Neural Network From Scratch - Report

Knut Raknes

September 24, 2025

UNIVERSIDADE FEDERAL
DE SANTA CATARINA
Department of Automation and Systems

Contents

1 Summary and Introduction

The goal of this project was to deepen my understanding neural networks by creating a multi-layer neural network from scratch using only the python library numpy. The task including implementing both forward and backwards-propagation algorithms. Then the network were to be used on a dataset "classification2.txt" whith two input features to classify one output feature (binary classification). Following this, the network was trained to classify handwritten digits from 0-9 from the dataset "classification3.mat". After training my network on "classification2.txt" I acheived a test accuracy of 79.17%. On "classification3.mat" I acheived a test accuracy of 77.1%.

This report will cover how the project is structured and the results acheived. [1]

2 Project structure

This section outlines how the project is structured.

2.1 Model.py

The file Model.py includes the code for the Neural Network class. The Neural Network class has 3 inputs arguments and 1 default argument that chooses if you want a sigmoid or a softmax output from the neural net. An untrained NN-model is initialized as follows: NN = NeuralNet(n_input, hidden_layers, n_outputs, output="sigmoid").

During initialization, the matrix **NN.W** is set that holds the weight values for each layer. The weights were initialized using the Xavier initialization.

It also sets up the list, NN.activations, that holds every activation for each layer in the network that gets calculated during the class member function **NN.forward_prop(x)**, where x is the feature vector.

The model also holds the member functions **NN.calculate_small_d**, that calculate the error terms and **NN.calculate_big_Delta** that is used for calculating the gradient of the cost function as seen here:

$$\begin{aligned} \triangle_{ij}^{(l)} &:= \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \\ D_{ij}^{(l)} &:= \tfrac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0 \\ D_{ij}^{(l)} &:= \tfrac{1}{m} \triangle_{ij}^{(l)} \qquad \text{if } j = 0 \end{aligned} \quad \Bigg| \quad \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Figure 1: Calculation of gradient where δ is calculated from NN.calculate_small_d() and a_j is activation in layer j stored in NN.activations.

The function returns Δ and that is again used for calculating D. This is then used in the function: NN.update_weights which updates the weights according to gradient descent using the calculated gradient.

2.2 grad_check.ipynb

The notebook grad_check.ipynb aims to approximate the gradient of the costfunction with the respect to the weights of the weight matrix NN.W. It does so by using the following formula:

$$\frac{\partial J}{\partial w_{ij}^{(l)}} \approx \frac{J\big(w_{ij}^{(l)} + \varepsilon\big) - J\big(w_{ij}^{(l)} - \varepsilon\big)}{2\,\varepsilon}, \quad \varepsilon > 0,$$

where $w_j^{(l)}$ is the weight from node i in layer l to node j in layer $l+1$.

2.3 binary_classification.ipynb and image_classification.ipynb

These notebooks show the implementations of the model where binary_net is the application of "classification2.txt" and image_classification is the implementation of "classification3.mat". In these notebooks the data is processed and prepared and the networks trained. They also include plots of their respective loss functions over epochs and binary_net also includes a plot of the descision boundary.

3 Data Preparation

The data preparation process involves several key steps to ensure that the datasets are properly formatted and ready for training the neural network models.

3.1 Training and Test Splits

The dataset is split into training and test sets, with splits of 80-20. The data is shuffled to ensure randomness in the train-test split, which helps in training the model effectively.

3.2 Feature Scaling

In the classification tasks, the features are standardized by mean normalization. This involves subtracting the mean of each feature and dividing by the standard deviation. This scaling process ensures that the features are on a similar scale, which improves the convergence of the neural network.

4 Results

This section shows the results acheived and experiments eith different hyperparameters.

4.1 Analytical vs. approximated gradient

To compare the difference between the analytical and numerical gradients I use the following formula:

$$E_{ij}^{(l)} = A_{ij}^{(l)} - N_{ij}^{(l)}, \qquad \max|E^{(l)}| = \max_{i,j}|E_{ij}^{(l)}| = \max_{i,j}|A_{ij}^{(l)} - N_{ij}^{(l)}|.$$

Where $A^{(l)}$ is the analytical gradient matrix for layer l and $N^{(l)}$ the numerical one.

```
X_batch = np.array([[1,2,3], [4,5,6], [7,8,9]])
y_batch = np.array([[1], [0], [1]])


NN_GradCheck = NeuralNet(X_batch.shape[1], [2,2,3],y_batch.shape[1])
diffs= grad_check(NN_GradCheck, X_batch, y_batch, epsilon=1e-5)
```

