Homework 3 Part 1

RNNs and GRUs and Search, Oh My!

11-785: Introduction to Deep Learning (Spring 2025)

Release Day: **March, 1st, 2025, 11:59 EST**
Early Submission Deadline: **March, 14th, 2025, 11:59 EST**
Final Submission Deadline: **March, 28th, 2025, 11:59 EST**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.
- **Directions:**
  - 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.

Homework objectives

If you complete this homework successfully, you would ideally have learned:

- How to write code to implement an RNN from scratch
  - How to implement RNN cells and how to build a simple RNN classifier using RNN cells
  - How to implement GRU cells and how to build a character predictor using GRU cells
  - How to implement CTC loss forward and backward pass
    (CTC loss is a criterion specific for recursive neural network, its functionality is similar to how we used cross-entropy loss for HW2P2)
- How to implement RNN, GRU and CTC loss using auto-differentiation - See the MyTorch section for explanation on this. (draft)
- How to decode the model output probabilities to get an understandable output
  - How to implement Greedy Search
  - How to implement Beam Search

TL;DR

This assignment provides an in-depth exploration of **Recurrent Neural Networks (RNNs)**, **Gated Recurrent Units (GRUs)**, **Connectionist Temporal Classification (CTC) loss**, and **decoding algorithms** for sequence modeling tasks. It is divided into several parts that build on each other, enabling you to construct models from scratch while learning foundational concepts. The key topics covered include:

RNNs

- Learn to implement RNN cells from scratch, including forward and backward passes.
- Understand how hidden states propagate through time and layers in an RNN.
- Use these cells to build an RNN-based phoneme classifier.

GRUs

- Implement a GRU cell, a variant of RNNs designed to handle vanishing/exploding gradients.
- Code its forward and backward passes with specialized gates (reset, update, and candidate gates).
- Construct a GRU-based character predictor for sequential data.

Connectionist Temporal Classification (CTC) Loss

- Implement CTC loss using the forward-backward algorithm to align model predictions with target sequences.
- Extend target sequences with blank symbols, calculate forward and backward probabilities, and compute posterior probabilities to derive loss.
- Use this loss to train models for sequence-to-sequence tasks with variable alignments.

Decoding Algorithms

- **Greedy Search:** Implement a simple decoding method that selects the most probable output at each timestep and collapses repeated symbols.
- **Beam Search:** Build a more sophisticated decoding method that considers multiple potential paths to find a high-probability output sequence.

Tools and Frameworks

- **No Auto-Differentiation:** You must manually compute gradients for all models (in the standard version) using NumPy, enhancing your understanding of the underlying computations.
- **Modular Library Design:** The components you develop (RNN, GRU, loss functions, and decoders) will integrate into *mytorch*, your custom deep learning library.

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.

This homework comes in two versions: the Standard version and the Autograd version. The Standard version requires you to manually code the backpropagation for all components. On the other hand, the Autograd version simplifies backpropagation. If you choose the Autograd version, you will use the gradients of primitive operations to automatically calculate the gradients of more complex functions, eliminating the need to manually implement backward propagation for each component. Both versions are included in the handout, and you can choose which version of the homework to work with based on your preference and desired level of challenge. Your choice of version will not affect your grade—both versions will be scored equally.

The files in your both versions of the homework are structured in such a way that you can easily import and reuse modules of code for your subsequent homework. For Homework 3, MyTorch will have the following structure:

```
HW3P1
├── mytorch
│   ├── nn
│   │   ├── linear.py
│   │   ├── activation.py
│   │   └── loss.py
│   ├── gru_cell.py
│   ├── rnn_cell.py
│   ├── autograd_engine.py
│   ├── funtional.py
│   └── utils.py
├── models
│   ├── char_predictor.py
│   └── rnn_classifier.py
├── MCQ
│   └── mcq.py
├── CTC
│   ├── CTC.py
│   └── CTCDecoding.py
└── autograder
    └── runner.py
```

- **Hand-in** your code by running the following command from the directory containing the handout, then **SUBMIT** the created *handin.tar* file to autolab:

  ```
  sh create_tarball.sh
  ```

- **Debug** individual sections of your code by running the following command from the top level directory:

  ```
  python3 autograder/toy_runner.py test_name
  ```

  The `test_name` are `rnn, gru, ctc, beam_search` based on which section you are testing.

- **Autograde** your code by running the following command from the top level directory:

  ```
  python3 autograder/runner.py
  ```

  Test individual sections of your code by running the following command from the top level directory:

  ```
  python3 autograder/runner.py test_name
  ```

  The `test_name` are `mcq, rnn, gru, ctc, search` based on which section you are testing.

- **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.
- **Autograd version (for students choosing this approach):** We have implemented all the rudimentary functions you need to complete the Autograd version of the homework, allowing you to focus on the key concepts of the homework instead. The autograd_engine.py holds the Autograd class, which is responsible for tracking the sequence of operations being performed and initiating the backpropagation algorithm once the forward pass is complete. The utils.py file contains various utility functions, and functional.py implements the backward pass for primitive operations. If you choose to solve the Autograd version of this homework, you will use these operations to automatically handle backpropagation.
- **Note!** *Please note that doing the autograd version of this assignment requires experience with the previous autograds (from HW1 and HW2).*
- **Autograd Example**
  - Suppose that we are building a single-layer MLP. As we know, this is an affine combination, for example,

$$y = x * W + b$$

  - This can be decomposed into the following two operations on x:

$$h = x * W \tag{1}$$

$$y = h + b \tag{2}$$

  - Using our autograd engine, we can implement this function using the following steps:
    * First, we need to create an instance of the autograd engine using:

```
autograd = autograd_engine.Autograd()
```

    * For equation (1), we need to add a node to the computation graph performing multiplication, which would be done in the following way:

```
autograd_engine.add_operation(
    inputs = [x, W], output = h,
    gradients_to_update = [None, dW],
    backward_operation = matmul_backward
)
```

    * Similarly, for equation (2):

```
autograd_engine.add_operation(
    inputs = [h, b], output = y,
    gradients_to_update = [None, db],
    backward_operation = add_backward
)
```

    * Invoke backpropagation by:

```
autograd_engine.backward(divergence)
```

* dW and db should be updated after this.

  The concept above could be leveraged in building more complex computation steps (with few lines of code).

Contents

1 Notation

****Numpy Tips:**

- Use `A * B` for element-wise multiplication $A \odot B$.
- Use `A @ B` for matrix multiplication $A \cdot B$.
- Use `A / B` for element-wise division $A \oslash B$.

