Deep Learning (CS324)
Assignment 2 Report

Fitria Zusni Farida
Student ID: 12112351

May 16, 2024

1 Introduction

(1) Multi Layer Perceptron (MLP)

Multi layer perceptron (MLP) is a supplement of feed forward neural network. It consists of three types of layers—the input layer, output layer and hidden layer. The input layer receives the input signal to be processed. The required task such as prediction and classification is performed by the output layer. An arbitrary number of hidden layers that are placed in between the input and output layer are the true computational engine of the MLP. Similar to a feed forward network in a MLP the data flows in the forward direction from input to output layer. The neurons in the MLP are trained with the back propagation learning algorithm.

(2) Convolutional Neural Network (CNN)

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be. A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.

(3) Recurrent Neural Network (RNN)

Recurrent neural networks (RNNs) are feed-forward neural networks that focus on modeling in the temporal domain. The distinctive feature of RNNs is their ability to send information over time steps. In their structure, RNNs have an additional parameter matrix for connections between time steps that promotes training in the temporal domain and exploitation of the sequential nature of the input. RNNs are trained to generate output where the predictions at each time step are based on current input and information from the previous time steps. RNNs are applicable to analysis of input in the time series domain. Data in this domain are ordered and context-sensitive, while elements in one timestep are related to elements in the previous time steps.

2 Motivation

The primary motivation of this assignment are as follow:

(1) Implement Multi-layer perceptron using PyTorch and compared the accuracy with Numpy implementation from assignment 1.

(2) Using CIFAR10 dataset, experiment the MLP model to obtain the best accuracy.

(3) Using CIFAR10 dataset, implement and train CNN model using reduced version VGG network architecture. Then, analyze the accuracy and loss.

(4) Implement Vanilla RNN to predict the last digit of palindrome. Then analyze the loss and accuracy.

(5) Analyze the accuracy of the RNN model with respect to input length.

3 Methodology

In this assignment, the task are divided into three parts: (1) PyTorch MLP, (2) PyTorch CNN, (3) PyTorch RNN.

(1) **Part I: PyTorch MLP**

Task 1: In this task, which is classification task, datasets provided by PyTorch is used to train and test the MLP model. The model is also implemented using PyTorch library. The model architecture is depicted in the following figure. Unlike Assignment 1 numpy array, this assignment used tensor from PyTorch.

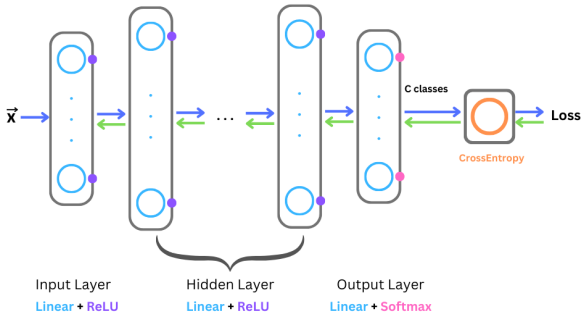


Figure 1: The architecture of the Multi layer perceptron.

Task 2: In this task, which is multi classification task, it implements the above MLP model for CIFAR10 datasets from PyTorch.

(2) **Part II: PyTorch CNN**

In this part, the task is to implement CNN model using PyTorch for multi classification task. The dataset used is CIFAR10 dataset from PyTorch. The model architecture is depicted as following Figure.

A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.

- Convolutional layer

  This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel or filter, and the other matrix is the restricted

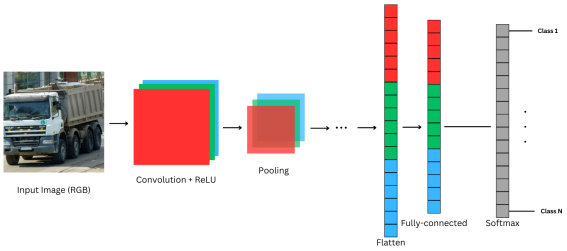


Figure 2: The architecture of the CNN

portion of the receptive field. The kernel applied can be figured as following. The number of learnable parameters of this layer can be calculated as follow:

number of parameters = (in_channel $\times$(kernel_width $\times$ kernel_height) $\times$ out_channel) + out_channel

From the convolutional layer, it will be obtained output dimension as following:

$$\text{Output Dimension} = \left\lfloor \frac{\text{Input Dimension} + 2 \times \text{Padding} - \text{Kernel Size}}{\text{Stride}} \right\rfloor + 1$$

The purpose of the convolutional operation is to extract higher-level features, such as edges, from an input image. Convolutional Neural Networks (ConvNets) are not restricted to just a single convolutional layer. Typically, the initial convolutional layer captures basic features like edges, colors, and gradient orientations. As more layers are added, the network begins to recognize more complex features, leading to a comprehensive understanding of the images in the dataset, much like human perception.

