Homework 3 Part 1

RNNs and GRUs and Search, Oh My!

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

Release Day: **October, 10th, 2025, 7:00PM EST**
Early Submission Deadline: **October, 31st, 2025, 11:59PM EST**
Final Submission Deadline: **November, 7th, 2025, 11:59PM 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 recurrent neural network, its functionality is similar to how we used cross-entropy loss for HW2P2)
- 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 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.

The files in 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
│   ├── funtional.py
│   └── utils.py
├── models
│   ├── char_predictor.py
│   └── rnn_classifier.py
├── MCQ
│   └── mcq.py
├── CTC
│   ├── CTC.py
│   └── CTCDecoding.py
├── autograder
│   └── runner.py
└── create_tarball.sh
```

- **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.

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, $\frac{1}{(1+\exp^{-x})}$
$\tanh(x)$	Hyperbolic tangent, $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
$\max_{\mathbb{S}} f$	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 $\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$ for $i = 1, ..., K$

Calculus

$\frac{dy}{dx}$	Derivative of scalar y with respect to scalar x
$\frac{\partial y}{\partial x}$	Partial derivative of scalar y with respect to scalar x
$\frac{\partial f(Z)}{\partial Z}$	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]**

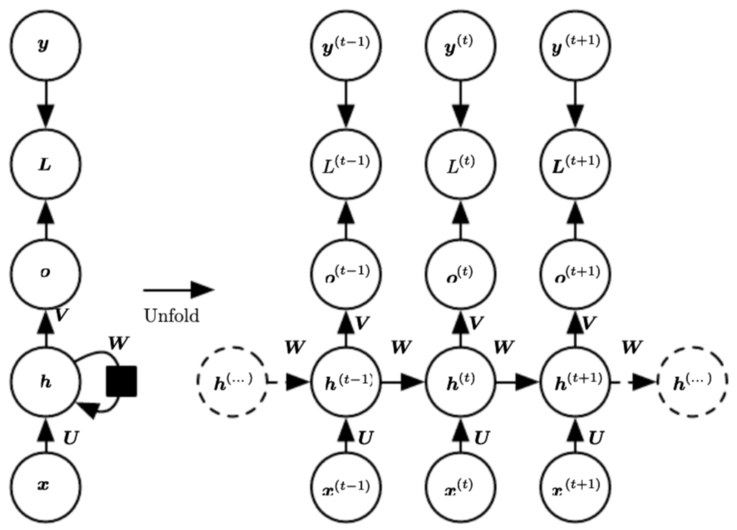


Figure 1: The high-level computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output values from http://www.deeplearningbook.org/contents/rnn.html. (Please note that this is just a general RNN, being shown as an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

To help orient yourself before diving into the implementation details, Figure 2 provides a high-level overview of the two modeling approaches implemented in this homework: the **RNN Phoneme Classifier** and the **GRU-based Character Predictor**.

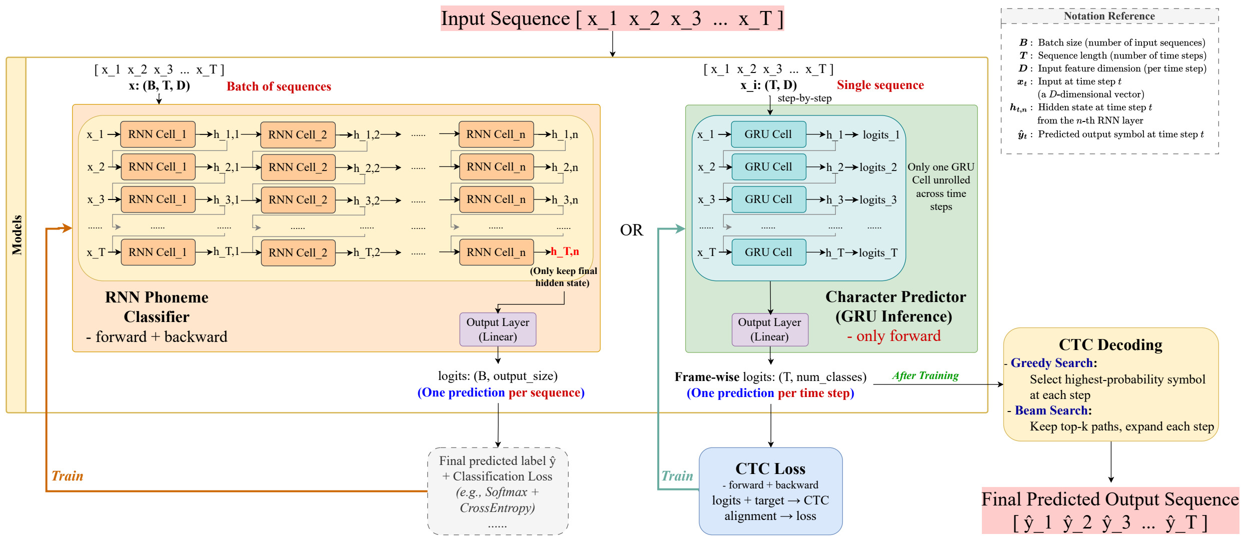

Input Sequence [x_1 x_2 x_3 ... x_T]
Models
[x_1 x_2 x_3 ... x_T]
x: (B, T, D)
Batch of sequences
RNN Cell_1
RNN Cell_2
RNN Cell_n
h_1,1
h_1,2
h_1,n
h_T,n
(Only keep final hidden state)
RNN Phoneme Classifier
- forward + backward
Output Layer (Linear)
logits: (B, output_size)
(One prediction per sequence)
OR
[x_1 x_2 x_3 ... x_T]
x_i: (T, D)
Single sequence
step-by-step
GRU Cell
logits_1
Only one GRU Cell unrolled across time steps
Character Predictor (GRU Inference)
- only forward
Output Layer (Linear)
Frame-wise logits: (T, num_classes)
(One prediction per time step)
After Training
Notation Reference
B : Batch size (number of input sequences)
T : Sequence length (number of time steps)
D : Input feature dimension (per time step)
x_t : Input at time step t (a D-dimensional vector)
h_t,n : Hidden state at time step t from the n-th RNN layer
ŷ_t : Predicted output symbol at time step t
CTC Decoding
- Greedy Search: Select highest-probability symbol at each step
- Beam Search: Keep top-k paths, expand each step
Train
Final predicted label ŷ + Classification Loss (e.g., Softmax + CrossEntropy) ......
