Lab report – TNM112

Deep learning for media technology, Lab 1

Alice Swanberg Roscher (alisw344)

Wednesday 22nd November, 2023 (11:36)

Abstract

This report explores the construction and training of an multilayer perceptron (MLP) in order to gain knowledge about the features of the MLP. Parameters of both the neural network and the optimization were analyzed and functions to simulate a network were implemented. Some combinations of parameters gave low accuracy results for the test set but setting the parameters in a concious manner gave good results for both a Keras network and the implementet network. The results imply that it is important to understand the function of the parameters and how they cooperate.

1 Introduction

Neural networks are in its construction inspired by the brain and consist of neurons and connections between them. Over time, the network denotes rules for solving a problem by processing training data. [2]

The core component of this idea is the perceptron which is an artificial neuron that takes in input values, weighs them in a linear combination (depending on the denoted rules), adds a tuning bias and sends the result through a activation function that outputs one value, the activation. The neuron is either activated or not depending on the weights and biases and the activation is used as input to the next perceptron. [2]

A simple neural network is structured in layers, is fully connected and uses a feed forward approach meaning that the output from a neuron in one layer is input to all neurons in the next. The layers between the input and output are called hidden layers [2]

To establish rules, i.e. optimize the model, the network keeps track of the loss in cost using a cost funtion that compares the correct values to the predicted one. This function is then analyzed, often times with a method called gradient descent which approaches the minimum cost by stepping in the negative direction of the loss functions gradient. The gradient is the derivative of the loss function with respect to the weights which can be backpropagated through the network by repeatingly using the chain rule. However, the weights and biases of a neural network are many which result in computationally heavy operations. The solution to this is to blance a set of parameters and add supporting tactics to streamline the process.[2]

This lab is performed to explore a fully connected, feedforward multilayer perceptron (MLP) network, understand the concept and get a sense of what effect the different parameters have on the result when the network is used to classify 2D dots. The lab is the foundation of deep learning and the concepts covered are crucial for further understanding in the field. [3]

2 Background

There are some important techniques and hyper-parameters being explored in this report while training the model with gradient descent. For instance, activation functions that enable complex relationship mapping of input and output. Another parameter is batch size which defines subsets of training data to use in one iteration of updating the weights. Going through all subsets once is called an epoch which can also be repeated. In optimization, the stepsize used to update the weights is called learning rate which can adapt or decay over time and the initialization of weights can be done in different ways using a normal distribution. An additional technique is momentum which utilizes past gradients to improve the search for the minimum cost. [2]

3 Method

This section describes the three tasks and how they were solved and implemented in python using the NumPy library for vectors and matrices.

3.1 Task 1 - Keras MLP

In this task a Keras MLP was trained on a linear dataset and a polar dataset respectively.

3.1.1 Task 1.1

Firstly a linear dataset with 512 points with 2 classes was generated to explore the effect of batch sizes [3]. The model was configured with zero hidden layers and the training used 4 epochs, a learning rate of 1 and batches of sizes 512 and 16 respectively. When using batches, the data order is firstly randomized and then divided into batches. The gradient is evaluated for each batch going through the data set and is thus an estimate of the actual gradient (the initial randomization should giva a sufficient estimate). Such an estimate approach while using gradient descent is called stocastic gradient descent (SGD). The process was repeated 4 times according to the number of epochs. Since a smaller batch size correspond to more evaluations of the gradient, this should lead to better results.[2]