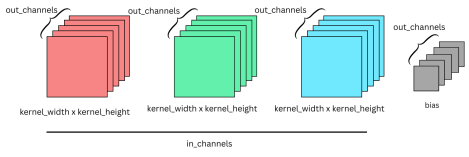


Figure 3: Convolutional layer

After convolution, it's common to apply ReLU function

for several benefits: introduce non-linearity, computational simplicity, avoid gradient vanishing problem, and faster convergence.

- Pooling

  Like the convolutional layer, the pooling layer serves to reduce the spatial dimensions of the convolved features. This reduction in size lowers the computational demand by decreasing the number of parameters, aiding in dimensionality reduction. Additionally, pooling helps in isolating dominant features that are invariant to rotation and position, thereby enhancing the model's training efficiency.

  There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

- Flatten

  The flatten operation transforms a multi-dimensional tensor into a one-dimensional tensor (or vector). This process is necessary when transitioning from convolutional layers (which produce multi-dimensional outputs) to fully connected layers, which require one-dimensional input.

- Fully-connected

  Fully connected layers are used to learn non-linear combinations of the high-level features extracted by the convolutional layers. The FC layer helps to map the representation between the input and the output. A fully connected layer performs a linear transformation. The number of learnable weights can be formulated as follow:

  Number of parameters = Number of inputs $\times$Number of neurons

  After fully-connected layer, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map. There are several types of non-linear operations, the popular ones being: Sigmoid, Tanh, and ReLU. For this classification task, Sigmoid activation will be implemented.

VGG (Visual Geometry Group) Neural Network

Following is breakdown of VGG network architecture:

1) Convolution layer: VGG uses a series of convolutional layers with small receptive fields (3x3 filters), which are followed by ReLU activation functions. The small filter size allows the network to capture fine details by increasing depth with a reduced parameter count compared to larger filters.

2) Pooling Layers: These are typically max pooling layers with a 2x2 window and a stride of 2, used to reduce the spatial dimensions (width and height) of the input volume for the next convolution layer.

3) Fully Connected Layers: After several convolutional and pooling layers, the architecture concludes with two or three fully connected layers. The last fully connected layer is used for classification, producing a distribution over class labels.

4) Fixed Input Size: VGG networks are typically trained with a fixed input size of 224x224 pixels.

(3) **Part III: PyTorch RNN** In this part, the task is to predict the last digit of a palindrome number or many-to-one sequence problem using Vanilla RNN.

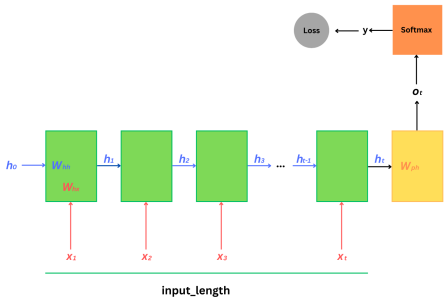


Figure 4: Many-to-one RNN architecture

Vanilla RNN is the simplest RNN among other types of RNN. The states are broken down into following:

- Hidden state

  h(t) is the hidden state of the RNN at time step t. It encapsulates the information that the network has processed up to that point in the sequence. The hidden state is updated at every time step based on the current input and the previous hidden state.

  The hidden state h(t) serves as the network's memory, carrying forward information from

one time step to the next. It accumulates and transforms information from the sequence, which is critical for tasks where context from the entire sequence affects the output.

$$h^{(t)} = \tanh(W_{hx}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

- Output

o(t) is the output of the network at time step *t*. In a many-to-one configuration like this problem, the final output o(T) (where T is the final time step) is typically used for making the final prediction. The output o(t) represents the model's prediction or decision at time *t*. In many-to-one RNNs, the primary focus is often on o(t), which uses the information to make a prediction based on the entire input sequence.

$$o^{(t)} = (W_{ph}h^{(t)} + b_o)$$

Backpropagation Through Time (BPTT)

In RNNs, each node in a layer is connected not only to the next layer but also to itself in the next timestep. This recurrent connection introduces dependencies not only across layers but also across timesteps. BPTT unrolls these temporal dependencies, treating them as additional layers in a deeper network, and applies the backpropagation algorithm:

(1) Forward Pass: The RNN processes inputs through its layers, updating its hidden states based on the previous hidden states and current input.