Train
CTC Loss
- forward + backward
logits + target → CTC alignment → loss
Final Predicted Output Sequence [ŷ_1 ŷ_2 ŷ_3 ... ŷ_T]

Figure 2: High-level flowchart for HW3P1.

The **RNN Phoneme Classifier** and the **GRU-based Character Predictor** are designed to solve different tasks, and this is reflected in their input-output behavior and training objectives.

- The **RNN Phoneme Classifier** produces a single prediction per input sequence. Specifically, it processes the entire input and uses the final hidden state to generate logits through a linear classification layer. These logits are then passed through a Softmax activation to obtain phoneme label probabilities or directly used with a Cross-Entropy Loss function during training to compute the loss.
- In contrast, the **GRU-based Character Predictor** generates a prediction at every time step. This design allows it to model sequences of varying lengths, such as character transcriptions, and enables the use of **Connectionist Temporal Classification (CTC) loss** and **CTC decoding**, as introduced in Homework 3 Part 2. These techniques are crucial when the alignment between input and output sequences is not known a priori.

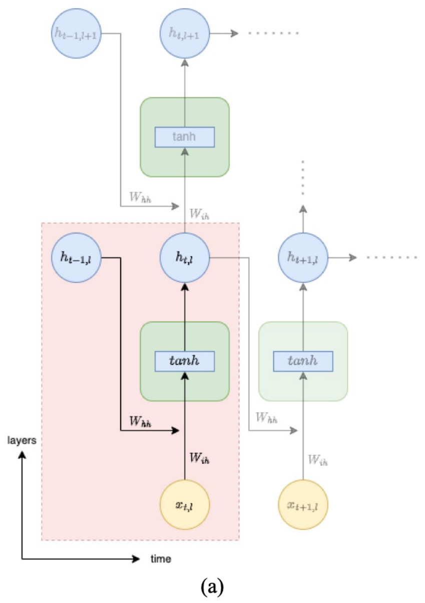
While the actual inference-time behavior may vary between the models, this distinction emphasizes two fundamental modeling paradigms:

- **Sequence-level classification**: using only the final hidden state.
- **Sequence-to-sequence prediction**: producing an output at every time step.

It is important to note that this distinction is not architecturally fundamental as one could replace GRU cells with vanilla RNN cells or vice versa. The design choices in this homework are pedagogical, intended to give you practical experience with both modeling styles and their corresponding learning objectives.

4 RNN Cell

Recurrent Neural Networks (RNNs) are designed to model sequential data by maintaining a "hidden state" that evolves over time. This "recurrent" nature allows the network to retain information from previous time steps, enabling it to learn temporal dependencies and context across a sequence. In practice, RNNs are usually unrolled over time and can be stacked across multiple layers. This might look a bit overwhelming at first.
To make this more approachable, this part of the homework focuses on the core unit of an RNN, i.e., **the RNN cell**. Specifically, we will implement a simple Elman RNN cell in `mytorch/rnn_cell.py`.
This cell processes one time step at a time, updating the hidden state using both the current input and the previous hidden state. Figure 3 provides a high-level overview of a typical RNN model, where the red box highlights the RNN cell that we will build and understand in detail.



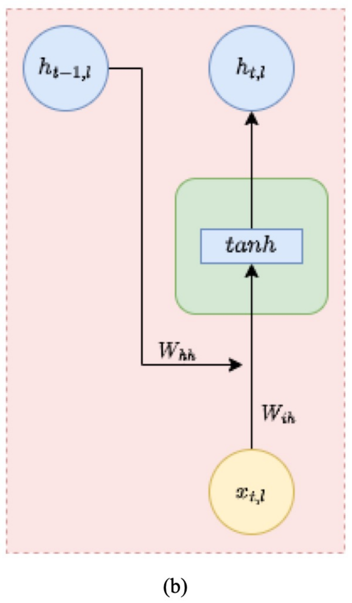


Figure 3: The red box shows one single RNN cell. RNNs can have multiple layers across multiple time steps. This is indicated by the two-axis in the bottom-left.

Below is the class structure to guide your implementation. Make sure to refer to the tables provided with each task as they serve as a key to help you interpret the pseudo code correctly and implement each part as intended.