3.1.2 Task 1.2

The second test was to compare different activation functions, linear, sigmoid and ReLu. The polar dataset was used with 512 points and 2 classes. The network used 1 hidden layer with 5 neurons. The network was trained for 20 epochs with batch size 16 and learning rate 1 [3]. The linear activation function

does not alter the weighted sum and is therefore not able to describe complex relationships like the non-linear desicion boundary required for this dataset. Sigmoid has a S-shape and is a non-linear activation function that outputs values between 0 and 1. The range makes it possible for the network to learn more complex relationships. The ReLu activation is a non-linear function that deactivates neurons that outputs negative values and has a non-saturating property for positive values. This is where the two non linear activation functions can be distinguished in the optimization, using SGD. The S shape of the sigmoid has a faint slope for very large (positive or negative) values resulting i a vanishing gradient for those values. A vanishing gradient results in slow learning due to the propagating gradient and negligible updates to weights. The ReLu however maintains the same slope for all positive values. [1]

3.1.3 Task 1.3

The next task was to try combinations of parameters and find the best possible composition. The polar dataset with 512 points and 5 classes was used aswell as a network with 10 hidden layers with 50 neurons each and ReLu activation [3]. Both initialization, learning rate, momentum, exponential decay, epochs and batch size was tweaked in iterations. The following code shows the combination of parameters that gave the best result.

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

# Hyper-parameters
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=0.02, stddev=0.1) #
    Initialization method (starting point for the optimization)
epochs = 120             # Number of epochs for the training
batch_size = 32      # Batch size to use in training
loss = keras.losses.MeanSquaredError() # Loss function
opt = keras.optimizers.SGD(learning_rate=0.01, momentum=0.5) #
    Optimizer
```

Evidently, the parameters need to be balanced against each other. For instance, it is 'prefered' to use small batches but an estimate of the gradient might do the job. In a likewise manner, training for many epochs should boost the result but when the batch size is small it gets computationally heavy. Learning rate and momentum is another pair of parameters that are balanced against each other. The learning rate is the stepsize used in the cost landscape toward the global minimum. A large value might miss the searched point and result in oscillation while a small value might take too much time. Momentum adds inertia and provide acceleration to the process. This allows for greater learning rate because it regulates the change of direction by considering the mean of past gradients. Additionally when approaching the minimum cost it is often suitable to reduce the learning rate to find a more exact point. Keras' ExponentialDecay was tested for this purpose. The initialization for the optimization, i.e. the initial values of the weights and biases are also of importance. A randomized approach using normal distribution will assign varying values which allows the neurons to learn differently. If instead they were initialized with zeros the acti-

vations would all be zero and the gradient would be unusable. A constant would result in the neurons learning to uniformly. [2]

3.1.4 Task 1.4

Lastly, task 1.4 compared the initialization and optimizer from task 1.3 to glorot normal and Adam respectively by replacing line 7 and 11 in task 1.3 [3]. Glorot initialization performs a normalization before using a normal distribution which prevents the vanishing gradient problem. The Adam optimizer is variant of SGD that uses momentum and a customized learning rate for each weight. These methods tunes the network more automatically.

3.2 Task 2 - Implementing an MLP

In this task, an MLP was implemented. Four functions were completed, activation, setup model, feedforward and evaluate.

3.2.1 activation

This function specifies different activation functions that is used for the neurons in the feedforward function. Equations (1), (2), (3) and (4) show linear, relu, sigmoid and softmax activation respectively.

$$f(x) = x \tag{1}$$

$$f(x) = max(0, x) \tag{2}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{4}$$

These are implemented in code and chosen depending on the activation variable.

```
1  def activation(x, activation):
2
3      #TODO: specify the different activation functions
4      # 'activation' could be: 'linear', 'relu', 'sigmoid', or '
       softmax'
5      if activation == 'linear':
6          # TODO
7          return x
8
9      elif activation == 'relu':
10         return np.maximum(0,x)
11
12     elif activation == 'sigmoid':
13         return 1 / (1 + np.exp(-x))
14
15     elif activation == 'softmax':
16         exp_x = np.exp(x - np.max(x))
17         return exp_x / exp_x.sum(axis=0)
18
```

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

3.2.2 setup model

This function sets up the model by calculating the hidden layer and the total number of weights. The MLP is described by equation (5) in matrix form. h is the activation column vector, W the weights matrix, b the bias column vector and sigma the activation function. The superscript is the layer index.

$$h^{(l)} = \sigma(W^{(l)}h^{(l-1)} + b^{(l)}) \tag{5}$$

Since each W matrix contains the weights between two layer, each matrix in the list of weight matrices represent one layer. Subtracting one omitts the input layer and calculates the number of hidden layers.