(2) Compute Loss: After processing all timesteps of an input sequence, a loss is computed. In many-to-one tasks, this might happen after the last timestep only; in many-to-many tasks, it could happen at each timestep.

$$y^{(t)} = \text{softmax}(o^{(t)})$$

(3) Backward Pass: The gradient of the loss function with respect to the output is calculated first. This gradient is then propagated backward through the network using the chain rule. For an RNN, this means moving backward through timesteps, recalculating and accumulating gradients for the weights at each timestep from the output back to the first input.

(4) Gradient Calculation: For each timestep, the partial derivatives of the loss with respect to the weights are computed by considering the contributions from the current timestep and also from the future timesteps, as the output at a given timestep can affect the loss in the future timesteps through the hidden states. This accumulation of gradients across timesteps is what makes BPTT computationally expensive and memory-intensive, as it essentially requires storing information from all timesteps during the forward pass.

(5) Update Weights: Once the gradients are calculated, weights are updated typically using a gradient descent method or any of its adaptive variants like SGD, RMSprop, or Adam.

4 Experiments

(1) **Part I: PyTorch MLP**

Task 1

1) Dataset Preparation

In this task, there are four datasets, in which each has two classes, trained and tested provided by PyTorch. The datasets are splitted into 80% for training and 20% fot testing. The datasets distribution for each classes are as follow:

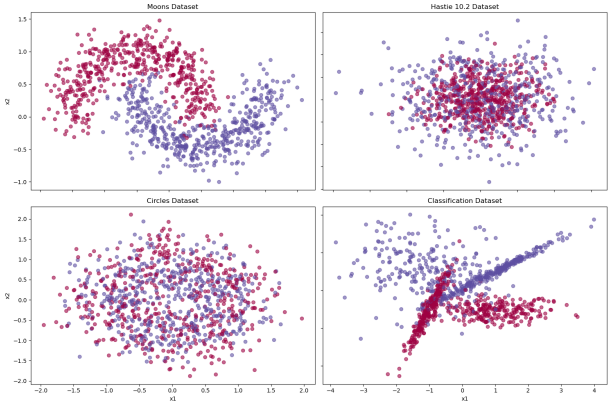


Figure 5: Dataset distribution

2) Hyperparameter configuration

The hyperparameter implemented in this experiment is as follow:

dnn hidden units = '20'

learning rate = 1e-2

max steps = 1500

eval freq = 10

mode = 'stochastic'

batch size = 1000

3) Performance Result

Table 1: Result in Numpy and PyTorch

Dataset	Numpy MLP		PyTorch MLP	
	Loss	Accuracy	Loss	Accuracy
Moon	0.2690	86.00%	0.2903	86.00%
Hastie	0.5759	69.50%	0.6106	69.50%
Circle	0.6846	56.49%	0.6882	56.00%
Multi	0.2610	87.50%	0.2769	87.50%

Task 2

1) Dataset preparation

The dataset is provided from CIFAR10. The CIFAR-10 dataset consists of 60,000 color images in 10 classes, with 6,000 images per class. The dimension of each images is 32 by 32 pixel. The dataset consist of RGB images which is represented as tensor with values within -1 and 1. In the formulation this input image is depicted as 2D array with 3 channels representing RGB (red, green, and blue primary color). To adjust with the model, the dataset is flatten. The dataset is divided into a training set of 50,000 images and a test set of 10,000 images.

2) Network Architecture

The MLP architecture used in this experiment has 3 hidden layers, first has 512 neurons, second has 256 neurons, and third has 128 neurons.

3) Hyperparameter configuration

The hyperparameter implemented in this experiment is as follow:

dnn hidden units = '512,256,128'

learning rate = 1e-2

max steps = 1500

eval freq = 10

mode = 'stochastic'

4) Performance result

In the step 1500, the loss and accuracy can be obtained are 1.8277 and 35.35% accordingly. Moreover, the training took 73 minutes and 59.1 seconds using CPU. After performing L2 regularization with weight weight decay 1e-5, the final accuracy slightly increase becomes 35.25% with 50.35 minutes.

