# Lab report – TNM112

Deep learning for media technology, Lab 1

Marcus Skoglund (Marsk090)
Lab partner: Anton Martic (Antma001)

Friday 28th November, 2025 (21:52)

**Abstract**

*This report investigates the architecture and behavior of the MultiLayer Perceptron (MLP) through several tasks. The study focuses on how hyperparameters such as batch size, number of epochs, activation functions, initialization methods, and optimizers impact model performance. In addition a custom MLP was implemented to deepen the fundamental understanding of MLPs and the feed-forward function, by manual matrix operations. Finally a weight matrix and bias vector was manually derived to demonstrate how these parameters define decision boundaries.*
*Results show that smaller batch sizes, more epochs, lower learning rates and added momentum improve convergence. The ReLU activation function was also proven to be essential for handling non-linear patterns, whereas the Sigmoid suffered from the vanashing gradient problem. Additionally, by manually specifying weights and biases the class labels can end up inverted.*

## 1 Introduction

This lab report focuses on the multilayer perceptron (MLP) which is a fundamental architecture in deep learning. It will demonstrate how to train MLPs for simple 2-dimensional datasets in Keras, as well as how to implement a MLP with matrix operations in numpy. It will examine subjects like the number of layers, layer size, activation function, optimization algorithm and loss function to better understand how they complement each other.

## 2 Background

The study of the MLP in this lab was executed inside a jupyter notebook environment, which simplified documentation, code and visualisations, and utilized an already provided semi-completed python codebase. This was done using Keras which is an API for Tensorflow.

## 3 Method

The lab was divided into three main parts, task 1.1 - 1.4 involved experimenting with hyperparameters to better understand them. Task 2 was about implement-

ing an MLP almost from scratch, with focus on the feed-forward function, and task 3 analyzed how weights and biases creates a decision boundary [1].

### 3.1 Task 1.1

The first task examines how differnet batch sizes affect model accuracy. A linear dataset with 2 classes and sigma of 0.1 was used and the models hyperparameters were set to 0 hidden layers, softmax activation function, 4 epochs and a stochastic gradient descent (SGD) optimizer with a learning rate of 1. First the confiruration used a batch size of 512 matching the dataset size, while the second configuration used a batch size of 16.

```
data.generate(dataset='linear', N_train=512, N_test=512, K=2, sigma
    =0.1) # Task 1.1

Hyper-parameters
hidden_layers = 0      # The number of hidden layers in the network
    (total number of layers will be L=hidden_layers+1)
layer_width = 0        # The number of neurons in each hidden layer
activation = 'softmax' # Activation function of hidden layers
init = keras.initializers.RandomNormal(mean=0.1, stddev=0.1) #
    Initialization method (starting point for the optimization)
epochs = 4             # Number of epochs for the training
batch_size = 16       # Batch size to use in training
loss = keras.losses.MeanSquaredError() # Loss function
opt = keras.optimizers.SGD(learning_rate=1, momentum=0.0) #
    Optimizer
```

### 3.2 Task 1.2

The second task changed the dataset from the linear one to a polar dataset, also with 2 classes and a sigma of 0.1. The network was updated to have 1 hidden layer with 5 neurons and a total of 20 epochs, while the learning rate stayed at 1 and batch size at 16. The task here was to compare three different activation functions: linear, Sigmoid and ReLU.

### 3.3 Task 1.3

Next the polar dataset was changed from 2 to 5 classes and sigma was set to 0.05. The network was configured with 10 hidden layers, each consisting of 50 neurons and a ReLU activation function. The task was to experiment with the hyperparameters, mean and standard deviation of the initialization, batch size and epochs, learning rate of SGD and to add a momentum and exponential decay to the SGD. This was done in order to improve the accuracy and minimize the loss.

### 3.4 Task 1.4

The last task of part 1 was to change the initialization of the previously well optimized task 1.3 to “glorot normal” and change the optimizer to Adam.

### 3.5 Task 2

The second part of the lab was to implement ones own MLP in the "mlp.py" file. First the different activation functions were created as followed.

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

After that the setup model was implemented where the number of hidden layers was calculated by taking the length of the incoming list of weight matrices and subtracting 1. Then the total number of weights in the model (both weight matrices and bias vectors) was calculated by iterating through them both and adding their sum of elements together.

The next step was to implement the feed-forward of information between the layers, described with:

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

But first the output matrix was initialized as followed:

```
y = np.zeros((x.shape[0], self.dataset.K))
```

Where "x.shape[0]" is the number of rows in the input data and corresponds to the number of data points. While "self.dataset.K" is the number of output classes the MLP is predicting.

