Lab report – TNM112

Deep learning for media technology, Lab 1

Emil Djurson (emidj236)
Lab partner: Astrid Helzén (asthe501)

Friday 28th November, 2025 (11:30)

Abstract

This lab explores different design choices in multilayer perceptrons (MLPs) influence their behavior and performance. Through a series of experiments conducted in Python using Jupyter Notebook, it first examines how individual hyper parameters can effect the training on both linear and non-linear datasets. The results highlight clear trends including importance of stable initialization, suitable activation functions and properly tuned optimization settings.

In the second part of the lab a custom MLP is implemented from a given skeleton code. Providing a deeper insight into the fundamentals of an MLP and especially the feedforward pass. By then manually specifying weight matrices and bias vectors in the last part to show how parameters directly shape the decision boundary. The lab highlights how both Keras models and manual implementation behave under different settings and brings up the importance of careful hyper parameter selection.

1 Introduction

Multilayer perceptrons (MLPs) are most commonly used for classification and regression tasks. They are a fundamental type of neural network that consists of an input layer, one or more hidden layers and an output layer. Each neutron applies a weighted sum of its inputs followed by an activation function, which enables the network to model complex and non-linear relationships in the data. This lab focuses on exploring how different design choices such as activation functions, number of hidden layers, weight initialization and hyper parameters effect the behavior and performance of an MLP. The work combines experiments with a prebuilt MLP with the Keras API and a custom MLP implementation including manually defined weight matrices and bias vectors, to provide insight how network parameters shape decision boundaries and influence learning.

2 Background

In order to experiment with the MLPs, the experiments were carried out in Python using a Jupyter Notebook environment. All models were implemented and trained using the Keras API that is built on top of TensorFlow. Other

key libraries used were, NumPy for numerical computations and Matplotlib for plotting results.

3 Method

To systematically investigate the behavior of the MLP, the followings tasks were done in subsequent order to explore the effects of different hyper parameters and configurations.

3.1 Task 1

The first set of tasks were to experiment and understand the hyper parameters of a MLP and to examine how each parameter affects model performance. For this Keras deep learning API was used with the following base configuration, seen in listing 1.

```
hidden_layers = 0
layer_width = 5
activation = 'linear'
init = keras.initializers.RandomNormal(mean=0.1, stddev=0.1)
epochs = 4
batch_size = 512
loss = keras.losses.MeanSquaredError()
opt = keras.optimizers.SGD(learning_rate=1.0, momentum=0.0)
```

Listing 1: Base configuration of the Keras MLP.

3.1.1 Task 1.1

The first experiment focused on understanding the effect of batch size on model performance. With the given data generator python script a linear dataset with 512 training and test samples divided into two classes were generated. The model was first trained on the base configuration, then the batch size was changed from 512 to 16, the performance of the two diffrent configurations were then evaluated.

3.1.2 Task 1.2

To investigate how different activation functions behave on non-linear data, the dataset was switched from the linearly separable case in task 1.1 to a polar configuration. The following code seen in listing 2 was implemented to generate the data.

```
data.generate(dataset='polar', N_train=512, N_test=512, K=2, sigma
    =0.1)
```

Listing 2: Code to generate a non-linear dataset with 2 classes.

The network was configured with one hidden layer containing five neutrons and trained for 20 epochs using SGD with a learning rate of 1.0 and a batch size of 16. This configuration was then tested with linear, sigmoid and ReLu as activation functions. The updated parameters are shown in listing 3 below.

```
activation = 'relu' # 'sigmoid' or 'linear'
hidden_layers = 1
layer_width = 5
epochs = 20
```

Listing 3: Changes made to the base configuration.

3.1.3 Task 1.3

A new dataset was generated with more classes and reduced standard deviation of the sample points. This was done with the code from listing 4.

```
data.generate(dataset='polar', N_train=512, N_test=512, K=5, sigma
    =0.05)
```

Listing 4: Code to generate a non-linear dataset with 5 classes.

The following lines of code were also changed in the base configuration, see listing 5.

```
hidden_layers = 10
layer_width = 50
activation = 'relu'
```

Listing 5: Base configuration changes made in task 1.3.

The experiment then examined how different hyper parameters affected performance. Specifically variations of the initialization mean and standard deviation, the learning rate and momentum in SGD as well as the number of training epochs and batch sizes were systematically tested. When a final configuration was decided, exponential decay was also added.

3.1.4 Task 1.4

For this task the initialization was changed to the Glorot Normal method and the Adam optimizer was used instead. Listing 6 displays changes made in the code, no other changes were made.

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

Listing 6: Changes made to the configuration for task 1.4.

