Lab report – TNM112

Deep learning for media technology, Lab 1

Emma Zettervall (emmze999)

Wednesday 4th December, 2024 (15:07)

Abstract

In this report, we explore the construction and training of a Multi-Layer Perceptron (MLP) for tasks involving non-linearly separable data. Using both the Keras framework and a custom implementation, we experiment with various hyperparameters, including activation functions, batch sizes, learning rates, and initialization methods, to understand their influence on model performance. Additionally, we demonstrate the effectiveness of different optimization strategies and test a custom feedforward neural network implementation. The results provide valuable insights into designing and optimizing neural networks, emphasizing the importance of hyperparameter tuning and initialization techniques for achieving high accuracy.

1 Introduction

The multi layer perception (MLP) is a powerfull neural network for data that is not linearly seperable. Building an effective MLP requires consideration of several factors, including number of layers, layer size, activation fuctions, optimization methods, loss functions and initialization. The aim of this lab is to experiment with these parameters to gain a deeper understanding of their roles and impacts on the performance of the MLP. This will be done by using Keras, as well as implementing our own MLP with numpy. [1]

2 Background

This lab explores Multi-Layer Perceptrons (MLPs), a type of feedforward neural network that uses weights, biases and activation functions to model data. Techniques like Stochastic Gradient Descent (SGD) optimize network parameters, with hyperparameters such as batch size, epochs, and learning rate. Activation functions introduce non-linearity, enabling the network to handle complex data. Proper weight initialization and loss functions, such as Mean Squared Error (MSE), ensure stable training and guide optimization. These techniques are essential for understanding and building effective neural networks.

3 Method

The lab was done in a jupyter notebook. Keras or python scripts was used to build MLP models.

3.1 Task 1

In this task we experimented with different datasets and parameters to understand the effect of them. The different datasets were linear and polar with different amounts of classes.

3.1.1 Task 1.1

The aim of the first task was to compare different batch sizes using a linear dataset with 512 training points and 2 classes. The model did not have any hidden layers so the activation is only applied to the output layer. Therefore softmax was used for activation. The model was trained for 4 epochs and a learning rate of 1. Fisrt the training was done with a batch size of 512 and then 16. The dataset used was:

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

The parameters put in code:

```
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=0.01, stddev=0.1)
epochs = 4             # Number of epochs for the training
batch_size = 512         # Batch size to use in training
loss = keras.losses.MeanSquaredError() # Loss function
learning_rate = 1.0
```

3.1.2 Task 1.2

For the next task the aim was to experement with the different activation functions linear, sigmoid and ReLU. This was done with a polar data set with 512 training points and 2 classes. The model was trained on a network with one hidden layes of five neurons for 20 epochs with a learning rate 1 and batch size 16. This was done one time for each activation functions. The dataset used was:

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

The parameters put in code with a linear activation function for example:

```
hidden_layers = 1      # The number of hidden layers in the network
    (total number of layers will be L=hidden_layers+1)
layer_width = 5        # The number of neurons in each hidden layer
activation = 'linear' # Activation function of hidden layers
init = keras.initializers.RandomNormal(mean=0.01, stddev=0.1)
epochs = 20             # Number of epochs for the training
batch_size = 16         # Batch size to use in training
loss = keras.losses.MeanSquaredError() # Loss function
learning_rate = 1.0
```

3.1.3 Task 1.3

After a deeper understanding of the different activation functions and the batch size, this task was ment to understand the remaining parameters. This was done

on a polar dataset with 512 training poins, 5 classes and sigma=0.05. The network had 10 hidden layers with 50 neurons each and ReLU activation. Different cominations of the mean and stddev in the normal initalizer, the learning rate and the momentum in the SGD was tested. Exponential decay shown below was also implemented and tested.

```
initial_learning_rate = 0.1
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate,
    decay_steps=100000,
    decay_rate=0.96,
    staircase=True)
```

3.1.4 Task 1.4

For this task, the same settings as the previous task was used exept for the normal initializer and the customized optimizer. The initializer was changed to glorot normal and the optimizer was changed to Adam as shown below.

```
init = keras.initializers.GlorotNormal() # Initialization method (
    starting point for the optimization)
opt = keras.optimizers.Adam(learning_rate=0.001) # Optimizer
```