```
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
```

The `RNNCell` class consists of the following methods:

- `__init__(input_size, hidden_size)`: Initializes weight and bias parameters, as well as their corresponding gradients. Also sets the activation function to `Tanh()`.
  (Already implemented; called when the object is created)
- `init_weights(W_ih, W_hh, b_ih, b_hh)`: Allows for manual initialization of weights and biases, useful for debugging or reproducibility.
  (Already implemented; does not return anything)
- `zero_grad()`: Resets all gradient buffers (`dW_ih`, `dW_hh`, `db_ih`, `db_hh`) to zero.
  (Already implemented; typically called before each backward pass)
- `__call__(x, h_prev_t)`: A wrapper that simply calls the `forward()` method on input `x` and previous hidden state `h_prev_t`.
  (Already implemented; returns `h_t`)
- `forward(x, h_prev_t)`: Computes the current hidden state h_t using the input x and the previous hidden state h_{t-1}.
  (To be implemented; returns `h_t`, the hidden state at the current time step)
- `backward(delta, h_t, h_prev_l, h_prev_t)`: Computes gradients of the loss with respect to the weights, biases, input, and previous hidden state.
  (To be implemented; returns `dx` and `dh_prev_t`)

In this homework, your main task is to implement the `forward` and `backward` methods of the `RNNCell` class.

4.1 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; the inputs are first passed through an affine transformation defined by the layer's weights and biases, followed by a non-linear activation function.

But how does the forward pass change for an RNN cell, which must also incorporate the hidden state from the previous time step?

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

Table 1: RNNCell Class Forward Components

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 transformation of the input:

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

Affine transformation of the previous hidden state:

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

We then add these two affine transformations and apply the tanh activation function. The resulting equation for the RNN cell's hidden state at time step t and layer l is:

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

Note that these equations are written for a single input example. In your implementation, you're working with batches of inputs, so be mindful of the tensor dimensions. **Ensure that your matrix multiplications are shape-compatible: you may need to transpose your weight matrices accordingly.**

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!

4.2 RNN Cell Backward (5 points)

Now that you've implemented the forward computation for a simple **Elman RNN Cell**, it's time to make it learn by implementing its **backward pass**.

Consider a task like *speech recognition*, where an RNN processes a sequence of audio frames and tries to predict phonemes or characters. The Elman RNN maintains a hidden state across time, allowing it to build temporal context. But during training, the model doesn't just need to *produce* outputs; It needs to *learn* from its mistakes.

This learning happens through **Backpropagation Through Time (BPTT)**. Since the same parameters (like W_{ih} and W_{hh}) are reused at every time step, we must accumulate gradients across time to correctly update them. The backward pass computes how the loss L changes with respect to these shared parameters, which allows the model to improve through gradient descent.

In this section, you'll compute the gradients for:

- the input-to-hidden and hidden-to-hidden weights,
- their associated biases,
- the input at the current time step,
- and the hidden state from the previous time step.

Recommended Reading

Before diving into the implementation, we highly recommend reviewing How to compute a derivative, especially pages **13–50**. This will walk you through backpropagation in LSTM cells, which shares many conceptual steps with the vanilla RNN cell we're implementing here.

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

Table 2: RNNCell Class Backward Components

How to approach this?

To compute the backward pass, start with the gradient $\frac{\partial L}{\partial h_{t,l}}$ (stored in `delta`) and the forward update equation:

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

Your goal is to compute the gradients listed below, based on the chain rule and the structure of the forward computation:

1. `self.dW_ih`
2. `self.dW_hh`
3. `self.db_ih`

4. `self.db_hh`
5. `dx` (returned by the method, explained below)
6. `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)

4.3 RNN Phoneme Classifier (10 points)

You will implement the forward and backward methods for the `RNN_Phoneme_Classifier` in

`models/rnn_classifier.py`.

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.

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

Table 3: RNNPhonemeClassifier Class Components

4.3.1 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.

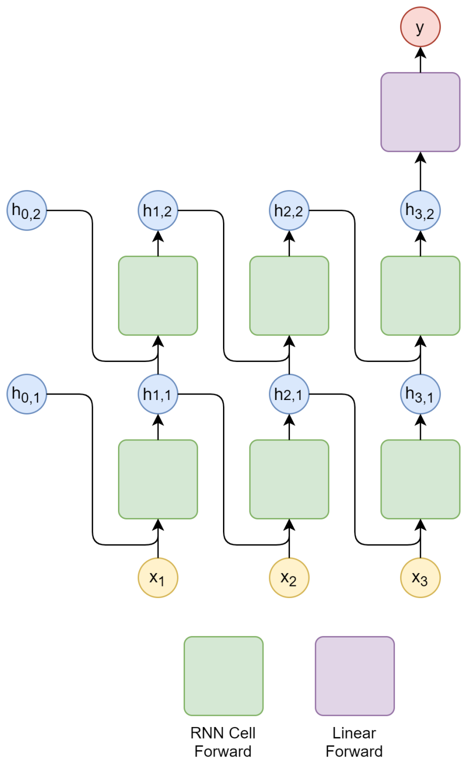


Figure 4: The forward computation flow for the RNN.

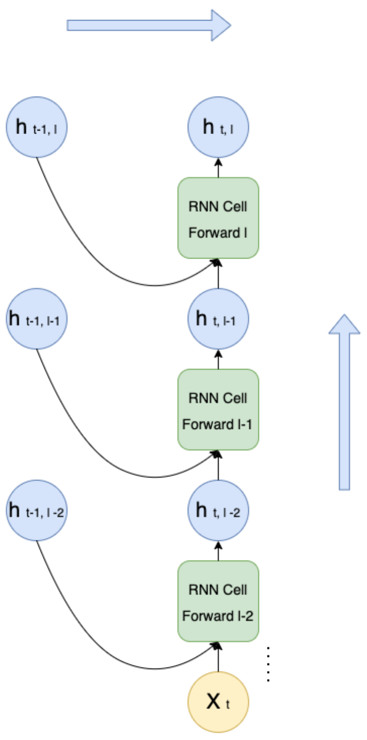


Figure 5: The forward computation flow for the RNN at time step t.

4.3.2 RNN Classifier Backward

While it might seem tricky to unravel what's going on in a backward pass of the RNN, we strongly recommend that you understand and implement the following pseudocode which encapsulates the logic outlined in the figures 6, 7.

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

