Lab report – TNM112

Deep learning for media technology, Lab 1

Filip Kayar (Filka428)

Sunday 1st December, 2024 (18:20)

Abstract

This report explores the construction and training of multi-layer perceptrons (MLPs) as part of a practical exercise in deep learning. Through a series of tasks, the lab investigates the impact of activation functions, weight initialization methods, and optimization algorithms on model performance. The study includes implementing an MLP from scratch and comparing its results to those from Keras. Key insights were gained into the trade-offs between hyperparameter choices, such as batch size and learning rate, and their effect on accuracy and loss. The report highlights the importance of theoretical understanding and experimentation in building effective neural network models.

1 Introduction

A neural network is often compared to a human brain but at a much smaller and less complex version. This is because it uses neurons to build a network which can be compared to how the human brain works. By connecting neurons to each other, we can achieve something that can take information as input and spit out something else. By training these networks we can manipulate each neuron independently and finally get a network that can for example tell us what an image contains or classify different classes.

A multilayer perceptron (MLP) is a neural network containing an input, output, and one or more hidden layers. MLPs are a fundamental type of feed-forward neural network, where information goes in a single direction, from the input layer to the output layer, passing through fully connected layers of neurons. A fully connected layer will have each neuron from the previous layer connected to all the other neurons in the next layer. Then each layer applies a linear transformation on the input followed by a nonlinear activation function on the output of the neuron, enabling the network to capture complex patterns in the data.

This lab will explore MLPs and how they work and also what hyperparameters can be utilized to train the MLP and achieve good results. By testing with different settings and parameters, such as varying the number of layers, the choice of activation functions, and the initialization of weights and biases. The report will aim to explore how these factors affect the learning process and performance of the network. The lab also experiments with how MLPs create decision boundaries to classify data into different classes. [1]

This lab is important to understand that MLPs are the building blocks of many advanced neural networks used in applications like computer vision, natural language processing, and reinforcement learning. Through this lab, we aim to understand both theoretical insights and practical implementations, enabling us to evaluate the strengths and weaknesses of MLPs in various contexts.

2 Background

The lab will be using premade MLP and implementing them self to understand the concept behind it. These models will then be initialized with different hyperparameters like layers, neurons per layer, initialization of weights, and biases. After initializing a model it has to be trained by using backpropagation and gradient descent to adjust it to the input data. The input data is also split into mini-batches which will be used to train the model where we update the weights and biases after each mini-batch. One pass through the whole training set or all the mini-batches is called one epoch.

3 Method

This section describes what has been done in the different tasks for this lab. The lab was implemented in Python with some already existing code for this lab. A library used extensively in these tasks was Numpy.

3.1 Dataset

The dataset was generated by using the code shown below where DataGenerator() is a class that can generate data in linear and polar format. The function generate() will be called either with linear or polar depending on the dataset required. Then the number of training and test data points we need, followed by the number of clusters, and lastly the deviation for the datasets. This will then create data points in a 2D-space.

```
data = data_generator.DataGenerator()
data.generate(dataset='linear', N_train=512, N_test=512, K=2, sigma
    =0.06)
```

3.2 Task 1

This task focuses on training an MLP in Keras and exploring the different hyperparameters. Using different datasets and model structures will give us a better understanding of how these models adjust to the data with respect to their hyperparameters.

3.2.1 Task 1.1

The first task was to create a linear dataset with two clusters where each cluster contained 512 data points. Then we classify these data points using Keras own implementation with zero hidden layers. This means that the input layer has two inputs connected directly to the output that only consists of two dimensions. This is because we only use two classes. When we run the model we use batch sizes 512 and 16 with the same data set, 4 epochs, and keep the learning rate to 1.0.

The two different batch sizes should give different results where when we use 512 we will compute the actual gradient. When using smaller batch sizes like 16, we calculate the stochastic gradient descent (SGD) since we introduce a bit of noise in the gradient. The noise comes from only choosing 16 data points of the 512 full dataset. Although this introduces noise it is better since it takes much less time to calculate compared to calculating the gradient from 512 data points. When using 512 data points of 512 in the training set we will get the correct gradient. Using a batch size of 16 should speed up the convergence of the model since using 16 data points for updating the gradient takes much less time to compute. The number of updates is four for 512 while the mini-batch size of 16 will update the gradient 128 times. This is due to the number of epochs.

