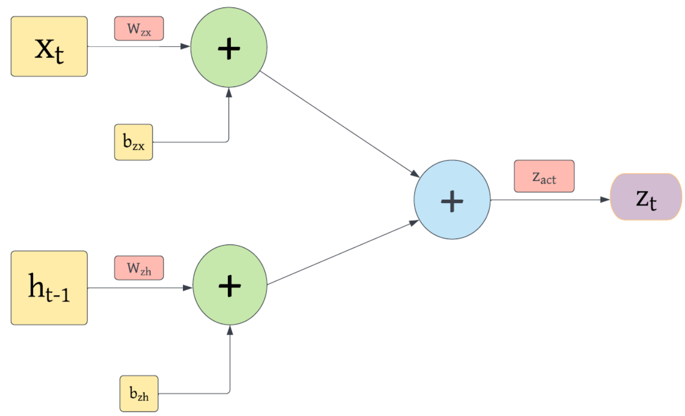

Figure 9: The computation for z_t

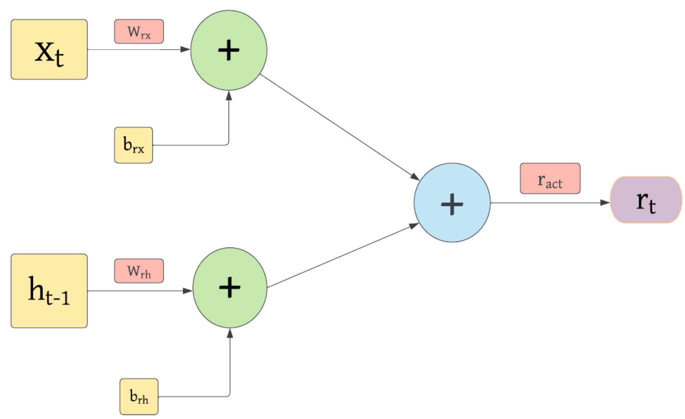

Figure 10: The computation for r_t

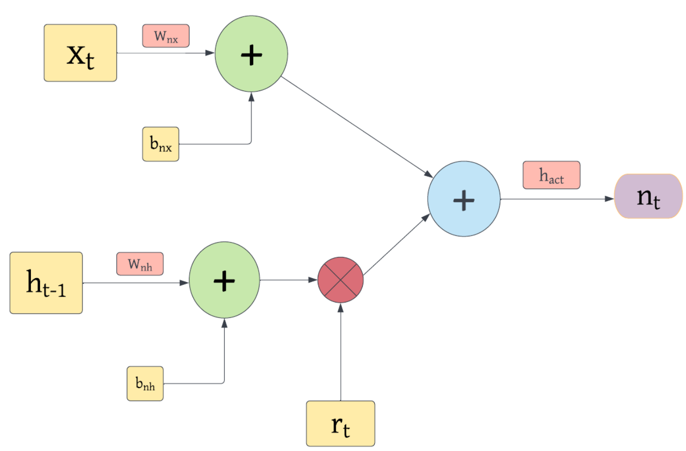

Figure 11: The computation for n_t

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 seems like the most time-consuming task in this homework because you have to compute 14 gradients but it is not difficult if you do it the right way.

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-1}, 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 fourteen gradients that need to be computed:

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

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

$\frac{\partial L}{\partial h_{t-1}}$ (number 14 above) refers to the derivative with respect to the input `h_prev_t` 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.

IMPORTANT NOTE: As you compute the above gradients, you will notice that a lot of expressions are being reused. Store these expressions in other variables to write code that is easier for you to debug. This problem is not as big as it seems. Apart from `dx` and `dh_prev_t`, all gradients can computed in 2-3 lines of code.