3.2 Task 2

The second set of tasks were to implement a custom MLP using the skeleton code given in the mlp.py file. This included implementing functionality for setting up a model, different activation functions, a feedforward function and an evaluate function.

3.2.1 Model setup function

For the model initialization function, in order to calculate the number of hidden layers the length of W was utilized. Since each entry in the W array corresponds to one layers weight matrix, the number of hidden layers can be calculated by

$\text{len}(W) - 1$. The total number of trainable parameters could then be computed by summing the number of elements in a weight matrices and bias vectors. See listing 7 for the code implementation.

```
self.hidden_layers = len(W) - 1
self.N = sum(W_L.size for W_L in W) + sum(b_L.size for b_L in b)
```

Listing 7: Changes made to the skeleton code for the initilization.

3.2.2 Feedforward function

The feedforward function implements the forward propagation through the network and is based on the standard layer transformation, shown below is the equation.

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

When implementing equation(1) in code, the function begins by allocating space for a $n \times k$ matrix where n is the number of data points and k is the number of classes. A for loop is then declared to loop over all of the data points where each input vector is reshaped into a 2×1 column vector. The function then iterates over each hidden layer calculating the new intermediate value by multiplying the current vector h with the corresponding weight matrix and adding the layer bias. The result is then processed through the chosen activation function.

After the final hidden layer the output performs the same weight-bias computation but applies a softmax activation to convert the result into class probabilities. This vector is flattened and stored in the correct row of y. Listing 8 shows the implementation.

```
def feedforward( self: 'MLP', x):
    y = np.zeros((len(x), self.dataset.K))

    for n in range(len(x)):
        h = x[n].reshape(2, 1)

        for l in range(self.hidden_layers):
            z = self.W[l] @ h + self.b[l]
            h = activation(z, self.activation)

        z_out = self.W[-1] @ h + self.b[-1]
        y[n, :] = activation(z_out, 'softmax').flatten()

    return y
```

Listing 8: Custom feedforward function.

3.2.3 Activation functions

The network supports four different activation functions which are sigmoid, softmax and ReLu. Each activation was implemented based on their mathematical definition, see listing 9 for the implementation.

```
def activation(x, activation):
    if activation == 'linear':
        return x
```

```
    elif activation == 'relu':
        return np.maximum(0, x)

    elif activation == 'sigmoid':
        return 1 / (1 + np.exp(-x))

    elif activation == 'softmax':
        exps = np.exp(x - np.max(x))
        return exps / np.sum(exps)

    else:
        raise Exception("Activation function is not valid",
    activation)
```

Listing 9: Activation functions.

3.2.4 Evaluate function

In order to examine the MLP performance the evaluate function was implemented by applying the feedforward pass to both the training and test datasets to compute the predictions. The loss is calculated as the mean square error between the predicted outputs and the target vectors. See listing 10 for the code implementation.

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

    output_train = self.feedforward(self.dataset.x_train)

    train_loss = np.mean((output_train - self.dataset.y_train_oh)
    **2)
    train_acc = np.mean(np.argmax(output_train, axis=1) == self.
    dataset.y_train) * 100

    print("\tTrain loss:     %0.4f"%train_loss)
    print("\tTrain accuracy: %0.2f"%train_acc)

    output_test = self.feedforward(self.dataset.x_test)

    test_loss = np.mean((output_test - self.dataset.y_test_oh)**2)
    test_acc = np.mean(np.argmax(output_test, axis=1) == self.
    dataset.y_test) * 100

    print("\tTest loss:      %0.4f"%test_loss)
    print("\tTest accuracy:  %0.2f"%test_acc)
```

Listing 10: Evaluation function.

3.3 Task 3

The goal of the final task was to no longer use the weights from the Keras MLP, but instead manually define a 2×2 weight matrix and 2×1 bias vector. The model was configured with no hidden layers, and the output used a softmax activation. The weights and biases were chosen to produce a decision boundary according to equation (2):

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

With no hidden layers, the network output is a linear combination of the inputs and biases:

$$\begin{aligned} z_1 &= w_{1,1}x_1 + w_{1,2}x_2 + b_1, \\ z_2 &= w_{2,1}x_1 + w_{2,2}x_2 + b_2. \end{aligned} \tag{3}$$

The decision boundary occurs where the two outputs are equal:

$$z_1 = z_2 \implies (w_{1,1} - w_{2,1})x_1 + (w_{1,2} - w_{2,2})x_2 + (b_1 - b_2) = 0. \tag{4}$$

To match the target boundary in equation (2), the weight and bias vector were chosen as:

$$W = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} -1 \\ 0 \end{pmatrix}. \tag{5}$$

Listing 11 shows the code implementation of the weight matrix and bias vector.