3.2.2 Task 1.2

In this task the dataset is set to polar with the two clusters and 512 data points for each cluster. Unlike Task 1.1, which used no hidden layers, this model includes one hidden layer with five neurons. The network was trained for 20 epochs using a learning rate of 1.0 and a batch size of 16. Three different activation functions were used to evaluate the differences between them.

The first one is the linear activation function which maps the input directly to the output and doesn't change the data. It does not alter the input and can't describe more complex data that is non-linear. The sigmoid activation function introduces a non-linear transformation that maps input values into the range of [0,1], allowing the network to learn more complex relationships in the data. Large values will map to 1 and large negative values will map to 0. Rectified Linear Unit (ReLU) activation function applies a non-linearity, where all negative values are set to zero and all values above 0 are kept as they are.

3.2.3 Task 1.3

In this task, the dataset was set to a polar distribution with five clusters and 512 data points for the training set for each cluster. The clusters also had smaller distribution by assigning sigma to 0.05 which gave each data point a smaller variance. The model had 10 hidden layers where each layer contained 50 neurons and used ReLU activation function. ReLU was specifically used in this model since it helped with the vanishing gradient problem, allowing gradients to propagate throughout all 10 layers. Otherwise, the gradient would become too small when backpropagation is done and we would not be able to update our weights. The complexity of the network allows it to model the non-linear relationships in the data introduced by the polar clustering. However, a deeper network requires careful tuning of hyperparameters to achieve good results.

We adjusted some different hyperparameters to get a good model. Adjusting the initialization by changing the mean and standard deviation of the weights using normal initialization. Modifying the learning rate and adding momentum to the stochastic gradient descent (SGD). Learning rate decay by using ExponentialDecay, which is a Keras function, that creates an exponential decay schedule for the learning rate. This allowed the learning rate to decrease over time, this allowed the model to fine-tune steps at the end of the training. Trying different batch sizes and numbers of epochs.

All these hyperparameters were then adjusted to train the model. By tweaking all these hyperparameters and trying many times to get better results from the model a good understanding could be achieved after all the trial and error.

3.2.4 Task 1.4

This task has the dataset and network architecture as in Task 1.3, with five polar clusters and 10 hidden layers of 50 neurons each using ReLU activation. The main change was the use of glorot_normal() initialization for the weights and the Adam optimizer instead of SGD optimizer. Using Glorot gives us a better initialization of the weights and biases because it scales with respect to the number of input and output neurons, ensuring stable variance throughout the network. By using Adam instead of SGD the need to set the learning rate and momentum is not needed. In Adam the learning rate is adaptive and the momentum is automatically integrated with it which means that we do not need to manually try to find good values with tries and errors.

3.3 Task 2

In this second task, the goal was to implement the functions `activation`, `setup_model`, `feedforward`, and `evaluate` in mlp.py and validate that our custom MLP implementation produced the same results as the Keras model. The weights and biases trained using Keras were transferred to our own implementation of MLP to test if it was correctly implemented.

First the implementation of `activation` was implemented and consisted of an if-statement that checks which type of activation the function has been called with. Then the other input was the actual data being used for the activation functions. For each activation function, it applies a different activation function. The activation functions were linear, sigmoid, ReLU, and softmax. The implementation of these activation functions can be seen below.

```python
def activation(x, activation):
    if activation == 'linear':
        return x
    elif activation == "sigmoid":
        return 1/(1+np.exp(-x))
    elif activation == "relu":
        return np.maximum(0, x)
    elif activation == "softmax":
        return np.exp(x)/np.sum(np.exp(x),axis=1, keepdims=True)
    else:
        raise Exception("Activation function is not valid",
    activation)
```

The second function implemented was `setup_model` which as the name states set up the model with corresponding parameters. It initializes the model with activation function, number of hidden layers, weight matrices, bias vectors, and total number of weights in the model. The implementation of these can be seen in the code below in the corresponding order.

```python
    def setup_model(self,W,b,activation='linear'):
        self.activation = activation
        self.hidden_layers = len(W)
        self.W = W
        self.b = b
        self.N = 0
        for i in range(len(self.W)):
            self.N += np.prod(self.W[i].shape) + np.prod(self.b[i].
    shape)
```