Linear Algebra Operations

A^T	Transpose of A
$A \odot B$	Element-wise (Hadamard) Product of A and B (i.e. every element of A is multiplied by the corresponding element of B. A and B must have identical size and shape)
$A \cdot B$	Matrix multiplication of A and B
$A \oslash B$	Element-wise division of A by B (i.e. every element of A is divided by the corresponding element of B. A and B must have identical size and shape)

Set Theory

$\mathbb{S}$	A set
$\mathbb{R}$	The set of real numbers
$\mathbb{R}^{N \times C}$	The set of N $\times$ C matrices containing real numbers

Functions and Operations

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$\log(x)$	Natural logarithm of x
$\varsigma(x)$	Sigmoid, $\dfrac{1}{(1 + \exp^{-x})}$
$\tanh(x)$	Hyperbolic tangent, $\dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
$\max_{\mathbb{S}} f$	The operator $\max_{a \in \mathbb{S}} f(a)$ returns the highest value $f(a)$ for all elements in the set $\mathbb{S}$
$\arg\max_{\mathbb{S}} f$	The operator $\arg\max_{a \in \mathbb{S}} f(a)$ returns the element of the set $\mathbb{S}$ that maximizes $f(a)$
$\sigma(x)$	Softmax function, $\sigma : \mathbb{R}^K \to (0,1)^K$ and $\sigma(x)_i = \dfrac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$ for $i = 1, ..., K$

Calculus

$\dfrac{dy}{dx}$	Derivative of scalar y with respect to scalar x
$\dfrac{\partial y}{\partial x}$	Partial derivative of scalar y with respect to scalar x
$\dfrac{\partial f(Z)}{\partial Z}$	Jacobian matrix $\mathbf{J} \in \mathbb{R}^{N \times M}$ of $f : \mathbb{R}^M \to \mathbb{R}^N$

2 Multiple Choice [5 points]

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

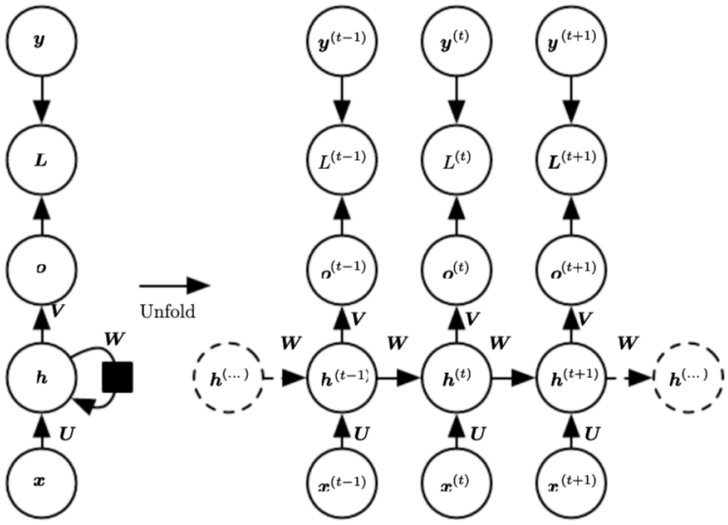

Figure 1: The high-level computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output values from http://www.deeplearningbook.org/contents/rnn.html. (Please note that this is just a general RNN, being shown as an 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: Read the following materials to better understand how to compute derivatives of vectors and matrices: Computing Derivatives, How to compute a derivative and answer the following question.**
Given matrices $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, and $C \in \mathbb{R}^{m \times p}$, and a scalar function $f(A, B, C) = (A \cdot B) \odot C$, where $\odot$ denotes element-wise multiplication, what is the derivative of $f(A, B, C)$ with respect to matrix B? Hint: How to compute a derivative page 5-6. [1 point]

(A) $A \cdot C^T$

(B) $C^T \cdot (A \odot I)$

(C) $C^T \cdot A$

(D) $A^T \cdot C$

(3) **Question 3: 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

(4) **Question 4: 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]`

(5) **Question 5: 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

3 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.

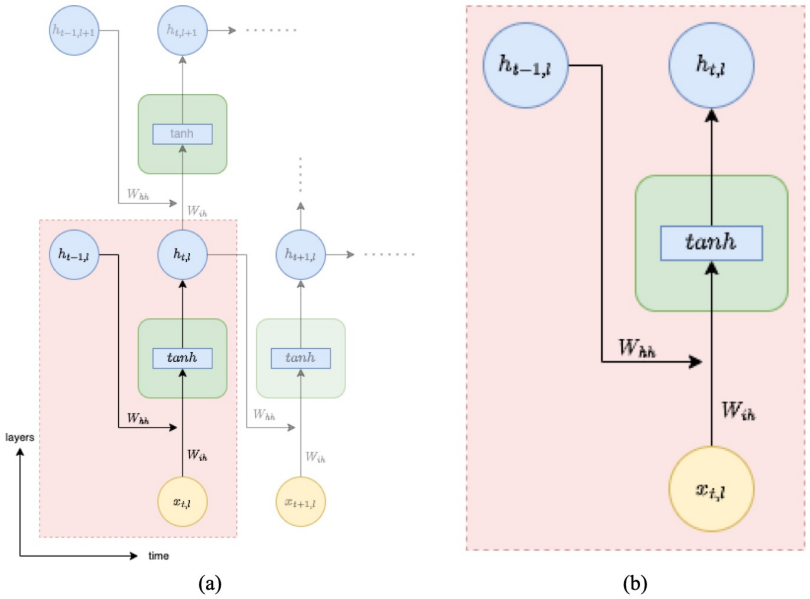

Figure 2: The red box shows one single RNN cell. RNNs can have multiple layers across multiple time steps. This is indicated by the two-axis in the bottom-left.

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

        return h_t
```

```
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`.

3.1 RNN Cell Forward (5 points)

Table 1: RNNCell Class Forward Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`input_size`	H_{in}	scalar	–	The number of expected features in the input x
`hidden_size`	H_{out}	scalar	–	The number of features in the hidden state h
`x`	x_t	matrix	$N \times H_{in}$	Input at current time step
`h_prev_t`	$h_{t-1,l}$	matrix	$N \times H_{out}$	Previous time step hidden state of current layer
`h_t`	h_t	matrix	$N \times H_{out}$	Current time step hidden state of current layer
`W_ih`	W_{ih}	matrix	$H_{out} \times H_{in}$	Weight between input and hidden
`b_ih`	b_{ih}	vector	H_{out}	Bias between input and hidden
`W_hh`	W_{hh}	matrix	$H_{out} \times H_{out}$	Weight between previous hidden and current hidden
`b_hh`	b_{hh}	vector	H_{out}	Bias between previous hidden and current hidden

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?

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.

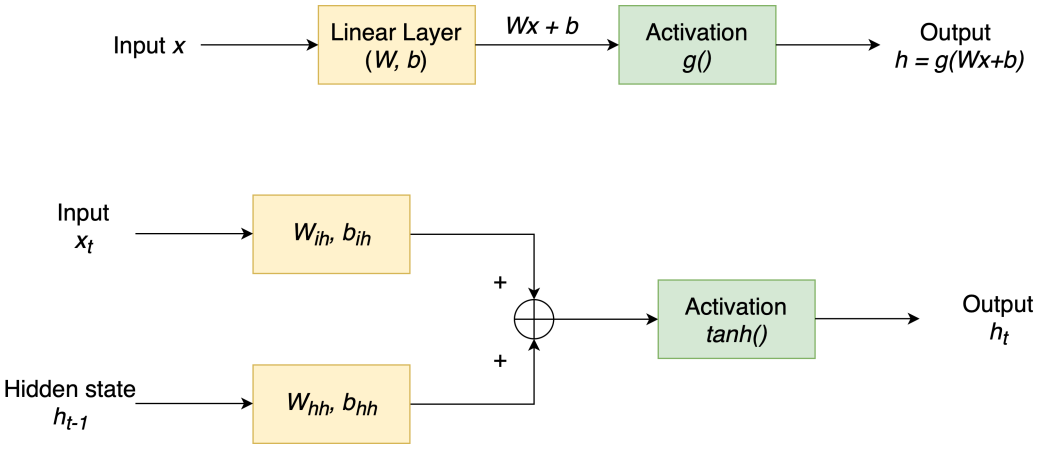

Figure 3: Perceptron (top) versus single layer, single time step RNN cell (bottom)

Affine function for inputs

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

Affine function for previous hidden state

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

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

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

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!