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 3: GRUCell Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`h`	hd	scalar	-	Hidden Dimension
`d`	id	scalar	-	Input Dimension
`x`	x_t	vector	id	observation at the current time-step
`h_prev_t`	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`	$\partial L/\partial W_{rx}$	matrix	$hd \times id$	Gradient of loss w.r.t W_{rx}
`dWzx`	$\partial L/\partial W_{zx}$	matrix	$hd \times id$	Gradient of loss w.r.t W_{zx}
`dWnx`	$\partial L/\partial W_{nx}$	matrix	$hd \times id$	Gradient of loss w.r.t W_{nx}
`dWrh`	$\partial L/\partial W_{rh}$	matrix	$hd \times hd$	Gradient of loss w.r.t W_{rh}
`dWzh`	$\partial L/\partial W_{zh}$	matrix	$hd \times hd$	Gradient of loss w.r.t W_{zh}
`dWnh`	$\partial L/\partial W_{nh}$	matrix	$hd \times hd$	Gradient of loss w.r.t W_{nh}
`dbrx`	$\partial L/\partial b_{rx}$	vector	hd	Gradient of loss w.r.t b_{rx}
`dbzx`	$\partial L/\partial b_{zx}$	vector	hd	Gradient of loss w.r.t b_{zx}
`dbnx`	$\partial L/\partial b_{nx}$	vector	hd	Gradient of loss w.r.t b_{nx}
`dbrh`	$\partial L/\partial b_{rh}$	vector	hd	Gradient of loss w.r.t b_{rh}
`dbzh`	$\partial L/\partial b_{zh}$	vector	hd	Gradient of loss w.r.t b_{zh}
`dbnh`	$\partial L/\partial b_{nh}$	vector	hd	Gradient of loss w.r.t b_{nh}
`dx`	$\partial L/\partial x_t$	vector	hd	Gradient of loss w.r.t x_t
`dh_prev_t`	$\partial L/\partial h_{t-1}$	vector	hd	Gradient 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.

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.

You have to complete the following in this section.

- The `CharacterPredictor` class by initializing the GRU Cell and Linear layer in the __init__ function
- 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. (We refer to the linear layer `self.projection` in the code because it is just a linear transformation between the hidden state to the output state)
- 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 lecture.

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 4: 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 is consist of initialization, get_forward_probs, and get_backward_probs attribute functions. Immediately once the class is instantiated, the code in __init__ will run. The initialization phase assigns the argument `BLANK` to variable self.blank.

Tip: You will be able to complete this section completely based on the pseudocodes given in the lecture slides.

1. Extend target 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.

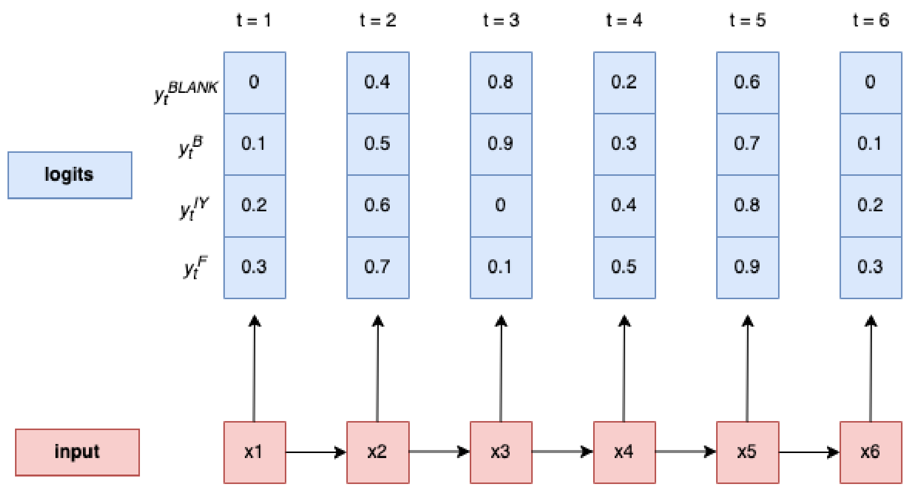

Figure 12: An overall CTC setup example

skipConnect: 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.

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`.