The `feedforward` function is supposed to feed forward the input through the MLP and create an output, which in this lab is a prediction of a class. The input is fed to the function to be predicted where the input is set to the current output (y), this y will be updated as we forward pass through the network. For each layer, we need to apply a matrix multiplication to get the output for this layer, which is the input for the next layer. A general layer of the MLP can be show in equation 1. The sigma function is the activation function applied to each output of a layer's neurons. $h^{(l)}$ is a (Lx1) vector containing activations of layer l, $W^{(l)}$ contains the learnable weights with size of (LxK) in the layer l, $h^{(l-1)}$ is a vector of the activations from the previous layer (l-1) with size (K$\times$1) and $b^{(l)}$ is a vector of the biases for the layer l with size of (L $\times$ 1). K is the number of outputs from layer (l-1) and L is the number of inputs in layer (l), these can be seen as the respective neurons in each layer. [1]

```
def feedforward(self,x):
        y = x
        for index in range(self.hidden_layers):
            weightTransposed = self.W[index].T
            z = np.matmul(y, weightTransposed) + np.squeeze(self.b[
    index])
            y = activation(z, self.activation)
        y = activation(y, "softmax")
        return y
```

$$h^{(l)} = \sigma\left(W^{(l)}h^{(l-1)} + b^{(l)}\right), \tag{1}$$

The last function implemented was `evaluate` and its purpose was to evaluate the model that was created with the previous functions. If the implementation is done correctly the expected output should be the same as for the previous task, since we have copied the structure of the model and all the weights from it.

The code below shows the implementation of the evaultation function where it begins with get the predicted values for the training and test set. This gives us what the model thinks will be the correct class for each data point. Then the loss is calculated for the training set by calculating the difference between the predicted value and the true value. To do this the true value has to be one hot encoded, which means that the true value will be a vector of size (2x1) and is the same size as the predicted value. This is done so that we can take the difference between them. Then we calculate the loss by using Mean Square Error (MSE) which is implemented in the code below and defined by equation 2, where y_t and y_p is the true value and predicted value respectively. The correct training samples are then calculated and summed up. We find these by taking argmax to get the class with the highest probability in the prediction. Comparing the predicted class to the true class becomes either 1 or 0 for true or false, which then sums up to give us how many correct predictions the model got. The correct number of guesses is then divided by the total number of data points to give an accuracy of the model. This process is then done the same for the test set.

```python
def evaluate(self):
        print('Model performance:')

        pred_x_train = self.feedforward(self.dataset.x_train)
        pred_x_test = self.feedforward(self.dataset.x_test)

        train_loss = np.mean((pred_x_train - self.dataset.y_train_oh)**2)
        train_correct = sum(np.argmax(pred_x_train,axis=1) == self.dataset.y_train)
        train_acc = train_correct/len(self.dataset.x_train)
        print("\tTrain loss:     %0.4f"%train_loss)
        print("\tTrain accuracy: %0.2f"%train_acc)

        test_loss = np.mean((pred_x_test -self.dataset.y_test_oh)**2)
        test_correct = sum(np.argmax(pred_x_test,axis=1) == self.dataset.y_test)
        test_acc = test_correct/len(self.dataset.x_test)
        print("\tTest loss:      %0.4f"%test_loss)
        print("\tTest accuracy:  %0.2f"%test_acc)
```

$$MSE(y_t, y_p) = \frac{1}{n}\sum_{i=1}^{n}(y_{t,i} - y_{p,i})^2 \tag{2}$$

3.4 Task 3

In the last task, the idea is to create a decision boundary between two separate classes so that one side of the boundary is assigned to one class and vice versa for the other side. To achieve this, we use the linear dataset with two classes (K=2) and want to manually specify the weights for a single-layer model with no hidden layers. This will give us the possibility to change a total of six parameters, four weights and two biases. See matrices in equation 3.

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{3}$$

Since the classes are distributed along the line for y = x with the interval of 0 to 1 then the equation for the boundary can be formulated as $x_2 = 1 - x_1$, where x_1 is one axis and x_2 is the second one. In figure 1 we can see the desired decision boundary to separate the two different classes.