3.2 RNN Cell Backward (5 points)

Table 2: RNNCell Class Backward Components

Code Name	Math	Type	Shape	Meaning
`h_t`	$h_{t,l}$	matrix	$N \times H_{out}$	Hidden state at current time step and current layer
`x`	x_t	matrix	$N \times H_{in}$	Input at current time step
`h_prev_t`	$h_{t-1,l}$	matrix	$N \times H_{out}$	Hidden state at previous time step and current layer
`delta`	$\partial L/\partial h$	matrix	$N \times H_{out}$	gradient wrt current hidden layer
`dx`	$\partial L/\partial x$	matrix	$N \times H_{in}$	gradient wrt hidden state at current time step and input layer
`dh_prev_t`	$h_{t-1,l}$	matrix	$N \times H_{out}$	gradient wrt hidden state at previous time step and current layer
`dW_ih`	$\partial L/\partial W_{ih}$	matrix	$H_{out} \times H_{in}$	Gradient of weight between input and hidden
`db_ih`	$\partial L/\partial b_{ih}$	vector	H_{out}	Gradient of bias between input and hidden
`dW_hh`	$\partial L/\partial W_{hh}$	matrix	$H_{out} \times H_{out}$	Gradient of weight between previous hidden and current hidden
`db_hh`	$\partial L/\partial b_{hh}$	vector	H_{out}	Gradient of bias between previous hidden and current hidden

Given $\frac{\partial L}{\partial h_{t,l}}$ (`delta`) and forward equation

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

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. $\frac{\partial L}{\partial x_t}$ (`dx`) (returned by the method, explained below)
6. $\frac{\partial L}{\partial h_{t-1,l}}$ (`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. 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? Read How to compute a derivative page 13-50 with an example of backprop of LSTM cell!

3.3 RNN Phoneme Classifier (10 points)

In `models/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 3: RNNPhonemeClassifier Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`x`	x	matrix	$N\times$ `seq_len` $\times H_{in}$	Input
`h_0`	h_0	matrix	`num_layers`$\times N \times H_{out}$	Initial hidden states
`delta`	$\partial L/\partial h$	matrix	$N \times H_{out}$	Gradient w.r.t. last time step output

3.3.1 RNN Classifier Forward

Follow the diagram given below to complete the forward pass of RNN Phoneme Classifier. Zero-initialize h_0 if no h_0 is specified.

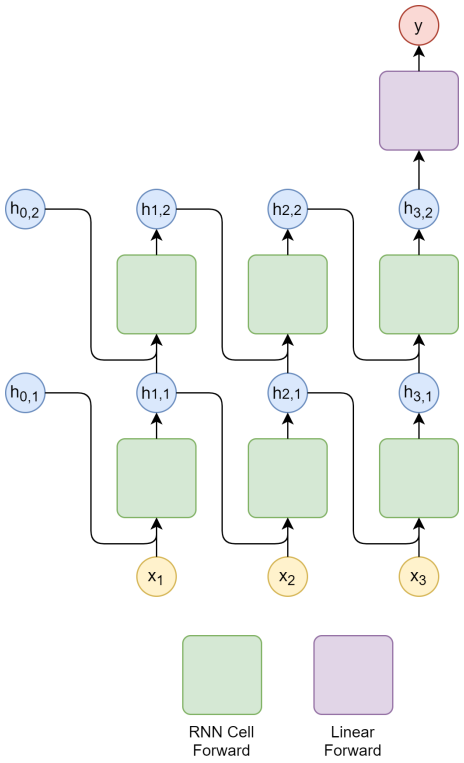

Figure 4: The forward computation flow for the RNN.

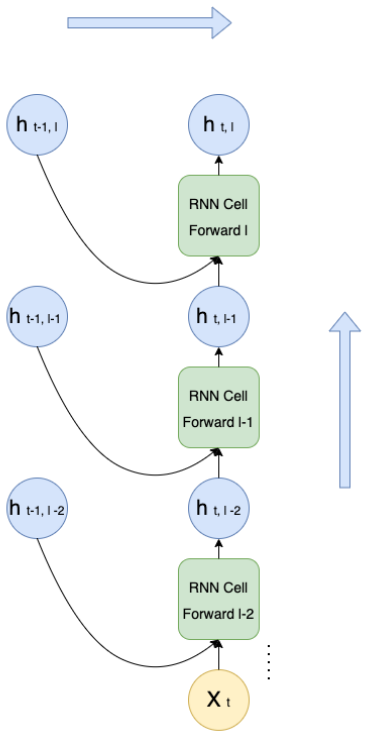

Figure 5: The forward computation flow for the RNN at time step t.

3.3.2 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.

Algorithm 1: Backward Pass for RNN

Input: delta: Gradient w.r.t. last time step output
Output: dx: Gradient of the loss with respect to the input sequence

```
for t ← seq_len − 1 to 0 do
    for l ← num_layers − 1 to 0 do
        // 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 l ≠ 0 then
            // If you aren't at the first layer, you will want to add dx to the dh from
               l-1th layer.
        end
    end
end
// Normalize dh by batch size since initial hidden states are also treated as
   parameters of the network (divide by batch size)
return dh ;            // Gradient of the loss with respect to the initial hidden states
```

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.

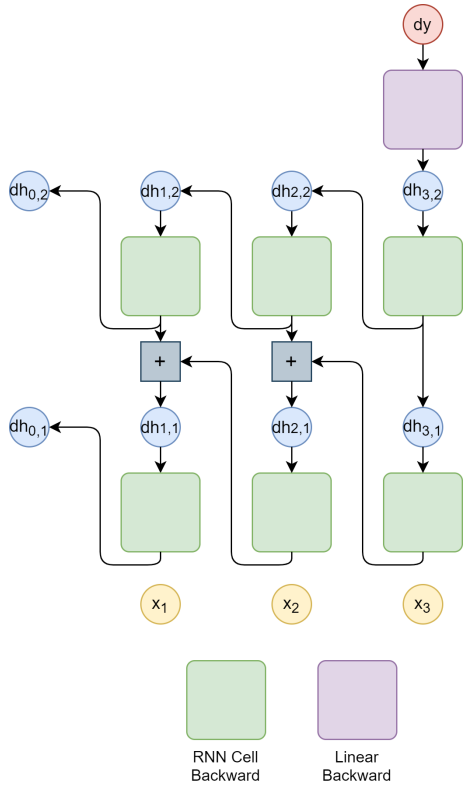

Figure 6: The backward computation flow for the RNN.

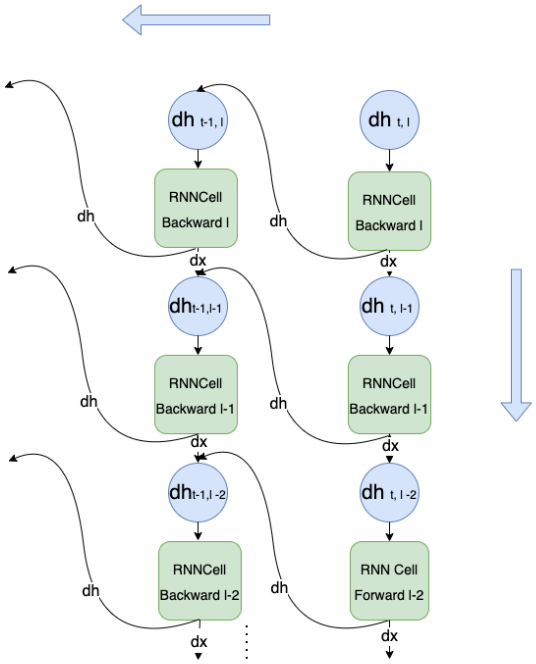

Figure 7: The backward computation flow for the RNN at time step t.