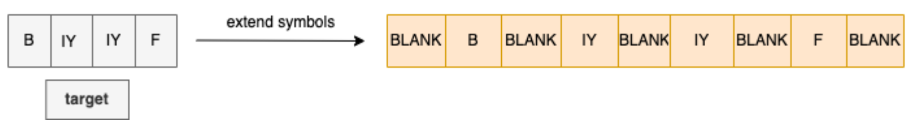

Figure 13: Extend symbols

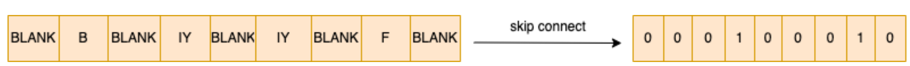

Figure 14: Skip connections

2. Forward Algorithm In forward, we calculate `alpha` $\alpha(t, r)$ (Fig.15).

$$\alpha(t, r) = P(S_0...S_r, s_t = S_r|X) = \sum_{q:S_q \in pred(S_r)} \alpha(t-1, q) y_t^{S_r}$$

$\alpha(t, r)$ is the total probability of all paths leading to the alignment of S_r to time t, $pred(S_r)$ is any symbol that is permitted to come before S_r and may include S_r.

The attribute for `get_forward_probs` 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`

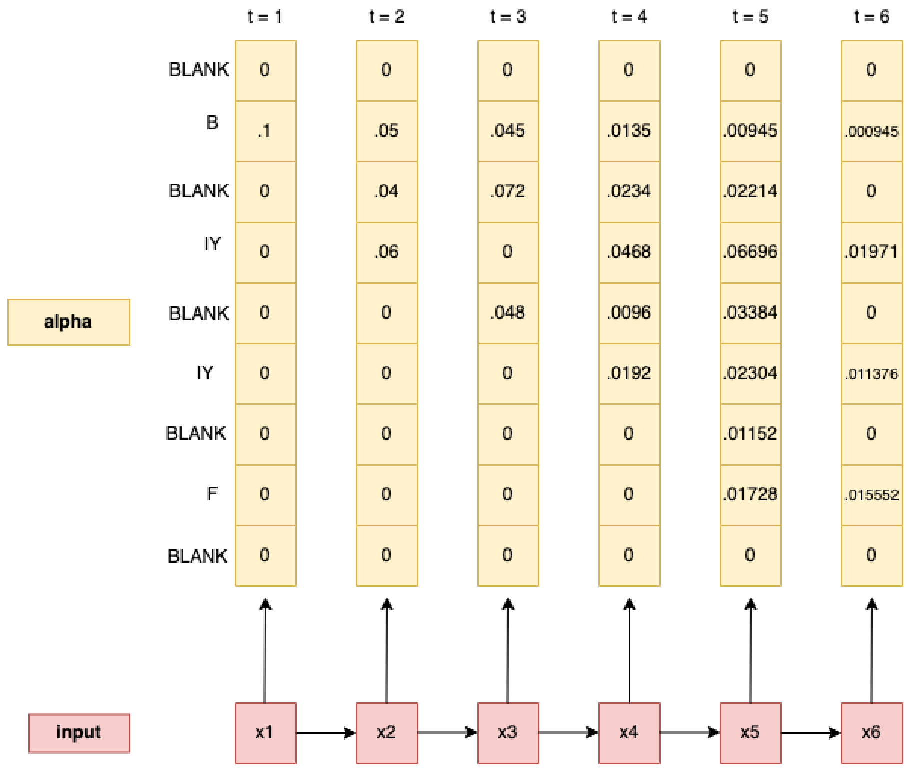

Figure 15: Forward Algorithm

3. Backward Algorithm In backward, we calculate `beta` $\beta(t,r)$ (Fig. 16), which is defined recursively in terms of the $\beta(t+1,q)$ of the next time step.

$$\beta(t,r) = P(s_{t+1} \in succ(S_r), succ(S_r), ..., S_{K-1}|X) = \sum_{q:S_q \in succ(S_r)} \beta(t+1,q) y_{t+1}^{S_q}$$

Where $succ(S_r)$ is any symbol that is permitted to come after S_r and does not include S_r. The attribute for `get_backward_probs` 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`

4. CTC Posterior Probability In posterior probability, we calculate `gamma` $\gamma(t,r)$ (Fig. 17). 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`

$$\gamma(t,r) = P(s_t = S_r|S,X) = \frac{\alpha(t,r)\beta(t,r)}{\sum_{r'} \alpha(t,r)\beta(t,r)}$$