```
1   # TODO: specify the number of hidden layers based on the length of
        the provided lists
2           self.hidden_layers = len(W)-1
```

The total number of weights are all the individual weights and biases. These are calculated by adding the size of all weight matrices of the weight matrix list aswell as adding the size of all bias vectors of the bias vector list and then sum them.

```
1   # TODO: specify the total number of weights in the model (both
        weight matrices and bias vectors)
2           self.N = sum(w_i.size for w_i in self.W) + sum(b_i.size for
         b_i in self.b)
```

3.2.3 feedforward

The feedforward function applys the layer operations of equation (5) to all layers. Firstly the output matrix is set up using NumPy to create a matrix of zeros. In the end each row will correspond to the predictions of probability of belonging to a class for a single datapoint. Each column corresponds to a class.

```
1   # TODO: specify a matrix for storing output values
2           y = np.zeros((x.shape[0], self.dataset.K))
```

There are two main loops, the outer one iterates over all datapoints to perform operations on each. The datapoint is extracted to inputLayer variable and transformed into a column vector, like h in (5), to prepare for the matrix multiplication. The nested loop performs the operations of (5) for each hidden layer using NumPy, i.e., multiplying the weight matrix, adding the bias vector and applying the activation function. The final layer is handled separately to apply the softmax activation which outputs predictions for the classes. The result is inserted in the output matrix, again using Numpy to remove possible added dimensions.

```
1   # TODO: implement the feed-forward layer operations
2           # 1. Specify a loop over all the datapoints
3           for point in range(x.shape[0]):
4               # 2. Specify the input layer (2x1 matrix)
5               inputLayer = x[point, :]
```

```
            inputLayer = inputLayer[:, np.newaxis]

            # 3. For each hidden layer, perform the MLP operations
            for layer in range(self.hidden_layers):
                # - multiply weight matrix and output from previous layer
                z = np.matmul(self.W[layer], inputLayer)
                # - add bias vector
                z = z + self.b[layer]
                # - apply activation function
                outputLayer = activation(z, self.activation)

                inputLayer = outputLayer # move forward

            # Apply softmax activation to the final layer
            z_output = np.matmul(self.W[-1], inputLayer) + self.b[-1]  # Weighted sum with bias
            outputLayer = activation(z_output, 'softmax')  # Apply softmax for the output layer

            y[point] = np.squeeze(outputLayer, axis=(1,))  # Store final output for the point
```

3.2.4 evaluate

This function evaluates the accuracy of the predictions of the network. Firstly the result of the train set are predicted by using the feedforward function. Then, the loss is computed with the mean square error method described by equation (6)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y - \hat{Y})^2 \tag{6}$$

where Y is the actual class, Y hat the prediction and n the number of datapoints. In the code NumPy was used to compute the mean and a one hot encoded vector holding a 1 for the correct class (and zeros for the rest) to compare against.

To extract the predicted class labels, NumPys argmax was used to fetch the highest predicted probability across columns. The mean accuracy was then computed by comparing the predicted labels with the actual labels. Finally the result was converted to represent percentage.