4 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.

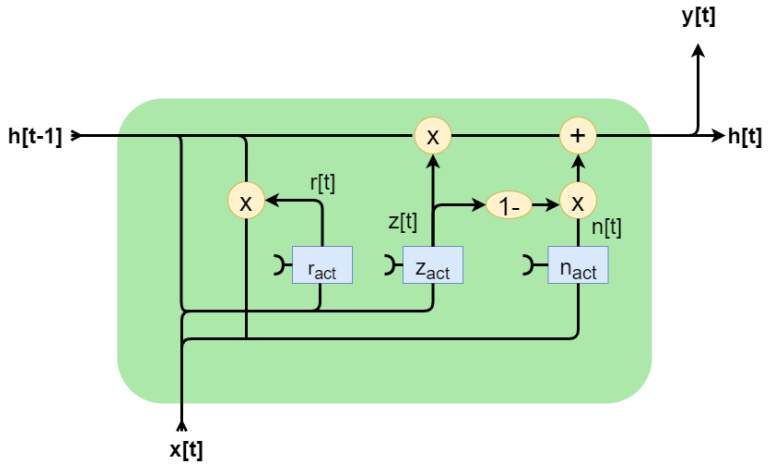

Figure 8: GRU Cell

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 arguments, forward expects input `x` and `h_prev_t`.
- As attributes, 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 an argument, 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.

4.1 GRU Cell Forward (5 points)

Table 4: GRUCell Forward Components

Code Name	Math	Type	Shape	Meaning
`input_size`	H_{in}	scalar	–	The number of expected features in the input x
`hidden_size`	H_{out}	scalar	–	The number of features in the hidden state h
`x`	x_t	vector	H_{in}	observation at the current time-step
`h_prev_t`	h_{t-1}	vector	H_{out}	hidden state at previous time-step
`Wrx`	W_{rx}	matrix	$H_{out} \times H_{in}$	Weight matrix for input (for reset gate)
`Wzx`	W_{zx}	matrix	$H_{out} \times H_{in}$	Weight matrix for input (for update gate)
`Wnx`	W_{nx}	matrix	$H_{out} \times H_{in}$	Weight matrix for input (for candidate hidden state)
`Wrh`	W_{rh}	matrix	$H_{out} \times H_{out}$	Weight matrix for hidden state (for reset gate)
`Wzh`	W_{zh}	matrix	$H_{out} \times H_{out}$	Weight matrix for hidden state (for update gate)
`Wnh`	W_{nh}	matrix	$H_{out} \times H_{out}$	Weight matrix for hidden state (for candidate hidden state)
`brx`	b_{rx}	vector	H_{out}	bias vector for input (for reset gate)
`bzx`	b_{zx}	vector	H_{out}	bias vector for input (for update gate)
`bnx`	b_{nx}	vector	H_{out}	bias vector for input (for candidate hidden state)
`brh`	b_{rh}	vector	H_{out}	bias vector for hidden state (for reset gate)
`bzh`	b_{zh}	vector	H_{out}	bias vector for hidden state (for update gate)
`bnh`	b_{nh}	vector	H_{out}	bias vector for hidden state (for candidate hidden state)

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 docu-

mentation.) The equations for a GRU cell are the following:

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

Derive the appropriate shape of r_t, z_t, n_t, h_t using the equation given. Note the difference between element-wise multiplication and matrix multiplication.
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:

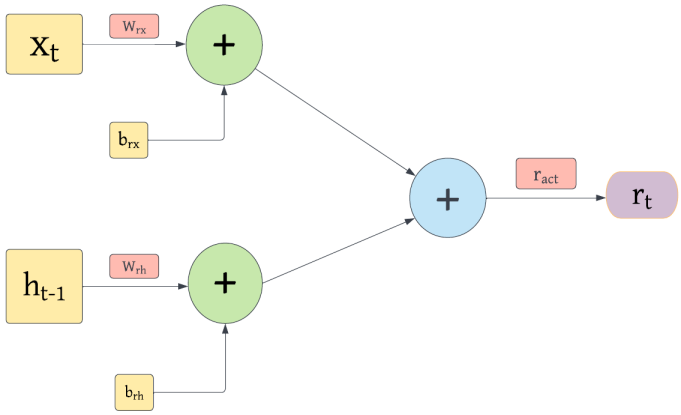

Figure 9: The computation for r_t

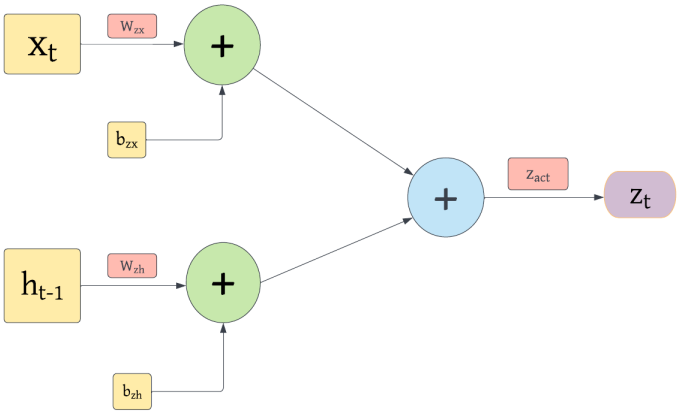

Figure 10: The computation for z_t

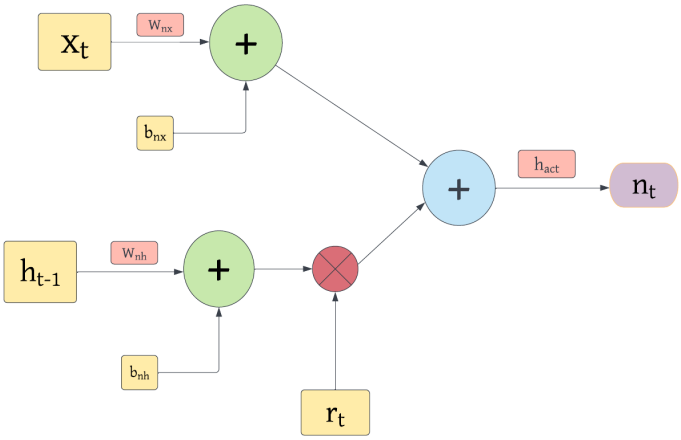

Figure 11: The computation for n_t

4.2 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.