Next was to create the feed-forward function. The first step was to itterate throught the data points and transpose each matrix from a row matrix to a column matrix. Next the MLP operations had to be performed for each hidden layer. This was done by itterating through the hidden layers and for each layer multiply the weight matrix with the previous layers output and adding the bias. Then the activation function was applied. The final layer of the feed-forward function used a softmax activation function, lastsly before returning the output (y) it was transposed to a column vector. The feed-forward operation code is shown below.

```
def feedforward(self, x # Input data points):
        # specify a matrix for storing output values
        y = np.zeros((x.shape[0], self.dataset.K)) # (number of
    input samples, number of output classes)
        # implement the feed-forward layer operations

        # 1. Specify a loop over all the datapoints
        for n in range(x.shape[0]):
            # 2. Specify the input layer (2x1 matrix)
            h = x[n,:].reshape(2,1) # row vector to column vector

            # 3. For each hidden layer, perform the MLP operations
```

```
            #     - multiply weight matrix and output from previous layer
            #     - add bias vector
            #     - apply activation function
            for l in range(self.hidden_layers):
                z = self.W[l] @ h + self.b[l]  # z = weightsen (w) in i layer h-1 + bias (matrix multiplication so all elements are multiplied simultaneously)
                h  = activation(z, self.activation)

            # 4. Specify the final layer, with 'softmax' activation
            z = self.W[-1] @ h + self.b[-1] # take last weight and bias
            h = activation(z,'softmax')

            y[n,:] = h.flatten() # creates a column vector

        return y
```

Lastly the evaluation function was created to calculate the training loss and accuracy as well as the test loss and accuracy. In order to compare the performance of the MLP to the Keras MLP. The evaluate code is shown below.

```
def evaluate(self):

        print('Model performance:')
        # formulate the training loss and accuracy of the MLP
        # Assume the mean squared error loss
        y_train_pre = self.feedforward(self.dataset.x_train)
        # Hint: For calculating accuracy, use np.argmax to get predicted class
        train_loss = np.mean((self.dataset.y_train_oh - y_train_pre)2)
        train_acc = np.mean(np.argmax(y_train_pre,1) == self.dataset.y_train)
        print("\tTrain loss:     %0.4f"%train_loss)
        print("\tTrain accuracy: %0.2f"%train_acc)
        # formulate the test loss and accuracy of the MLP
        y_test_pre = self.feedforward(self.dataset.x_test)
        test_loss = np.mean((self.dataset.y_test_oh - y_test_pre)2)

        test_acc = np.mean(np.argmax(y_test_pre, 1) == self.dataset.y_test)
        print("\tTest loss:      %0.4f"%test_loss)
        print("\tTest accuracy:  %0.2f"%test_acc)
```

### 3.6 Task 3

The final part of the lab was to manually derive a weight matrix and a bias vector to specify a model that draws a decision boundary at:

$$x_2 = 1 - x_1. \tag{2}$$

First the dataset was changed to linear with 2 classes, no hidden layers and a softmax activation function. Next the weight matrix was specified to a 2x2 and the bias vector to a 2x1. They were derived from the decision boundary equation together with the equation for a 2-class problem with softmax, shown below.

$$\begin{aligned} z_1 &= w_{11} \cdot x_1 + w_{12} \cdot x_2 + b_1 \\ z_2 &= w_{21} \cdot x_1 + w_{22} \cdot x_2 + b_2 \end{aligned} \tag{3}$$

The step by step solution is shown below:

$$x_2 = 1 - x_1 \Leftrightarrow x_1 + x_2 - 1 = 0. \tag{4}$$

A decision boundary occurs when:

$$z_1 = z_2 \tag{5}$$

This give:

$$\begin{aligned} w_{11} \cdot x_1 + w_{12} \cdot x_2 + b_1 = w_{21} \cdot x_1 + w_{22} \cdot x_2 + b_2 \\ \Leftrightarrow \\ (w_{11} - w_{21}) \cdot x_1 + (w_{12} - w_{22}) \cdot x_2 + b_1 - b_2 = 0 \end{aligned} \tag{6}$$

This should equal (4), which gives:

$$(w_{11} - w_{21}) = 1 \Leftrightarrow w_{11} = 1 \quad \& \quad w_{21} = 0, \tag{7}$$

$$(w_{12} - w_{22}) = 1 \Leftrightarrow w_{12} = 1 \quad \& \quad w_{22} = 0, \tag{8}$$

$$b_1 - b_2 = -1 \Leftrightarrow b_1 = 0 \quad \& \quad b_2 = 1 \tag{9}$$

as one of the solutions.
From the equations 7, 8 and 9 the weight matrix and bias vector was created, as shown below.

$$\begin{aligned} W = \begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{pmatrix} \implies \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \\ b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \implies \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \end{aligned} \tag{10}$$

