Assignment 4: *Recurrent Neural Networks*

Date: *Mar 21, 2025* Due : *April 4, 2024*

Preface

This is the last series of assignments for the course *ECE1508: Applied Deep Learning*. The exercises are aimed to review recurrent neural networks studied in Chapter 6. Below, you can find the information about contents of these exercises, as well as instructions on how to submit them.

General Information The assignments are given in two sections. In the first section, you have written questions that you can answer in words or by derivation. The questions are consistent with the material of Chapter 4 and you do not need any further resources to answer them. The second section includes the programming assignments. For these assignments, you are assumed to have some very basic knowledge of programming in Python. For those who are beginners, a quick introduction to the install and use of Python libraries has been given in a separate file. In the case that a question is unclear or there is any flaws, please contact over Piazza. Also, in case that any particular assumption is required to solve the problem, feel free to consider the assumption and state it in your solution. The total mark of the assignments is **100 points** plus **5 extra points** with total mark of written questions adds to **30 points** and the total mark of the programming assignments adds to **70 points** plus **5 points bonus** for optional items.

How to Submit A notebook has been provided with basic code that you can complete. The submission is carried out through the Crowdmark. Please submit the answer of each question **separately**, following the **steps below**. Please note that *failure to meet the formatting can lead to mark deduction*.

1. For Written Questions, you can submit handwritten or typed answers as a `.pdf`, `.jpg` or `.png` file.
2. For Programming Questions, the answer should **complete the Python Notebook** shared with this assignment. For *each question*, please print the corresponding part of the notebook as a `.pdf` file and upload it in the corresponding field on Crowdmark. Your uploaded `.pdf` should contain **all figures, diagrams and codes requested** in the question.
3. The completed Notebook, i.e., the `.ipynb` file, including all the codes and the outputs should also be submitted as the attachment to the last item on Crowdmark.

When submitting your notebook, please pay attention to the following points:

1. The file should be named `Lastname_Firstname_Assgn4.ipynb`
2. Please make sure to name the files with the name that is displayed on the Quercus account

The deadline for your submission is on **April 4, 2025 at 11:59 PM**.

- You can delay up to two days. After this extended deadline no submission is accepted.
- For each day of delay, 5% of the mark after grading is deducted.

Please submit your assignment **only through Crowdmark, and not by email.**

1 Written Exercises

Question 1 [15 Points] **(Realizing Sum with RNN)** We intend to compute the "exclusive OR" (XOR) of a sequence of T binary numbers. This means that we are given by

$$x[1], \ldots, x[T] \in \{0, 1\}$$

and wish to compute

$$y[T] = x[1] \oplus x[2] \oplus \ldots \oplus x[T].$$

We solved this problem with a deep feedforward neural network (FNN) in Assignment 1. There, we gave the complete sequence as the input, i.e.,

$$\mathbf{x} = \begin{bmatrix} x[1] \\ x[2] \\ \vdots \\ x[T] \end{bmatrix},$$

and returned the XOR via a deep FNN. In this assignment, we want to solve it with a shallow RNN. To this end, answer the following items

1. Re-write the shallow neural network (NN) we used in Chapter 1 to XOR two binary numbers. Call the binary inputs x and s, and the output y. We want to use this NN it as an RNN.
2. Initiate the NN at time $t = 1$ with a binary value $s[1]$, such that the output at time $t = 1$ equals the input at time $t = 1$, i.e., $y[1] = x[1]$.
3. For $t > 1$, specify the choice of $s[t]$, such that after T times using this RNN, we get

$$y[T] = x[1] \oplus x[2] \oplus \ldots \oplus x[T].$$

4. What do we call $s[t]$ and what components does it represent?
5. We read the output at $t = T$, i.e., $y[T]$. Nonetheless, we have an sequence of outputs $y[1], \ldots, y[T]$. Explain what does $y[t]$ represent at time $t \neq 1, T$?

Question 2 [15 Points] **(**$\tanh$ **Activation)** As we mentioned in the course, we often use $\tanh$ activation in RNNs to reduce the impact of vanishing gradient. In this question, we have a very basic analysis on the property of this activation. Before we start with the question, let us recall that

- The derivative of $\tanh$ is given as

$$\frac{\mathrm{d}}{\mathrm{d}x} \tanh x = 1 - \tanh^2 x.$$

- For a given function $f(\cdot)$, a linear approximation at $\varepsilon \ll 1$ is given by the first two terms of Maclaurin expansion as

$$f(\varepsilon) = f(0) + \dot{f}(0)\, \varepsilon.$$

Now, answer the following items

1. For both *sigmoid* and $\tanh$ activation functions, use the linear approximation given by the Maclaurin expansion and approximate the derivative at $\varepsilon \ll 1$ with a polynomial of ε.