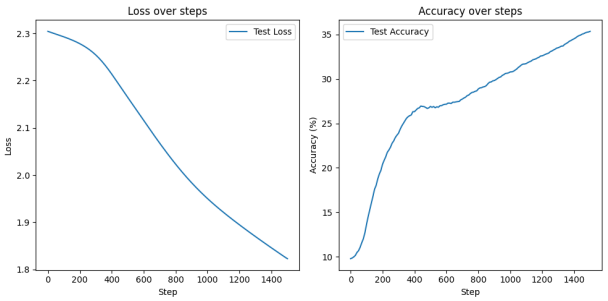


Figure 6: Loss and Accuracy Result Part I Task 2

(2) **Part II: PyTorch CNN**

1) Dataset Preparation

The dataset is provided from CIFAR10. The CIFAR-10 dataset consists of 60,000 color images in 10 classes, with 6,000 images per class. The dataset consist of RGB images which is represented as tensor with values within -1 and 1. The dimension of each images is 32 by 32 pixel. In the formulation this input image is depicted as 2D array with 3 channels representing RGB (red, green, and blue primary color). The dataset is divided into a training set of 50,000 images and a test set of 10,000 images.

2) Network architecture

The network architecture used in this experiment is relatively small as shown in the Figure 7.

3) Hyperparameter configuration

learning rate = 1e-4

max epoch = 50

eval freq = 10

optimizer = 'ADAM'

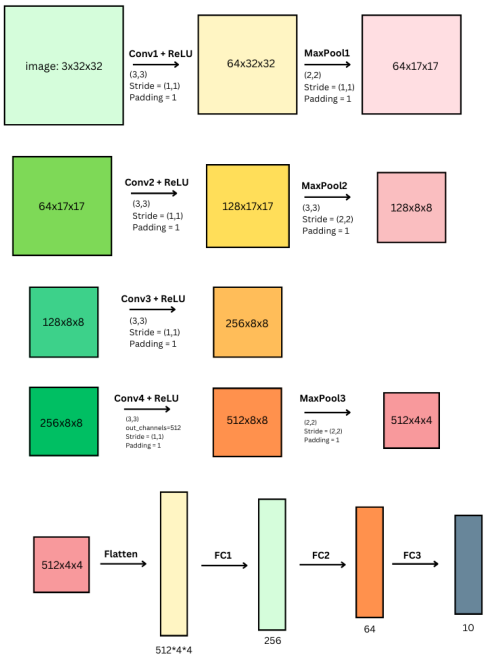


Figure 7: CNN Network architecture Part II

4) Performance result

In the epoch 50 as shown in Figure 8 and 9, the train accuracy and train loss can be obtained 78% and 2.06 accordingly. Whereas, the test accuracy and test loss are 72% and 2.07 accordingly.

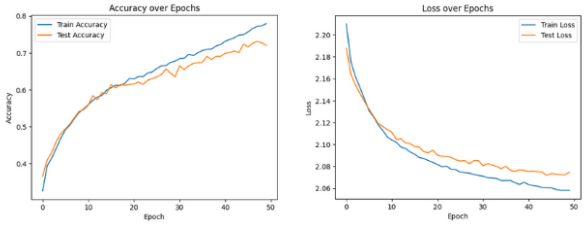


Figure 8: Part II Accuracy and Loss using ADAM

From the same model and dataset, this experiment also experiment using stochastic gradient descent as optimizer, and obtained following result: In epoch 50 as shown in the Figure 9 and 10, the train Accuracy is very small which is 0.09, train Loss is 2.30, test accuracy is 0.09 and test Loss is 2.30.

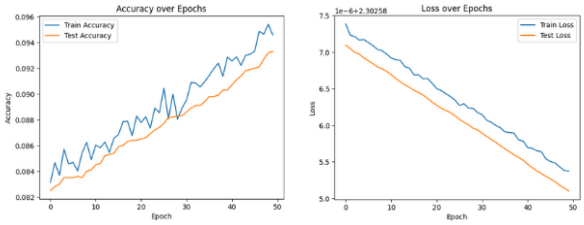


Figure 9: Part II Accuracy and Loss using SGD

(3) Part III: PyTorch RNN

1) Dataset preparation The dataset is palindrome dataset with a digit label which consist of digit numbers with input length N. In this experiment, there are 1,000,000 samples with input length is 19 for each sample. The dataset is divided into training and validation set with ratio 8:2.

