Machine Learning Course

Exercise 3

Name:

Eden Koresh

ID:

206845315

Date:

27/01/2025

Contents

Section 1

In this problem we modified the ANN from “Implementing a Multi-layer Artificial Neural Network from Scratch”- chapter 11 of the book “Machine Learning with PyTorch and Scikit-Learn” by Raschka et al. (2022) used to classify handwritten digits.

Chapter 11 focuses on building a multilayer ANN from the ground up. They are inspired by the human brain and designed to solve complex problems. Backpropagation is used to train ANNs efficiently. It involves propagating the error backwards through the network to adjust the weights and biases.

The relevant code is single-layer Neural Networks. The chapter briefly reviews the Adaptive Linear Neuron (Adaline) algorithm from Chapter 2, which uses gradient descent to learn model weights. In Adaline the activation function is a linear function. A threshold function squashes the continuous output into binary class labels for prediction. Stochastic gradient descent (SGD) is a method to accelerate model learning by approximating the loss from a single training sample or a small subset of training examples, which also helps in escaping local loss minima.

Multilayer Perceptrons (MLPs) are feedforward ANNs where layers are fully connected. A two-layer MLP includes an input layer, a hidden layer, and an output layer. The number of layers and units in an NN are hyperparameters.

- Each node in a layer is connected to all nodes in the next layer via a weight.
- The activation of a hidden layer is calculated using a differentiable activation function such as the sigmoid function.

Input data is passed through the network to generate an output. The activation of the hidden layer is calculated by applying an activation function to the net input of each node. This process is then repeated for the output layer. The chapter uses linear algebra concepts to vectorize the code implementation using NumPy, which makes it more efficient. Matrix-vector multiplication is used to compute the net input and activation of each layer.

The dataset used in the ANN we modified is the MNIST dataset of handwritten digits. The dataset is pre-processed by unrolling the 28x28 pixel images into one-dimensional row vectors, normalizing the pixel values to a range between -1 and 1, and splitting the dataset into training, validation, and test sets. The Mean Squared Error (MSE) loss is used, which simplifies gradient calculations.

The backpropagation algorithm calculates the gradients of the loss with respect to the weight and bias parameters. These gradients are then used to update the parameters via gradient descent. The backpropagation process is an application of the chain rule, calculating partial derivatives of the loss function with respect to the weights and biases by propagating error from the output layer to the input layer. The weights are updated using a stochastic gradient descent method.

Mini-batch Learning: Instead of using the full training set for calculating the gradient, the chapter uses mini-batches, which are small subsets of the training data. This allows for faster learning and better computational efficiency.

The chapter notes that the gap between training and validation accuracy can increase as the model trains for more epochs, indicating the model is overfitting to the training data. This can be reduced by techniques such as increasing regularization or using dropout.