To create this boundary with the help of the weights and biases we have to calculate the outputs so that each class is assigned to their respective class. In equation 4 the matrix multiplication behind the small neural network can be seen where z_1 and z_2 is the output from the neural network for the respective class. The outputs will then be sent through softmax to get the probability for each class and lastly, we choose the highest probability and assign the input to the class with the highest probability. To calculate it we first know that z_1 has to be equal to z_2 for data points on the line $x_2 = 1 - x_1$. This gives that if we have an input on the line then our z_1 and z_2 will be equal, which lets us set equation 5 and 6 equal to each other. With the input as x_1 and $1 - x_1$ we get every point on the line and end up with the equation 7 and 8 where z_1 and z_2 will be equal and give us equation 9. Now we can extract the dependent variables from the equation which can be seen in equation 10. We also know

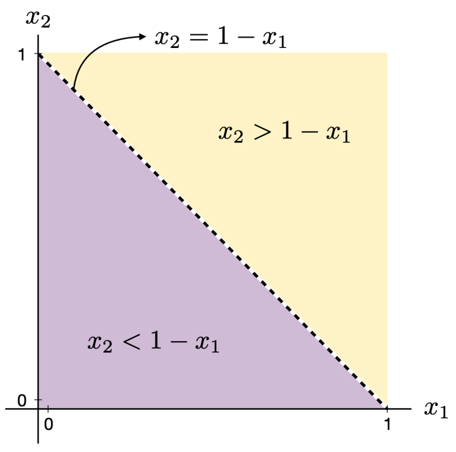

Figure 1: Illustration of the decision boundary to separate the classes by manually specifying weights and biases.

that b_1 has to be bigger than b_2 for the model to classify class 1 to the correct class since z_1 is the first class.

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4}$$

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1 \tag{5}$$
$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2 \tag{6}$$

$$w_{11}x_1 + w_{12} - w_{12}x_1 + b_1 = z_1 \tag{7}$$
$$w_{21}x_1 + w_{22} - w_{22}x_1 + b_2 = z_2 \tag{8}$$

$$(w_{11} - w_{12})x_1 + w_{12} + b_1 = (w_{21} - w_{22})x_1 + w_{22} + b_2 \tag{9}$$

$$\begin{cases} w_{11} - w_{12} = w_{21} - w_{22} \\ w_{12} + b_1 = w_{22} + b_2 \end{cases} \tag{10}$$

4 Results

This section will go through all of the results that was achieved in the different tasks. We will not go into much detail of why since this has been discussed in previous section.

4.1 Task 1.1

In figure 2 and 3 the decision boundary can be seen with each color being one class. The circles with a red outline are data points that have been wrongly classified. In table 1 the accuracy of each batch size can be seen and we can see that batch size of 16 gives better accuracy compared to 512 when using four epochs.

