Lab report – TNM112

Deep learning for media technology, Lab 1

Oliver Lundin (olilu316)

Lab partner: Filip Bägén (filma379)

Wednesday 22nd November, 2023 (11:49)

Abstract

This lab explores Multilayer Perceptrons (MLP) using Keras and a custom-built model. It investigates dataset configurations, network settings, and their impact on classification accuracy. Results highlight the influence of batch size, activation functions, and hyperparameters on model performance. Comparisons between different initializers and optimizers reveal subtle variations. Overall, the study emphasizes the critical role of network configurations in achieving better accuracy in classification tasks.

1 Introduction

A multi-layer perceptron (MLP) is a fundamental type of artificial neural network known for its ability to solve complex problems in machine learning and pattern recognition. Composed of multiple layers of interconnected nodes (neurons), an MLP consists of an input layer, one or more hidden layers, and an output layer. Each neuron in the network processes information by applying a weighted sum of inputs, followed by an activation function that introduces non-linearities, enabling the network to learn and represent intricate relationships within the data. The aim of this lab is to get hands on with building and configuring a feedforward network, first using Keras and then by implementing our own MLP using python and the numpy library.

2 Background

The MLP layers can be implemented using a equation of how information is fed forwards between layers. This equation along with different optimization techniques such as altering the number of layers, batch size and epochs while also changing the activation function, are techniques used in this lab [1].

3 Method

To experiment with the MLP we use the Python API Keras, which is a deep-learning API built on top of tensorflow. Our developement enviroment is a jupyter notebook which has different codeblocks that can be compiled individually.

3.1 Task 1

The aim of this task was to understand the connection between datasets and how they will be perform in combination with the network along with what settings for the network suit which dataset. Below are three different dataset configurations used and fed to the network.

```
data.generate(dataset='linear', N_train=512, N_test=512, K=2, sigma
    =0.1) #Task 1.1
data.generate(dataset='polar', N_train=512, N_test=512, K=2, sigma
    =0.1)  # Task 1.2
data.generate(dataset='polar', N_train=512, N_test=512, K=5, sigma
    =0.05) # Task 1.3
```

3.1.1 Task 1.1

For the first task linear data was generated and fed to a network without hidden layers and trained for 4 epochs with a learning rate of 1.0 and first a batch size of 512 and then compared to a network with the same settings but a batch size of 16.

3.1.2 Task 1.2

In this task the data was configured to be polar using the same amount of training points and classes as in the previous task. This time the network was configured to have one layer with 5 neurons, trained for 20 epochs with the same learning rate of 1.0 and a batch size of 16. The 3 different activation functions were applied with the same settings, linear, sigmoid and relu.

```
hidden_layers = 1     # Number of hidden layers
layer_width = 5         # Number of neurons in each layer
activation = 'linear' # Sigmoid, Relu
epochs = 20            # Number of epochs for the training
batch_size = 16       # Batch size to use in training
```

3.1.3 Task 1.3

This time the same datatype as in task 1.2 was used but with 5 classes and sigma=0.05. The network was configured to use 10 hidden layers with 50 neurons each and relu activation. Then a selection of hyper parameters were changed to try to achieve a better accuracy, such as the standard deviation and mean, the learning rate and momentum of the SGD. The most important one being standard deviation.

```
hidden_layers = 10    # Number of hidden layers
layer_width = 50         # Number of neurons in each layer
activation = 'relu' # Sigmoid, Relu
epochs = 20            # Number of epochs for the training
batch_size = 16       # Batch size to use in training

keras.initializers.RandomNormal(mean=0.1, stddev=0.1) #Init method
keras.optimizers.SGD(learning_rate=1.0, momentum=0.0) # Optimizer
```

3.1.4 Task 1.4

In this part we changed the initialization to glorot.normal and optimizer to Adam. Instead of RandomNormal and Stochastic Gradient Descent.

```
init = keras.initializers.glorot_normal() # Initialization method
opt = keras.optimizers.Adam(learning_rate=1.0) # Optimizer
```