Therefore, after reading and consideration of the written above, we modified mainly the model class of the original code:

```
class NeuralNetMLP:

    def __init__(self, num_features, num_hidden_1, num_hidden_2,
num_classes, random_seed=123):
        super().__init__()

        self.num_classes = num_classes
        rng = np.random.RandomState(random_seed)

        # hidden1
        self.weight_h1 = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden_1, num_features))
        self.bias_h1 = np.zeros(num_hidden_1)

        # hidden2
        self.weight_h2 = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden_2, num_hidden_1))
        self.bias_h2 = np.zeros(num_hidden_2)

        # output
        self.weight_out = rng.normal(
            loc=0.0, scale=0.1, size=(num_classes, num_hidden_2))
        self.bias_out = np.zeros(num_classes)

    def forward(self, x):
        # Hidden layer 1
        # input dim: [n_examples, n_features] dot [n_hidden1, n_features].T
        # output dim: [n_examples, n_hidden1]
        z_h1 = np.dot(x, self.weight_h1.T) + self.bias_h1
        a_h1 = sigmoid(z_h1)

        # Hidden layer 2
        # input dim: [n_examples, n_hidden1] dot [n_hidden2, n_hidden1].T
        # output dim: [n_examples, n_hidden2]
        z_h2 = np.dot(a_h1, self.weight_h2.T) + self.bias_h2
        a_h2 = sigmoid(z_h2)

        # Output layer
        # input dim: [n_examples, n_hidden2] dot [n_classes, n_hidden2].T
        # output dim: [n_examples, n_classes]
        z_out = np.dot(a_h2, self.weight_out.T) + self.bias_out
        a_out = sigmoid(z_out)
        return a_h1, a_h2, a_out
```

```
    def backward(self, x, a_h1, a_h2, a_out, y):

        ########################
        ### Output layer weights
        ########################

        # onehot encoding
        y_onehot = int_to_onehot(y, self.num_classes)

        # Part 1: dLoss/dOutWeights
        ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
        ## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet
        ## for convenient re-use

        # input/output dim: [n_examples, n_classes]
        d_loss__d_a_out = 2.*(a_out - y_onehot) / y.shape[0]

        # input/output dim: [n_examples, n_classes]
        d_a_out__d_z_out = a_out * (1. - a_out) # sigmoid derivative

        # output dim: [n_examples, n_classes]
        delta_out = d_loss__d_a_out * d_a_out__d_z_out # "delta (rule)
placeholder"

        # gradient for output weights

        # [n_examples, n_hidden_2]
        d_z_out__dw_out = a_h2

        # input dim: [n_classes, n_examples] dot [n_examples, n_hidden_2]
        # output dim: [n_classes, n_hidden_2]
        d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)
        d_loss__db_out = np.sum(delta_out, axis=0)

        #################################
        # Part 2: dLoss/dHiddenWeights
        ## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet *
dHiddenNet/dWeight

        # [n_classes, n_hidden_2]
        d_z_out__a_h2 = self.weight_out

        # output dim: [n_examples, n_hidden_2]
        d_loss__a_h2 = np.dot(delta_out, d_z_out__a_h2)

        # [n_examples, n_hidden_2]
        d_a_h2__d_z_h2 = a_h2 * (1. - a_h2) # sigmoid derivative

        # [n_examples, n_hidden_1]
        d_z_h2__d_w_h2 = a_h1

        # output dim: [n_hidden, n_features]
        d_loss__d_w_h2 = np.dot((d_loss__a_h2 * d_a_h2__d_z_h2).T,
d_z_h2__d_w_h2)
        d_loss__d_b_h2 = np.sum((d_loss__a_h2 * d_a_h2__d_z_h2), axis=0)

        #################################
```

```
        # Part 3: dLoss/dFirstHiddenWeights
        ## = DeltaHidden_2 * dHiddenNet_2/dHiddenAct_1 *
dHiddenAct_1/dHiddenNet_1 * dHiddenNet_1/dWeight_1

        # [n_hidden_2, n_hidden_1]
        d_z_h2__a_h1 = self.weight_h2

        # output dim: [n_examples, n_hidden_1]
        d_loss__a_h1 = np.dot((d_loss__a_h2 * d_a_h2__d_z_h2),
d_z_h2__a_h1)

        # [n_examples, n_hidden_1]
        d_a_h1__d_z_h1 = a_h1 * (1. - a_h1)  # sigmoid derivative

        # [n_examples, n_features]
        d_z_h1__d_w_h1 = x

        # output dim: [n_hidden_1, n_features]
        d_loss__d_w_h1 = np.dot((d_loss__a_h1 * d_a_h1__d_z_h1).T,
d_z_h1__d_w_h1)
        d_loss__d_b_h1 = np.sum((d_loss__a_h1 * d_a_h1__d_z_h1), axis=0)

        return (d_loss__dw_out, d_loss__db_out,
                d_loss__d_w_h2, d_loss__d_b_h2,
                d_loss__d_w_h1, d_loss__d_b_h1)
```

First in the _init_ functionality we added the initial weights and bias for the second layer. We carefully changed the size input of the whole layers to match the addition of the second layer.

In the forward functionality we added the connection of the second layer by dot producting the output of the first layer and the weights of the second layer. We added the bias of the second layer. In the end, we used the Sigmoid activation function and forwarded the result to the output layer.

The main modification happened in the backward functionality.