3.2 Task 2

In task 2, the aim was to implement our own MPL. The functions "activation"," setup_model", "feedforward" and "evaluate" was implemented in the file mlp.py. The weights from the previos task was used in the new model. In the "activation" function, the equations for the different activation functions were implemented as below.

```
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))  # Stability trick
        return exps / np.sum(exps)
    else:
        raise Exception("Activation function is not valid",
    activation)
```

In the function "setup_model", the number of hidden layers and the number of weights in the model were specified as below.

```
    self.hidden_layers = len(W) - 1

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

In the "feedforward" function, a matrix for storing output values was implemented aswell as a for-loop for the feed forward algorithm. This iterates over all the data points and each hidden layer and applies matrix multiplication on the output from the prevoius layer and the current layer and adds the activation function. Softmax was added on the final layer and the data was returned flattened.

```
    def feedforward(
        self,
        x       # Input data points
    ):
        y = np.zeros((len(x), self.dataset.K))

        for i, x_i in enumerate(x):
            h = np.reshape(x_i, (2, 1))  # Assuming each input is
    2-dimensional

            for j in range(len(self.W)-1):
                h = np.dot(self.W[j], h) + self.b[j]
                h = activation(h, self.activation)

            h = np.dot(self.W[-1], h) + self.b[-1]
            h = activation(h, 'softmax')

            y[i] = h.flatten()
        return y
```

In the "evaluate" function, the "feedforward" function was applied to the training and test data. Then the loss and accuracy of the model was calculated for the training data and the test data.

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

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

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

        y_test_pred = self.feedforward(self.dataset.x_test)
        test_loss = np.mean((y_test_pred - self.dataset.y_test_oh)
    **2)
        test_acc = np.mean(np.argmax(y_test_pred, 1) == self.
    dataset.y_test)

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

3.3 Task 3

The same implementation as in Task 2 was used in this task. This time the weights and biases was hard coded to achieve a decision boudry at

$$x_2 = 1 - x_1$$

The implemented weights and biases was:

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

4 Results

The results from the tasks and the answered questions from the lab documentation [1] will be displayed below.

4.1 Task 1

4.1.1 Task 1.1

Batch size	Loss	Accuracy
16	0.0142	0.9854
512	0.2354	0.6465

Table 1: Loss and accuracy from Task 1.1

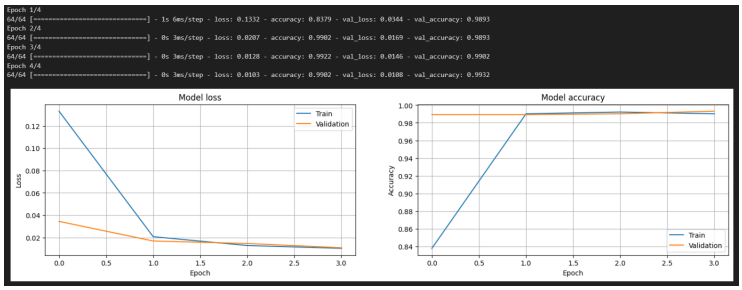

Figure 1: Optimization with batch size 16.

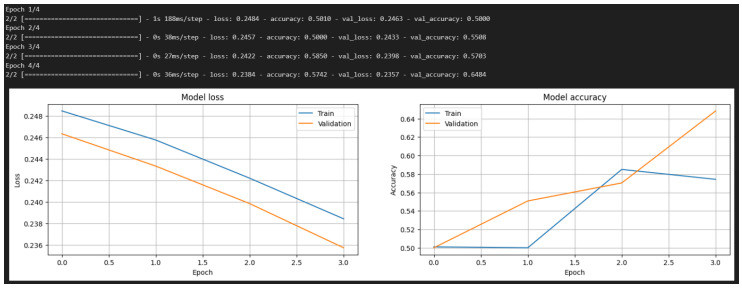

Figure 2: Optimization with batch size 512.

Q1: Which model is best at separating the two classes (provide the accuracy on the test set)? Elaborate on why there is a difference between the results, by comparing not only accuracy but also the behavior of the optimization.

The model with a batch size of 16 performed better since the accuracy is higher and the optimization was faster. This is due to 4x512/16 updates of the weights and biases compared to 4x512/512 updates. The model with a batch size of 16

achieved high accuracy and low loss after the first epoch, while the model with a batch size of 512 changed drastically after each epoch.

Q2: Why is it possible to do the classification without non-linear activation function (there's only a softmax activation)?

Since the model has no hidden layers, only the softmax activation is applied to the output layer. The data is linearly separable, so the softmax can directly divide the two classes linearly.

4.1.2 Task 1.2

Activation function	Loss	Accuracy
Linear	0.2514	0.5
Sigmoid	0.2501	0.5
ReLU	0.0102	0.9912

Table 2: Loss and accuracy from task 1.2

Q1: Why does linear activation not work?

Linear activation does not work because there is no linear decision boundary that can separate the two classes in a polar dataset. In a polar datasets, the decision boundary between classes is often curved, and a linear activation function can not achieve that.

Q2: On average, what is the best classification accuracy that can be achieved with a linear activation function on this dataset?

The best classification accuracy that can be achieved on average is 0.5. This is because the model can only correctly classify half of the samples on average, making it no better than random guessing.

Q3: Can you find an explanation for the difference comparing sigmoid and ReLU activation?

The ReLU activation function produces a gradient of 0 for negative values and 1 for positive values. In contrast, the sigmoid activation function has a steep gradient around input values close to 0 but the gradient becomes very small for large positive or negative inputs. This can cause vanishing gradient problems during backpropagation, where the gradients decrease significantly, making it difficult for the network to update the weights effectively deeper layers.

4.1.3 Task 1.3

The optimal parameters were:

```
hidden_layers = 10    # The number of hidden layers in the network
    (total number of layers will be L=hidden_layers+1)
layer_width = 50       # The number of neurons in each hidden layer
activation = 'relu' # Activation function of hidden layers
```

```
#init = keras.initializers.RandomNormal(mean=0.01, stddev=0.1) #
    Initialization method (starting point for the optimization)
init = keras.initializers.RandomNormal(mean=0.0, stddev=0.1) #
    Initialization method (starting point for the optimization)
epochs = 80             # Number of epochs for the training
batch_size = 32         # Batch size to use in training
loss = keras.losses.MeanSquaredError() # Loss function

initial_learning_rate = 0.1
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate,
    decay_steps=100000,
    decay_rate=0.96,
    staircase=True)

opt=keras.optimizers.SGD(learning_rate=lr_schedule,
               momentum=0.9)
```

Q1 What combination worked best, and what was your best classification accuracy (on the testset)?

A smaller learning rate worked good, (0.01) and add some momentum in SGD. The learning decay was the biggest change. It gave a loss of 0.0185 and accuracy of 0.9422

Q2 Can you find any patterns in what combinations of hyperparameters work and doesn't work?

Smaller learning rates with momentum in SGD lead to better results. Learning rate decay achieved stable convergence. Weight initialization with a mean of 0.0 and standard deviation of 0.1 improved the performance. ReLU activation works better than sigmoid for deeper networks.

4.1.4 Task 1.4

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

Yes, using the Glorot Normal initialization improved the model's performance slightly, compared to the results in Task 1.3. With the Glorot Normal initialization, the model achieved a lower loss of 0.0144 and higher accuracy of 0.9539.

4.2 Task 2

Model	Loss	Accuracy	Number of weights
Keras	0.0105	0.9844	23 202
Own implementation	0.0105	0.9844	23 202

Table 3: Loss, accuracy and number of weights from Task 2

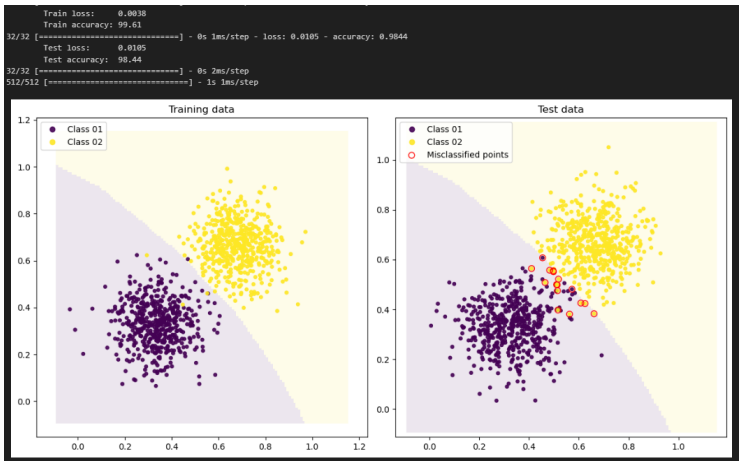

Figure 3: Output from Keras implementation for Task 2.

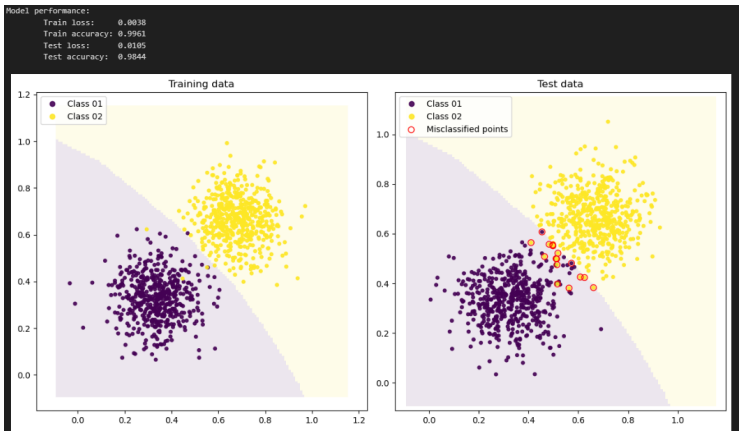

Figure 4: Output from own implementation for Task 2.

The input data, weights and biases in this task comes from the model in the previous task. As a result, the loss, accuracy and decision boundaries are excactly the same for the models.