```
W = [np.array([[1, 1],
               [0, 0]])]

b = [np.array([[-1],
               [0]])]
```

Listing 11: Weight matrix and bias vector implementation.

4 Results

This section will present the outcomes of the experiments conducted as well as the parameters used in task 1.3 mentioned in section 3.1.3.

4.1 Task 1.1

With a batch size of 16, the model performance significantly improved. A lower batch size resulted in both a lower test loss and higher test accuracy, as seen in table 1.

Batch size	Test loss average	Test accuracy average (%)
512	0.2383	64.61
16	0.0138	98.71

Table 1: Average performance comparison on five test for different batch sizes on the linear dataset.

The performance difference is explained by the number of SGD updates per epoch. With 512 samples in the dataset, a batch size of 16 produces 32 weight updates per epoch, whereas a batch size of 512 results in only a single update.

More frequent updates allow the model to adjust the parameters gradually and converge more efficiently. Since the dataset is linearly separable, even this one-layer softmax model is sufficient to learn the correct linear boundary, and the smaller batch size enables the optimization process to reach that solution more reliably.

4.2 Task 1.2

The polar dataset used in task 1.2 is non-linear, which makes a linear activation function ineffective or unpredictable, as it can only form a decision boundary in a straight line. This resulted in the accuracy ranging from 35.55% up to 69.40% as shown in table 2.

Test number	Test accuracy (%)
1	50.00
2	45.51
3	50.00
4	69.04
5	35.55

Table 2: 5 tests with linear as the activation function.

When switching to a sigmoid activation function, the performance was not improved. Although the sigmoid function is non-linear, the vanishing gradient problem occurs. This is due to the sigmoid function approaching 0 or 1 for almost all input values as seen in figure 1. This makes the gradient for the sigmoid function have a value close to zero. When this is then feed through the network, each layer is multiplied with this small value making them approach zero as well, which significantly slows down or stops learning.

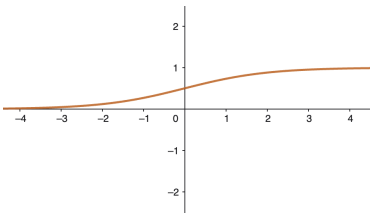

Figure 1: Plot of the sigmoid function.

Using the ReLu function as activation function instead, the performance of the model was significantly improved. This can be explained by how the ReLu function is defined, when a positive value is feed through, the gradient is always one since the output is linear for values greater than zero. On the other hand, if a negative value is feed through the function, the result will always be zero making the gradient also be zero, this is visualized in figure 2. When this is then sent through all of the layers in the network, the network can efficiently separate the data.

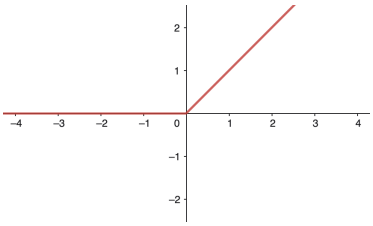

Figure 2: Plot of the ReLu function.

4.3 Task 1.3

From testing different values on each hyperparameter, the configuration that yielded the best result most consistently was established as shown in listing 12. The configuration resulted in a test accuracy of 94.30% with a test loss of 0.0175.

```
hidden_layers = 10
layer_width = 50
activation = 'relu'
init = keras.initializers.RandomNormal(mean=0.01, stddev=0.3)
epochs = 100
batch_size = 16
loss = keras.losses.MeanSquaredError()
opt = keras.optimizers.SGD(learning_rate=0.1, momentum=0.5)
```

Listing 12: Final configuration for task 1.3.

When exponential decay was added, it was concluded that the configuration shown in listing 13 yielded the best result. The exponential decay gave a marginally better test accuracy and a slightly lower test cost as seen in table 3.