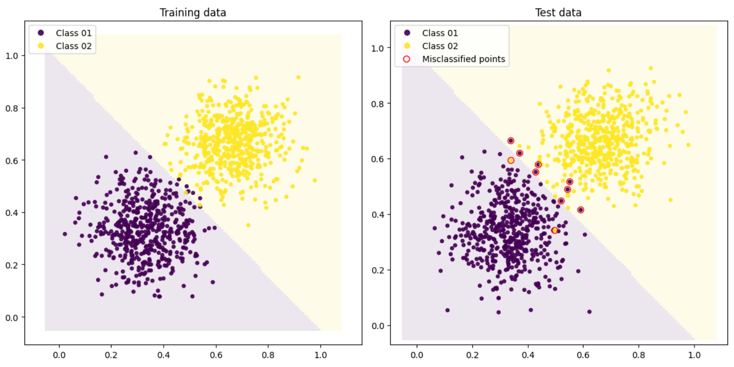

Figure 2: Classification with batch size of 16

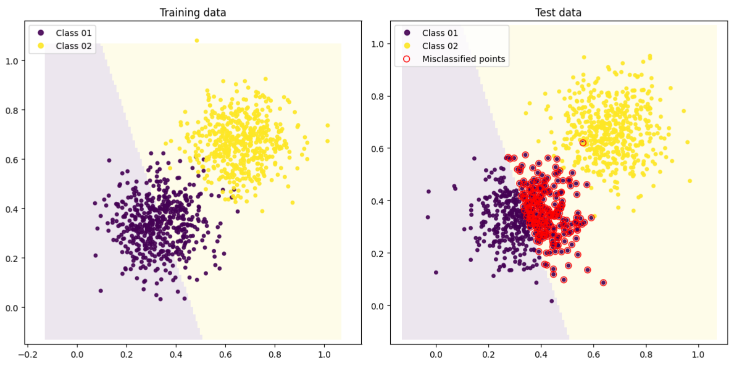

Figure 3: Classification with batch size of 512

Table 1: Accuracy vs. Batch Size

Batch Size	**Accuracy (%)**
16	98.83%
512	75.68%

4.2 Task 1.2

In this task we use the different activation functions on a polar data set. The results off linear, sigmoid and ReLU activation functions can be seen in figure 4, 5 and 6, respectively. We see that linear and sigmoid can not adjust the weights to match the polar dataset which ReLU can and this gives the best result, see table 2 for the accuracy.

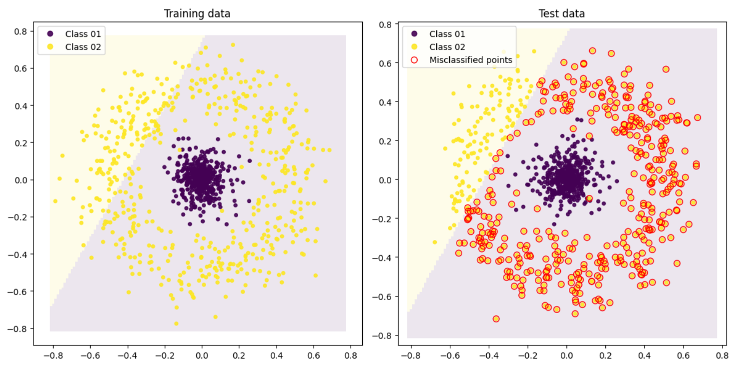

Figure 4: Classification using linear activation function

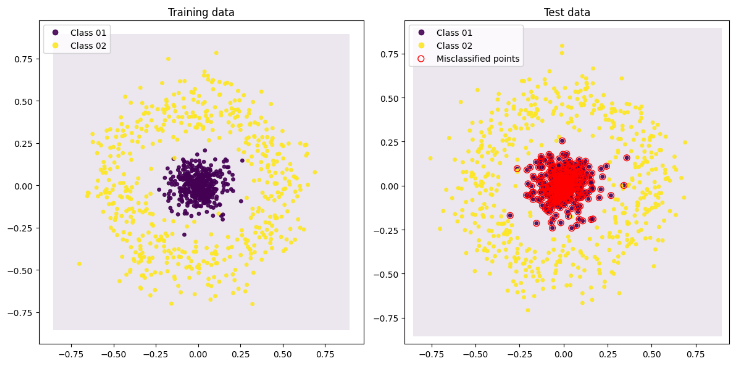

Figure 5: Classification using sigmoid activation function

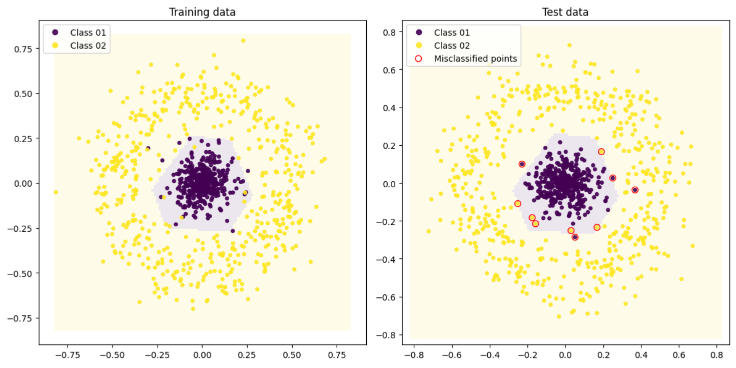

Figure 6: Classification using ReLU activation function

4.3 Task 1.3

Here we set the hyper-parameters manually by adjusting them in the code and see what works best. In the code below we can see what parameters gave the best result with the 10 hidden layers and 50 neurons in each layer. In figure 7 the result can be seen where the right image shows the test data and each data