```
# TODO: formulate the training loss and accuracy of the MLP
        # Assume the mean squared error loss
        # Hint: For calculating accuracy, use np.argmax to get predicted class
        ypTrain = self.feedforward(self.dataset.x_train)
        train_loss = np.mean((self.dataset.y_train_oh - ypTrain)**2)

        ypTrain = np.argmax(ypTrain,1)
        train_acc = np.mean(ypTrain == self.dataset.y_train)
        train_acc = train_acc*100
```

3.3 Task 3 - Disecting an MLP

For task 3 the decision boundary was manually set for a linear data set with 2 classes by deriving the weights for the boundary described by (7). [3]

$$x_2 = 1 - x_1 \tag{7}$$

With starting point from equation (5), omitting the activation function, an expression for each of the two outputs was derived. On the boundary (7) the expressions can be set equal, resulting in (8)

$$(w_{1,1} - w_{2,1})x_1 + (w_{1,2} - w_{2,2})x_2 + b_1 - b_2 = 0 \tag{8}$$

By matching the coefficients to equation (7) the weights and biases could be determined. Infinite options for the weights and and biases were possible. The solution chosen can be found in (9) where the weights control the slope of the boundary and b the sift.

$$W = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, b = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{9}$$

The corresponding code uses NumPy to get the correct dimensions.