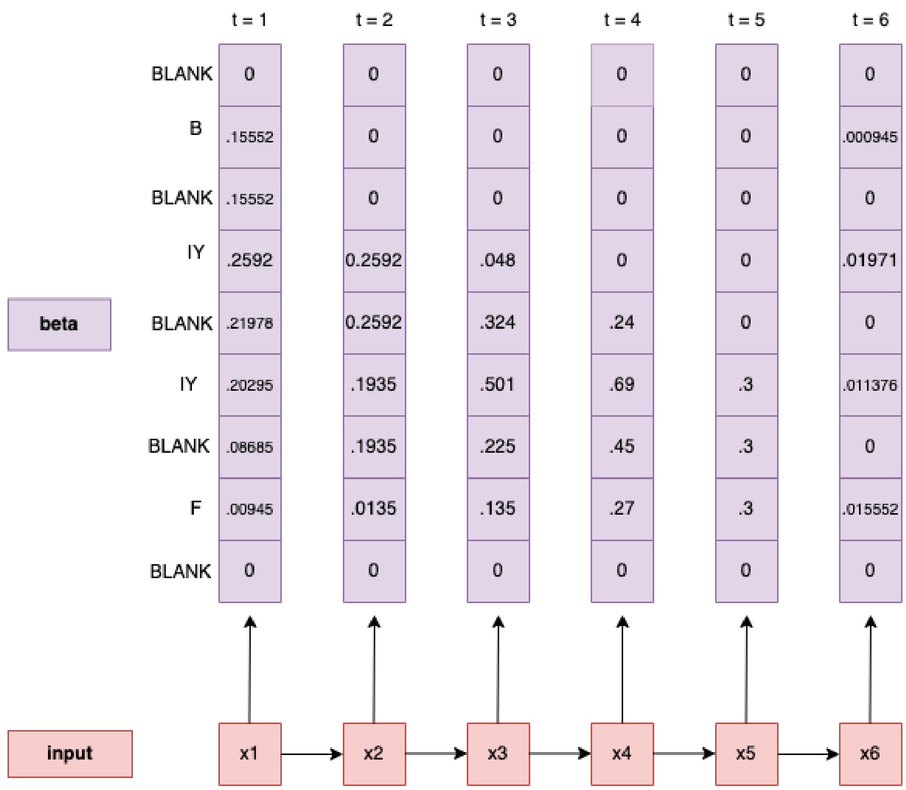

Figure 16: Backward Algorithm

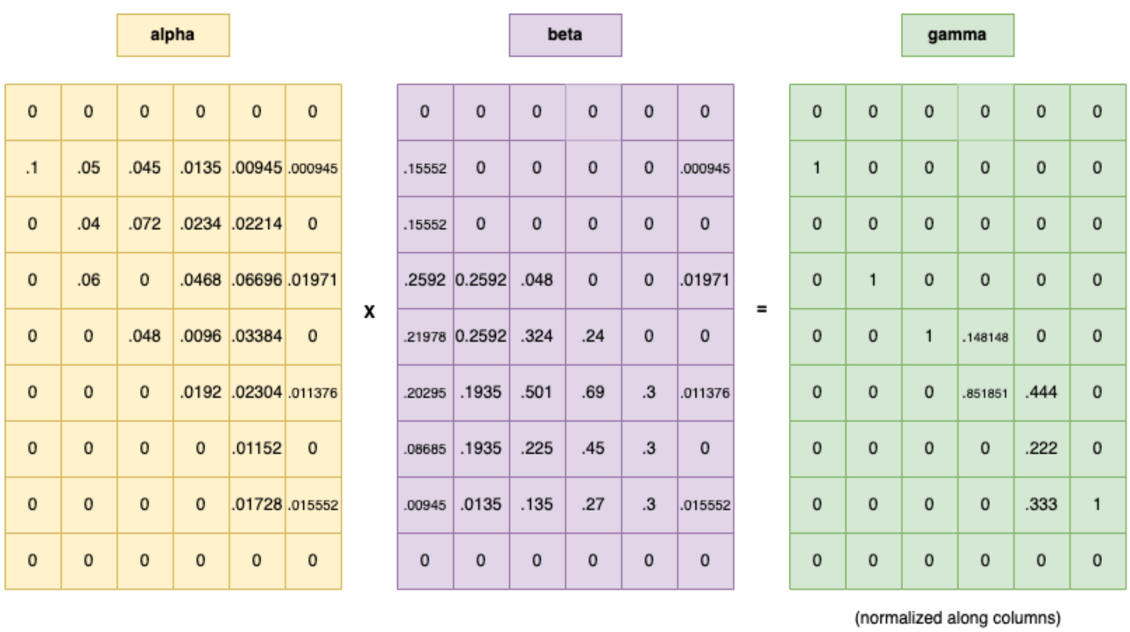

Figure 17: Posterior Probability

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 5: 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.

