Deep Learning - Assignment 1

Ayça Avcı - S4505972
Sytse Oegema - S3173267

April 19, 2022

1 Introduction

This report compares the performance of two multi-layer perceptron networks on a binary classification problem. The first of the two networks has been developed from scratch by the authors. The second network has been constructed with Keras module[1]. Both networks have the same node architecture and use the same activation function. The networks were also trained with the same loss function on the same data set. The aim of this setup is to test and show the capabilities of the self developed neural network, have an understanding of how Stochastic Gradient Descent (SGD) algorithm works, and compare the performance of both neural networks by using evaluation metrics.

2 Method

This section starts with a description of the used data set. Thereafter the design of the self implemented neural network is described. This includes the structure, weight initialisation and mathematics. Also, the setup of the experiment where the two neural networks are compared is explained.

2.1 Data

The data set is drawn from a 2-dimensional Gaussian distribution. The code used for generation the data is added to appendix A. We generated 500 data points with 2 features. Figure 1, shows a scatter plot of the data samples. The scatter plot show a quite good distinction between the samples of class 0 (yellow dots) and class 1 (purple dots). We have a balanced data set with 250 data points are belong to class 0, and 250 are belong to class 1. Classes are separable in multidimensional space (in our case, 3-dimensional space is enough for separating the classes). Therefore it should be possible to train a classifier on this binary classification problem.

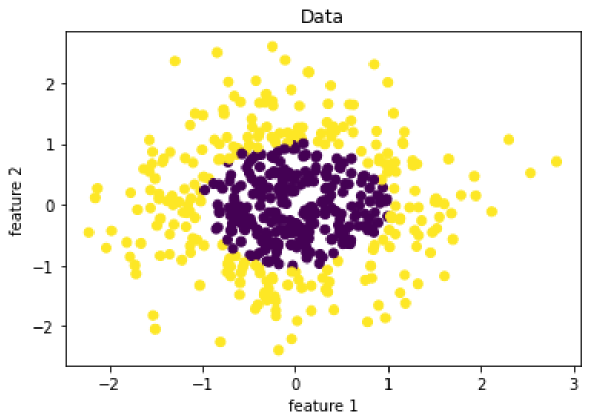

Figure 1: A scatter plot of the distribution of the data set. The samples of class 0 are marked with yellow and the samples of class 1 are marked with purple.

2.2 Neural network implementation

For the purpose of the comparison, experiment with a 3-layer neural (one input layer with 2 units, one hidden layer with 8 units and one output layer with one unit) network has been designed. The network's setup makes it possible to dynamically configure the number of perceptron units (nodes) in the hidden layer. We tried different number of hidden layer units (8, 28, 56 and 112), and observed that performance of the classifier (accuracy) dropped significantly when the number of hidden layer increases. therefore, we decided to use 8 hidden units in the hidden layer.

We split our data set into training and test set, which has 400 and 100 samples respectively. We initialized an neural network class called "NN" with initializing weights and biases. To test the working of the networks, we used 10 different weight initialization kernels and compare of average of the accuracy scores. The 10 different weight initializations are described in Table 1.

Uniform, min value:0, max value:1
Uniform, min value:-1, max value:1
Uniform, min value:-0.1, max value:0.1
Uniform, min value:0, max value:1
Uniform, min value:-0.05, max value:0.05
Normal, mean:0, standard deviation:1
Normal, mean:0, standard deviation:0.5
Normal, mean:1, standard deviation:1
Normal, mean:-1, standard deviation:1
Normal, mean:0.5, standard deviation:1

Table 1: Kernel initializer settings used for initialisation of the weights in the neural networks.

