Lab report – TNM112

Deep learning for media technology, Lab 1

Filip Hamrelius (Filha243)
Lab partner: Anna Granberg (Anngr950)

Thursday 21st November, 2024 (09:22)

Abstract

This laboration aims to provide a fundamental understanding of the multilayer perceptron (MLP). The lab shows that an MLP can be implemented utilizing the Python library Keras as well as a simple custom implementation. Furthermore, the report and the lab discuss the use of decision boundaries and their corresponding weights and biases.

The results include outputs from the Keras model, where the best performance was achieved using the Adam optimizer, with a training accuracy of 95%. The custom MLP yielded similar results. The last part shows how a decision boundary can be created by hand for a simple network were the weights and biases can be defined without algorithms.

Overall, the lab provides a comprehensive understanding of the importance of hyperparameters, weights, biases, layers, and the choice of activation functions.

1 Introduction

This lab is the first in a series within the course Deep Learning for Media Technology. It was conducted to develop a fundamental understanding of the multilayer perceptron within a neural network. The tasks cover the multilayer perceptron in a fundamental and basic, yet comprehensive way. The purpose of the lab was to build an understanding of the underlying mechanics of the multilayer perceptron, such as weights, feedforward functions, bias, and basic implementation techniques for a simple neural network within artificial intelligence, specifically deep learning.

2 Background

This lab was implemented using Python 3.8.16 in combination with Jupyter Notebook, a tool for creating interactive and isolated code blocks that helps with testing and implementing code while avoiding unnecessary recompilation. Additionally, other libraries were used, with the most prominent being Keras within TensorFlow, NumPy, and Matplotlib, the latter two primarily for computations and graphical representations.

3 Method

3.1 Task 1

This chapter goes trough the subtasks regarding training a MLP in Keras.

3.1.1 Task 1.1

This task involved testing two different batch sizes while keeping the rest of the hyperparameters the same. The dataset used contained 512 training points with 2 classes. This problem did not require any additional implementation. Instead, it involved researching and understanding batch size for this implementation. This was achieved by reviewing the lectures and discussing with the lab assistant. The network has no hidden layers and therefore used the softmax activation function.

```
data.generate(dataset='linear', N_train=512, N_test=512, K=2, sigma
    =0.1)
epochs = 4
batch_size = 16 # 16 or 512
```

3.1.2 Task 1.2

This task, similar to Task 1.1, only involved testing. In this case, the goal was to test different activation functions—sigmoid, ReLU, and linear activation functions. Once again, the task focused more on reading about the different activation functions. The dataset contained 512 points and was this time polar.

```
data.generate(dataset='polar', N_train=512, N_test=512, K=2, sigma
    =0.1)
hidden_layers = 1
layer_width = 5
activation = 'linear'# 'linear', 'sigmoid' and 'ReLu'
epochs = 20
batch_size = 16
```

3.1.3 Task 1.3

This task aimed to understand the impact of different hyperparameters by testing various configurations to find the best results. Initially, this was done through trial and error, which did not yield any high scores but helped somewhat in understanding how they correlate.

```
data.generate(dataset='polar', N_train=512, N_test=512, K=5, sigma
    =0.05)
init = keras.initializers.RandomNormal(mean=0.1, stddev=0.1)
epochs = 20
batch_size = 16
opt = keras.optimizers.SGD(learning_rate=1.0, momentum=0.0)
```

The next step involved following the instructions regarding exponential decay using the Keras documentation and implementing it as described.

```
decay = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.01,
    decay_steps=10000,
    decay_rate=0.96,
```

```
    staircase=True)
opt = keras.optimizers.SGD(learning_rate=decay, momentum=0.9)
```

By the end of the task, the different combinations were discussed with classmates and the lab assistant to gain a broader understanding of the patterns between different hyperparameters.

3.1.4 Task 1.4

The final task for the first part included changing the optimizer to *Adam* and the initialization to *Glorot Normal*. No additional implementation was required beyond evaluating the results and methods.

```
init = keras.initializers.GlorotNormal(seed=None)
opt = keras.optimizers.Adam()
```

3.2 Task 2

The second task involved implementing an custom MLP using a provided skeleton code. This included defining different activation functions, the feed forward model, and an implementation to evaluate the performance of the model, as well as the necessary steps in between these implementations to run it.

For activation functions, relu, linear, sigmoid and softmax was implemented. The function for choosing an activation function relied on basic if-else statements using the argument to decide which activation function to return by.

