192.151 Introduction to Deep Learning

2025W

Problem Sheet 7

Problem 7.1

Through this assignment, you will progressively build 'MyTorch,' your own deep learning library. Conceptually, it will be similar to existing frameworks such as PyTorch or TensorFlow, and the files should be well organized to enable importing and reuse of code components. MyTorch will adopt the specific structure detailed as follows:

- rnn_cell.py: implement an Elman RNN cell,
- rnn_classifier.py: implement an RNN as described for the RNN Phoneme Classifier.

This assignment must be completed using the Python programming language, specifically version 3. The use of automatic differentiation frameworks such as PyTorch, TensorFlow, or Keras is prohibited. For all computations, you are permitted and indeed encouraged to leverage the Numpy library to implement vectorized solutions.

Regarding your approach to the problems, we recommend an initial review of all questions before beginning work on the first. However, it is then strongly recommended that you complete the problems in the presented order. This sequential approach is beneficial, as the problems are designed with progressively increasing difficulty, and later questions frequently build on the solutions or concepts developed in earlier ones.

A RNN Cell

The RNN Cell is the fundamental unit within a Recurrent Neural Network (RNN). Designed for time-dependent or sequence-based tasks, the 'recurrent' nature of RNNs stems from an internal memory used to predict future states. An RNN architecture typically spans multiple time steps and layers, as depicted in Figure 1. In these models, hidden states from earlier time steps influence subsequent ones, while outputs from prior layers are utilized by later layers. However, this section and the next will focus specifically on the single RNN cell highlighted by the red box. Besides, the RNNCell class components are shown in Table 1

In **mytorch/rnn_cell.py**, you need to write an Elman RNN cell.

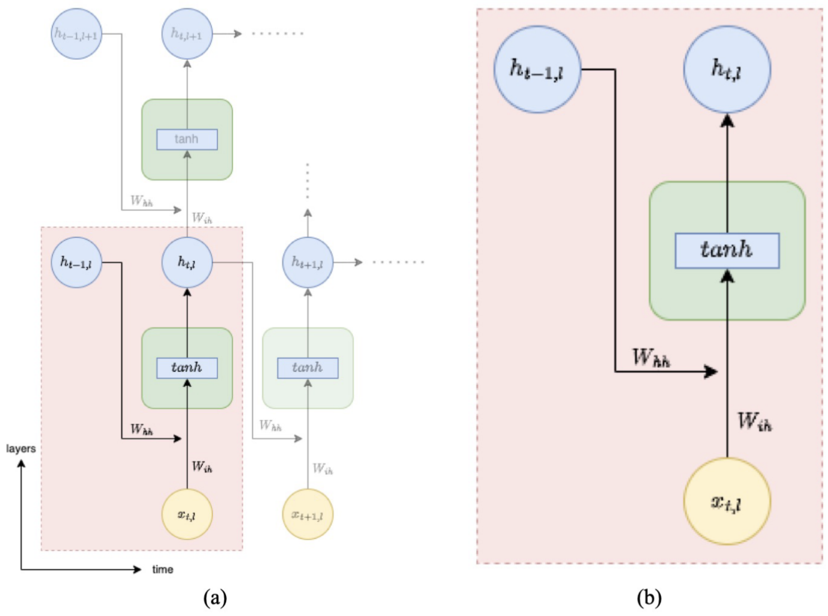


Figure 1: The red box shows one single RNN cell. RNNs can have multiple layers across multiple time steps. This is indicated by the two-axis in the bottom-left.