```
1 for $t \leftarrow$ seq_len $- 1$ to $0$ do
2     for $l \leftarrow$ 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
3         if $l \neq 0$ then
              // If you aren't at the first layer, you will want to add $dx$ to the $dh$ from
                 l-1th layer.
4         end
5     end
6 end
  // Normalize dh by batch size since initial hidden states are also treated as
     parameters of the network (divide by batch size)
7 return $dh$ ;           // Gradient of the loss with respect to the initial hidden states
```

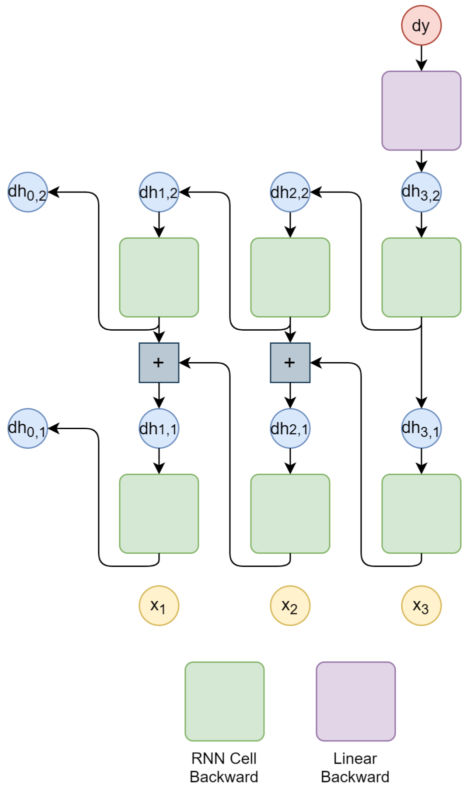


Figure 6: The backward computation flow for the RNN.

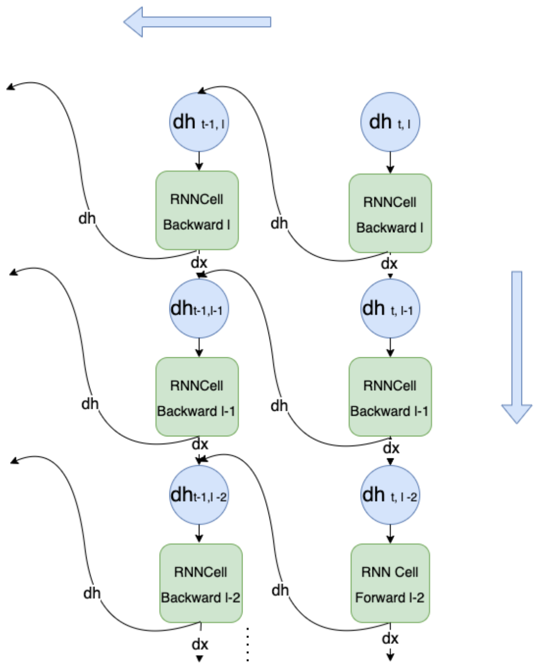


Figure 7: The backward computation flow for the RNN at time step t.

5 GRU Cell

Training standard Recurrent Neural Networks (RNNs) over long sequences often leads to the problem of **vanishing** or **exploding gradients** during backpropagation through time (BPTT). This issue arises due to the repeated multiplication of gradients through time via the recurrent weight matrices. As gradients are propagated backward through many time steps, they can either shrink exponentially towards zero (vanishing gradients) or grow without bound (exploding gradients), making it difficult for the model to learn long-term dependencies.

To address this, the **Long Short-Term Memory (LSTM)** network was introduced, incorporating gating mechanisms that regulate the flow of information and enable more stable gradient propagation across long sequences. A simplified alternative, the **Gated Recurrent Unit (GRU)**, was later proposed. GRUs retain the benefits of gated memory but use fewer parameters and are computationally more efficient than LSTMs, often achieving comparable performance in practice.

GRUs are widely used in tasks where temporal modeling is essential, such as **optical character recognition (OCR)** and **speech recognition from spectrograms**. In this section, you will develop a foundational understanding of how a GRU cell performs both the forward and backward passes, setting the stage for implementing GRU-based sequence models.

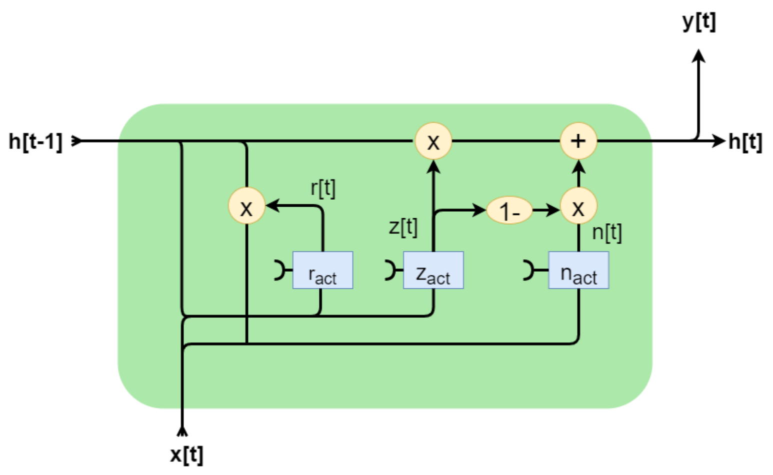


Figure 8: GRU Cell

Below is the class structure to guide your implementation of the GRU Cell.