Table 5: GRUCell Backward Components

Code Name	Math	Type	Shape	Meaning
`delta`	$\partial L/\partial h_t$	vector	H_{out}	Gradient of loss w.r.t h_t
`dWrx`	$\partial L/\partial W_{rx}$	matrix	$H_{out} \times H_{in}$	Gradient of loss w.r.t W_{rx}
`dWzx`	$\partial L/\partial W_{zx}$	matrix	$H_{out} \times H_{in}$	Gradient of loss w.r.t W_{zx}
`dWnx`	$\partial L/\partial W_{nx}$	matrix	$H_{out} \times H_{in}$	Gradient of loss w.r.t W_{nx}
`dWrh`	$\partial L/\partial W_{rh}$	matrix	$H_{out} \times H_{out}$	Gradient of loss w.r.t W_{rh}
`dWzh`	$\partial L/\partial W_{zh}$	matrix	$H_{out} \times H_{out}$	Gradient of loss w.r.t W_{zh}
`dWnh`	$\partial L/\partial W_{nh}$	matrix	$H_{out} \times H_{out}$	Gradient of loss w.r.t W_{nh}
`dbrx`	$\partial L/\partial b_{rx}$	vector	H_{out}	Gradient of loss w.r.t b_{rx}
`dbzx`	$\partial L/\partial b_{zx}$	vector	H_{out}	Gradient of loss w.r.t b_{zx}
`dbnx`	$\partial L/\partial b_{nx}$	vector	H_{out}	Gradient of loss w.r.t b_{nx}
`dbrh`	$\partial L/\partial b_{rh}$	vector	H_{out}	Gradient of loss w.r.t b_{rh}
`dbzh`	$\partial L/\partial b_{zh}$	vector	H_{out}	Gradient of loss w.r.t b_{zh}
`dbnh`	$\partial L/\partial b_{nh}$	vector	H_{out}	Gradient of loss w.r.t b_{nh}
`dx`	$\partial L/\partial x_t$	vector	H_{in}	Gradient of loss w.r.t x_t
`dh_prev_t`	$\partial L/\partial h_{t-1}$	vector	H_{out}	Gradient of loss w.r.t h_{t-1}

The table above lists the 14 gradients to be computed, and `delta` is the input of the backward function.

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. For your convenience, the forward equantions are listed here:

$$\mathbf{r}_t = \sigma(\mathbf{W}_{rx} \cdot \mathbf{x}_t + \mathbf{b}_{rx} + \mathbf{W}_{rh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{rh}) \tag{8}$$

$$\mathbf{z}_t = \sigma(\mathbf{W}_{zx} \cdot \mathbf{x}_t + \mathbf{b}_{zx} + \mathbf{W}_{zh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{zh}) \tag{9}$$

$$\mathbf{n}_t = tanh(\mathbf{W}_{nx} \cdot \mathbf{x}_t + \mathbf{b}_{nx} + \mathbf{r}_t \odot (\mathbf{W}_{nh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{nh})) \tag{10}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{n}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1} \tag{11}$$

In the backward calculation, we start from terms involved in equation 11 and work back to terms involved in equation 8.

1. **Forward Eqn:** $\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{n}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1}$
   (a) $\dfrac{\partial L}{\partial z_t} = \dfrac{\partial L}{\partial h_t} \times \dfrac{\partial h_t}{\partial z_t}$
   (b) $\dfrac{\partial L}{\partial n_t} = \dfrac{\partial L}{\partial h_t} \times \dfrac{\partial h_t}{\partial n_t}$
2. **Forward Eqn:** $\mathbf{n}_t = tanh(\mathbf{W}_{nx} \cdot \mathbf{x}_t + \mathbf{b}_{nx} + \mathbf{r}_t \odot (\mathbf{W}_{nh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{nh}))$
   (a) $\dfrac{\partial L}{\partial W_{nx}} = \dfrac{\partial L}{\partial n_t} \times \dfrac{\partial n_t}{\partial W_{nx}}$
   (b) $\dfrac{\partial L}{\partial b_{nx}} = \dfrac{\partial L}{\partial n_t} \times \dfrac{\partial n_t}{\partial b_{nx}}$
   (c) $\dfrac{\partial L}{\partial r_t} = \dfrac{\partial L}{\partial n_t} \times \dfrac{\partial n_t}{\partial r_t}$
   (d) $\dfrac{\partial L}{\partial W_{nh}} = \dfrac{\partial L}{\partial n_t} \times \dfrac{\partial n_t}{\partial W_{nh}}$

(e) $\frac{\partial L}{\partial b_{nh}} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial b_{nh}}$

3. **Forward Eqn:** $\mathbf{z}_t = \sigma(\mathbf{W}_{zx} \cdot \mathbf{x}_t + \mathbf{b}_{zx} + \mathbf{W}_{zh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{zh})$

(a) $\frac{\partial L}{\partial W_{zx}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial W_{zx}}$

(b) $\frac{\partial L}{\partial b_{zx}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial b_{zx}}$

(c) $\frac{\partial L}{\partial W_{zh}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial W_{zh}}$

(d) $\frac{\partial L}{\partial b_{zh}} = \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial b_{zh}}$

4. **Forward Eqn:** $\mathbf{r}_t = \sigma(\mathbf{W}_{rx} \cdot \mathbf{x}_t + \mathbf{b}_{rx} + \mathbf{W}_{rh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{rh})$

(a) $\frac{\partial L}{\partial W_{rx}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial W_{rx}}$

(b) $\frac{\partial L}{\partial b_{rx}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial b_{rx}}$

(c) $\frac{\partial L}{\partial W_{rh}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial W_{rh}}$

(d) $\frac{\partial L}{\partial b_{rh}} = \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial b_{rh}}$

5. **Terms involved in multiple forward equations:**

(a) $\frac{\partial L}{\partial x_t} = \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial x_t} + \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial x_t} + \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial x_t}$

(b) $\frac{\partial L}{\partial h_{t-1}} = \frac{\partial L}{\partial h_t} \times \frac{\partial h_t}{\partial h_{t-1}} + \frac{\partial L}{\partial n_t} \times \frac{\partial n_t}{\partial h_{t-1}} + \frac{\partial L}{\partial z_t} \times \frac{\partial z_t}{\partial h_{t-1}} + \frac{\partial L}{\partial r_t} \times \frac{\partial r_t}{\partial h_{t-1}}$

4.3 GRU Inference (10 points)

In `models/char_predictor.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[1] 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 `models/char_predictor.py` and use the inference function (also in `models/char_predictor.py`) to use the neural network that you have created to get the outputs.

[1] In deep learning, "logits" refer to the raw, unnormalized predictions generated by the output layer of a neural network.

5 CTC

Connectionist temporal classification (CTC) is a type of neural network output and associated scoring function, for training recurrent neural networks (RNNs) such as LSTM networks to tackle sequence problems where the timing is variable.
In Homework 3 Part 2, for the utterance to phoneme mapping task, you will utilize CTC Loss to train a seq-to-seq model. In this part, **CTC/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 `(CTC/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 6: CTC Components

Code Name	Math	Type	Shape	Meaning
`target`	-	matrix	(target_len,)	Target sequence
`logits`	-	matrix	(input_len, len(Symbols))	Predicted 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 consists 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.

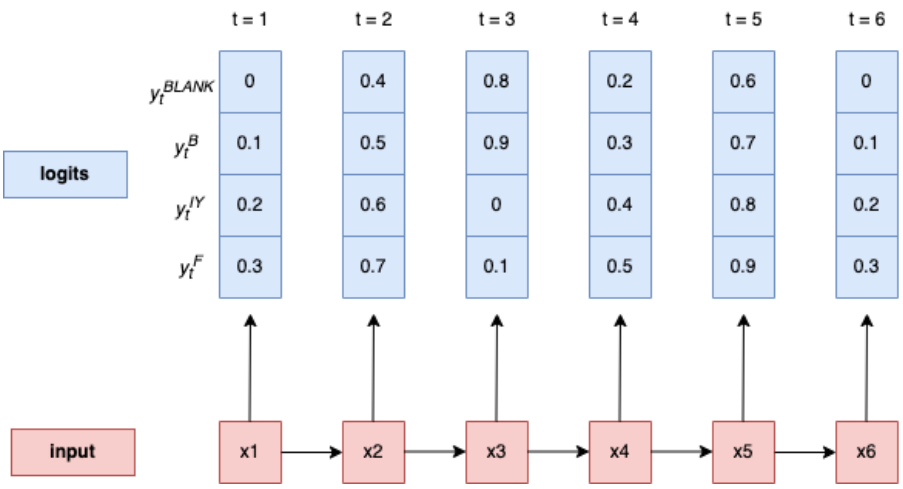

Figure 12: An overall CTC setup example

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.