$$\nabla_{Y_t}\text{DIV} = \left[\frac{d\text{DIV}}{dy_t^0} \quad \frac{d\text{DIV}}{dy_t^1} \cdots \frac{d\text{DIV}}{dy_t^{L-1}}\right]$$
$$\frac{d\text{DIV}}{dy_0^l} = -\sum_{r:S(r)=l} \frac{\gamma(t,r)}{y_t^l}$$

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

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 CTC Decoding: Greedy Search and Beam Search (20 points)

After training your sequence model, the next step to do is to decode the model output probabilities to get an understandable output. Even without thinking explicitly about decoding, you have actually done a simple version of decoding in both HW1P2 and HW2P2. You take the predited class as the one with the highest output probability by searching through the probabilities of all classes in the final linear layer. Now we will learn about decoding for sequence models.

- In `mytorch/CTCDecoding.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:

$$Y^* = \underset{Y}{\arg\max}\ p(Y|X)$$

We will cover two approaches for the inference step:

- Greedy Search
- Beam Search

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

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:

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

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.

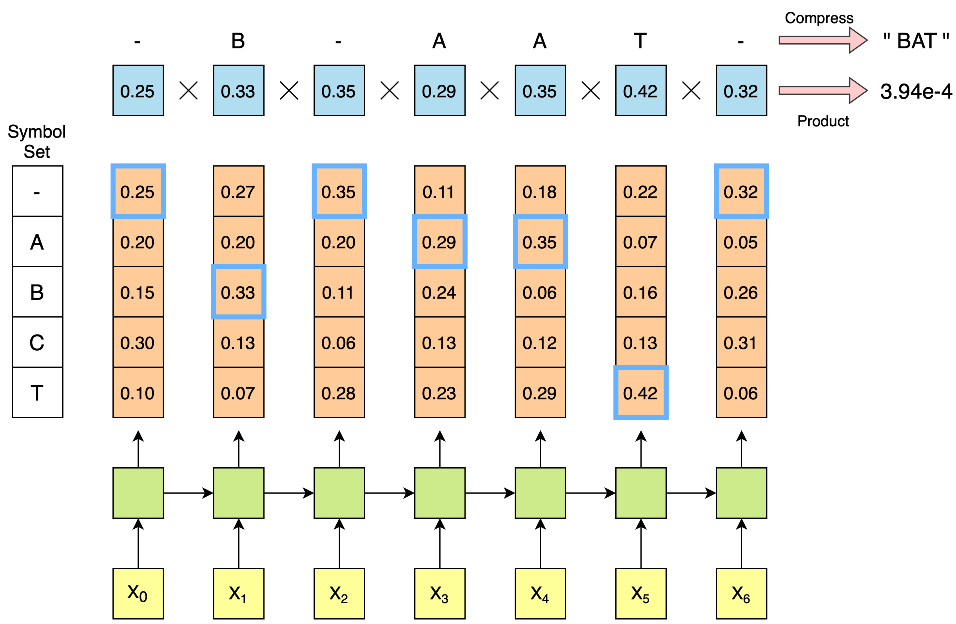

Figure 18: Greedy Search

Consider the example in Figure 18. The output is given for 7 time steps. Each probability distribution has 5 element (4 for each symbol and 1 for the blank). Greedy decode chooses the most likely time-aligned sequence by choosing the symbol corresponding to the highest probability at that time step. The final output is obtained by compressing the sequence to remove the blanks and repetitions in-between blanks. The class is given below.

```
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
# 5. Compress sequence (Inside or outside the loop)


return decoded_path, path_prob
```

Note: Detailed pseudo-code for Greedy Search can be found in the lecture slides.