```
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
```

The `GRUCell` class consists of the following methods:

- `forward(x, h_prev_t):` Computes the hidden state `h_t` at the current time step using the input `x` and previous hidden state `h_prev_t`. Stores intermediate variables `x`, `hidden`, `r`, `z`, and `n` needed for the backward pass.
  (To be implemented; returns `h_t`)
- `backward(delta):` Computes gradients of the loss with respect to the input and previous hidden state, as well as updates gradients of all weights and biases. Calculates `dr`, `dz`, `dn`, `dx`, and `dh_prev_t`.
  (To be implemented; returns `dx` and `dh_prev_t`)

Note: Unlike the `RNNCell`, the GRU implementation in this homework uses a different approach for managing and storing gradients. This is a deliberate design choice to expose you to a variety of architectural and programming patterns in neural networks.

5.1 GRU Cell Forward (5 points)

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)

Table 4: GRUCell Forward Components

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:

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

Derive the appropriate shape of r_t, z_t, n_t, h_t using the above equations. 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.

The computations given in the above equations are illustrated in the following figures:

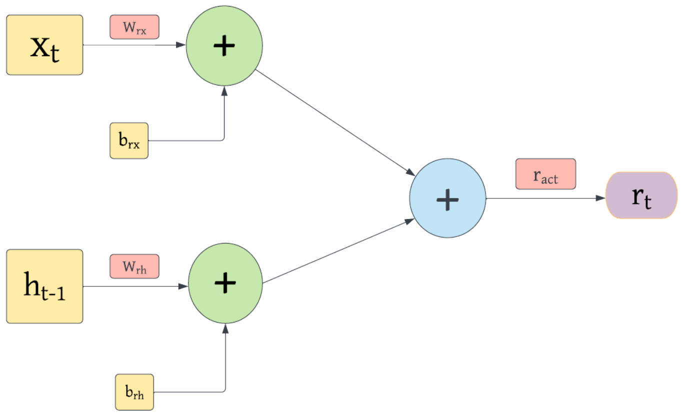


Figure 9: The computation for r_t

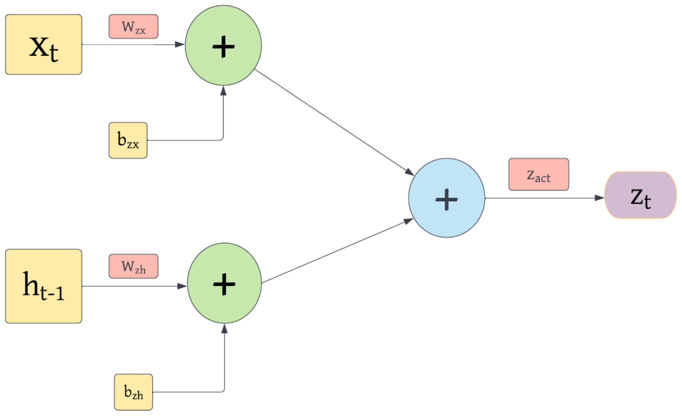


Figure 10: The computation for z_t

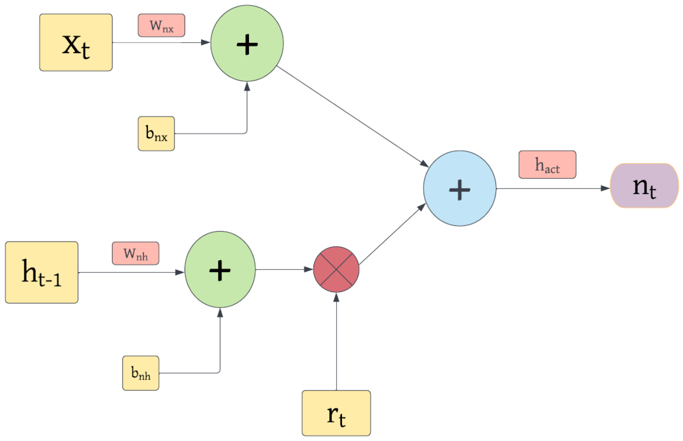


Figure 11: The computation for n_t

5.2 GRU Cell Backward (15 points)

In `mytorch/gru.py`, you are required to implement the `backward` method for the `GRUCell` class. This component is one of the more involved parts of the homework, as it requires computing gradients with respect 14 parameters in total. While this task may appear time-consuming, it is conceptually straightforward if approached methodically.

The `backward` method receives as input a tensor `delta`, which represents the upstream gradient flowing into this GRU cell. More precisely, `delta` is the **sum of two components**:

- the derivative of the loss with respect to the input of the next layer at the same time step, $\frac{\partial \mathcal{L}}{\partial x_{l+1,t}}$,
- the derivative of the loss with respect to the hidden state at the next time step, $\frac{\partial \mathcal{L}}{\partial h_{l,t+1}}$.

Using this combined gradient, your task is to compute the partial derivatives of the loss with respect to:

- the six parameter matrices of the GRU: W_z, U_z, W_r, U_r, W_h, U_h,
- the corresponding six bias vectors: b_z, b_r, b_h,
- the input at the current time step: x_t,

- the hidden state from the previous time step: h_{t-1}.

Your implementation should return the gradients with respect to x_t and h_{t-1}, which are required for propagating the gradients backward in time and across layers. Make sure to follow the computational graph defined in the forward pass and apply the chain rule carefully.

Hint: A clean and modular implementation will help you debug and validate each gradient computation efficiently.

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}

Table 5: GRUCell Backward Components

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 might seem inundating at first but apart from `dx` and `dh_prev_t`, all gradients can b computed in 2-3 lines of code! For your reference, the forward equations 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) $\dfrac{\partial L}{\partial b_{nh}} = \dfrac{\partial L}{\partial n_t} \times \dfrac{\partial n_t}{\partial b_{nh}}$

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) $\dfrac{\partial L}{\partial W_{zx}} = \dfrac{\partial L}{\partial z_t} \times \dfrac{\partial z_t}{\partial W_{zx}}$

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

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

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

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) $\dfrac{\partial L}{\partial W_{rx}} = \dfrac{\partial L}{\partial r_t} \times \dfrac{\partial r_t}{\partial W_{rx}}$

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

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

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

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

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

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

5.3 GRU Inference (10 points)

In `models/char_predictor.py`, you will implement a simple neural network using your `GRUCell` from the previous section and a linear layer to produce per-time-step logits for character prediction.

Unlike the RNN Phoneme Classifier:

- You are not training the model: only running inference.
- The network consists of a **single GRU layer**, unrolled over the input sequence.

You will complete the following components:

1. `CharacterPredictor` class

- In the `__init__` method, initialize:
  - a `GRUCell` with the given `input_dim` and `hidden_dim`.
  - a Linear layer `self.projection` that maps from `hidden_dim` to `num_classes`.
- Implement the `forward` method that:
  - Takes in the current input and hidden state.
  - Returns the updated hidden state and the output logits.

2. `inference(net, inputs)` function

- **Inputs**:
  - `net`: an instance of `CharacterPredictor`.
  - `inputs`: a tensor of shape `(seq_len, feature_dim)` representing the input sequence.
- **Output**:
  - `logits`: a tensor of shape `(seq_len, num_classes)`, containing one set of logits per time step.
- In this function, unroll the GRU over the sequence:
  - Initialize the hidden state to zeros.
  - For each time step, call `net.forward()` and collect the logits.
  - Stack the logits across time and return.

Note: The complete forward unroll in `inference()` can be implemented in under 10 lines of code, and the `CharacterPredictor.forward()` method should be just 2–3 lines.

6 CTC (Connectionist Temporal Classification)

CTC is a loss function and scoring method used for training RNNs (e.g., LSTMs, GRUs) on sequence tasks where input-output alignment is unknown or variable in time, like speech or handwriting recognition. Reference recitation 0.19.

Task Overview

You'll implement CTC loss in `CTC/CTC.py` using the **Forward-Backward algorithm** (this is covered in lectures).
You are given:

- The output of an RNN/GRU: a sequence of probability distributions over symbols (including the `blank` symbol), one per timestep.
- A ground-truth label sequence (e.g., phonemes).

Your goal is to:

- Compute the CTC loss by summing over all valid alignments.
- Implement the forward and backward passes to compute the posterior probability $\gamma(t, r) = Pr(s_t = S_r \mid S, X)$: the probability that the symbol at time t aligns to the r-th position in the modified label sequence.

Reminder: Make sure to include the `blank` symbol when constructing the extended target sequence, and account for repeated symbols correctly during alignment.