3.2.1 ReLU Activation Function

The Rectified Linear Unit (ReLU) activation function was defined as:

$$\text{ReLU}(x) = \max(0, x)$$

3.2.2 Linear Activation Function

The linear activation function was defined as:

$$\text{Linear}(x) = x$$

3.2.3 Sigmoid Activation Function

The sigmoid activation function was defined as:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

3.2.4 Softmax Activation Function

The softmax activation function was defined as:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

where K is the number of classes and x_i is the input to the i-th class.

3.2.5 Feed Forward Function

The computation part of the implementation was done in the feed forward function, i.e the function to process the data trough the network. This was done with a basic understanding of the process from the first lecture, together with some questions to the lab assistant and a basic outline produced together with Chat-GPT.

The feed forward function works as the following:

```
def feedforward(self, x):
    y = np.zeros((len(x), self.dataset.K))
    h = 0

    for i, x_i in enumerate(x):
        h = np.reshape(x_i, (2, 1))

        for j in range(len(self.W)):
            h = np.dot(self.W[j], h) + self.b[j]
            h = activation(h, self.activation)

        h = activation(h, 'softmax')
        y[i] = h.flatten()

    return y
```

The feedforward function works by first allocating space for the output matrix y, using the size of the input data and the number of labels (classes) from the dataset. It then initializes a variable h to temporarily store the output for one layer. The outer loop iterates through the dataset, processing each input data point individually. For each input, it reshapes the data into a 2×1 matrix. The inner loop iterates through the layers of the network. For each layer, it takes the matrix multiplication of the weight matrix and the current layer's output, adds the corresponding bias, and then applies the specified activation function. (The *np.dot* is part of the *Numpy* library and performs the dot product if the inputs are vectors and matrix multiplication if the inputs are matrices.) After the inner loop completes, i.e all layers have been processed, the final output is passed through the softmax activation function to produce a probability distribution for classification. This final result is stored in the output matrix y for the current data point. The process repeats for all inputs in the dataset, ensuring that each input produces an output stored in y.

3.2.6 Measure performance of model Function

To measure the performance of the model a specific function had to be implemented. In this case only the test and training dataset was used, no validation set.

```
outputs = self.feedforward(self.dataset.x_train)
outputs_test = self.feedforward(self.dataset.x_test)

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

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

```

```
test_loss = np.mean((outputs_test - self.dataset.y_test_oh)**2)
test_acc = np.mean(np.argmax(outputs_test, 1) == self.dataset.
    y_test)

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

The loss is calculated using the `outputs` variable, which contains the predictions. Using the built-in function *argmax()*, it returns the index of the largest value, i.e, the index for the class with the highest score. A conditional check is then used to determine how many times this is true, and the mean is computed to obtain the accuracy.

Furthermore the mean loss value is derived from the mean squared error as shown in equation 1.

$$\mathrm{E} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

3.3 Task 3

The final section of the lab introduced a minimal variant of the MLP to get an understanding of the decision boundary. Here a linear dataset with two classes where to be classified. This was done with a 2x2 weight matrix and a 2x1 bias matrix, defined by hand. No activation function was specifically defined since no hidden layers were introduced, i.e the last layer uses the activation function softmax at all times.

The lab document suggested a decision boundary as straight line:

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

The line rearranged and weights plus bias added yields this:

$$\mathrm{a} \cdot x_1 + \mathrm{b} \cdot x_2 + \mathrm{b} = 0 \tag{3}$$

Or in vector form

$$\mathrm{W} \cdot \mathrm{x} + \mathrm{b} = 0 \tag{4}$$

- W is a 2×2 weight matrix.
- b is a 2×1 bias vector.

- $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ is the input vector.

The first output follows the decision boundary and is $-x_1 - x_2 + 1 = 0$.
The second output will be the opposite of that $x_1 + x_2 - 1 = 0$.

4 Results

4.1 Task 1

4.1.1 Task 1.1

Which model is best at separating the two classes (provide the accuracy on the test set)? Why is this the case? (Hint: how many iterations of SGD are performed in the two cases?)

As shown in Figure 1, a batch size of 16 outperformed a batch size of 512 by a significant margin. This is because, with a dataset size of 512, using a batch size of 512 results in only one pass through the data, updating the weights once per epoch. With a batch size of 16, however, the weights are updated 32 times per epoch.

Batch_size = 16