2) Network architecture The network architecture used in this problem is shown in the Figure 4. For each neuron, it has input-to-hidden weights (Whx) and hidden-to-hidden weights (Whh), and hidden-to-input in the last neuron output prediction (Wph). The input dimesion is 1 since it implemented scalar digit input. The num classes or output dimension is 10, since there are 10 digit integers.

3) Hyperparameter configuration

input dim = 1

num classes = 10

num hidden = 128

batch size = 128

learning rate = 0.001

max epoch = 100

max norm = 10.0

data size = 100000

portion train = 0.8

4) Performance result

As shown in Figure 10, With input length 19, the accuracy reach the highest during epoch=10, which is

20% training 23% validation set. It starts to decrerase the accuracy until last epoch 30 with training accuracy 10% and validation accuracy 10%.

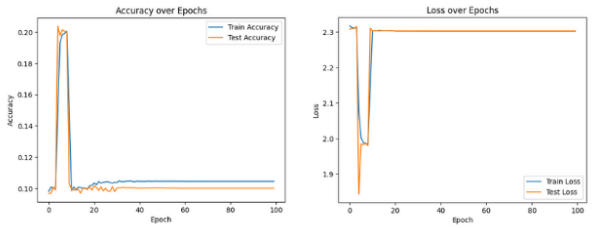


Figure 10: Part III input length 19 Loss and Accuracy

To draw relationship between the accuracy and input length, various input lengths are also experimented using same number of epochs. The result is shown in the Figure 11.

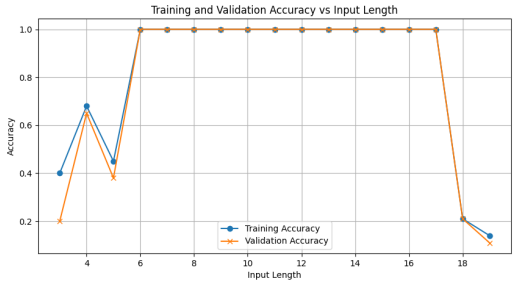


Figure 11: Part III Accuracy in epoch 30

5 Discussion and Insights

(1) **Part I: PyTorch MLP**

From the experiment result, it can be obtained following analysis:

- Task 1

In this MLP task, the configuration and network architecture is experimented similar with in Assignment 1. Based on Table 1, for each dataset, the result show that both implementation using numpy and PyTorch have the same accuracy and loss, which should be. However, over the epoch, they show different behavior. These two model use the same optimizer, which is stochastic gradient descent. Numpy implementation show more distinguishable fluctuation over the epoch compared with PyTorch implementation. Despite using identical hyperparameters, optimizer, loss function, and architecture, it can be attributed to several factors. Firstly, random initialization of network weights, which may differ slightly between PyTorch and NumPy due to their internal handling. Secondly, numerical precision and floating-point arithmetic discrepancies can cause the two implementations to diverge over time. Also, subtle differences in the implementation details of backpropagation and gradient calculations between the two libraries may impact how the model update the weights.

- Task 2

In this task, which is multi classification task for CIFAR10 dataset, use the same MLP model as Task 1. The model use more complex architecture than Task 1, which is 512, 256, and 128 neurons in the hidden layers. Using Adam optimizer, despite the low accuracy in the first epoch, which is only 9.79%, the loss decrease periodically over the steps, and the accuracy increase regularly as well. In the step 1500, it can be seen that the accuracy is 35.35%. After performing L2 regularizatin with weight decay 1e-5, the accuracy slightly increase 36.25% and the training takes less time. Although it may be smaller than as expected, the result shows improvement of the model over the steps. It can be assumed that higher steps and more proper complex architecture may lead to higher accuracy.

(2) **Part II: PyTorch CNN**

Based on the Figure 7, which is the implemented CNN architecture using Adam as optimizer, it is obtained 78% in the final epoch. It is also shown that the accuracy increase regularly over the epoch. In another scenario using stochastic gradient descent, with the same model, datasets, and hyperparameter, it is shown that the accuracy obtained is much worse than using Adam optimizer. Furthermore, it can be seen from the diagram that Adam optimizer con-

verges faster than stochastic gradient descent. The sgd implementation increase constantly if it is drawn a straight line. However, for this image classification task, it can be drawn a conclusion that CNN model is more suitable rather than MLP because CNN provides more features to learn the images.