```
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
```

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.

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

Table 6: CTC Components

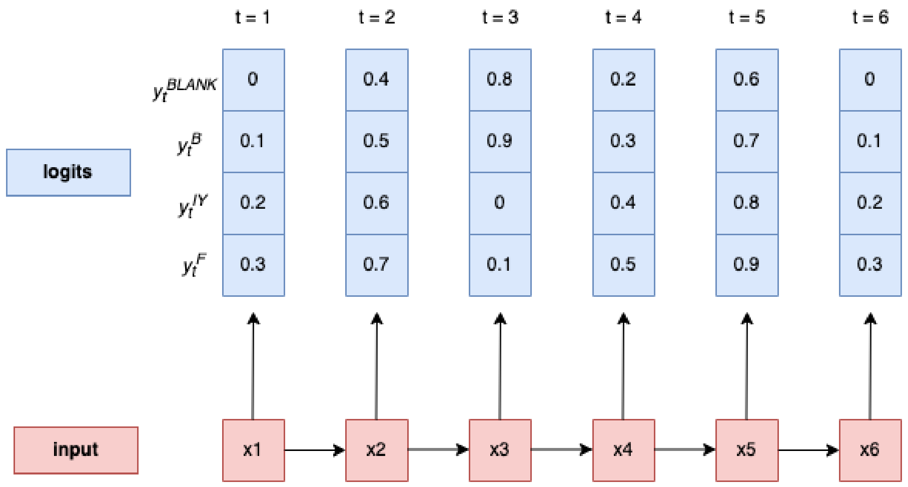


Figure 12: An overall CTC setup example

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`.

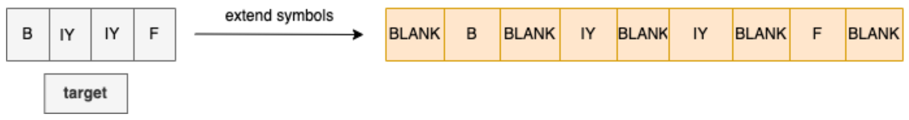


Figure 13: Extend symbols

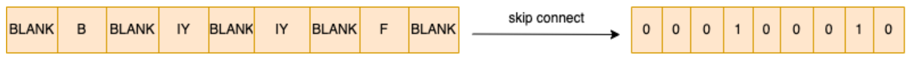
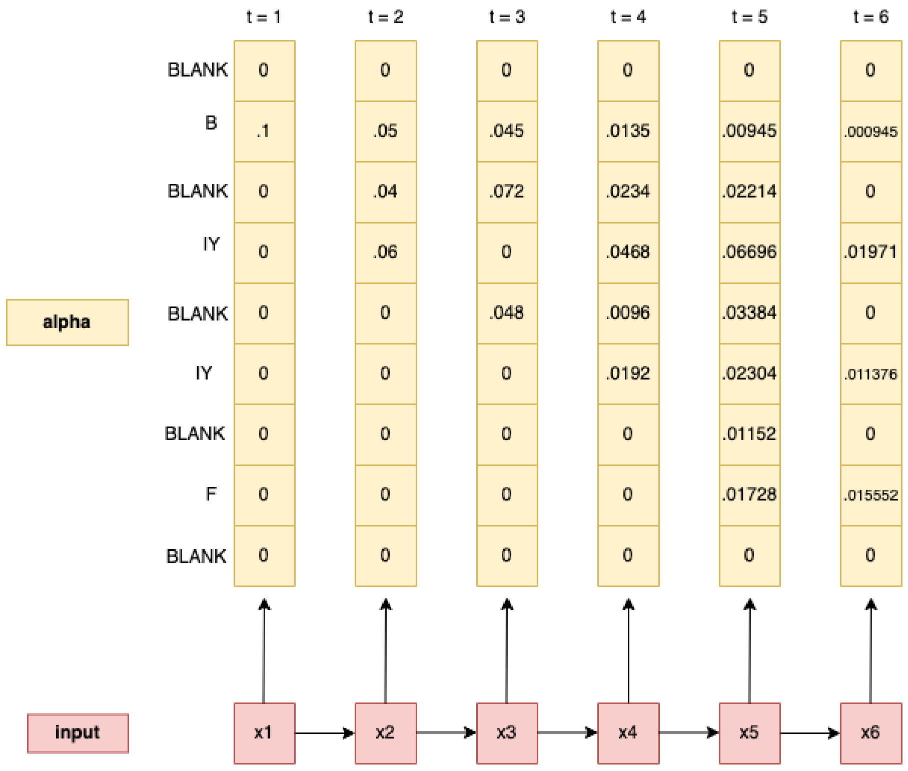


Figure 14: Skip connections

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.



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

Figure 15: Forward Algorithm

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`

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`

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

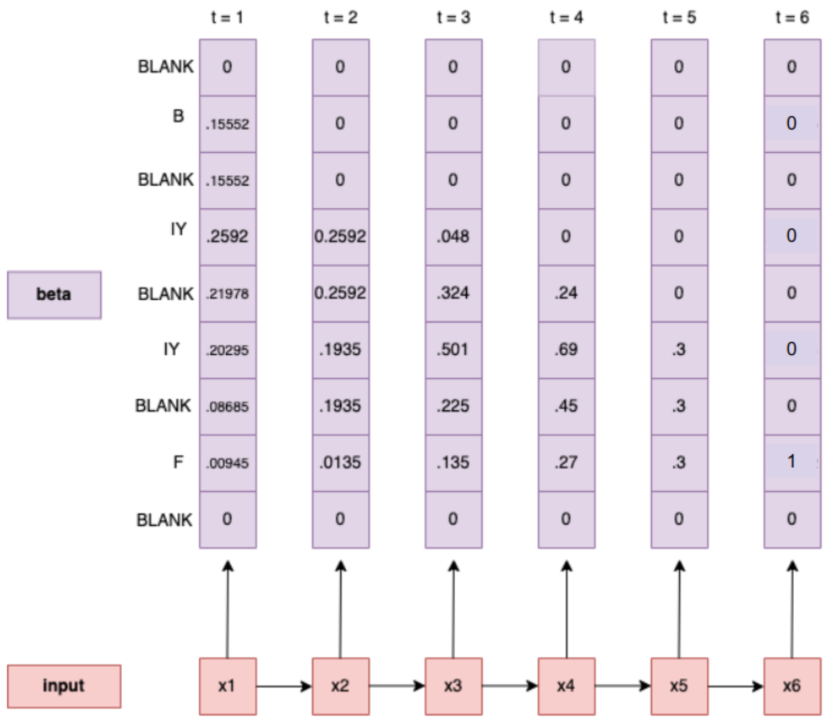
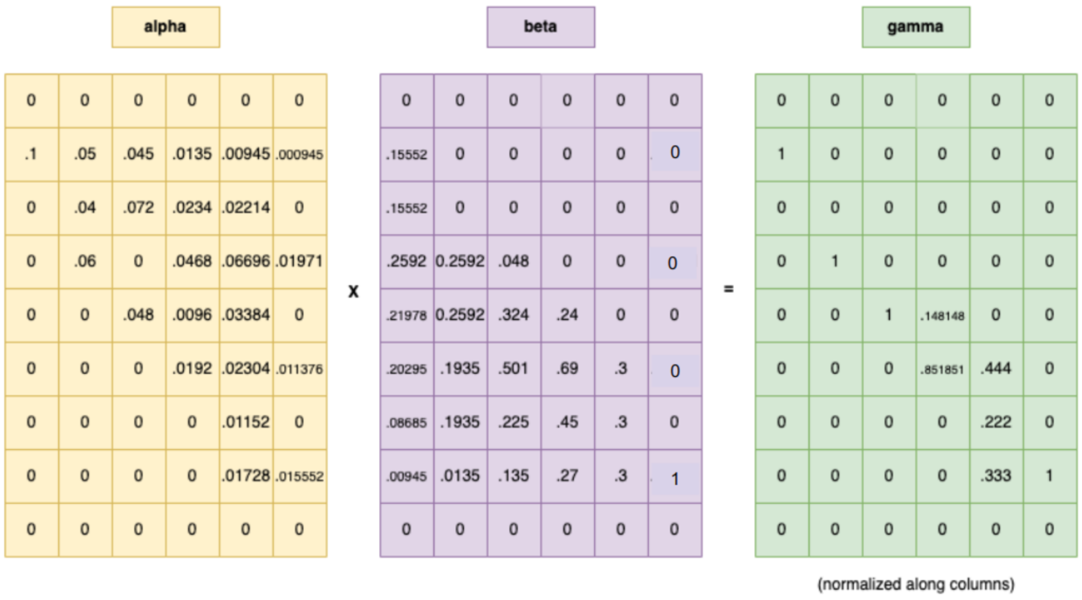

t = 1
t = 2
t = 3
t = 4
t = 5
t = 6
BLANK 0 0 0 0 0 0
B .15552 0 0 0 0 0
BLANK .15552 0 0 0 0 0
IY .2592 0.2592 .048 0 0 0
beta
BLANK .21978 0.2592 .324 .24 0 0
IY .20295 .1935 .501 .69 .3 0
BLANK .08685 .1935 .225 .45 .3 0
F .00945 .0135 .135 .27 .3 1
BLANK 0 0 0 0 0 0
input
x1
x2
x3
x4
x5
x6

Figure 16: Backward Algorithm


alpha
x
beta
=
gamma
(normalized along columns)

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

0	0	0	0	0	0
.15552	0	0	0	0	0
.15552	0	0	0	0	0
.2592	0.2592	.048	0	0	0
.21978	0.2592	.324	.24	0	0
.20295	.1935	.501	.69	.3	0
.08685	.1935	.225	.45	.3	0
.00945	.0135	.135	.27	.3	1
0	0	0	0	0	0

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

Figure 17: Posterior Probability

6.1 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
```