4.3 Task 3

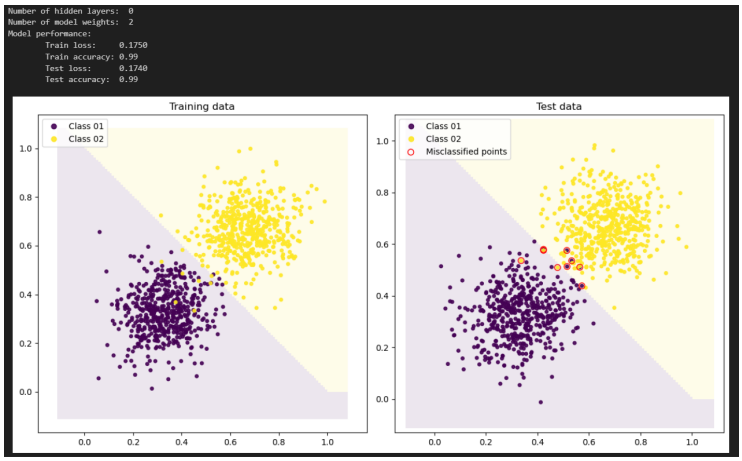

Figure 5: Output from task 3 with decision boundary.

Manually derive weights and biases to specify a model that draws a decision boundary at x2 = 1-x1. What is the simplest possible solution? How many additional solutions are possible?

The simplest possible solution involves a single layer with weights and biases as describes below:

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

Infinite solutions exist sinse the same result will be achived by for example scaling both the weights and biases by a constant factor. As long as z1 = z2.

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

By switching the signs of the weights and flipping the biases to:

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

By writing out the output of the MLP, motivate why your choice of weights and biases creates a decision boundary at x2 = 1-x1. Can you find a general formula for specifying which combinations of weights and biases will generate the decision boundary?

The decision boundary equation can be rewritten as $x_2 + x_1 - 1 = 0$. Since the decision boudaray is where $z_1 = z_2$ the weights needed to be:

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

with

$$\mathbf{z} = W \cdot \mathbf{x} + b$$

we apply matrix multiplication and get:

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} -x_1 + 1 \\ x_2 \end{bmatrix}.$$

5 Conclusion

This lab provided valuable insights in MLPs and the role of different hyperparameters in their performance. Through experimentation, we observed that:

- Smaller batch sizes lead to faster optimization due to more frequent weight updates.
- Activation functions like ReLU significantly outperform linear or sigmoid functions in deeper networks, as they enable better representation of non-linear decision boundaries.
- Effective weight initialization (e.g., Glorot Normal) and adaptive learning rates (e.g., using decay or Adam optimizer) enhances performance.
- Our custom implementation closely matched the Keras model's performance, demonstrating the fundamental principles behind neural network design..

Use of generative AI

The abstract, background and conclusion was written with the help of ai, aswell as some rephrasing.

References

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