- The code computes the gradient of the loss with respect to the weights (d_loss__d_w_h2) and biases (d_loss__d_b_h2) in the second hidden layer.
- It propagates the error from the output layer (via delta_out) backward through the second hidden layer by taking into account the activation of the first hidden layer (a_h1) and the weights connecting the two hidden layers.
- The code further propagates the error from the second hidden layer back to the first hidden layer.
- It calculates the gradient of the loss with respect to the weights (d_loss__d_w_h1) and biases (d_loss__d_b_h1) in the first hidden layer using the input data (x) and the activations of the first hidden layer.
- Returns gradients for output layer weights, second hidden layer weights, and first hidden layer weights.

Let’s compare the results to the original single layer ANN:

Table 1: Comparison between the original accuracies and the modified code accuracies

Model	Train Accuracy	Validation Accuracy	Test Accuracy
Single-Layer ANN	95.61%	94.78%	94.54%
Multi-Layer ANN	96.10%	95.22%	95.22%

In this table we can see that all the accuracy values have increased from the single-layer to the multi-layer ANN. Moreover, we can see that there is not an overfitting since the accuracy values are high in both the validation and test subsets.

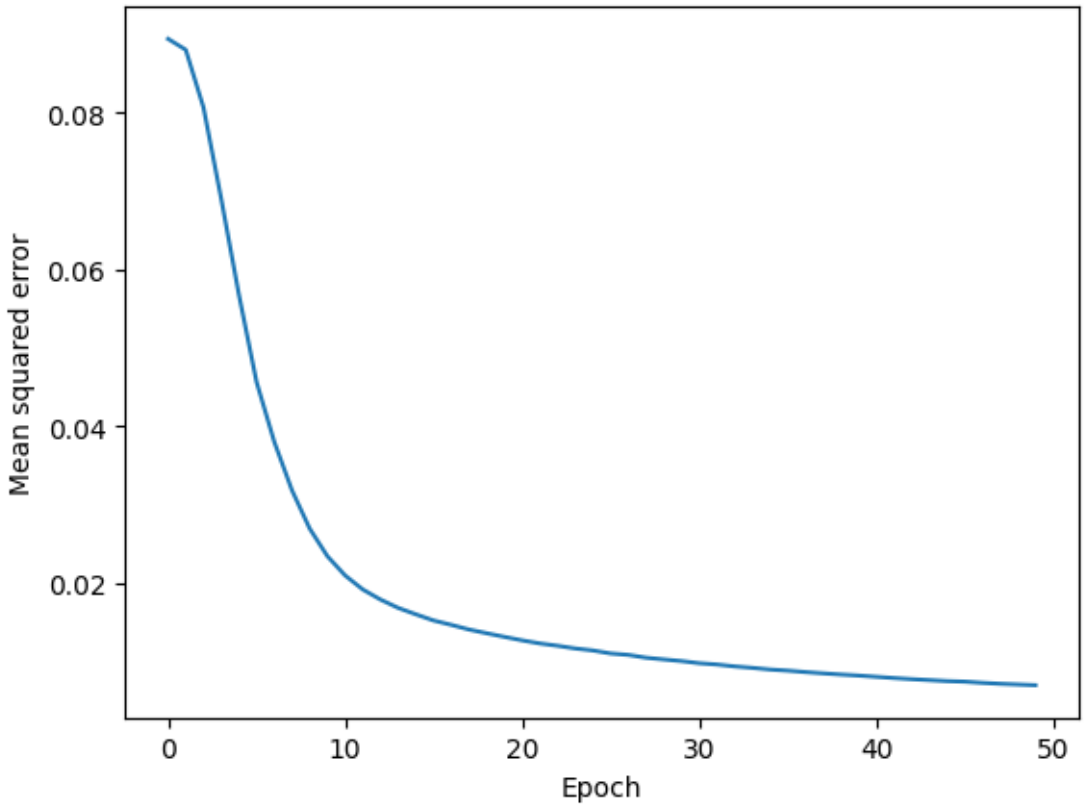

Figure 1: MSE as a function of epoch number

In this figure we can see that as the epochs continue the MSE is decreasing monotonically. In the first epochs, the slope is much steeper than at progressing epochs. The decrease is still quite substantial in the later epochs; hence we can infer that if we increase slightly the number of epochs, we can get even higher accuracy than what we already got.