Figure 2: Dummy dataset used for testing analytical gradient

```
Layer 0: max |E|=2.454e-11
Layer 1: max |E|=1.185e-11
Layer 2: max |E|=1.969e-11
Layer 3: max |E|=1.203e-11
```

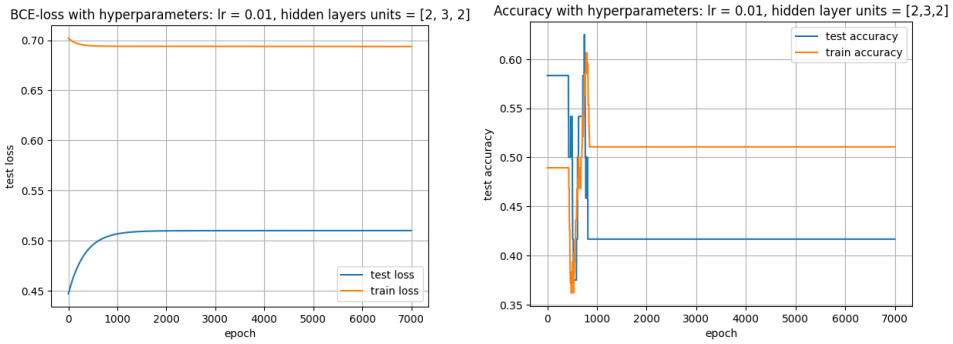
Figure 3: Output

Since the maximum error between the numerical and analytical gradient matricies is extremely close to 0 we can conclude that the analytical gradient computation is correct.

4.2 Binary classification

Experiment 1 - lr = 0.01- hidden layerunits = [2,3,2]

Similar results to **Experiment 1**



(a) loss plot (b) accuracy plot

Figure 4: Experiment 1

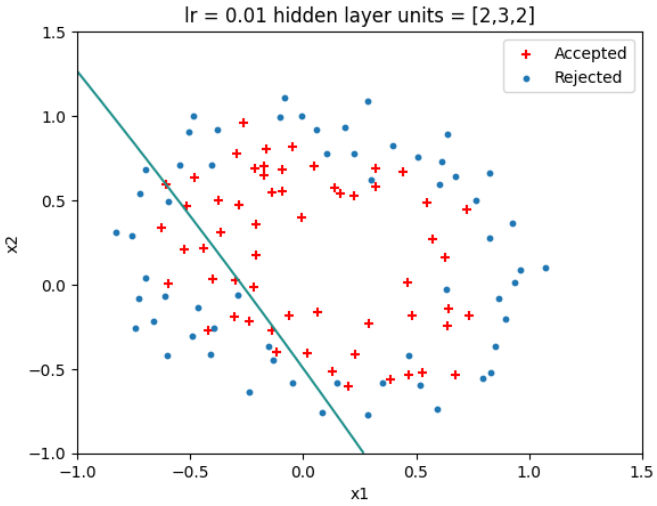
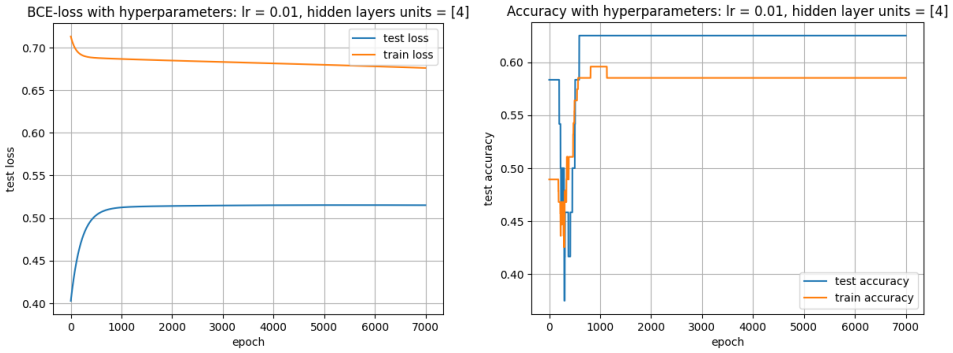


Figure 5: Decision boundary experiment 1

Experiment 2 - lr= 0.01 - hidden layer units = [4]

This experiment was conducted with only one hidden layer with 4 units and a learning rate of 0.01.



(a) loss plot (b) accuracy plot

Figure 6: Experiment 2

As shown in figure 6 both test and training loss and accuracy converges at around 1000 epochs. This means that it the gradient descent algorithm has reached a local minimum and the learning rate and small gradient makes almost no changes to the weights over the epochs.