```
lr_schedule = keras.optimizers.schedules.ExponentialDecay(0.1,
    decay_steps=500, decay_rate=0.92, staircase=True)
opt = keras.optimizers.SGD(learning_rate=lr_schedule, momentum=0.5)
```

Listing 13: Exponential decay configuration.

While tuning the model some trends became clear. High learning rate or too large initial weights often caused unstable updates, while low learning rates without momentum led to slow or premature convergence. The most reliable setups combined a lower learning rate with strong momentum and small initial weights.

Model	Test loss	Test accuracy (%)
Base configuration	0.3200	20.00
Final configuration	0.0175	94.30
Final configuration with exponential decay	0.0130	95.90

Table 3: Performance comparison between the different model configurations.

4.4 Task 1.4

Switching to the Glorot Normal initialization together with the Adam optimizer resulted in a small but clear improvement compared to the configuration used in task 1.3, as shown in table 4.

Model	Test loss	Test accuracy (%)
SGD & RandomNormal	0.0175	94.30
Adam & Glorot Normal	0.0162	95.23

Table 4: Performance comparison between optimizers and initialization.

4.5 Task 2

The custom MLP that was implemented performed the same as the Keras MLP when classifying a polar dataset as shown in figure 3 and figure 4.

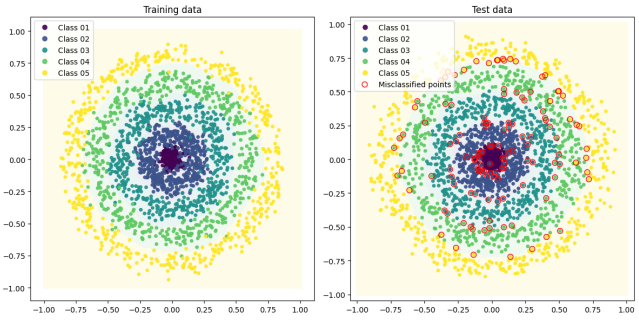

Figure 3: Plot of Keras MLP classification of points.

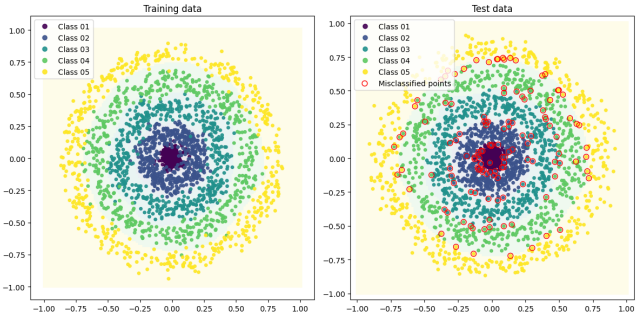

Figure 4: Plot of custom MLP classification of points.

4.6 Task 3

With the initial weight and biases from equation 5 the MLP gave the following result shown in figure 5. This resulted in a test accuracy of 0.59% and a test loss of 0.3414. Although the decision boundary was correct, it was inverted for the classification, resulting in a poor performance.

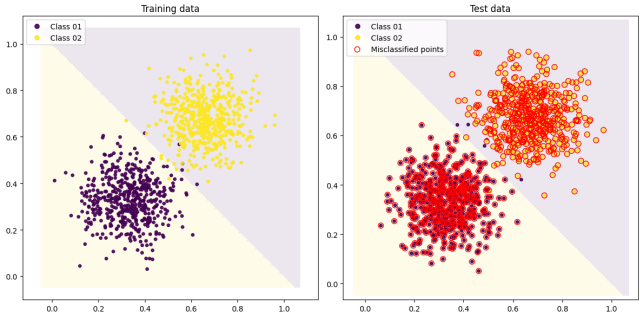

Figure 5: Plot of original weight and biases.

Equation 4 shows that there are infinite number of solutions to create the desired decision boundary. By then flipping the weights and biases to the ones shown in equation 6.

$$W = \begin{pmatrix} -1 & -1 \\ 0 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \tag{6}$$

Resulted in the correct label assignment and yielded a test accuracy of 99.41% and a test loss of 0.1748. Classification of the dataset with the new weights and biases is shown in figure 6.

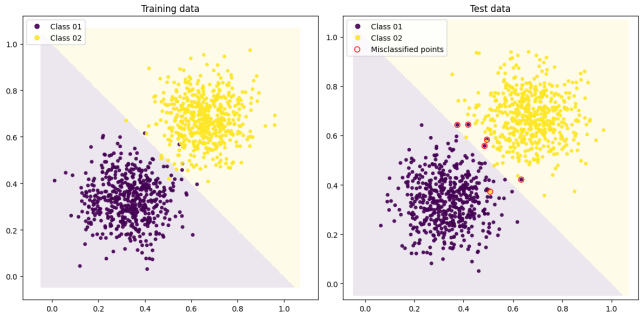

Figure 6: Plot of flipped weights and biases.

5 Conclusion

The tasks shows how different design choices can shape the behavior and performance of an MLP. When experimenting with different hyper parameters showed how sensitive the model is to things such as learning rate, momentum and batch sizes but also training epochs and optimizers. That there isn't a specific configuration that works for all data, but that the hyper parameters have to be tweaked according to the properties of the dataset.

By going from the Keras MLP to implementing a custom MLP we gained a better understanding for the fundamentals of an basic MLP. The calculations that are being made, especially for how feedforward pass works. We also gained

a deeper understanding in task 3 for how weight matrices and bias vectors shape the decision boundary.

Use of generative AI

Generative AI has been used to assist in writing the abstract based on the method, results and conclusion of this report. It was also used to better understand the vanishing gradient problem and to explore approaches or manually deriving weights and biases for task 3.

References