6.1.1 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 averaged over the batch. Within the loop, follow the steps:

1. set up a CTC

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.

Table 7: CTC Loss Components

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`.

6.1.2 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`

7 CTC Decoding: Greedy Search and Beam Search

After training a sequence model, the next critical step is to *decode* the model's output; that is, to convert the predicted probability distributions over the output vocabulary into a meaningful sequence of tokens. Although this may initially seem like a new task, you have already implemented a basic decoding strategy in earlier assignments (HW1P2 and HW2P2), where the class with the highest predicted probability was selected via an `argmax` over the final linear layer. This form of greedy decoding, while simple, serves as a foundation for more advanced sequence decoding methods.

In the context of sequence models, especially those used for tasks such as speech recognition or machine translation, more sophisticated decoding algorithms (such as Beam Search) are often employed to balance accuracy and computational tractability, taking into account the sequential structure and dependencies across time steps.

- `CTC/CTCDecoding.py` contains two classes: `GreedySearchDecoder` and `BeamSearchDecoder`, both of which implement a `decode` method for decoding CTC outputs.
- The `decode` method in both classes takes the following inputs:
  - `symbol_set` is a list of all symbols in the vocabulary, **excluding the blank symbol**.
  - `y_probs` is an array of shape `(len(SymbolSets) + 1, seq_length, batch_size)` representing the probability distribution over all symbols **including the blank symbol** at each time step.
    - Index 0 in `y_probs` always corresponds to the blank symbol.
    - The batch size is 1 for all test cases, but if you plan to use your implementation for HW3P2 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:

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

7.1 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.

7.1.1 Example

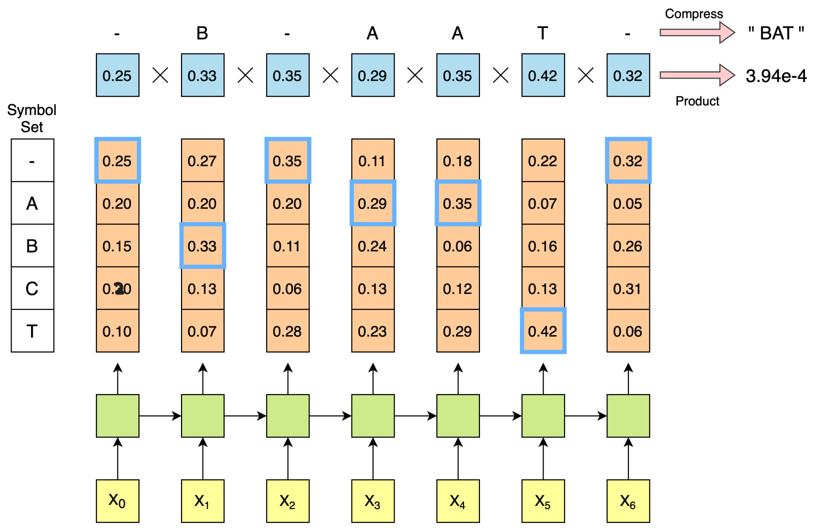


Figure 18: Greedy Search

Consider the example in Figure 18. The output is given for 7 time steps. Each probability distribution has 5 elements (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.

7.1.2 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.

7.2 Beam Search (15 points)

Greedy Search is a simple decoding strategy that selects the most probable token at each time step. However, it can lead to suboptimal results because early decisions are fixed and may prevent the formation of a more coherent sequence later.

Beam Search addresses this by maintaining the **top-k** most probable sequences (where k is the beam width) at each time step, rather than just one. At every step, it expands each sequence in the beam with all possible next tokens and retains only the best k candidates based on the total probability. This allows the decoder to explore multiple promising paths in parallel.

In CTC-based models (e.g., speech recognition), Beam Search is especially useful. It handles blank tokens and repeated characters by collapsing paths that yield the same final output and summing their probabilities. This ensures that the model considers all equivalent interpretations of the same sequence.

Beam Search is more computationally intensive than Greedy Search, but it offers a much better trade-off between performance and efficiency compared to exhaustive search.

You can also read more about this here.

7.2.1 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

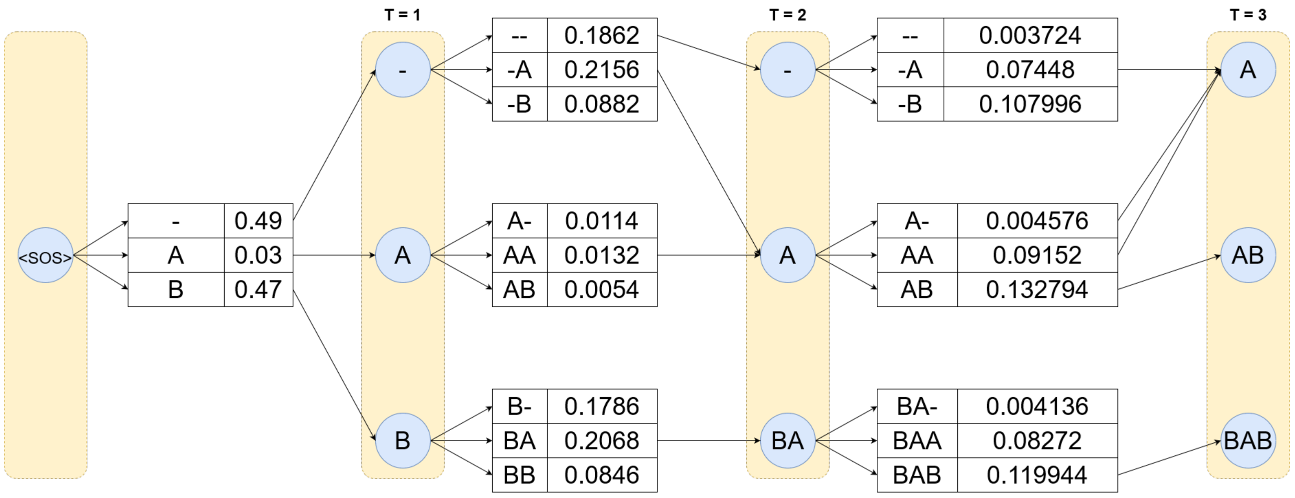


Figure 19: Beam Search over a vocabulary / symbols set = {-, A, B} with beam width (k) = 3. (" - " == BLANK). The blue shaded nodes indicate the compressed decoded sequence at given time step, and the expansion tables show the probability (right column) and symbols (left column) at each time step pre-pended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

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 (illustrated in 19 at time step = 3) will be:

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

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.

7.2.2 Pseudo-code

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

```
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
```

The `decode` function in your `BeamSearchDecoder` class will accept the following arguments:

- `symbol_set`: 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.

`beam_width` limits the number of partial sequences (or 'paths') that you keep track of at each timestep.

How to Approach this?

- Initialize the beam search with one path consisting of a 'blank' symbol. At each timestep, iterate over your current best paths.
- Extend 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.
- Return the highest scoring path and the scores of all final paths.

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). We recommend following the below pseudocode.