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`.

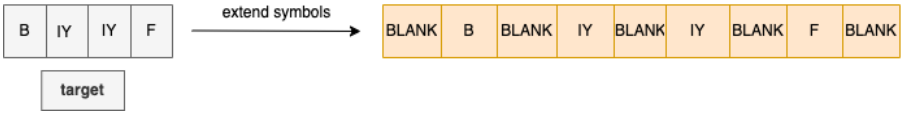

Figure 13: Extend symbols

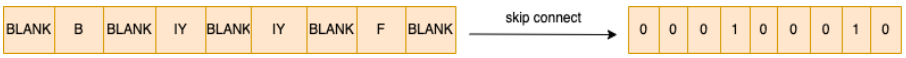

Figure 14: Skip connections

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`

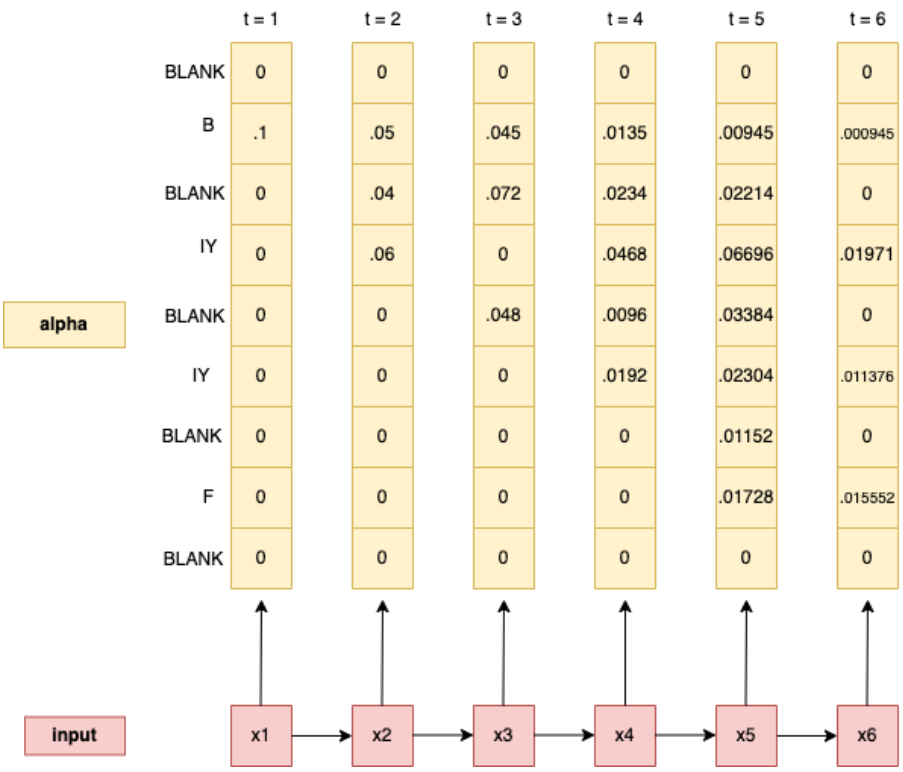

Figure 15: Forward Algorithm

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 (in other words, permitted to succeed) 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`