3.2 Task 2

For task 2 we implemented our own MLP without the help of Keras. First an activation function was implemented to be able to choose from different activation methods.

```
def activation(x, activation):
    if activation == 'linear':
        return x
    elif activation == 'sigmoid':
        return 1 / (1 + math.exp(-x))
    elif activation == 'relu':
        return np.maximum(0.0, x)
    elif activation == 'softmax':
        return (np.exp(x)/np.exp(x).sum())
    else:
        raise Exception("Activation function is not valid",
    activation)
```

Then some calculations were needed to setup the model.

```
 W,  # List of weight matrices
 b,  # List of bias vectors
self.hidden_layers = len(W)-1
```

The calculation above is performed to get the number of hidden layers in the network. Since we have a list of the weight matrices, the length of the list - the last output layer will be the number of hidden layers in the network.

```
self.N = sum([w.size for w in W]) + sum([bi.size for bi in b])
```

The above equation is to find the total number of weights in the network. Since we have a list of weight matrices we have to visit every matrix and check it's size and the sum them up with the sum of the size of all the bias vectors.

We then implemented a feedforward algorithm to feed information from the input layer through the hidden layers to the output layer.

```
for i in range(x.shape[0]):

            input = x[i, :]
            input = input[:, np.newaxis]

            for layer in range(self.hidden_layers):
                z = np.dot(self.W[layer], input) + self.b[layer]
                input = activation(z, self.activation)

            z = activation(
                np.matmul(self.W[-1], input) + self.b[-1], 'softmax
    ')
            y[i, :] = z[:, 0]
```

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

The loop is implemented using equation 1 [1].

3.3 Task 3

First our own MLP was tested using the similar data as in task 1, linear data with 512 datapoints with 2 classes, using the same weights and biases as in the keras mlp. Then we manually implemented our own weight and bias matrices:

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

Using the linear equation:

$$x2 = 1 - x1 \tag{2}$$

4 Results

4.1 Task 1

Batch Size	Accuracy	Loss
512	71.78	0.2119
16	99.12	0.0425

Table 1: Results for Task 1.1

The results in 1 show that the model using a batch size of 16 is better at classifying since the accuracy is noticeably higher. This is expected since a batch size of 16 means that we capture the datapoints more individually and are able to catch nuances of the data. Using 16 as batch size with 512 datapoints means $512/16 = 32$ iterations of SGD. It is possible to do this classification with a linear activation function since this model is a one-layer network and the dataset can be separated linearly.

Activation	Accuracy	Loss
Linear	56.45	0.2532
Sigmoid	50.00	0.2571
Relu	99.02	0.0096

Table 2: Results for Task 1.2

In the table of results for task 1.2 we observe that using both linear or sigmoid as activation function the accuracy is poor compared to using relu as an activation function. This is because we have introduced a hidden layer in our model and are no longer using linear data.

For task 1.3 the hyperparameters we changed were: mean=0.02, learning rate=0.1, momentum=0.5 and 260 epochs. This resulted in a accuracy of 0.9488 and a loss of 0.0337, as seen in table 3.

Methods	Accuracy	Loss
RandomNormal and SGD	0.9488	0.0337
glorot.Normal and Adam	0.9863	0.009

Table 3: Results for Task 1.4

Comparing the model with changed hyperparameters to using a different initializer (glorotNormal) and a different optimizer (adam) we can see that the second model performed slightly better.

4.2 Task 2

Model	Accuracy	Loss
kera-smlp	0.99	0.0082
custom-mlp	0.99	0.0052
custom-mlp-manualweights	0.99	0.1750

Table 4: Results for Task 2,3

As seen in table 4 the accuracy for all the models are the same the only small but noticeable in terms of numbers, difference can be observed in the loss of the model using our own MLP with manually derived weights. The weights can also be observed in figure 3.

