

Homework 3 Part 1

RNNs and GRUs and Search, Oh My!

11-785: INTRODUCTION TO DEEP LEARNING (SPRING 2020)

OUT: **March 8, 2020**

DUE: **April 4, 2020, 11:59 PM**

Start Here

- **Collaboration policy:**

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

- **Overview:**

- **MyTorch**
- **Multiple Choice**
- **RNN**
- **GRU**
- **Greedy Search and Beam Search**

- **Directions:**

- You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.
- We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.
- If you haven't done so, use pdb to debug your code effectively.

1 MyTorch

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

- mytorch
 - rnn_cell.py
 - gru_cell.py
 - search.py
 - linear.py
 - activation.py
 - loss.py
- hw3
 - hw3.py
 - rnn_classifier.py
 - mc.py
- autograder
 - hw3_autograder
 - * runner.py
- create_tarball.sh

-
- **Install** Python3, NumPy and PyTorch in order to run the local autograder on your machine:

```
pip3 install numpy
pip3 install torch
```

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

```
sh create_tarball.sh
```

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

```
python3 autograder/hw3_autograder/runner.py
```

- **DO NOT:**

- Import any other external libraries other than numpy, as extra packages that do not exist in autolab will cause submission failures. Also do not add, move, or remove any files or change any file names.

2 Multiple Choice [5 points]

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

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

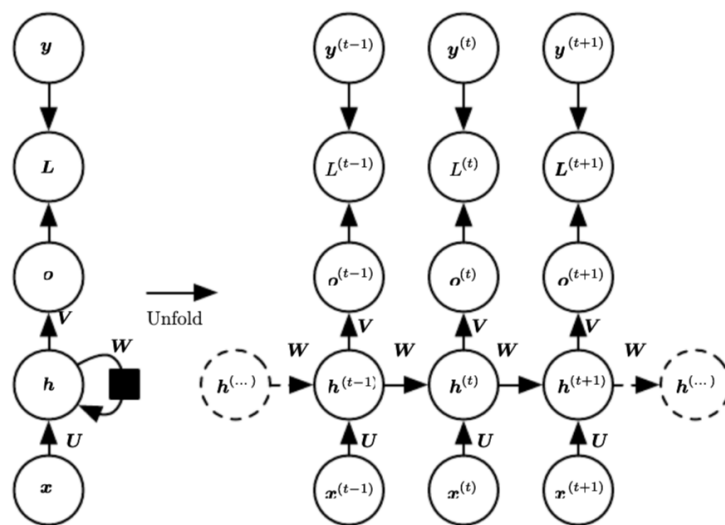


Figure 1: The high-level computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output values from <http://www.deeplearningbook.org/contents/rnn.html>. (Please note that this is just a general RNN, being shown as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

- (A) I have decided to forgo the reading of the aforementioned chapter on RNNs and have instead dedicated myself to rescuing wildlife in our polluted oceans.
- (B) I have completed the optional reading of <http://www.deeplearningbook.org/contents/rnn.html> (Note the RNN they derive is different from the GRU later in the homework.)
- (C) Gravitational waves ate my homework.
- (2) **Question 2: In an RNN with N layers, how many unique RNN Cells are there? [1 point]**
- (A) 1, only one unique cell is used for the entire RNN
- (B) N , 1 unique cell is used for each layer
- (C) 3, 1 unique cell is used for the input, 1 unique cell is used for the transition between input and hidden, and 1 unique cell is used for any other transition between hidden and hidden

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

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

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

- (A) [3,0,3,0,3,1,1,0,2,0]
- (B) [3,0,3,1,3,0,1,0,2,0]
- (C) [3,0,3,1,3,0,0,2,0,1]
- (4) **Question 4:** I have watched the lectures for Beam Search and Greedy Search? Also, I understand that I need to complete each question for this homework in the order they are presented or else the local autograder won't work. Also, I understand that the local autograder and the autolab autograder are different and may test different things- passing the local autograder doesn't automatically mean I will pass autolab. [1 point]
- (A) I understand.
- (B) I do not understand.
- (C) Potato

3 RNN Cell

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

3.1 RNN Cell Forward (5 points)

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

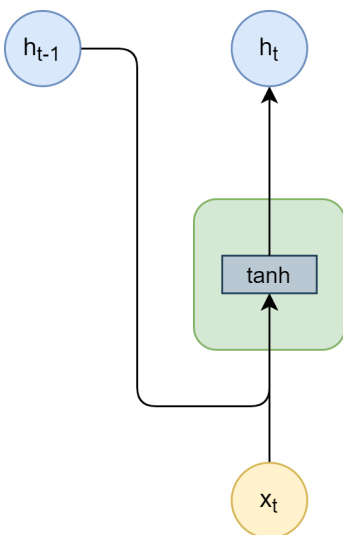


Figure 2: The computation flow for the RNN Cell forward.

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

The equation you should follow is given in equation 1.

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

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

Inputs

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

Outputs

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

3.2 RNN Cell Backward (5 points)

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

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

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

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

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

Inputs

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

Outputs

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

How to start? We recommend drawing a computational graph.