(3) **Part III: PyTorch RNN**

From the result in the Using default hyperparameters with input length 19, epoch 100 and 100,000 generated samples, in the epoch 100, it can be obtained roughly 20% accuracy both in training and validation set. This result is slightly similar with input length 18. However, input length 6 to 17 are able to achieve perfect accuracy with different epoch converged. From the result, it can be seen that they are able to converge in around epoch 3-5, except for input length 6 and 7, which require more than 10 epoch to converge. Input length 3 to 6 achieve low accuracy but 2-3 times higher than 18-19 input length. This may be due to the dataset generated according to the input length, in which if the input length is too short, it will generate smaller number dataset (in which short input length surely will have many duplicates samples), which is not comparable. However, in this experiment, it is also experimented to generate the same number of samples regardless of the input length. The result is in the input length 3 to 6, it is able to achieve perfect accuracy as well. Moreover, in this experiment use 100 epoch, however, the model is able to converge even in small epoch.

6 Conclusions

In this series of experiments, we explored the performance of three different neural network architectures—Multi-Layer Perceptrons (MLP), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN) on various tasks. MLPs are versatile and can handle a variety of tasks including classification and regression. They are easy to implement and serve as a good baseline model. MLPs struggle with high-dimensional data and complex spatial hierarchies due to the lack of spatial invariance. They also do not effectively capture temporal dependencies in sequential data. CNNs excel in tasks involving spatial data, such as image classification and object detection. Their ability to capture local spatial features through convolutional layers and pooling operations makes them highly effective for image-related tasks. CNNs are less effective for tasks involving sequential data or where temporal relationships are crucial. They require substantial computational resources and can be complex to train. RNNs are specifically designed to handle sequential data, making them ideal for tasks like language modeling, time series prediction, and sequence classification. They can capture temporal dependencies and context over time.

7 References

[1] ScienceDirect. (n.d.). Multilayer perceptron. Retrieved from https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron

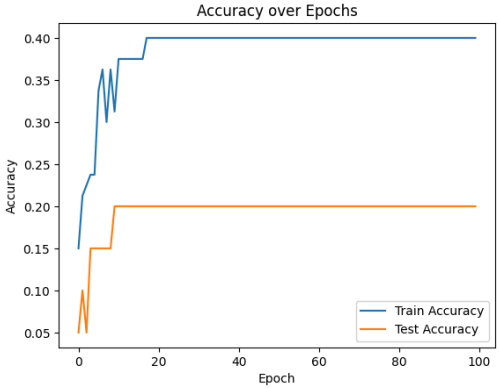
[2] Alammar, J. (2018, December 31). A comprehensive guide to convolutional neural networks—the ELI5 way. Towards Data Science. Retrieved from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[3] Patil, M. (2018, October 31). Convolutional neural networks explained. Towards Data Science. Retrieved from https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

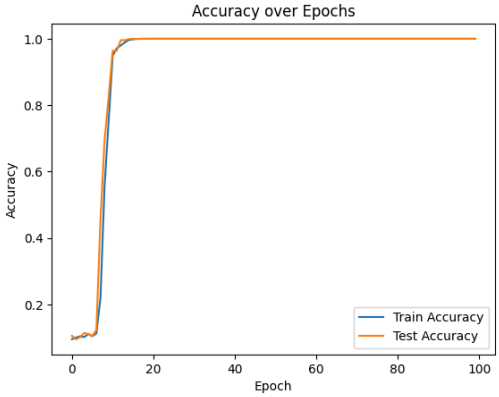
[4] ScienceDirect. (n.d.). Recurrent neural network. Retrieved from https://www.sciencedirect.com/topics/engineering/recurrent-neural-network