Upon initial testing, the predicted class labels were found to be inverted compared to the desired configuration. Therefore the simplest change was made to the derived weights and bias, by swapping the rows it correctly switched the predicted class labels. The full code is shown below.

```
import numpy as np
import mlp
importlib.reload(mlp)

# Get the weight matrices and biases of the trained Keras model
W, b = model_k.get_weights()

Task 3: specify a weight matrix and a bias vector
#W = [np.array([[1,1],[0,0]])]
#b = [np.array([[0],[1]])]
# swapp
W = [np.array([[0,0],[1,1]])]
b = [np.array([[1],[0]])]

# This is our implementation of an MLP, which we set to use the
    dataset we generated
model = mlp.MLP(data)

# Assign the weights and biases to the MLP and specify the
    activation function
model.setup_model(W, b, activation=activation)

# Evaluate the model (accuracy on the training and test data)
model.evaluate()

# Plot the dataset with decision boundaries generated by our MLP
data.plot_classifier(model)
```

# 4 Results

Below, the result of each task will be discussed.

## 4.1 Task 1.1

The model using the smaller batch size of 16 had better accuracy than the one using 512. This can be explained by the amount of times each model performs stochastic gradient descent. The model performs SGD for each batch per epoch, meaning that the smaller batch of 16 performs 128 SGD since it has $512/16 = 32$ batches and 4 epochs. While the larger batch size of 512 only performs 4, $512/512 = 1$. This means that the model's loss function becomes more optimized and reduced for a batch size of 16.

Since the data was linear and the output after every neuron was linear, the final output was like adding linear to linear meaning it was possible to do the classification without a non-linear activation function. When looking at the data it was clear that a linear line could separate the 2 classes with a good accuracy, stating that the data was linear separable.

## 4.2 Task 1.2

When looking at the linear activation function the accuracy was around 50%, meaning that the output is not any better than just random guessing. This is the result of using a non linear input for a linear activation function, meaning that the data can not be divide with a single separation line.

The polar dataset contained negative values which made the Sigmoid function ineffective due to the vanishing gradient problem. Because the derivative of the Sigmoid function is small, the gradients shrink rapidly as they are multiplied during backpropagation, becoming smaller and smaller and eventually close to 0. This resulted in the network failing to learn anything new. ReLU on the other hand outperformed Sigmoid because it does not suffer from vanishing gradient. ReLU assigns a gradient of 1 to all positive values and clamps negative values to 0, ensuring that the gradient remains.

## 4.3 Task 1.3

The final combination of hyperparameters was:

- mean = 0.01
- stddev = 0.1
- epochs = 100
- batch size = 16
- learning rate = 0.06
- momentum = 0.8.

And for the added exponential decay:

- initial learning rate = 0.1
- decay steps=5000
- decay rate=0.96.

The result was: Test loss: 0.0152, Test accuracy: 95.08, which is visualised in Figure 1.

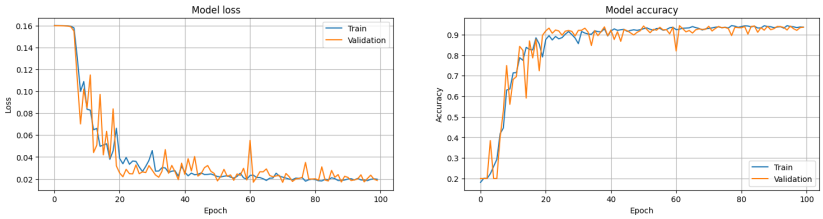



Figure 1: Results for task 1.3.

A lower learning rate and a higher momentum of 0.8 made the convergence smoother, with less oscillation and allowed the optimizer to pass over small "bumps" or local minimas. Whereas a high learning rate and no mementum made the model oscillate and become unstable. A larger number of epochs helped the model to fully converge but it increased the run time. A smaller initialization mean worked better, by reducing the chance of saturation. The addition of exponential decay made the model fine tune itself by starting of with a fast initial movement and towards the end become more precise.

## 4.4 Task 1.4

By replacing the "RandomNormal" initializer to "glorot_normal" and swapping the SGD optimizer with Adam resulted in a small improvement in accuracy, however the convergence became significantly more stable, as shown in Table 1 and Figure 2 below.

| Optimizer | Initializer 2 | Test loss | Test Accuracy(%) |
|---|---|---|---|
| SGD | RandomNormal | 0.0152 | 95.08 |
| Adam | glorot_normal | 0.0153 | 95.31 |

Table 1: Comparisson of results from Task 1.3 & Task 1.4.

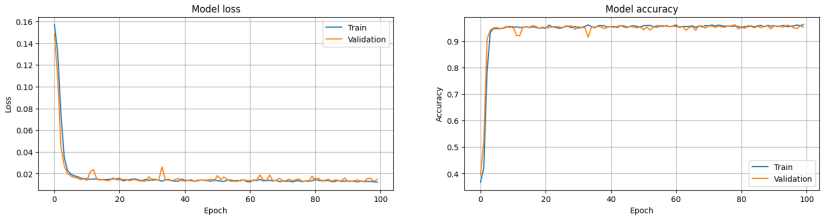



Figure 2: Results for task 1.4.