4.3 Task 3

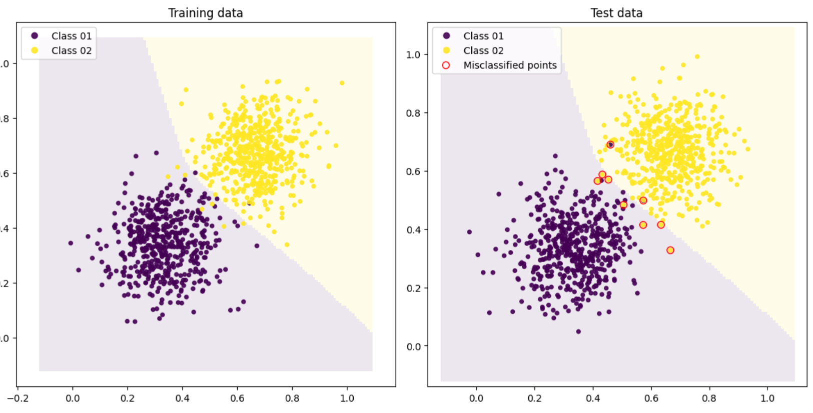

Figure 1: Classification using keras-mlp

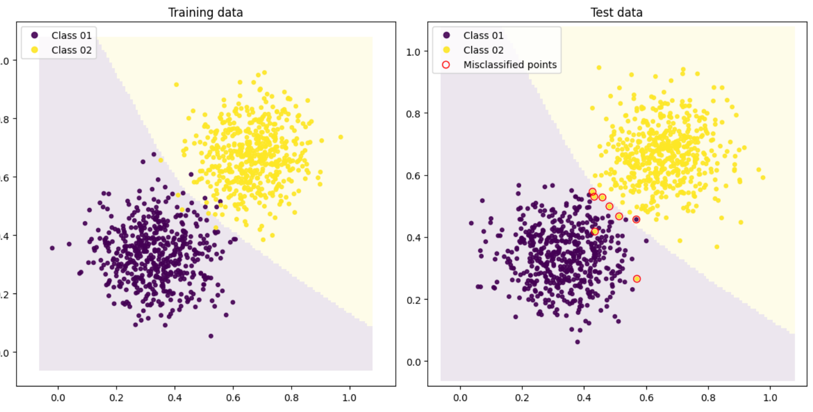

Figure 2: Classification using our own mlp with the same weights as in the keras model

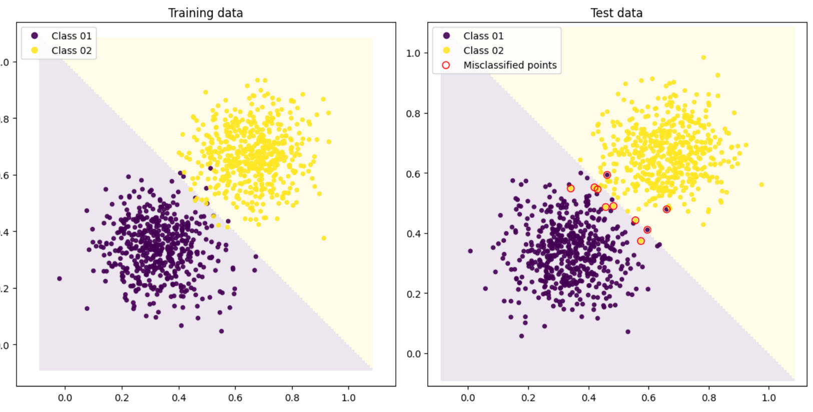

Figure 3: Classification using our own mlp with custom weights

5 Conclusion

The tasks underscore the significance of various factors in MLP performance, such as batch size, activation functions, and hyperparameter optimization. They highlight the nuanced interplay between model architecture, data complexity, and algorithmic choices. Furthermore, while Keras offers convenience, the custom MLP showcased its ability to yield competitive results, albeit with slightly divergent weight outcomes. These findings collectively emphasize the importance of thoughtful parameter selection and the adaptability of both standard libraries like Keras and bespoke implementations in addressing diverse learning tasks.

Use of generative AI

Generative AI has been used to write the abstract and the conclusion of this report.

References

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