Appendix

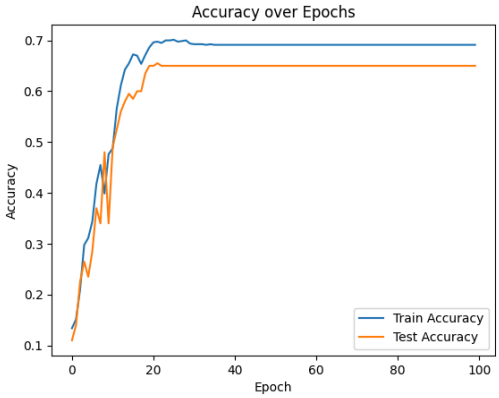
Part III: PyTorch RNN



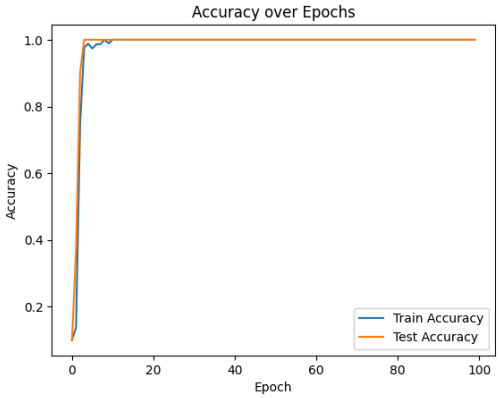
input length = 3



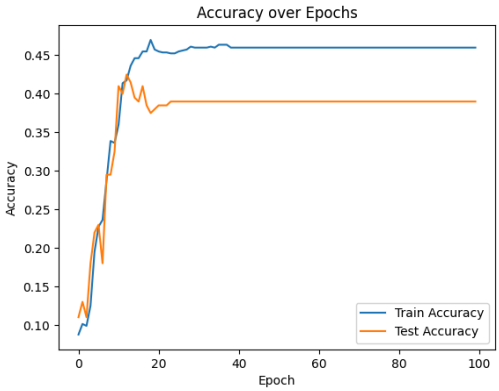
input length = 7



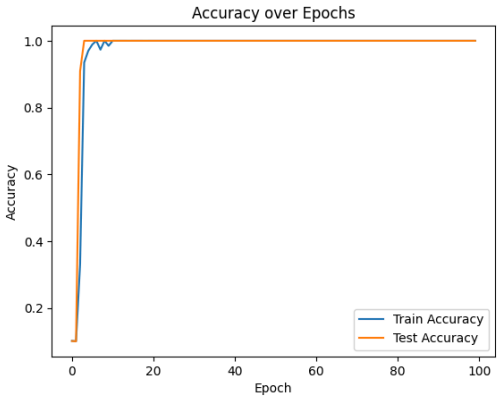
input length = 4



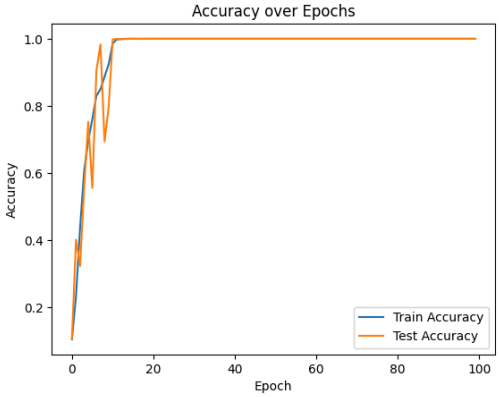
input length = 8



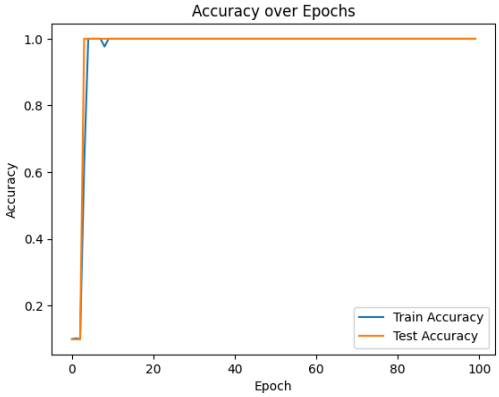
input length = 5



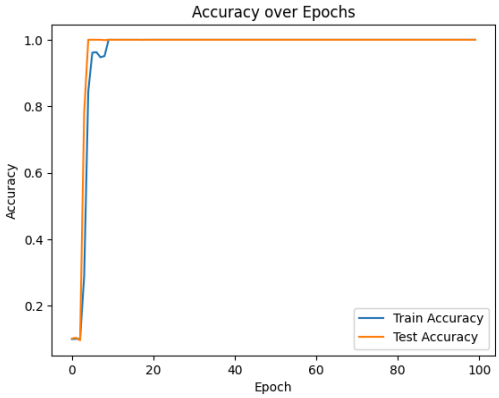
input length = 9



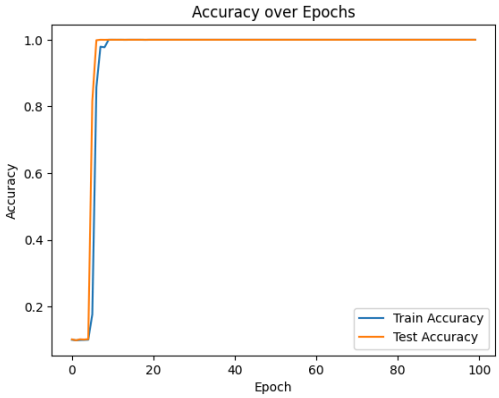