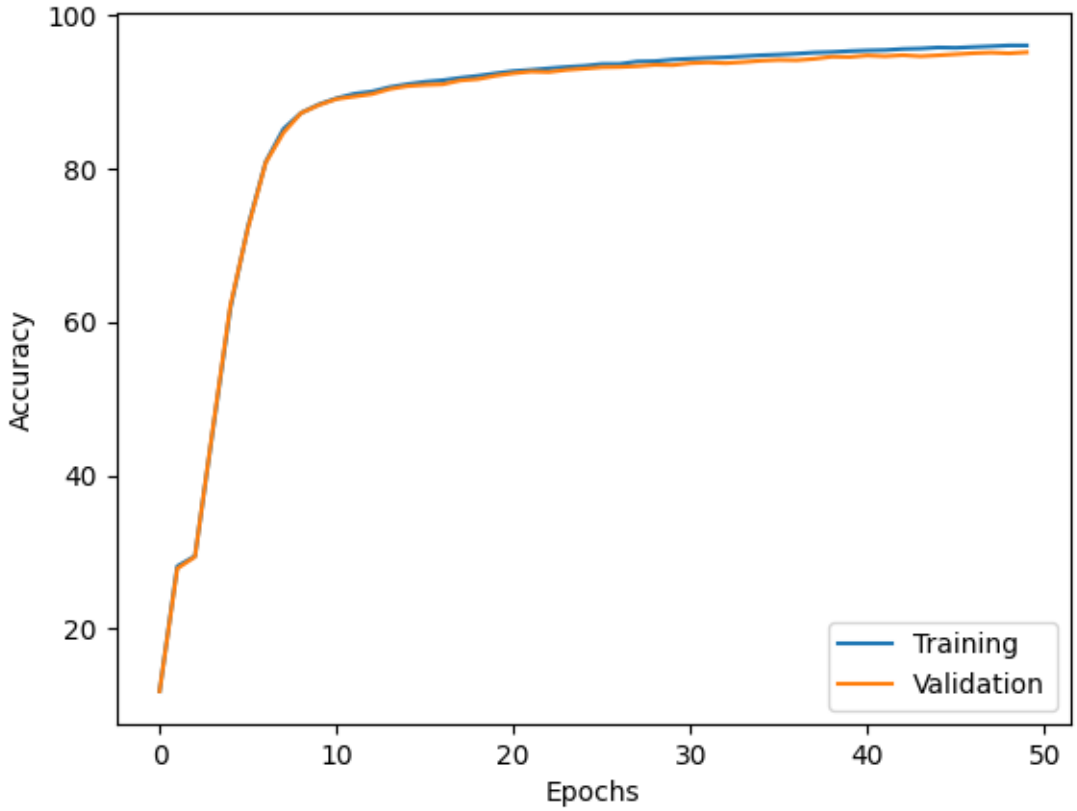

Figure 2: Training and validation accuracy as a function of number of epochs

In this figure we can see the accuracy increasing at each epoch. The validation and training behave basically the same.

Section 2

In this section we modified slightly the code to match the ANN presented in the lectures. Mainly all the characteristics are the same, the gradient calculation, the loss measurements, optimizer and the learning rate are the same. What we changed are the size of the hidden layers, replaced the activation function of the output layer from Sigmoid to Softmax and changed the number of epochs to 20.

- Larger hidden layers provide more representational power to learn intricate relationships. Conversely, smaller hidden layers can prevent overfitting on simpler datasets or when working with limited data. Adjusting hidden layer sizes ensures a balance between underfitting and overfitting.
- Sigmoid outputs values between 0 and 1 for each class independently, making it suitable for binary classification but problematic for multi-class tasks because the probabilities do not sum to 1. Softmax is specifically designed for multi-class classification. It converts raw scores into probabilities that sum to 1 across all classes, making it easier to interpret predictions and compute a proper cross-entropy loss.

We'll dive into the accuracy results of the code in the next section.