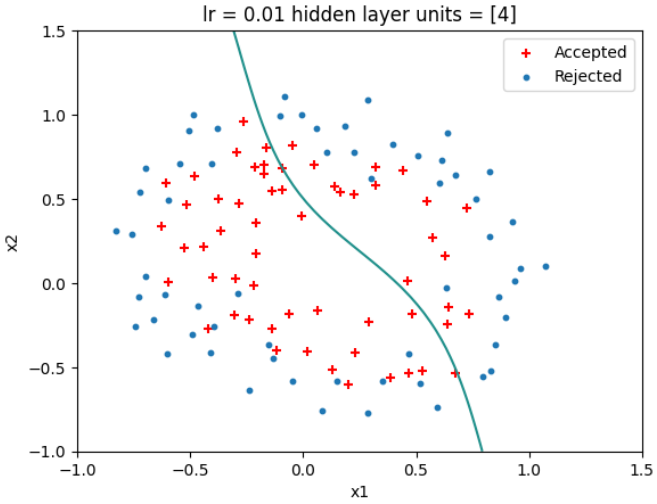
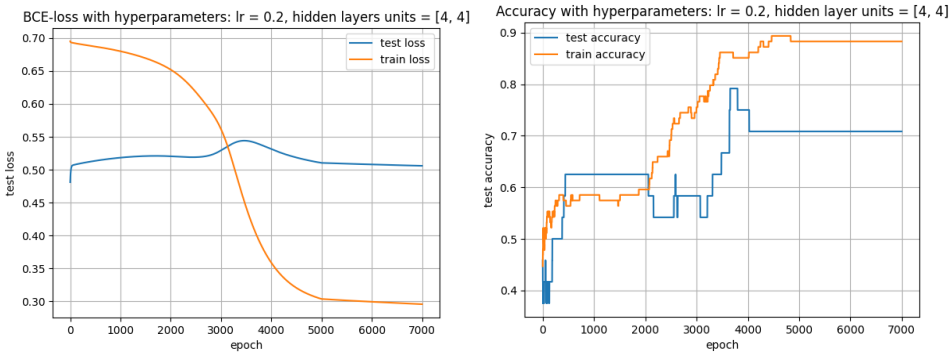


Figure 7: Decision boundary experiment 2

Experiment 3 - lr = 0.1 - hidden layer units = [4, 4]

This experiments holds my **best result** that I acheived for the binary classification task using "classifiaction2.txt". It is shown in figure 8 (experiment 3) Here I acheived a test accuracy of 79.17% after training for 3655 epochs.



(a) loss plot (b) accuracy plot

Figure 8: Experiment 3

In this experiment the learning rate was lowered to 0.04 after 5000 epochs, but I choose the best trained model at 3655 epochs. This is because training accuracy did not improve after training for more epochs while training accuracy improved. The model became overfitted.

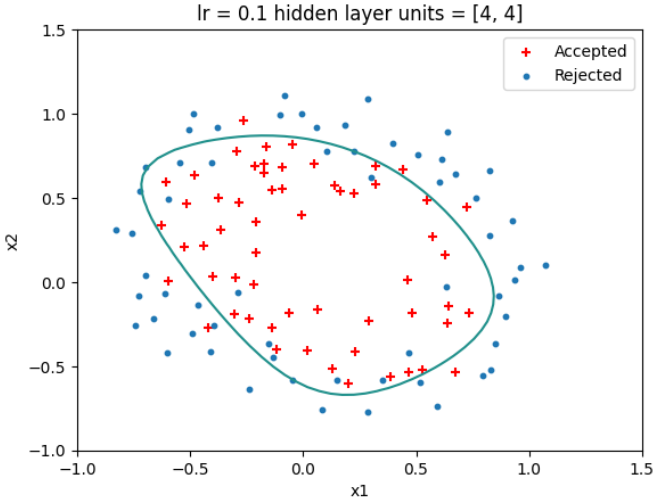


Figure 9: Decision boundary experiment 3

Table of all results

This table 2 shows the characteristics of the best model from each experiment.

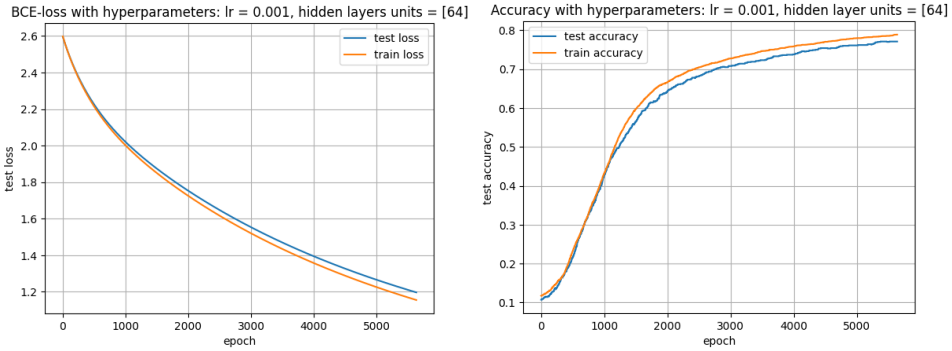
Table 1: Results for Experiments 1–3

Experiment	Train Loss	Train Acc	Test Loss	Test Acc	Best Epoch
1	0.6941	56.38%	0.5030	0.625%	742
2	0.6865	59.57%	0.5124	62.50%	1000
3	0.4100	86.17%	0.5414	79.17%	3655

The decision boundaries from figure 5, 7, 9 are also plottet using the results from the table below.