2. Let $\dot{\sigma}_{\text{approx}}(\varepsilon)$ and $\dot{\tanh}_{\text{approx}}(\varepsilon)$ be the approximative term for *sigmoid* and $\tanh$ activation at ε, respectively. Find and interval for ε, in which the $\dot{\sigma}_{\text{approx}}(\varepsilon) \leq \dot{\tanh}_{\text{approx}}(\varepsilon)$.

3. Define the approximative ratio of the activations as

$$\rho_{\text{approx}}(\varepsilon) = \frac{\dot{\sigma}_{\text{approx}}(\varepsilon)}{\dot{\tanh}_{\text{approx}}(\varepsilon)}.$$

   Plot $\rho_{\text{approx}}(\varepsilon)^T$ for $\epsilon = 0.05$ against T for T ranging between 1 and 20. Also, plot the true ratio, i.e., $\rho(\varepsilon)^T$ for

$$\rho(\varepsilon) = \frac{\dot{\sigma}(\varepsilon)}{\dot{\tanh}(\varepsilon)},$$

   against T, and compare it to the approximative curve. How do they drop against time?

4. What does your observation imply for training of an RNN? Use your observation to explain why $\tanh$ helps slowing down the vanishing gradient in RNNs.

2 Programming Exercises

Question 1 [40 + 5 (Optional) Points] **(Classifying MNIST via Basic Shallow RNN)** It is not only the XOR computation that could be much easier via RNN. In fact, using RNNs we can perform conventional classification also efficiently with simple NNs. In this question, we aim to classify MNIST via a shallow vanilla RNN. Let's first load MNIST dataset and break it into training and testing datasets.

1. Write a code that loads the MNIST training and test sets and split them into mini-batches of size `batch_size`. You may complete the reference code.

To this end, we need first to interpret MNIST dataset as a set of sequence data. Read the following text to find out how we do it.

MNIST Images as Sequences We can see an image as a sequence, where each sub-part of the image describes an entry of the sequence. For MNIST, we represent each image as a sequence of pixel vectors with each vector being 28-dimensional, i.e., $\mathbf{x}[t] \in \mathbb{R}^{28}$. We label each sequence with a single label which is the label of the image. With this interpretation, an MNIST image is given by a sequence $\mathbf{x}[1], \ldots, \mathbf{x}[T]$ and has only one label at the last time instant, i.e., $y[T]$, that represent its class.

Considering the above interpretation, answer the following items:

2. What is the length of each sequence in the MNIST dataset?

3. What type of sequence to sequence problem, e.g., many-to-many, one-to-many, etc, is this problem?

We aim to use a shallow RNN, i.e., one input layer, one recurrent unit, and one output layer. The time diagram of this RNN is shown in Figure 2.1. We assume that the size of the hidden state is 150, i.e., $\mathbf{h}[t] \in \mathbb{R}^{150}$.

Given this architecture, answer the following items:

4. Specify the input and output dimension of the hidden and output layers of this RNN.

5. Explain how each layer is activated in this RNN.

6. Assume that the RNN is trained. Explain how we can use the trained RNN to classify a new image.

We now implement the RNN, but not from scratch. We use some built-in modules. Read the following text to learn how you could do it.

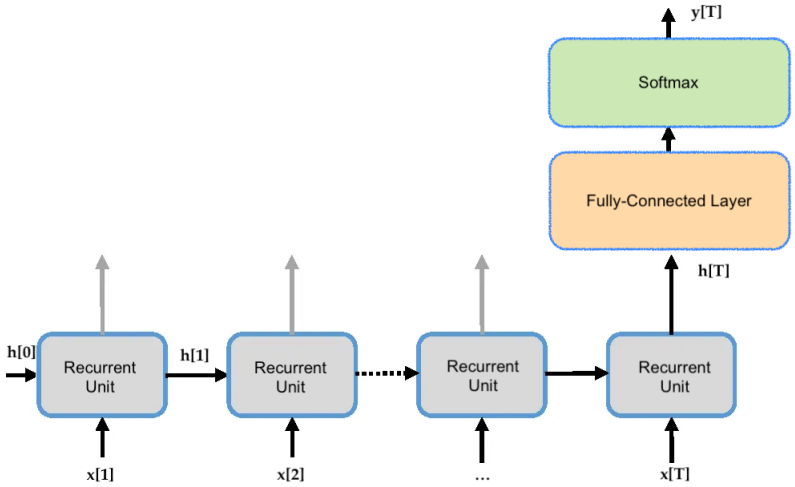

Figure 2.1: Architecture of out target RNN used for MNIST classification.