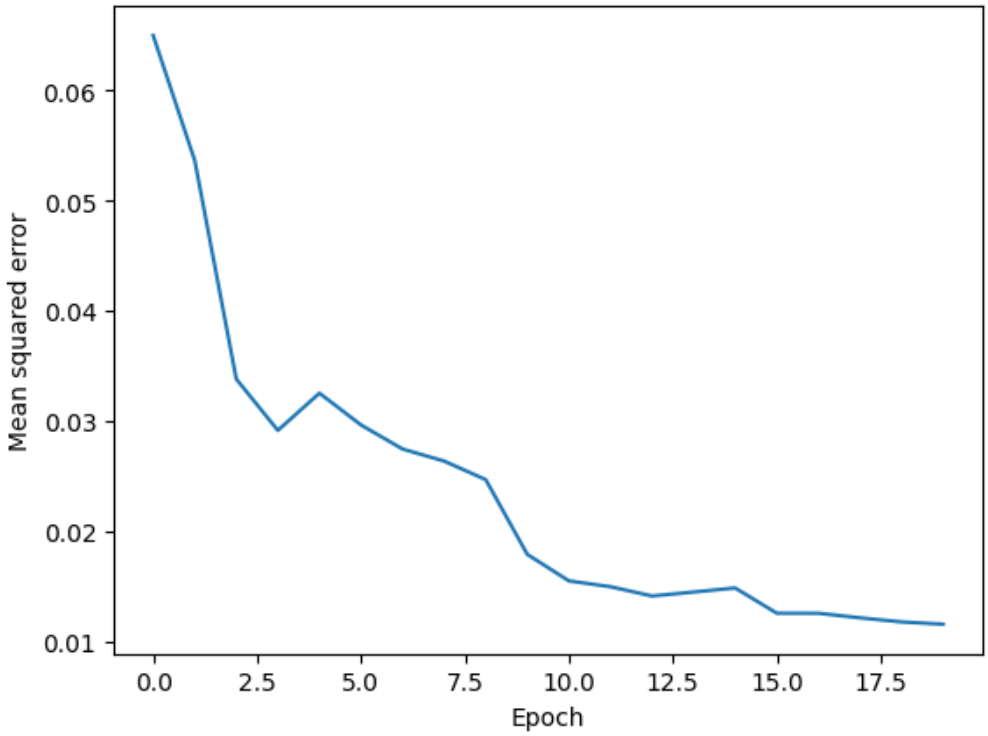

Figure 3: MSE as a function of number of epochs

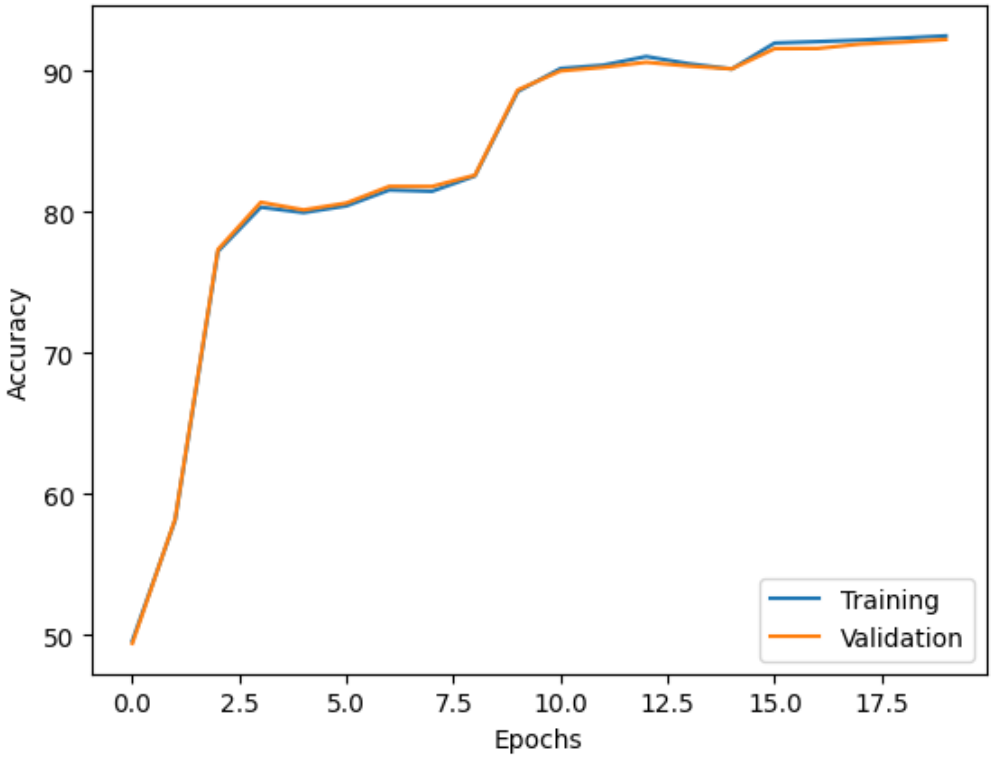

Figure 4: Accuracy as a function of number of epochs

From the 2 figures above, we can see that the learning was not optimal. The loss derogation was not monotonically decreasing, and the accuracy was also not rising monotonically. This could be affected by increasing the size of the hidden layers which increased also the complexity of the model. Therefore, we have not optimized learning curves.

Section 3

In this section we implemented the same ANN structure from the lectures in PyTorch.

We used the same architecture and hyperparameters but increased the learning rate. With a learning rate of 0.1, the model learned and trained very slowly, and the loss degradation were miniscule each epoch. We changed it to 0.8 and left the other hyperparameters the same.

This is the architecture of the ANN:

```
class NeuralNetMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Define feature extractor
        self.feature_extractor = nn.Sequential(
            nn.Linear(28*28, 500),
            nn.Sigmoid(),

            nn.Linear(500, 500),
            nn.Sigmoid(),
        )
        # Define classifier
        self.classifier = nn.Sequential(
            nn.Linear(500, 10),
            nn.Softmax(dim=1),
        )

    def forward(self, x):
        # Pass input through feature extractor and classifier
        x = self.feature_extractor(x)
        x = self.classifier(x)
        return x
```

We can see that this architecture is simpler than the manual architecture. All the propagation is performed by using loss.backward() and we don’t need any other calculation.