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

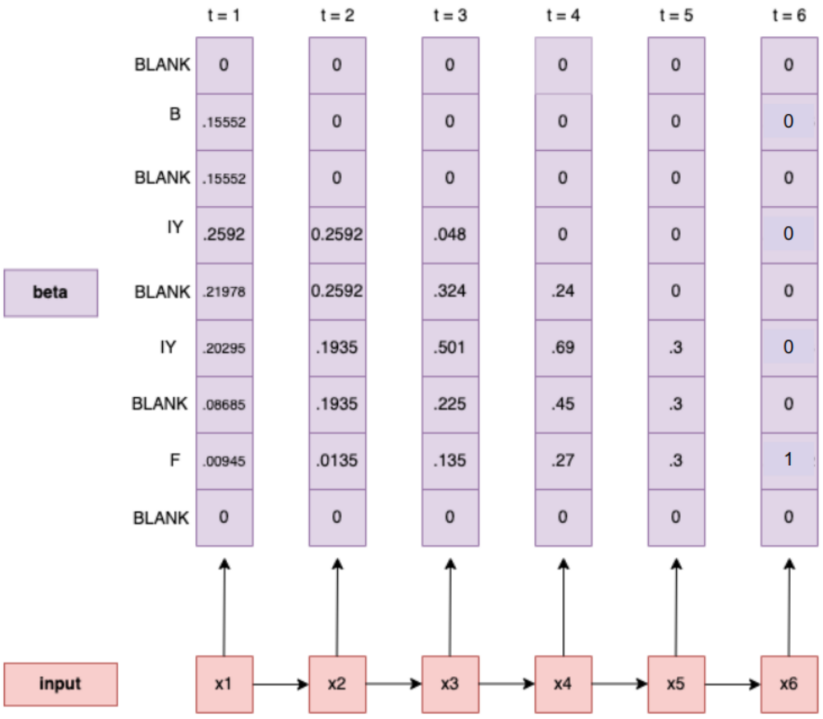

Figure 16: Backward Algorithm

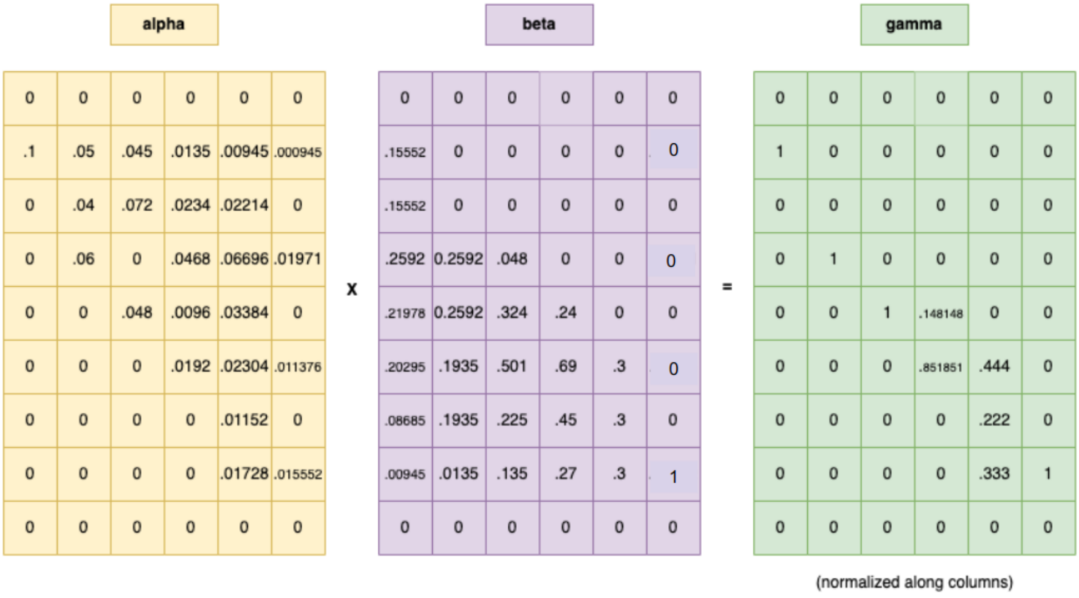

Figure 17: Posterior Probability

5.1 CTC Loss (25 points)

```
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 7: CTC Loss Components

Code Name	Math	Type	Shape	Meaning
`target`	-	matrix	(batch_size, paddedtargetlen)	Target sequences
`logits`	-	matrix	(seqlength, batch_size, len(Symbols))	Predicted 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.

5.1.1 CTC Forward

In `CTC/ctc.py`, you will implement **CTC Loss** using your implementation of the forward method.

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`.

5.1.2 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`

6 CTC Decoding: Greedy Search and Beam Search

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 `CTC/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:

$$Y^* = \arg\max_Y \ p(Y|X)$$

We will cover two approaches for the inference step:

- Greedy Search
- Beam Search

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

6.1 Greedy Search (5 points)

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.

6.1.1 Example

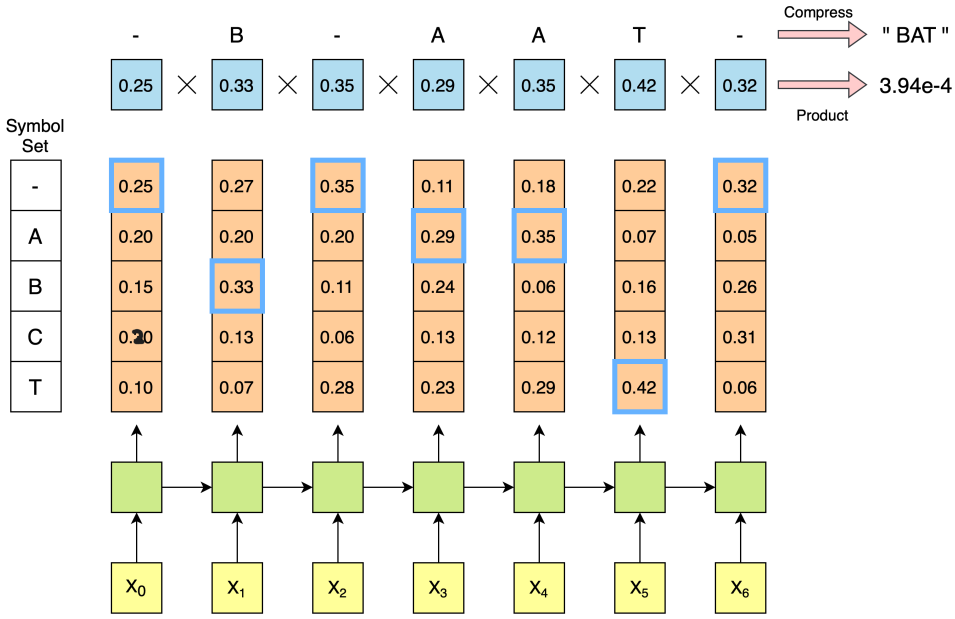

Figure 18: Greedy Search

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.

6.1.2 Pseudo-code

```
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.

6.2 Beam Search (15 points)

A straightforward approach, Greedy Search, selects the most probable token at each step without considering the broader context, often leading to suboptimal results. This is because an early decision might seem optimal at the moment but could prevent the model from forming a more coherent and accurate sequence in later steps. To address this limitation, Beam Search is used as a more effective decoding technique that balances computational efficiency and accuracy. Instead of selecting just one token per step like Greedy Search, Beam Search maintains the *top-k highest-scoring sequences (where k is called the beam width)* and expands only these promising candidates in the next step. This approach ensures that multiple potential sequences are explored simultaneously, increasing the likelihood of selecting a more accurate final output while avoiding the exhaustive computations required to evaluate all possible sequences. The probability of each sequence is computed based on the product of token probabilities, and longer sequences are often normalized using a length penalty to prevent bias toward shorter outputs.

Beam Search is particularly important in Connectionist Temporal Classification (CTC)-based models, such as those used in speech recognition, where blank symbols and repeated characters need to be collapsed to form a valid output. In these cases, Beam Search not only keeps track of the highest probability sequences but also merges equivalent sequences by summing their probabilities, ensuring that different paths leading to the same final result are appropriately considered.

Compared to Greedy Search, Beam Search produces significantly better results by maintaining flexibility in early decisions, while its computational efficiency makes it more practical than exhaustive search, which evaluates all possible sequences. 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. You can also read more about this here.

6.2.1 Example

Fig. 19 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 8 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 8: 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 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 19, where the sequence "A" at time step = 3 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.

6.2.2 Pseudo-code

Here's a skeleton of what you're expected to implement:

Your function `BeamSearch(SymbolSets, y_probs, BeamWidth)` should accept three arguments:

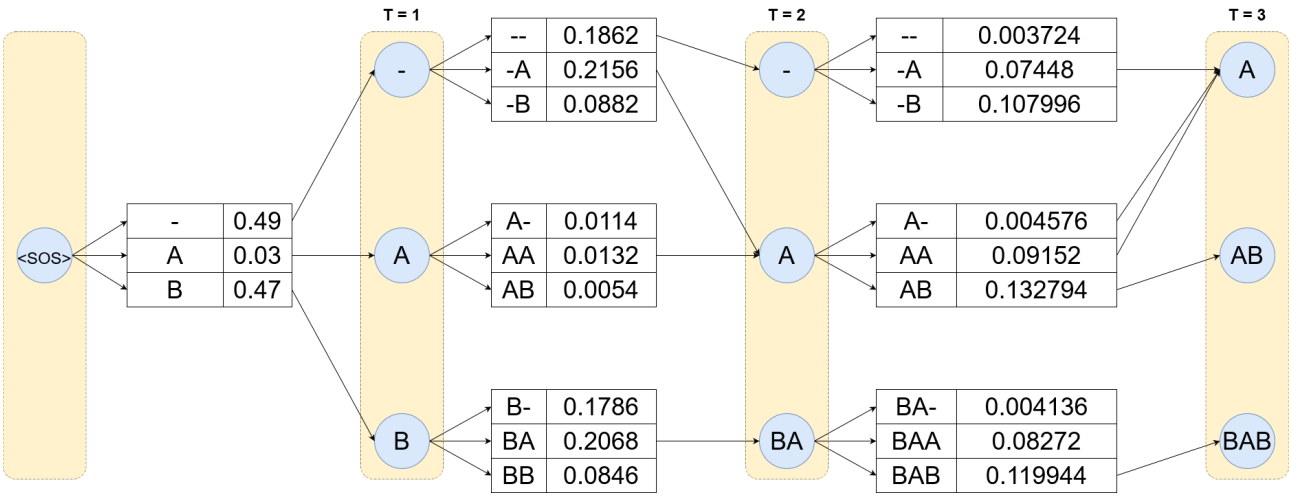

Figure 19: Beam Search over a vocabulary / symbols set = {-, A, B} with beam width (k) = 3. (" - " == BLANK). The blue shaded nodes indicate the compressed decoded sequence at given time step, and the expansion tables show the probability (right column) and symbols (left column) at each time step pre-pended with the current decoded sequence

- `SymbolSets`: Think of this as your alphabet, the range of all possible symbols that can be produced.
- `y_probs`: This is a tensor containing the probabilities for each symbol at each timestep.
- `BeamWidth`: This parameter limits the number of partial sequences (or 'paths') that you keep track of at each timestep.