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.

Beam search might be a difficult concept to understand at first. We recommend you to watch the lectures and recitations pertaining to beam search to better understand the concept. 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. The Figure 19 gives a clearer understanding of the pseudo-code. Please take some time to go through it, try doing the calculations on your own and verifying them. *It uses the first test example from the local autograder.* **It is also to be noted that the given figure only shows the output after time step t=1**.

Debugging tips:

- Assign T=2 and complete the code before proceeding
- Don't write the whole beam search code and then try running the local autograder. After you complete writing one of the functions from the pseudocode, make sure to check it
- Print intermediate outputs, i.e, outputs returned by each small function. The green boxes in Figure 19 tells you the expected output values of each function. **Make sure you get these output when t=1**

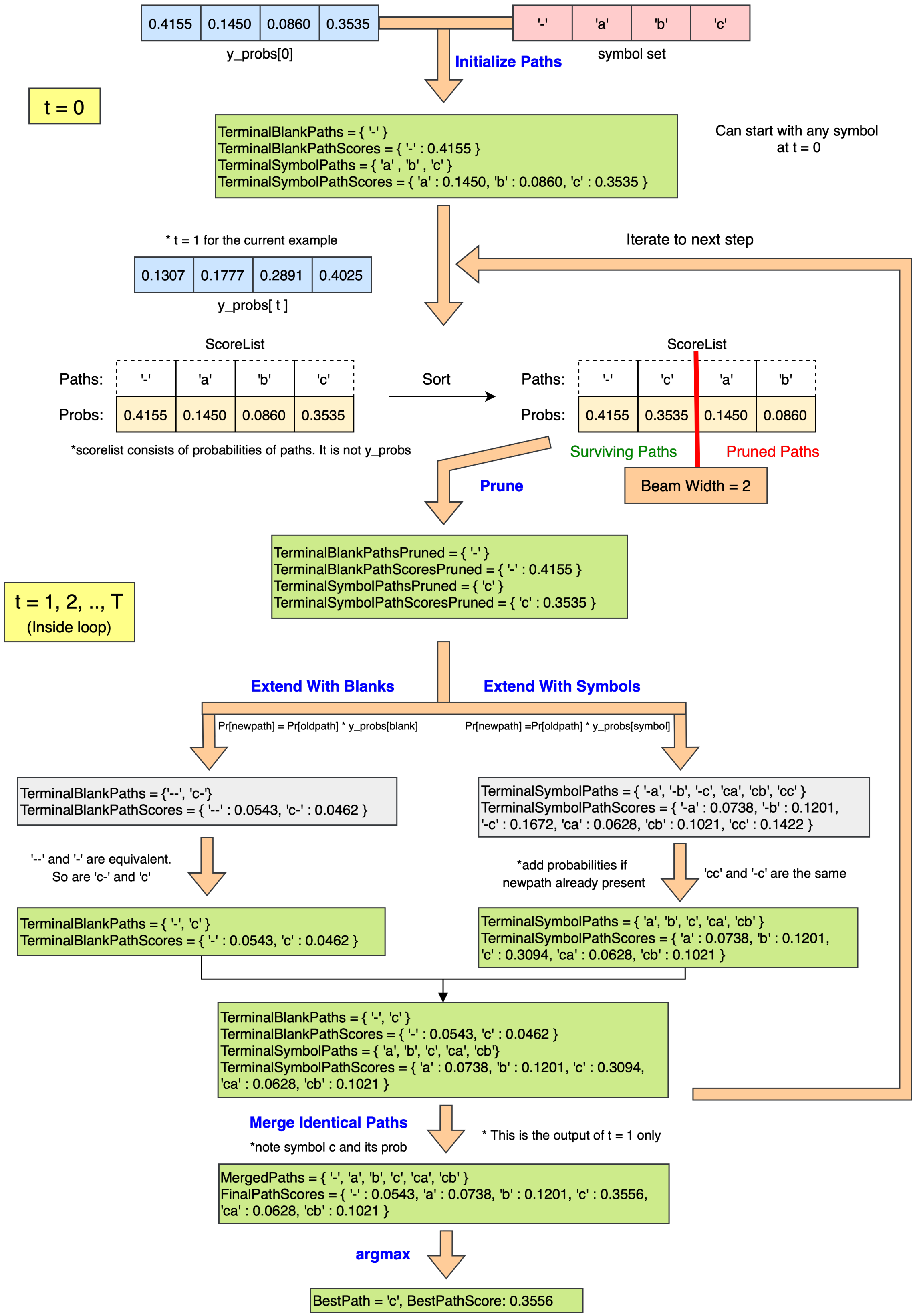

Figure 19: Beam Search

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). As mentioned earlier, the figure explains the pseudocode followed in the lectures. You are also welcomed to try a more efficient way. One of which has the following pseudocode.

Efficient Beam Search:

```
0. Intialize:
    decoded_path = list()
    sequences = [[list(), 1.0]]
    ordered = None
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'
6. Merge paths in 'ordered', and get merged paths scores