Table 2: Accuracy of Different Activation Functions

Activation Function	Accuracy (%)
Linear	61.62%
Sigmoid	50.00%
ReLU	99.02%

point circled with red is misclassified. In figure 8 we can see the model's loss and accuracy over 100 epochs. The model reaches good results with about 96% accuracy and about 0.04 loss, see table 4.

```
# Hyper-parameters
hidden_layers = 10
layer_width = 50
init = keras.initializers.RandomNormal(mean=0.035, stddev=0.1)

epochs = 100
batch_size = 32
loss = keras.losses.MeanSquaredError()
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate = 0.05,
    decay_steps=400,
    decay_rate=0.85,
    staircase=True)
opt = keras.optimizers.SGD(learning_rate=lr_schedule, momentum=0.3)
```

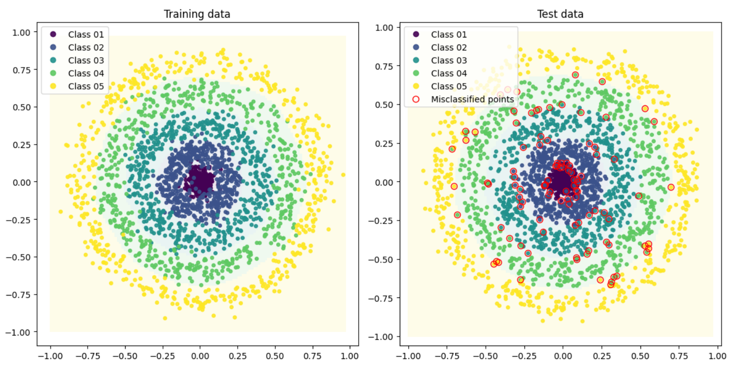

Figure 7: Classification by using the hyper-parameters from the code above

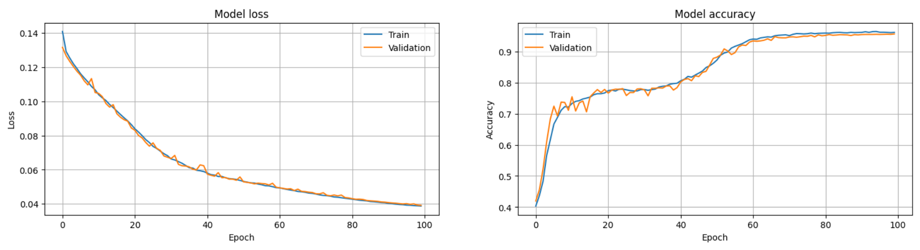

Figure 8: The model loss and accuracy over 100 epochs

Test Accuracy	Test Loss
96.09%	0.0389

Table 3: Final test loss and accuracy of Task 1.3

4.4 Task 1.4

In this task, we keep the code from task 1.3 but change the initializer and the optimizer. Only the changes made to the code from task 1.3 are in the code below, and we use the Adam optimizer with a glorot normal initializer. In figure 9 we can see the classification and in figure 10 we can see the loss and accuracy over 20 epochs. There is not much difference here regarding the accuracy but a little improvement in the loss. The big difference here compared to task 1.3 is that we only use 20 epochs, which actually is too much since we can see in figure 10 that we reach good results in just around 3-5 epochs.

```
init = keras.initializers.glorot_normal()
opt = keras.optimizers.Adam()
```