input length = 6



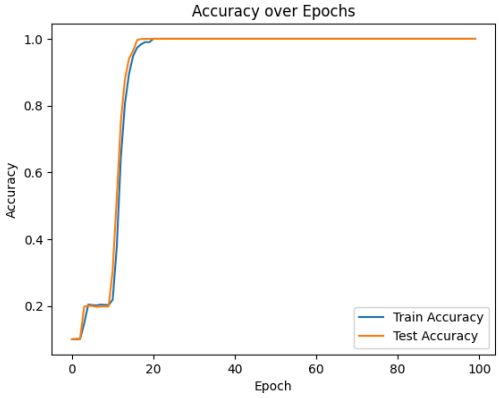
input length = 10



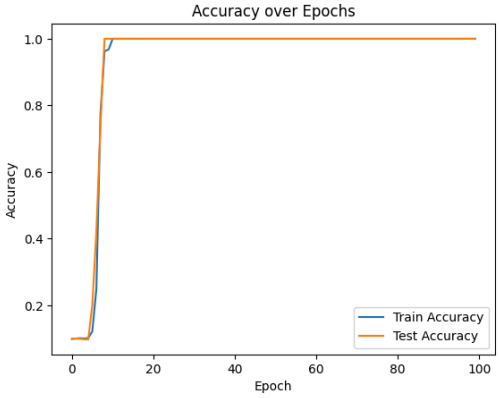
input length = 11



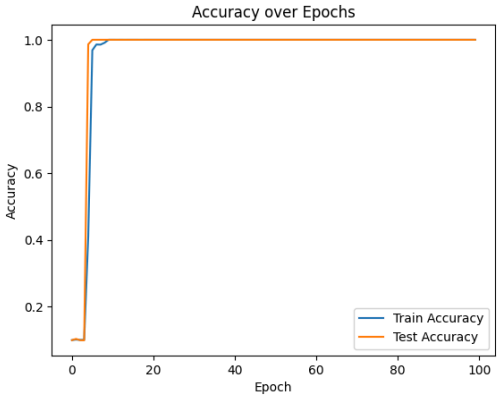
input length = 15



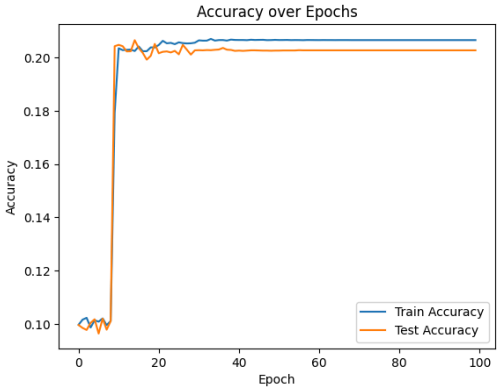
input length = 12



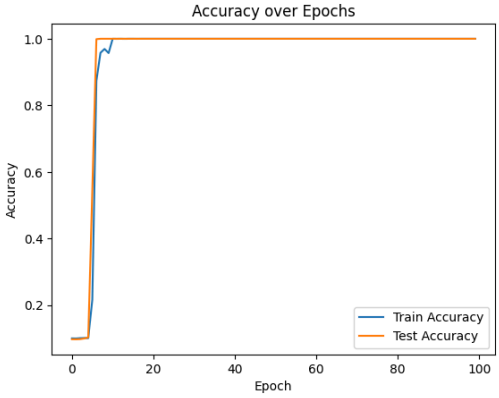
input length = 16



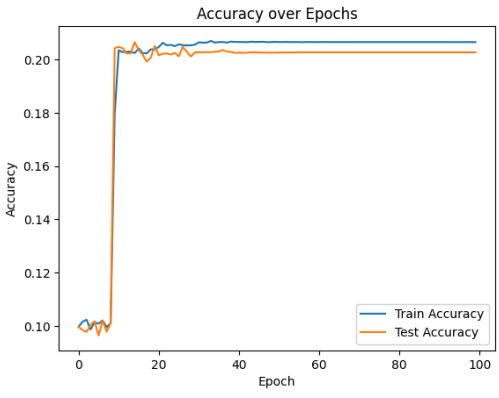
input length = 13



input length = 17



input length = 14



input length = 18