Implementing Basic RNN A basic Elman-like recurrent unit in the `nn` module of PyTorch by calling `nn.RNN()`. This class gets the dimension of input and hidden state as well as other hyperparameters to make an Elman-like unit for RNN. Note that this is *only the recurrent unit*, and we yet need to implement output layer. We could make it deep by setting the number of hidden layers to be more than one. However, we will not do it in this assignment. Important thing that we should consider is to set `batch_first = True`. This is to make sure, that our recurrent unit will consider the first dimension of the input to be the batch-size. This option is by default `False`, so we should make sure to set it on. For instance, if we want to make a single-layer recurrent unit with 2-dimensional input and 3-dimensional hidden states, we write:

```
RNN_layer = nn.RNN(2, 3, batch_first = True)
```

Now, if we pass a sequence `x` forward in this unit, we will get two outputs:

1. The *sequence of hidden states* which has the same time length as `x`.
2. The *last* hidden state, i.e., $\mathbf{h}[T]$ whose dimension is the same as *one time entry* of `x`.

We now start with implementing our RNN via this basic recurrent unit.

Answer the following items:

7. Write the class `myRNN` that realizes the components of the RNN. You can do it by completing the reference code.
8. Add function to this class that generates an initial *zero* hidden state. The hidden state should be a tensor of size `(1, batch_size, size_of_hidden_state)`.
9. Add a forward pass to the class.

We next need a function that takes the output of the RNN to an input mini-batch and computes the accuracy of classification by comparing the output with the true label. The implementation is slightly different to FNNs, since in this case we classify the image based on the last entry in the output sequence.

10. Write a function that gets the output of the last time step for a complete mini-batch along with the list of true labels and returns the average error over the mini-batch. You can do it by completing the reference code.
11. Confirm that your implementation returns correct dimensions.

We now complete the training loop for this RNN and train the model.

12. Write the function `train()` that gets an instant model, a loss function and a number of epochs and trains the model using the given loss function for the specified number of epochs. You can do this by completing the reference code.
13. Instantiate `model = myRNN()` and pass it to the function `train()` to train it for 10 epochs with cross-entropy loss function.
14. Do you think that the same shallow FNN without any recurrence could return such result? How do you think the RNN understands the class of an image?

In case interested, you could further complete the following optional task.

15*. (Optional) Implement the recurrent unit, i.e., `nn.RNN()`, from scratch. Replace it in your implementation and compare the result with what you observed via `nn.RNN()`.

Question 2 [15 Points] **(Classifying MNIST via Shallow GRU)** We now modify our implementation by replacing the basic RRN with a gated recurrent unit (GRU). We can access a GRU directly in the `torch.nn` module as

```
nn.GRU(... , ... , batch_first = True)
```

The input and output of this unit are read similar to the basic recurrent unit.

1. Write the class `myGatedRNN` that realizes the RNN in Figure 2.1 with its recurrent unit being a single layer (shallow) GRU. You can do it by completing the reference code.
2. Add function to this class that generates an initial zero hidden state.
3. Add a forward pass to the class.

We now train this gated RNN. For this, we could use our `train()` function from Question 1.

4. Instantiate `model = myGatedRNN()` and pass it to the function `train()` to train it for 10 epochs with cross-entropy loss function.
5. Compare your result with the previous implementation via basic recurrent unit. Explain your observation.

Question 3 [15 Points] **(Classifying MNIST via Shallow LSTM)** As the last exercise, we want to modify our implementation by replacing the basic recurrent unit with an LSTM. We can access an LSTM directly in the `torch.nn` module by

```
nn.LSTM(... , ... , batch_first = True)
```

The input and output of this unit are read similar to the basic recurrent unit and GRU. However, we need to further input the *cell state*. As mentioned in the course, this is the state that remains in the LSTM and does not go to the upper layers.

1. Write the class `myLSTM` that realizes the RNN in Figure 2.1 with its recurrent unit being a single layer (shallow) LSTM. You can do it by completing the reference code.
2. Add function to this class that generates an initial zero hidden state and *cell state*.
3. Add a forward pass to the class.

We now train this gated RNN. For this, we could use our `train()` function from Question 1.

4. Instantiate `model = myLSTM()` and pass it to the function `train()` to train it for 10 epochs with cross-entropy loss function.
5. Compare your result with the previous two implementations. Explain your observation.