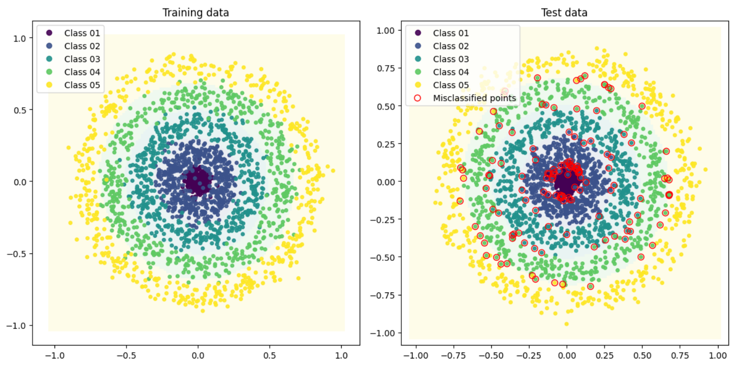

Figure 9: Classification by using Adam and glorot normal

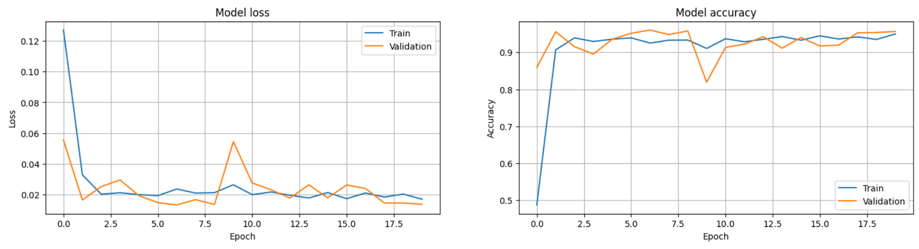

Figure 10: The model loss and accuracy over 20 epochs

Test Accuracy	Test Loss
94.80%	0.0152

Table 4: Final test loss and accuracy of Task 1.4

4.5 Task 2

For task 2 the model from task 1.4 is copied with its structure, weights, and biases. As we can see in figure 11 the classification is the same as in task 1.4 and the same goes for our train and test data, which can be seen in table 5. We get the exact same results since we just copy the whole model and its weights, so this proves that our implemented MLP works as it should.

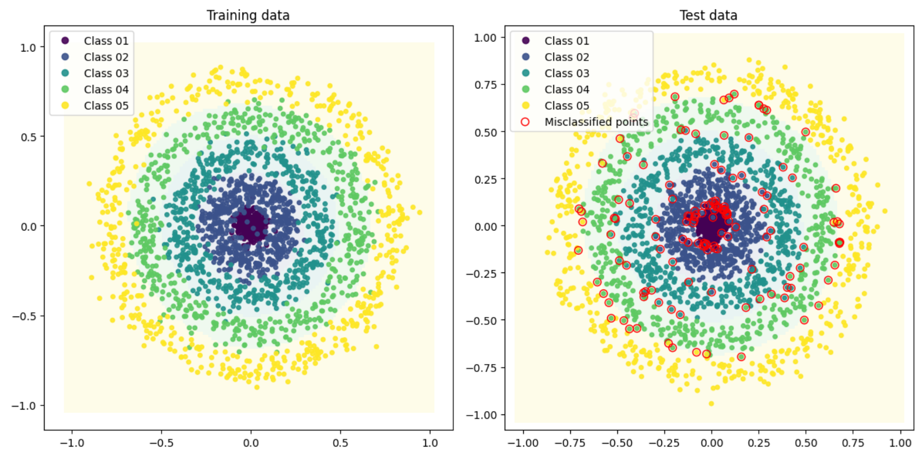

Figure 11: The result of our own made MLP with weights from Keras model

Table 5: Comparison of train and test performance between our MLP and Keras implementation

	Our MLP	**Keras Implementation**
Train Loss	0.0163	0.0163
Train Accuracy (%)	94.92	94.92
Test Loss	0.0182	0.0182
Test Accuracy (%)	94.53	94.53

4.6 Task 3

By using the formula in equation 10 we can calculate some of the unlimited number of solutions for this task. In equation 11 and equation 12 we can see two possible solutions to the equation and these fulfill the constrains given from task 3, section 3.4.

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{11}$$

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix}. \tag{12}$$

The results from these calculations can be seen in figure 12 which uses the weights and biases from equation 11 while in figure 13 we use the weights from equation 12. These do have the same decision boundaries but different weights, and they also use two separate randomized dataset with the same specifications.