We implemented the forward pass between layers using the Equation 1. W^T, x and b represent weight vector, input vector and bias (which is a constant, in our case, it is 1 initially) respectively. σ represents the activation function, which is ReLU (see Equation 4) between input layer and hidden layer, and, sigmoid (see Equation 2) between hidden layer and output layer in our experiments. As a loss function, we used mean squared error (MSE) (see Equation 3). In Equation 3, N, Y_i and $\hat{Y}_i$ represent number of data points, observed values and predicted values respectively.

$$h = \sigma(W^T x + b) \tag{1}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

$$MSE = \frac{1}{N} \sum_{N}^{i=1} (Y_i - \hat{Y}_i)^2 \tag{3}$$

$$ReLUs(x) = max(0, x) \tag{4}$$

Then, we implemented the backward pass algorithm using Stochastic Gradient Descent (SGD) algorithm. We trained our network with 100 epochs setting the $learning_rate$ to 0.05. We run out algorithm 10 time, and took the average of the errors and accuracies for training and test set. Results can be seen in Table 2.

3 Results

We plotted Figure 2 and Figure 3, to demonstrate change in the accuracy and error in the network for both our own ANN implementation and Keras implementation respectively. In Table 2, we showed accuracy and error averages with their standard deviations for both networks mentioned above.

Metric	Our implementation	Keras
train error avg	0.17	0.02
train error std	0.09	0.01
train accuracy avg	0.71	0.96
train accuracy std	0.2	0.03
test error avg	0.18	0.02
test error std	0.09	0.01
test accuracy avg	0.68	0.97
test accuracy std	0.2	0.03

Table 2: Mean and standard deviation scores of the self developed and Keras neural network implementations for 10 different kernel initializations.

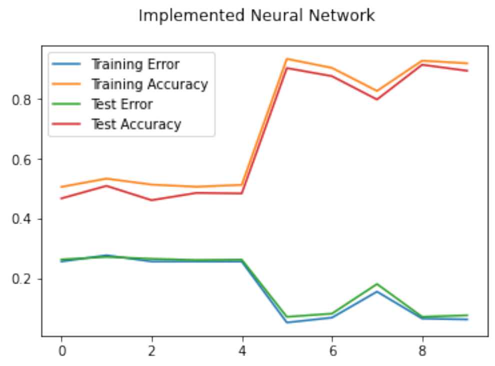

Figure 2: The train and test score for a 100 epoch training procedure of our own neural network implementation.

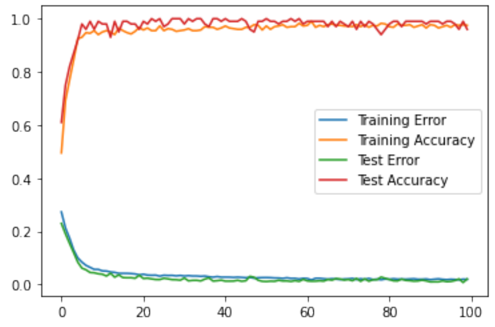

Figure 3: The train and test score for a 100 epoch training procedure of the Keras neural network implementation.

4 Discussion

During the experimentation, we have realized important things. How we initialized the weight and biases have significant affect on training and test set accuracies and how loss changes during each epoch. That is why in our implementation we initialized weights diffrently each time and try to get reasonable accuracy compared to Keras implementation. Second important thing is which activation function to use. Initially we tried sigmoid activation function for both between input and hidden layer, and, hidden layer and output layer. We saw that accuracy was stuck around 0.5. Then we decided to use ReLU activation function between input and hidden layer to observe any increase in the accuracy. It worked quite well, and we believe that it has to do with reduced likelihood of vanishing gradient. The another important point is how top set learning rate and epochs to run. We realized that higher learning rate converges quickly so model cannot learn the classification task properly (accuracy becomes too low). If it is too low, it does not converge at all. For model to learn the task, epoch size should be set to correct value as well (not too low since model cannot be able to learn the task fully).

References

[1] Module: tf.keras. 2022.

A Data Generation

The data that is used for the experiments in this report were generated with the code shown below.

```
import sklearn.datasets

def load data ():
    N = 500
    gq = sklearn.datasets.make_gaussian_quantiles(
    mean=None ,
    cov=0.7 ,
    n_samples=N,
    n_features=2,
    n_classes=2,
    shuffle=True ,
    random_state=None)
    return gq
```