3.3 RNN Phoneme Classifier (15 points)

In `hw3/rnn_classifier.py` implement the forward and backward methods for the `RNN_Phoneme_Classifier`. Read over the `init` method and uncomment the `self.rnn` and `self.output_layer` after understanding their initialization.

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

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

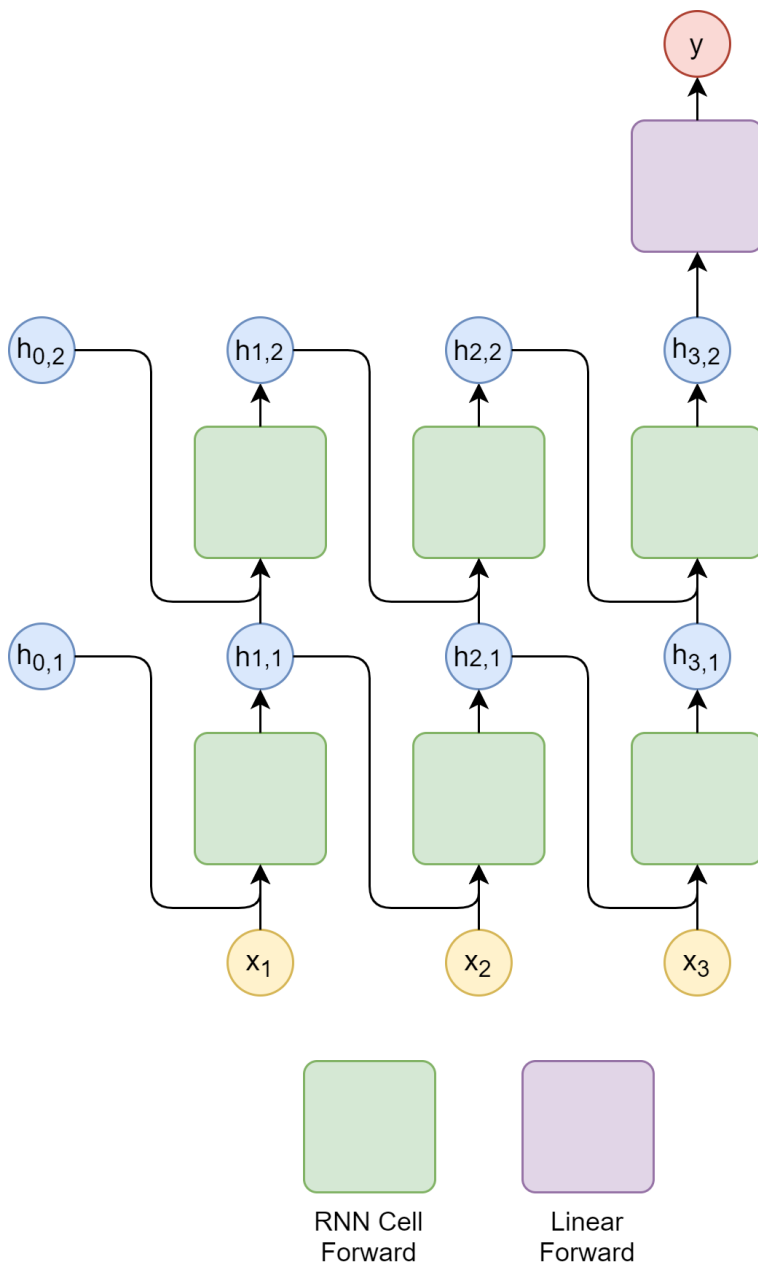


Figure 3: The forward computation flow for the RNN.