7. Select best path based on merged path scores, and return
```

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

```
class BeamSearchDecoder(object):

    def __init__(self, symbol_set, beam_width):

        self.symbol_set = symbol_set
        self.beam_width = beam_width

    def decode(self, y_probs):

        T = y_probs.shape[1]
        bestPath, FinalPathScore = None, None


        #return bestPath, FinalPathScore
```

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 4 and Figure 6.

```
*** 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. 20 depicts a toy problem for understanding Beam Search. Here, we are performing Beam Search over a vocabulary of {-, A, B}, where " - " is the BLANK character. The **beam width is 3** and we perform three decoding steps. Table 6 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 6: Probabilities of each symbol in the vocabulary over three consecutive decoding steps

To perform beam search with a beam width = 3, we select the top 3 most probable sequences at each time

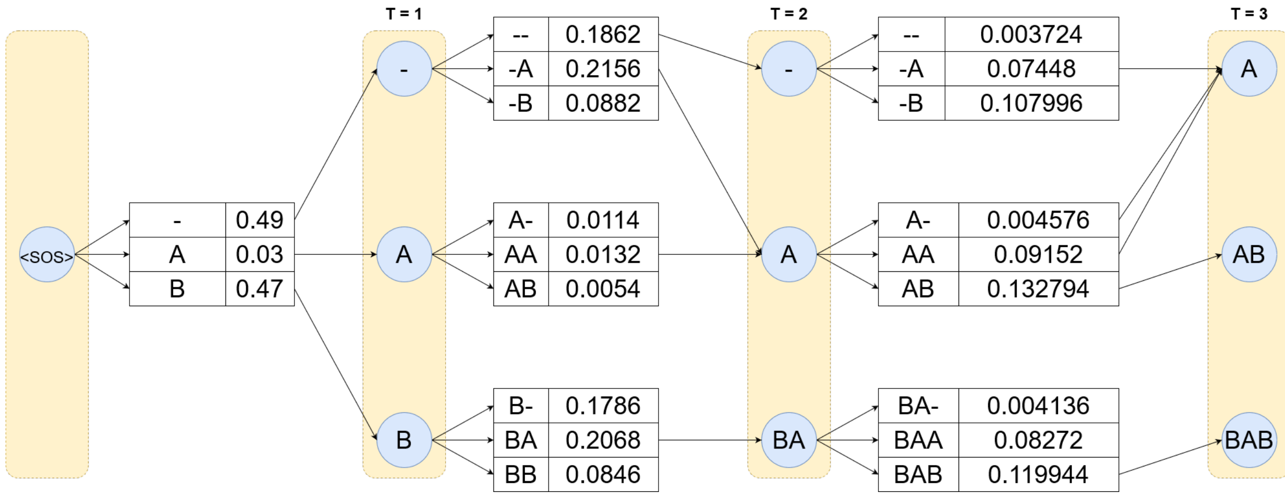

Figure 20: Beam Search over a vocabulary / symbols set = {-, A, B} with beam width (k) = 3. (" - " == BLANK). The blue shaded nodes indicate the compressed decoded sequence at given time step, and the expansion tables show the probability (right column) and symbols (left column) at each time step pre-pended with the current decoded sequence

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 20, 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.