```
# TODO: Task 3: specify a weight matrix and a bias vector
W = [np.array(([0,0],[1,1]))]
b = [np.array(([[0],[-1]]))]
data.generate(dataset='linear', N_train=32, N_test=512, K=2, sigma
    =0.1)
```

4 Results

Your discussions around the results should be formulated in enough details to clearly demonstrate that you have understood the results and the reason for why they look as they do. Report results using, e.g., tables with numerical comparisons, such as in Table **??**, as well as figures of training behavior and resulting model performance, such as in Figure **??**.

In this section the result of the tasks are presented and discussed.

4.1 Task 1

Table 1 show that a smaller batch size estimates the gradient more accurately. This is because the weights are updated more frequently and a smaller size also introduces noise that prevents overfitting by providing a broader view of the data. [2]

Batch size	Test accuracy (%)
512	72.66
16	99.22

Table 1: Test accuracy with varying batch size

As expected, as shown by Table 2, the linear activation function can not label more datapoints correct than the ones that happened to land in the correct class with the linear boundary. The sigmoid activation aquires a similar result due to the vanishing gradient which slows down the training. ReLu activation function aquires 98.93% test accuracy since the slope of the function for activated values does not decrease and training can proceed.

Activation function	Test accuracy (%)
Linear	50.00
Sigmoid	50.00
ReLu	98.93

Table 2: Test accuracy of activation functions.

When manually finding the best parameters the accuracy reached about 95% as Figure 1 shows. Both the test and validation curves follow a similar path which indicate that the model is good at generalize to unseen data. Using Glorot initialization and Adam optimizer also reached about 95% accuracy (slightly higher) according to Figure 2 but it gets to this point in only a few epochs due to their advantages, far sooner than in Figure 1.

As discussed in Task 1.3, parameters somewhat opposite were balanced against each other. For instance batch size and epoch, aswell as learning rate and momentum.

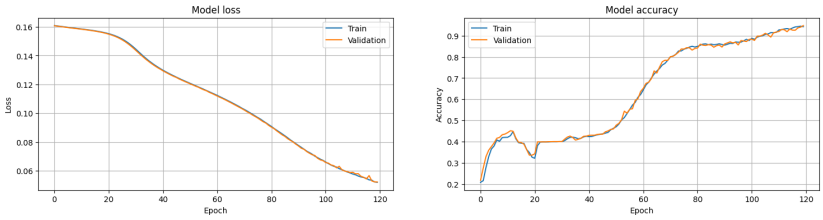

Figure 1: Model loss and accuracy for best parameters

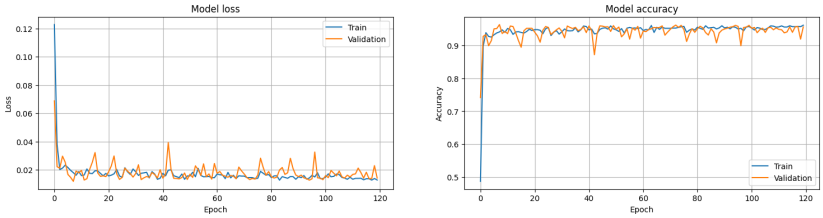

Figure 2: Model loss and accuracy using Glorot initialization and Adam optimizer

4.2 Task 2

The implemented MLP achives a similar result to the Keras model (with Glorot initialization and Adam optimizer) for all tested configurations of hidden layers,

layer width and activation function. Figure 3 shows the result when using Relu activation and 10 hidden layers with 5 neurons each. The implemented functions seem to work well even though the code (in hindsight) is relatively simple.

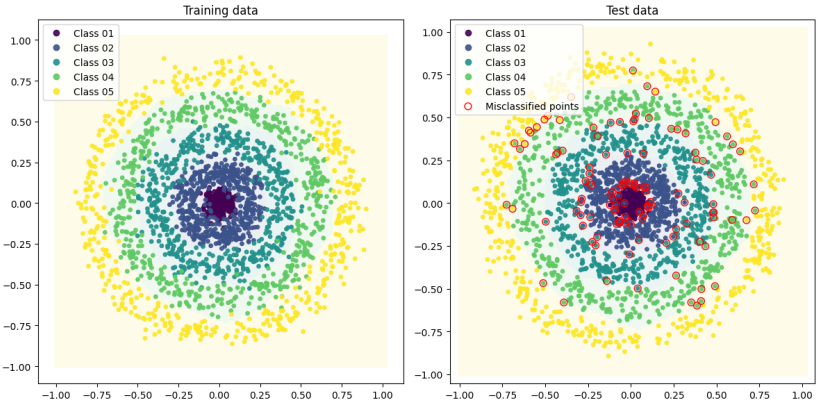

Figure 3: Resulting plot of the implemented MLP using ReLu activation with 10 hidden layers of layer width 5

4.3 Task 3

The derived weight matrix W and bias vector b did create the expected decision boundary as can be seen in Figure 4. The training data was correctly predicted while the test data had a few falsely predicted points.

Figure 4: Decision boundary with derived weight matrix and bias vector

Since the weight matrix and bias vector control the slope and shift of the decision boundary respectively, by multiplying them with -1 inverts the classes and most points are labeled falsely according to Figure 5.

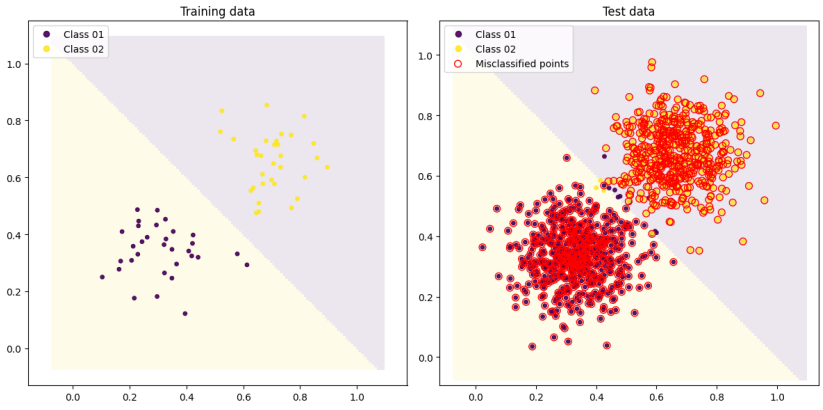

Figure 5: Decision boundary with derived weight matrix and bias vector

5 Conclusion

The conclusion of this lab and report is that it is important to learn the effect of the parameter to be able to adjust them toward a better result. The same settings also does not yield the same result each time.

The lab provided more clarity to both constructiong and training an MLP aswell as managing the training.

Use of generative AI

Generative AI was used to clarify the concepts and code of this report with examples or alternative explainations, to gain a solid understanding of the construction and training of MLPs' theoretically and in code.

References

[1] Pragati Baheti. Activation functions in neural networks [12 types use cases].

[2] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

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