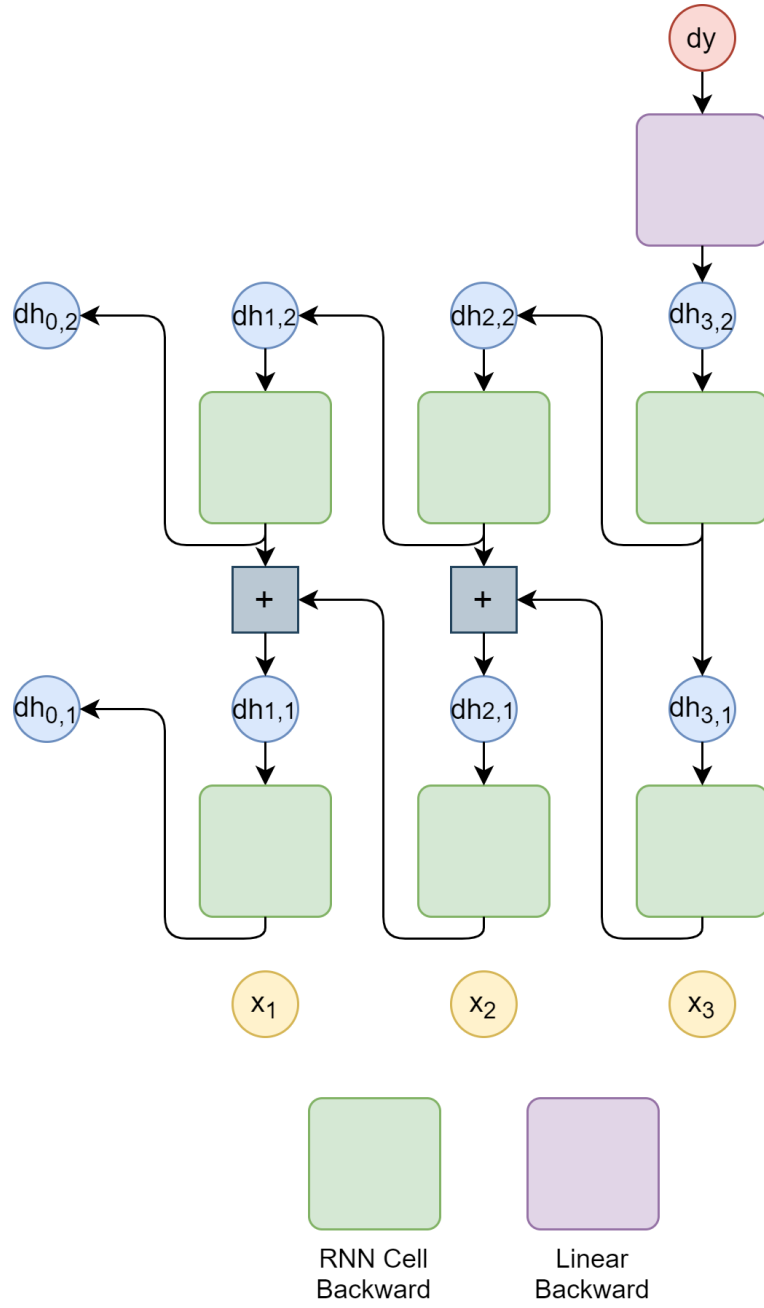


Figure 4: The backward computation flow for the RNN.

4 GRU Cell

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

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

4.1 GRU Cell Forward (10 points)

In `mytorch/gru.py` implement the forward pass for a GRUCell using Numpy, analogous to the Pytorch equivalent `nn.GRUCell`. (Though we follow a slightly different naming convention than the Pytorch documentation.) The equations for a GRU cell are the following:

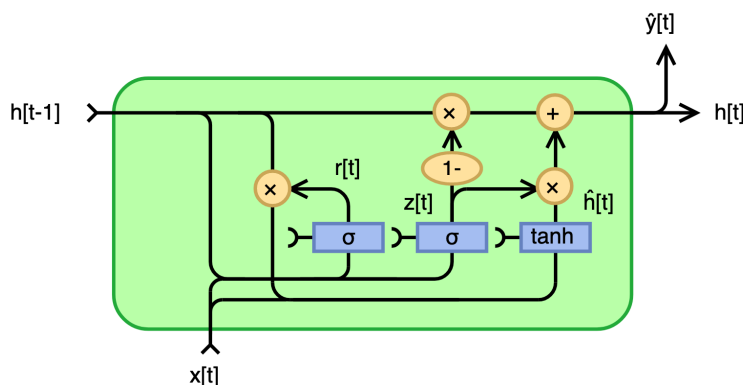


Figure 5: The computation for our GRU

$$\mathbf{z}_t = \sigma(\mathbf{W}_{zh}\mathbf{h}_{t-1} + \mathbf{W}_{zx}\mathbf{x}_t) \quad (2)$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{W}_{rx}\mathbf{x}_t) \quad (3)$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h(\mathbf{r}_t \otimes \mathbf{h}_{t-1}) + \mathbf{W}_x\mathbf{x}_t) \quad (4)$$

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

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

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

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

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

4.2 GRU Cell Backward (20 points)

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

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

The partial derivative input you are given, *delta*, is the summation of: the derivative of the loss w.r.t the input of the next layer $x_{l+1,t}$ and the derivative of the loss w.r.t the input hidden-state at the next time-step $h_{l,t+1}$.

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

Specifically, there are eight gradients that need to be computed:

1. $\frac{\partial L}{\partial W_{rx}}$ (`self.dWrx`)
2. $\frac{\partial L}{\partial W_{rh}}$ (`self.dWrh`)
3. $\frac{\partial L}{\partial W_{zx}}$ (`self.dWzx`)
4. $\frac{\partial L}{\partial W_{zh}}$ (`self.dWzh`)
5. $\frac{\partial L}{\partial W_x}$ (`self.dWx`)
6. $\frac{\partial L}{\partial W_h}$ (`self.dWh`)
7. $\frac{\partial L}{\partial x_t}$ (returned by method)
8. $\frac{\partial L}{\partial h_t}$ (returned by method)

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

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

How to start? You will need to derive the formulae for the back-propagation in order to complete this section of the assignment. We recommend creating your own computation graph, and refreshing yourself on the rules for gradients from an earlier recitation.

4.3 GRU Inference (10 points)

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

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

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

- The forward method of `CharacterPredictor` should return both the logits and the next hidden state.
- Any code given in the `init` method that is commented, should be uncommented and then used. Changing names of variables will result in lost points.

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

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

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

5 Greedy Search and Beam Search

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

5.1 Greedy Search (10 points)

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

5.2 Beam Search (20 points)

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