4.3 Multi-class classification

lr = 0.001 - hidden layer units = [64]



(a) loss plot

(b) accuracy plot

Figure 10: Image classification

As seen in the figure above, both test and train loss decrease in a similar rate, with a slightly larger decrease for the train data. After 5000+ epochs the test and train accuracy converge but the difference between them increase, which is why I stopped training to avoid overfitting. I did not experiment with hyperparameters for this experiment due to the long training times (120 minutes for 5000 epochs), which also shows that my model is quite computationally inefficient.

Table 2: Results for image classification

Experiment	Train Loss	Train Acc	Test Loss	Test Acc	Best Epoch
1	1.1547	78.88%	1.1964	0.771%	5130

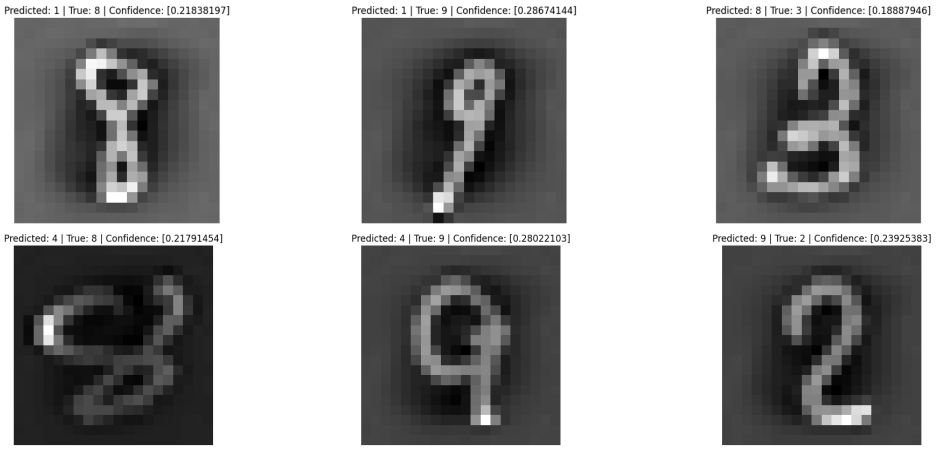


Figure 11: Examples of misclassified images

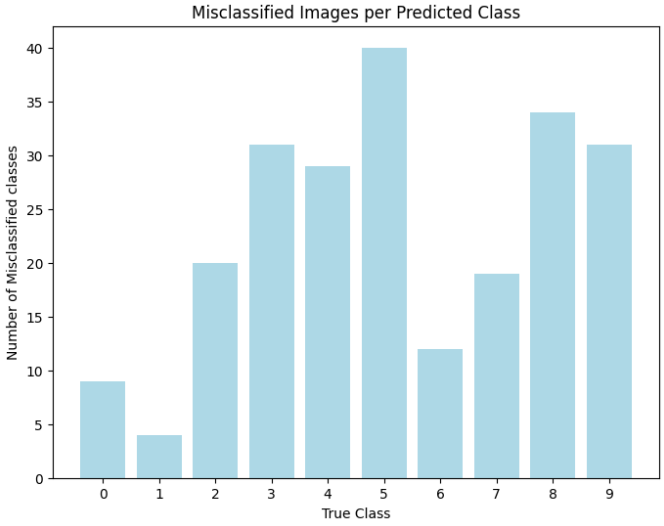


Figure 12: Frequency of misclassified images

The barplot in figure 12 displays which classes gets misclassified the most, showing that my model has the biggest struggle with classifying the number 5 and easiest classifying the number 1. This indicates that my model could be biased towards classifying the number 1.

5 Conclusion

The neural network model was successfully implemented from scratch using numpy, achieving a test accuracy of 79.17% on a binary classification task and 77.1% on a multi-class classification task. The model's performance varied with hyperparameters, showing that smaller learning rates and optimized hidden layers improved accuracy. However, the model struggled with certain classes, like '5', indicating a possible bias in classification. Further optimizations, including hyperparameter tuning, could improve its performance and efficiency

References

[1] Eric Aislan. Redes neurais, 2025. [Online; read 24.09.2025].