Consider initializing the beam search with one path consisting of a 'blank' symbol. At each timestep, iterate over your current best paths. Consider extending each path by every possible new symbol, and calculate the score of the new paths.

Remember, when extending a path with a new symbol, you'll encounter three scenarios:

1. `The new symbol is the same as the last symbol on the path.`
2. `The last symbol of the path is blank.`
3. `The last symbol of the path is different from the new symbol and is not blank.`

After extending the paths, retain only the best paths, as determined by the 'beam width'.

At the end of all timesteps, remove the 'blank' symbol if it's at the end of a path. Translate the numeric symbol sequences to string sequences, sum up the scores for paths that end up with the same final sequence, and find the best overall path.

Remember to return the highest scoring path and the scores of all final paths.

Note: The provided y_probs are designed for a batch size of 1 for simplicity. Consider how you might adapt this function to handle larger batch sizes.

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). You are also welcomed to try a more efficient way. One of which has the following pseudocode.

Algorithm 2: Beam Search Decoding Algorithm

Input: $SymbolSet, y\_probs, BeamWidth$
Output: $BestPath, MergedPathScores$

Initialization: Create an empty dictionary **ActivePaths** with an initial blank path mapped to a score of 1.0 Create an empty dictionary **TempPaths** for temporary storage during each timestep
foreach *timestep* in *y_probs* **do**
  Extract the symbol probabilities at the current timestep;
  Sort ActivePaths by score in descending order and keep only the top $BeamWidth$ paths;
  foreach $(path, score)$ in ActivePaths **do**
    foreach *symbol* in *SymbolSet* including blank **do**
      Compute the new path based on the last symbol of *path*;
      Calculate the updated score: $new\_score = score \times probability(symbol)$;
      if *new_path* exists in TempPaths **then**
        Add *new_score* to the existing score of *new_path* in TempPaths;
      end
      else
        Store *new_score* in TempPaths with *new_path*;
      end
    end
  end
  Replace ActivePaths with TempPaths;
  Clear TempPaths for the next timestep;
end
Final Merging of Paths: Initialize **BestPath** as an empty string and **BestScore** as 0;
foreach $(path, score)$ in ActivePaths **do**
  Remove empty spaces at the beginning and end of the *path*;
  if *path* in MergedPaths **then**
    Add *score* to *path* in $MergedPathScores$;
  end
  else
    Store *score* in new path in $MergedPathScores$;
  end
  if *score* is greater than *BestScore* **then**
    BestPath $\leftarrow$ *path*;
    BestScore $\leftarrow$ *score*;
  end
end
Convert Best Path to Symbols: Convert $BestPath$ to their corresponding symbols in $SymbolSet$; Convert all *path* in $MergedPathScores$ to their corresponding symbols in $SymbolSet$
return $BestPath, MergedPathScores$;

(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
# your implementation

return bestPath, FinalPathScore
```

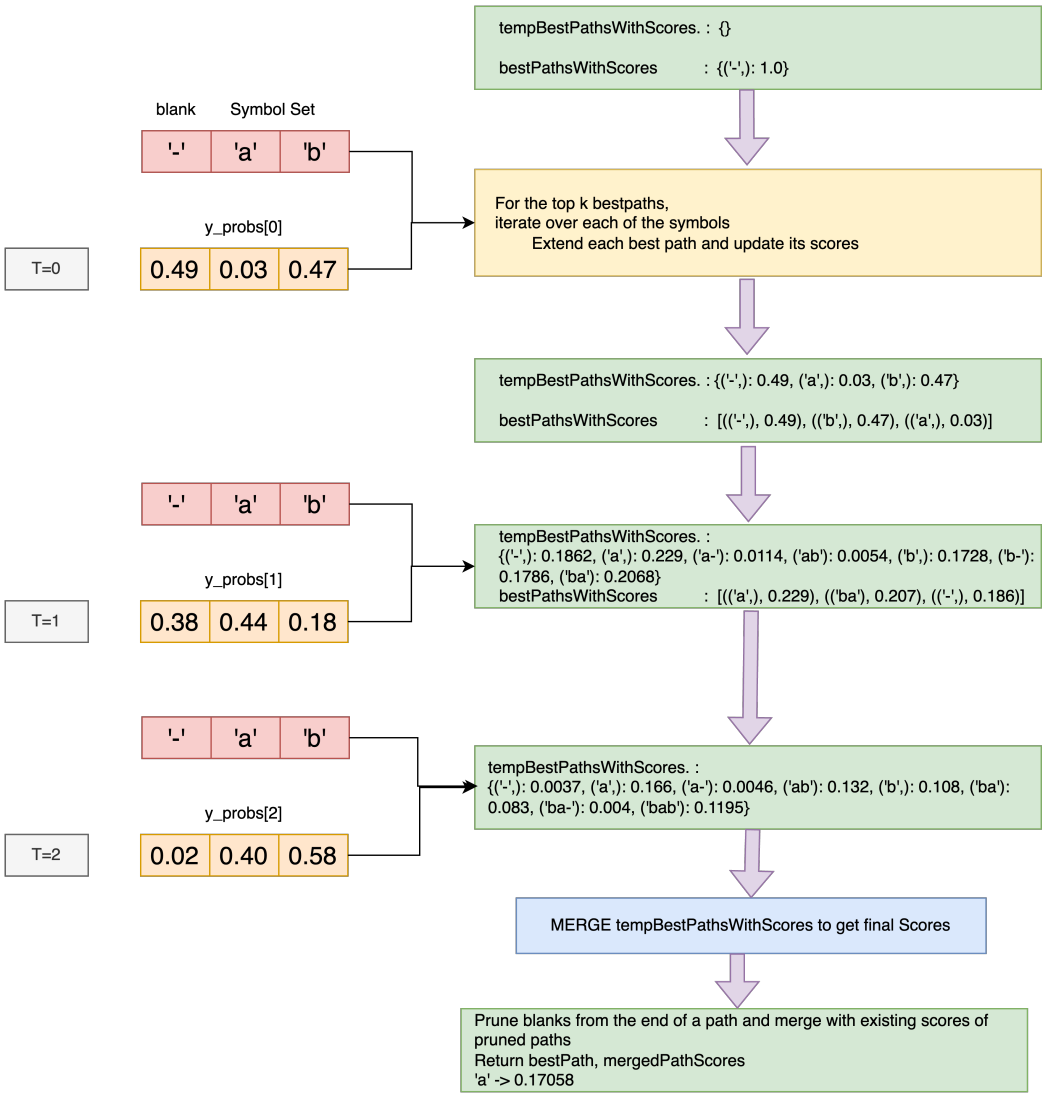

Figure 20: Efficient Beam Search procedure

7 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/toy_runner.py

You can also debug individual sections of your code by running the following command from the top level directory:

```
python3 autograder/toy_runner.py test_name
```

7.1 RNN

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

python3 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]]
```

7.2 GRU

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

python3 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]
```

7.3 CTC

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

python3 autograder/toy_runner.py ctc

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

```
--------------------
Section 4 - Extend Sequence with Blank
```

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

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

7.4 Beam Search

You can run tests for Beam Search only with the following command:

python3 autograder/toy_runner.py beam_search

The details of the toy test case is explained in Section 6.2.1.