```
Model performance:
32/32 [==============================] - 0s 2ms/step - loss: 0.0370 - accuracy: 0.9922
        Train loss:     0.0370
        Train accuracy: 99.22
32/32 [==============================] - 0s 2ms/step - loss: 0.0410 - accuracy: 0.9863
        Test loss:      0.0410
        Test accuracy:  98.63
32/32 [==============================] - 0s 2ms/step
512/512 [==============================] - 1s 1ms/step
```

Batch size = 512

```
Model performance:
32/32 [==============================] - 0s 2ms/step - loss: 0.2165 - accuracy: 0.6201
        Train loss:     0.2165
        Train accuracy: 62.01
32/32 [==============================] - 0s 2ms/step - loss: 0.2172 - accuracy: 0.6191
        Test loss:      0.2172
        Test accuracy:  61.91
32/32 [==============================] - 0s 2ms/step
512/512 [==============================] - 1s 1ms/step
```

Figure 1: Task 1.1 - 16 or 512 in batch size

Why is it possible to do the classification without non-linear activation function (there's only a softmax activation)?

Since the dataset is linear with only two classes, it is possible to classify the data with a single line. The softmax activation function works by creating a probability distribution for the output, but the actual output is still assigned to either the first or the second category.

4.1.2 Task 1.2

Why does linear activation not work?

Since the data cannot be separated by a straight line (i.e, the dataset is polar), using a linear activation function, which produces a straight-line decision boundary, is insufficient to fit the data.

On average, what is the best classification accuracy that can be achieved with a linear activation function?

In this case, an accuracy of around 50% was achieved because the data is polar, meaning the clusters are mixed together. The average accuracy of 50% occurs because, with the decision boundary positioned entirely outside, it classifies only one of the classes correctly; if it is centred, half of each class will be classified correctly. Accuracy can vary depending on the distribution of data points and the placement of the decision boundary.

Can you find an explanation for the difference comparing sigmoid and relu activation?

The sigmoid activation function always outputs values between 0 and 1, which makes it optimal for binary classification tasks, especially in the output layer where the model needs to produce a probability value. The ReLU activation function, on the other hand, outputs values between 0 and infinity. This allows it to handle a wider range of values and makes it suitable for hidden layers in deep networks.

4.1.3 Task 1.3

What combination worked best, and what was your best classification accuracy (on the test set)?

The combination that yielded the best result was as follows:

- Standard Deviation = 0.1
- Learning Rate = 0.01
- Momentum in SGD = 0.9
- Batch size = 32
- Epochs = 120
- Exponential Decay active

The result can be seen in Figure 2.

```
Model performance:
80/80 [==============================] - 0s 1ms/step - loss: 0.0185 - accuracy: 0.9391
        Train loss:     0.0185
        Train accuracy: 93.91
80/80 [==============================] - 0s 1ms/step - loss: 0.0228 - accuracy: 0.9305
        Test loss:      0.0228
        Test accuracy:  93.05
80/80 [==============================] - 0s 1ms/step
512/512 [==============================] - 1s 1ms/step
```

Figure 2: Task 1.3 results

Can you find any patterns in what combinations of hyper parameters work and doesn't work?

Some pattern that was recognised was that a high momentum with a high learning rate can lead to the model moving to quickly. There should also be a balance between epochs and batch size, it is unnecessary to maximize one while keeping the other low. Learning rate decay is beneficial as it reduces the step size when approaching the minimum, helping the model converge more smoothly.

4.1.4 Task 1.4

Does this perform better compared to your results in Task 1.3?

Yes, while using glorot normal and the optimizer Adam from Keras, a slightly better result was produced. It performed 2 percentage points better, as can be seen comparing Figure 2 and 3.