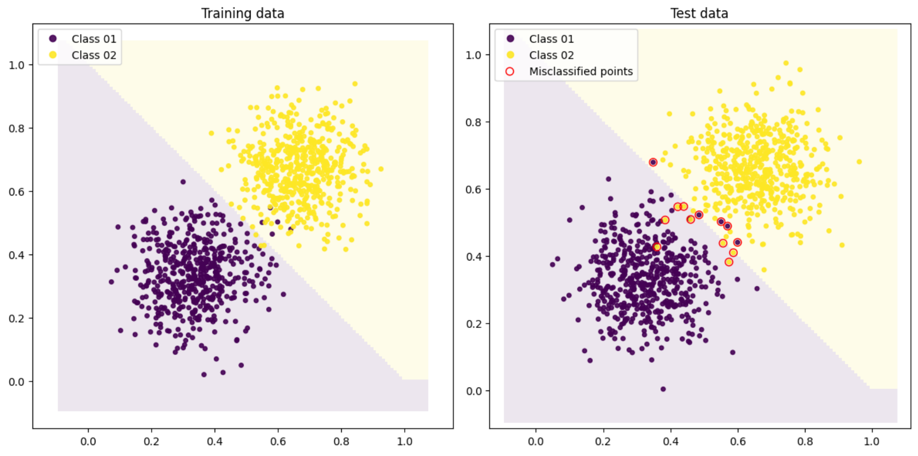

Figure 12: Decision bouderies by using weights from equation 11

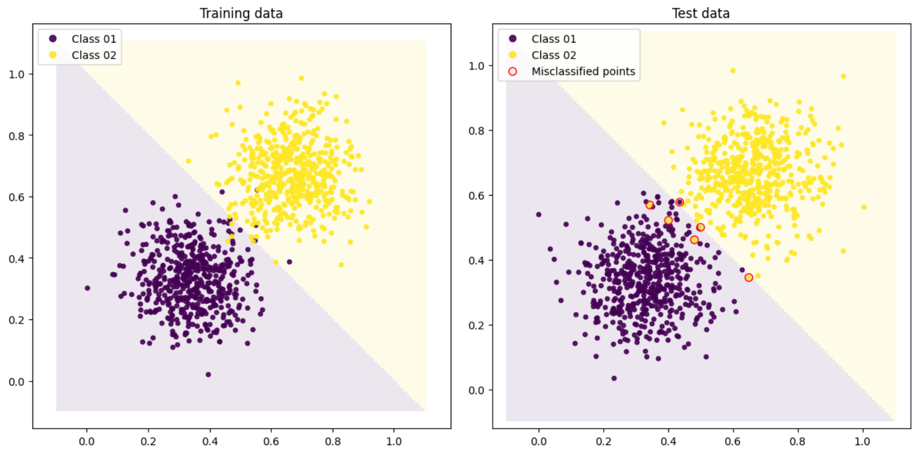

Figure 13: Decision bouderies by using weights from equation 12

5 Conclusion

From this lab, a deeper understanding of how multi-layer perceptrons (MLPs) are constructed, trained, and evaluated. Through the various tasks in this lab, the fundamental concepts such as dataset generation, the impact of different activation functions, weight initialization methods, and optimization algorithms. By playing around and testing with different settings we have gotten a better understanding of how MLPs work and what things can be changed to improve the model.

Understanding how batch size impacts training has also given a better understanding. Smaller batch sizes provided more frequent updates but introduced more noise, while larger batch sizes offered smoother gradients but fewer updates per epoch. These observations highlight the trade-offs involved in hyperparameter tuning.

One of the most important findings was the significance of choosing appropriate activation functions. For instance, while the ReLU activation performed well on complex datasets, the sigmoid activation struggled to fit the non-linear polar dataset and had a problem with its vanishing gradient. Additionally, we observed how weight initialization methods, such as glorot normal and advanced optimizers such as Adam, improved training stability and faster convergence compared to simpler approaches like Stochastic Gradient Descent (SGD).

Finally, implementing an MLP from scratch reinforced the theoretical foundations behind feedforward computations, matrix operations, and the role of activation functions. Comparing the results of my implementation with Keras' implementation demonstrated how consistent these frameworks can be when configured correctly.

Use of generative AI

In this lab, ChatGPT has been used to help with debugging the code for the lab. This was very helpful when considering problems related to the structure of the variables, such as the weights and biases. AI has also been used to implement most of the images, tables and equations in this report to save time. While most of the text has been written by me, AI has been helping with suggesting what to write, the overall structure of the text, and also checking grammar and sentence structures. Lastly, AI has been used to summarize the report and write the whole abstract to create an overall summarizing abstract.

References

[1] TNM112. Lab 1 – the multilayer perceptron, 2023. Lab description – Deep learning for media technology.