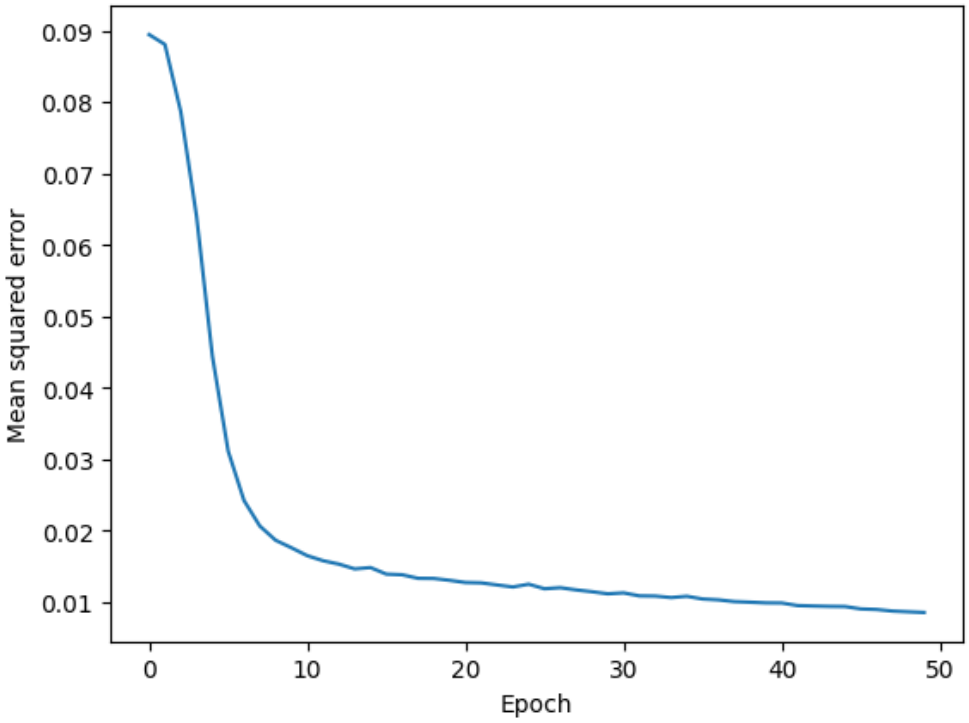

Figure 5: MSE as a function of number of epochs

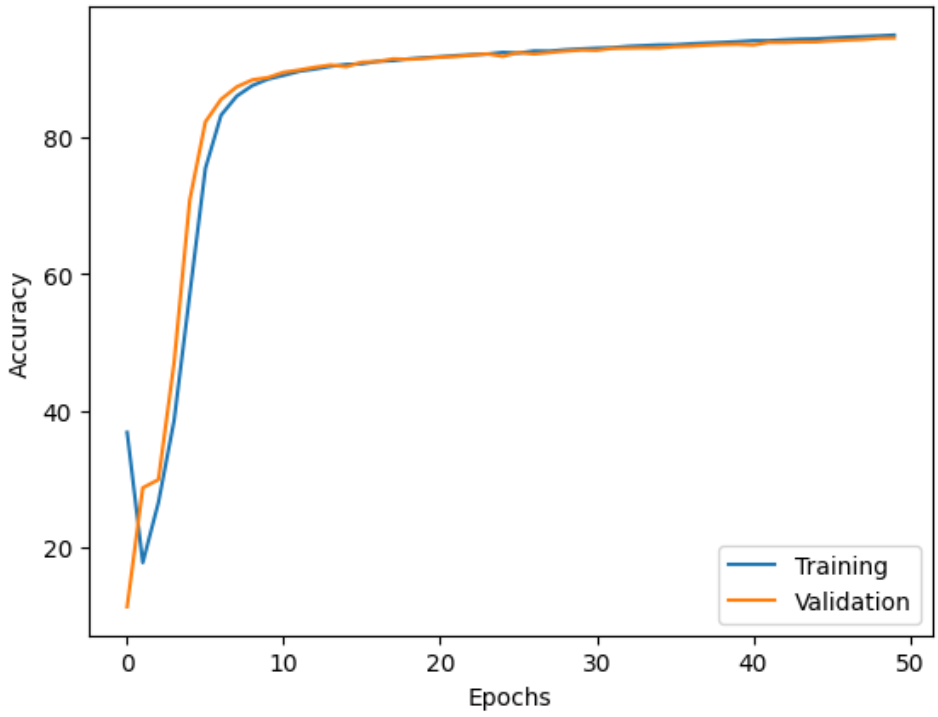

Figure 6: Accuracy as a function of number of epochs

In the figures above we can see that as the epochs continue the MSE is decreasing monotonically. There are some small bumps and occasionally increases but mostly they are minuscule. In the first epochs, the slope is much steeper than at progressing epochs. The decrease is still quite substantial in the later epochs; hence we can infer that if we increase slightly the number of epochs, we can get even higher accuracy than what we already got. The

accuracy figures show the accuracy increasing at each epoch beside for the first epoch. After that the validation and training behave basically the same.

Table 2: Model evaluations and comparison

Model	Train Accuracy	Validation Accuracy	Test Accuracy
Single-Layer ANN	95.61%	94.78%	94.54%
Multi-layer lecture ANN	92.50%	92.24%	92.10%
PyTorch implementation	94.99%	94.58%	94.31%

We can see from the table that the single layer were the most successful one in classifying the digits. This can be by some conditions; the lecture implementation were only 20 epochs, the accuracy could increase with more epochs (but with the tradeoff of possibly overfitting) and the PyTorch implementation of the ANN were not perfect and could be more optimized than what I did.

The other implementations are not perfect and could be improved by changing the activation functions, the size of the hidden layers and optimizing the learning rate.

In conclusion, despite seeing that the multi-layer did not perform as well for the single-layer, we can see the advantages of the multi-layer ANNs and that with some more tweaking, we can surpass easily the single layer, as we saw in Section1.