```
Model performance:
80/80 [==============================] - 0s 1ms/step - loss: 0.0127 - accuracy: 0.9590
        Train loss:     0.0127
        Train accuracy: 95.90
80/80 [==============================] - 0s 1ms/step - loss: 0.0181 - accuracy: 0.9414
        Test loss:      0.0181
        Test accuracy:  94.14
80/80 [==============================] - 0s 1ms/step
512/512 [==============================] - 1s 1ms/step
```

Figure 3: Task 1.4 results

4.2 Task 2

The custom MLP performed highly with a train and test accuracy of both 99%. The results can be seen in deatil in Figure 4 and Figure 5.

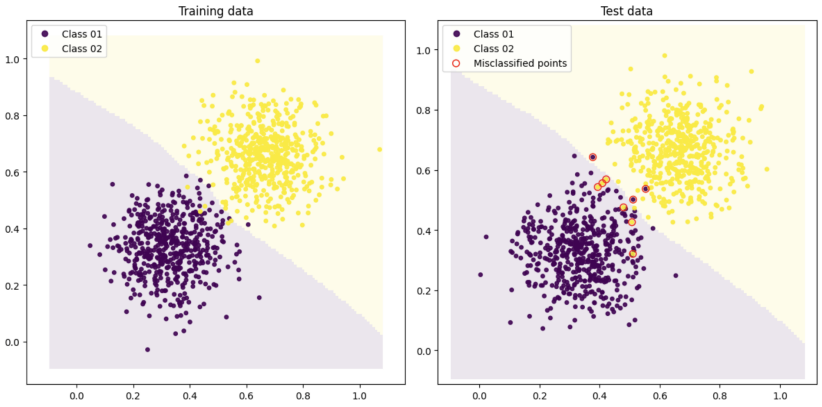

Figure 4: Task 2 numerical representation of result

```
Number of hidden layers:  10
Number of model weights:  550
Model performance:
        Train loss:     0.0127
        Train accuracy: 0.99
        Test loss:      0.0142
        Test accuracy:  0.99
```

Figure 5: Task 2 graphical representation of result

4.3 Task 3

What is the simplest possible solution?

$$W = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \tag{5}$$

$$b = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{6}$$

How many additional solutions are possible?

There are an infinite amount of solutions for this problem. The combination of the weights and biases can be kept the same by adjusting the amount in the same way on all points, yielding the same result but with different inputs.

How can you, in the simplest way, change the weights/biases to switch the predicted class la- bels?

In this case, if the decision boundary would switch to the second output the labels would switch, that is if one were to change all the signs of the weight and bias matrices, the labels would switch.

Can you find a general formula for specifying which combinations of weights

and biases will generate the decision boundary?

A general formula would be the relationship shown in Equation 5 and 6 but with a constant changing the weights and biases proportionally the same, as shown in Equation 7 and 8.

$$W = \alpha \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}, \tag{7}$$

$$b = \alpha \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \tag{8}$$

The results for the decision boundary can be seen in Figure 6.

```
Number of hidden layers:  0
Number of model weights:  2
Model performance:
        Train loss:     0.1752
        Train accuracy: 0.99
        Test loss:      0.1752
        Test accuracy:  0.99
```

Figure 6: Task 3 numerical representation of result

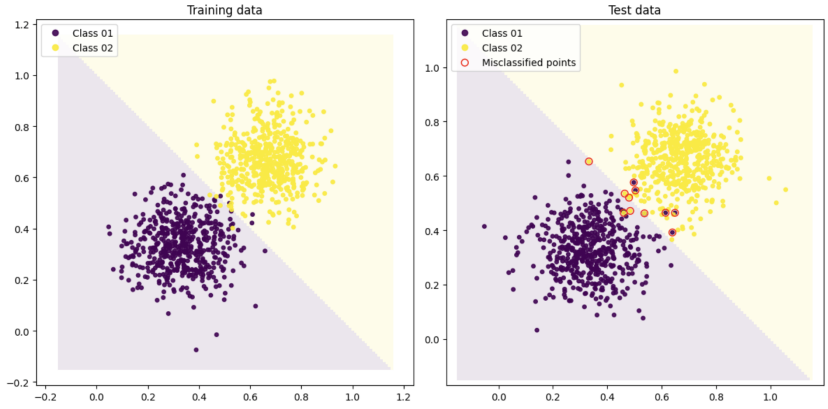

Figure 7: Task 3 graphical representation of result

5 Conclusion

In conclusion, this lab provided a fundamental understanding of hyperparameters, the feedforward method, and, specifically, different activation functions and their corresponding applications. The most challenging part was implementing the MLP in part two, especially given the initial lack of comprehensive understanding of the MLP. Real insights were gained after completing the implementation, and it would have been beneficial to study MLP implementations in pure code beforehand. That said, diving straight into the task proved to be an efficient way to learn through hands-on experience. Conclusions regarding

the aim of the lab can be said that the lab gave a great start for deep learning and the multilayer percpetron.

Use of generative AI

Generative AI, specifically ChatGPT, was used to provide a starting point for the feedforward loop in Task 2.0. The tool was also used to gain a deeper understanding of Task 3.0 and to brainstorm ideas. Furthermore the tool was used to format some parts in Latex, mostly regarding equations, notation and code structure.