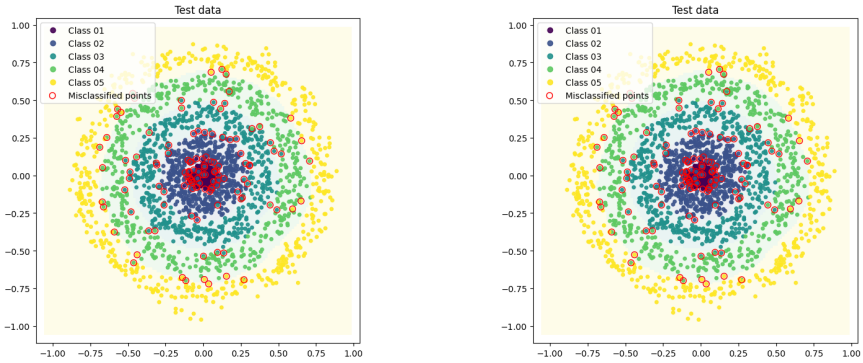
## 4.5 Task 2

The result for task 2 together with task 1.4 is shown in Table 2 below.

| Initializer/optimizer | Test loss | Test Accuracy(%) |
|---|---|---|
| glorot_normal/Adam | 0.0153 | 95.31 |
| Custom MLP | 0.0153 | 95.00 |

Table 2: Results from Task 2.

When looking at the decision boundaries of the Adam optimizer and the custom one, there is no difference as shown in Figure 3 below.




(a) Adam. (b) Custom MLP.

Figure 3: Decision boundaries.

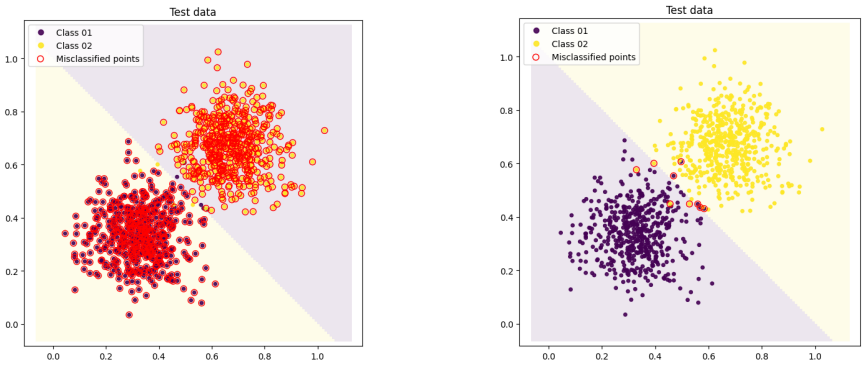
This indicates that the custom MLP is implemented correctly.

## 4.6 Task 3

The initial outcome from implementing Task 3 showed that the output was inverted, resulting in a test accuracy of only 1%. However this result still confirmed that the decision boundary itself was accurately implemented and that it clearly separated the 2 classes. After swapping the rows of both the weight matrix and the bias vector, the class assignment was corrected, immediately improving the accuracy to 99%. The final results and the generated decision boundary can be seen in Table 3 and Figure 4 below.

| Labels | Test loss | Test Accuracy(%) |
|---|---|---|
| Wrong | 0.3406 | 1.00 |
| Correct | 0.1754 | 99.00 |

Table 3: Results from Task 3.




(a) Wrong class label.

(b) Correct class label.

Figure 4: Comparisson between class labels for custom MLP with manual wheigh matrix and bias vector.

# 5 Conclusion

This lab with the different tasks provided has given a deeper understanding of how Multilayer perceptrons (MLP) operate and how tuning the hyperparameters affects the model. By choosing a low learning rate and higher momentum, combined with a small batch size and an exponential decay the model can achieve a more stable and accurate convergence. When pairing this with an activation function like ReLU the model becomes capable of learning more complex patterns for decision boudaries.

The implementation of the custom MLP gave a greater understanding of how each hidden layer is dependent on the previous one and a better overall understanding of the feed-forward function. Additionally by manually specifying the weight matrix and bias vector, it was clear that the class labels had to be tweak manually compared to the Keras MLP where this was handled automatically.

Finally this lab has provided a better understanding of how neural networks work and how they differ in both architectural and technical aspects from traditional machine learning (ML).

## Use of generative AI

During the lab generative AI has been used to understand the vanashing gradient problem as well as the manual derivation of the weight matrix and bias vector. For the report, generative AI was used to assist with sentence structure and phrasing. It was also used to help write the abstract and introduction.

# References

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