Algorithm 2: Beam Search Decoding Algorithm

```
Input: $SymbolSet, y\_probs, BeamWidth$
Output: $BestPath, MergedPathScores$
1  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
2  foreach timestep in y_probs do
3      Extract the symbol probabilities at the current timestep;
4      Sort ActivePaths by score in descending order and keep only the top $BeamWidth$ paths;
5      foreach $(path, score)$ in ActivePaths do
6          foreach symbol in $SymbolSet$ including blank do
7              Compute the new path based on the last symbol of path;
8              Calculate the updated score: $new\_score = score \times probability(symbol)$;
9              if new_path exists in TempPaths then
10                 Add new_score to the existing score of new_path in TempPaths;
11             end
12             else
13                 Store new_score in TempPaths with new_path;
14             end
15         end
16     end
17     Replace ActivePaths with TempPaths;
18     Clear TempPaths for the next timestep;
19 end
20 Final Merging of Paths: Initialize BestPath as an empty string and BestScore as 0;
21 foreach $(path, score)$ in ActivePaths do
22     Remove empty spaces at the beginning and end of the path;
23     if path in MergedPaths then
24         Add score to path in $MergedPathScores$;
25     end
26     else
27         Store score in new path in $MergedPathScores$;
28     end
29     if score is greater than $BestScore$ then
30         BestPath ← path;
31         BestScore ← score;
32     end
33 end
34 Convert Best Path to Symbols: Convert $BestPath$ to their corresponding symbols in $SymbolSet$;
35   Convert all path in $MergedPathScores$ to their corresponding symbols in $SymbolSet$
36 return $BestPath, MergedPathScores$;
```

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

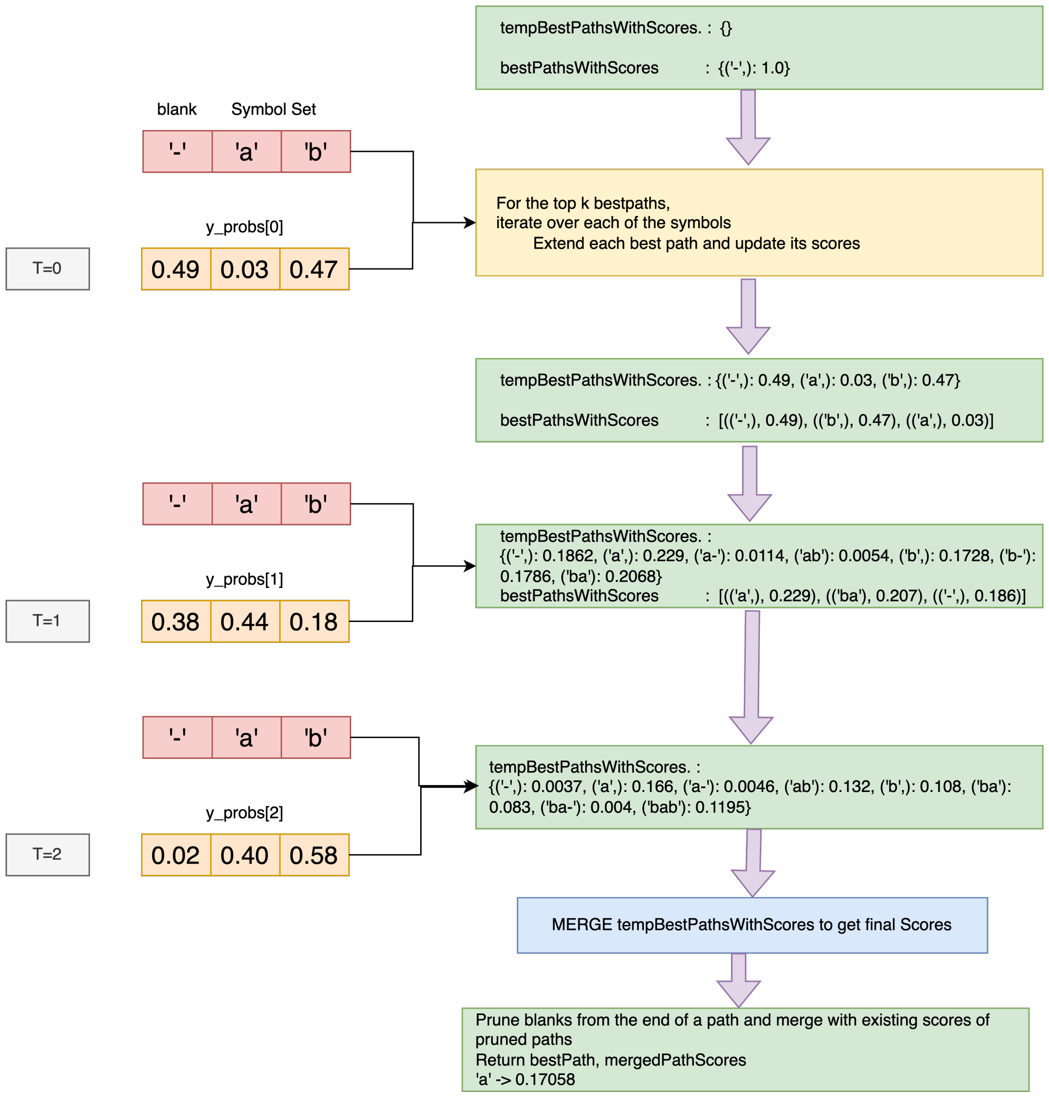

tempBestPathsWithScores. : {}
bestPathsWithScores : {('-',): 1.0}
blank
Symbol Set
'-'
'a'
'b'
y_probs[0]
T=0
0.49
0.03
0.47
For the top k bestpaths,
iterate over each of the symbols
Extend each best path and update its scores
tempBestPathsWithScores. : {('-',): 0.49, ('a',): 0.03, ('b',): 0.47}
bestPathsWithScores : [(('-',), 0.49), (('b',), 0.47), (('a',), 0.03)]
'-'
'a'
'b'
y_probs[1]
T=1
0.38
0.44
0.18
tempBestPathsWithScores. :
{('-',): 0.1862, ('a',): 0.229, ('a-'): 0.0114, ('ab'): 0.0054, ('b',): 0.1728, ('b-'): 0.1786, ('ba'): 0.2068}
bestPathsWithScores : [(('a',), 0.229), (('ba'), 0.207), (('-',), 0.186)]
'-'
'a'
'b'
y_probs[2]
T=2
0.02
0.40
0.58
tempBestPathsWithScores. :
{('-',): 0.0037, ('a',): 0.166, ('a-'): 0.0046, ('ab'): 0.132, ('b',): 0.108, ('ba'): 0.083, ('ba-'): 0.004, ('bab'): 0.1195}
MERGE tempBestPathsWithScores to get final Scores
Prune blanks from the end of a path and merge with existing scores of pruned paths
Return bestPath, mergedPathScores
'a' -> 0.17058

Figure 20: Efficient Beam Search procedure

8 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
```

8.1 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]]
```

8.2 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]
```

8.3 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 ***
--------------------
```

8.4 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 7.2.1.