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

```
# Define RNNCell class
class RNNCell:
  def __init__(self, input_size, hidden_size):
           <Weight definitions>
           <Gradient definitions>

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

       def zero_grad(self):
           <zeroing gradients>

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

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

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

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

The RNNCell class has initialization, forward, and backward attribute functions. The **__init__** code is executed when the class is instantiated.

The forward method calculates **h_t** and is defined as follows:

- **Input**: x and h_prev_t.
- **Stored Attributes**: none during this operation.
- **Output**: h_t.

The backward method computes gradients for optimization:

- **Input**: delta, h_t, h_prev_l, and h_prev_t.
- **Stored Attributes**: dW_ih, dW_hh, db_hh, and db_ih.
- **Output**: dx and dh_prev_t.

Table 1: RNNCell Class Components

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

A.1 RNN Cell Forward

The fundamental process for calculating a cell's forward output is analogous to what you have learned with other neural networks: weights and biases are applied in an affine transformation, and the result is then passed through an activation function.

Each input has an associated weight and bias. The first step in the computation is to apply an affine transformation to these inputs. The specific calculation is as follows:

Affine function for inputs:

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

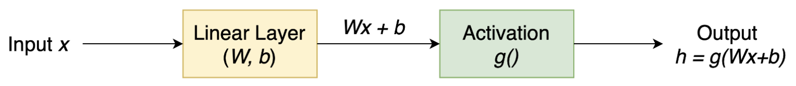
Affine function for previous hidden state:

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

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

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

Note that these equations address a single element; handling batch dimensions in your code will likely require transpositions. The PyTorch nn.RNNCell documentation also shows the forward pass for an Elman RNN cell with tanh activation, which can serve as a reference. Ensure that you use the activation attribute and all pre-defined weights and biases from the __init__ method.



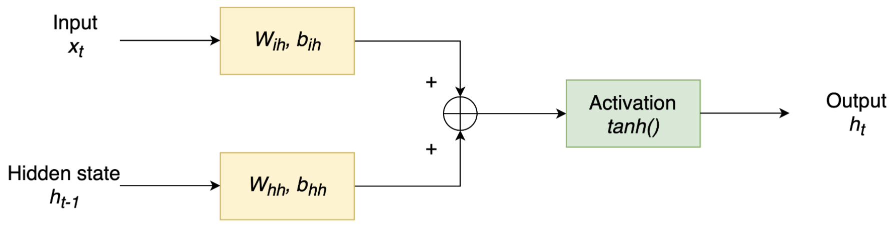


Figure 2: Comparison of a perceptron with a single-layer, single-time-step RNN cell.

A.2 RNN Cell Backward

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

a) $\frac{\partial L}{\partial W_{ih}}$ (self.dW_ih),

b) $\frac{\partial L}{\partial W_{hh}}$ (self.dW_hh),

c) $\frac{\partial L}{\partial b_{ih}}$ (self.db_ih),

d) $\frac{\partial L}{\partial b_{hh}}$ (self.db_hh),

e) $\frac{\partial L}{\partial x}$ (dx) (returned by the method, explained below),

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

In our **RNNCell** implementation, newly calculated gradients must be accumulated by adding them to any existing gradients for the cell's parameters. This accumulation is necessary, because a single RNNCell instance, with its set of weights, is applied repeatedly across different time steps within an RNN layer. This iterative application of the same cell is visualized in Figure 3.

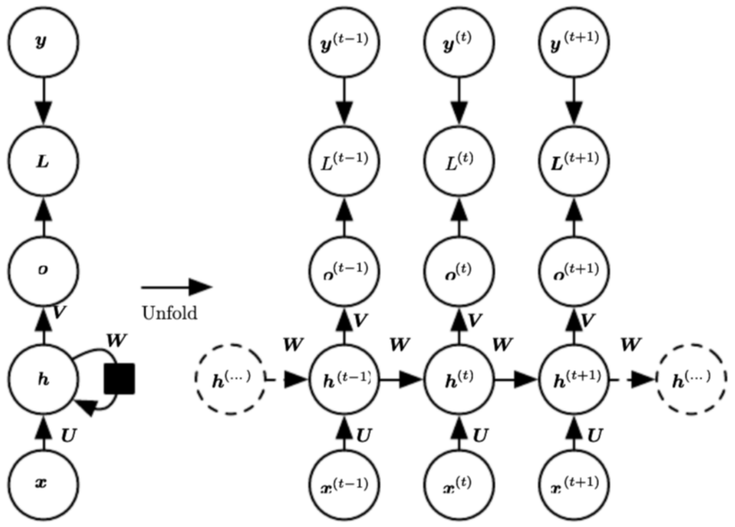


Figure 3: The high-level computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output values. Note that this is just a general RNN, being shown as an example of loop unrolling.

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.

A.3 RNN Phoneme Classifier

You need to implement an RNN (forward and backward methods) as described in the images below for the RNN Phoneme Classifier. Below is the reference:

```
class RNNPhonemeClassifier(object):
    """RNN Phoneme Classifier class."""

    def __init__(self, input_size, hidden_size, output_size, num_layers=2):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # TODO
        self.hiddens = []

    def init_weights(self, rnn_weights, linear_weights):
        for i, rnn_cell in enumerate(self.rnn):
            rnn_cell.init_weights(*rnn_weights[i])
        self.output_layer.W = linear_weights[0]
        self.output_layer.b = linear_weights[1].reshape(-1, 1)

    def __call__(self, x, h_0=None):
        return self.forward(x, h_0)

    def forward(self, x, h_0=None):
        batch_size, seq_len = x.shape[0], x.shape[1]
        if h_0 is None:
            hidden = np.zeros((self.num_layers, batch_size, self.hidden_size), dtype=float)
        else:
            hidden = h_0

        self.x = x
        self.hiddens.append(hidden.copy())
        logits = None

        # TODO

        raise NotImplementedError

    def backward(self, delta):
        batch_size, seq_len = self.x.shape[0], self.x.shape[1]
        dh = np.zeros((self.num_layers, batch_size, self.hidden_size), dtype=float)
        dh[-1] = self.output_layer.backward(delta)

        # TODO

        # return dh / batch_size
        raise NotImplementedError
```

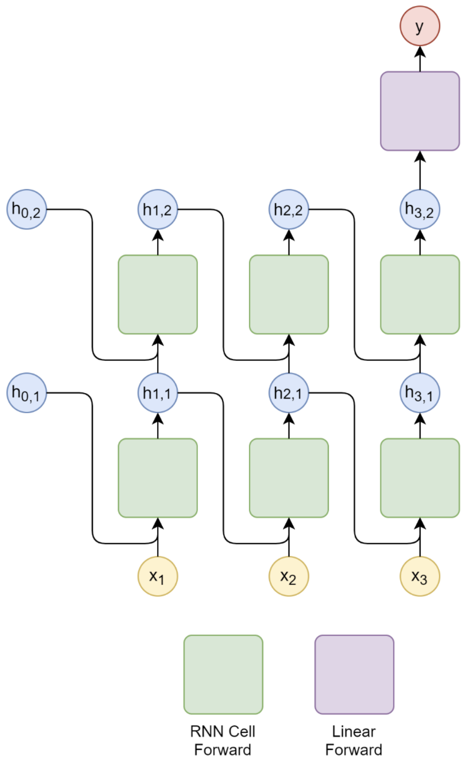
Below are visualizations of the forward and backward computation flows. Your RNN Classifier is expected to execute given with an arbitrary number of layers and time sequences.

Table 2: RNNPhonemeClassifier Class Components

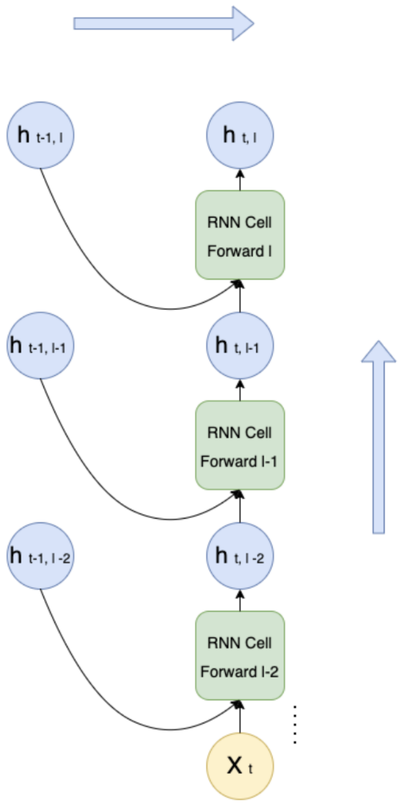
Code Name	Math	Type	Shape	Meaning
x	x	matrix	(batch_size, seq_len, input_size)	Input
h_0	h_0	matrix	(num_layers, batch_size, hidden_size)	Initial hidden states
delta	$\partial L/\partial h$	matrix	(batch_size, hidden_size)	Gradient w.r.t. last time step output

RNN Classifier Forward

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



(a) The forward computation flow for the RNN.



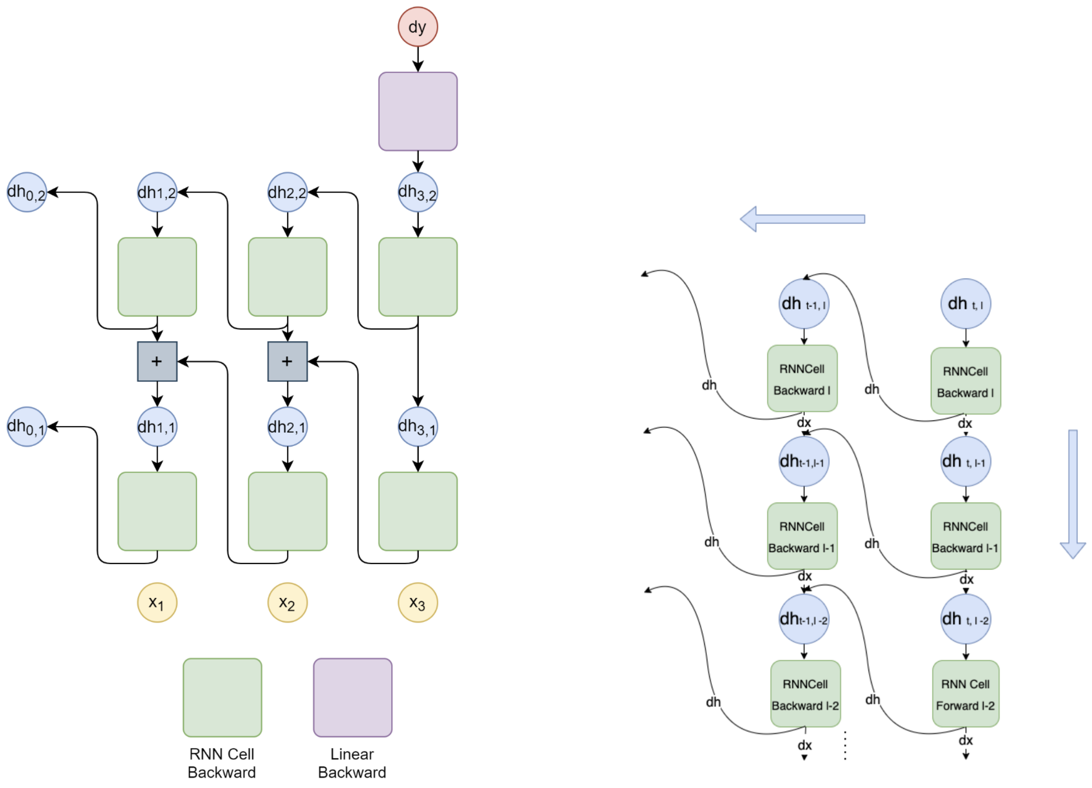
(b) The forward computation flow for RNN at time step t.

Figure 4: Illustration of RNN forward computation flows.

RNN Classifier Backward

To complete this question, first please understand the flow with the help of Figure 5, and then follow the pseudocode.

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



(a) The backward computation flow for the RNN. (b) The backward computation flow for RNN at time step t.

Figure 5: Illustration of RNN backward computation flows.