Neural Networks - Deep Learning 1st Project

Iordanis Kokkinidis - 3412

This is the first project for the Neural Networks - Deep Learning course in the Computer Science Department of the Aristotle University of Thessaloniki.

In this project we are going to create a Multilayer-Perceptron Neural Network from scratch, using python and the NumPy library, in order to classify the handwritten digits from the MNIST dataset of handwritten digits. This is a classic classification problem in the field of Deep Learning.
Below, we will talk a little bit about the dataset, the Neural Network we created and how we implemented its algorithms, while also comparing their performance to the SKLearn classifiers we used in the intermediary project.

We will be building a basic Deep Neural Network with 4 layers in total: 1 input layer, 2 hidden layers and 1 output layer. All layers will be fully connected.

MNIST dataset

The MNIST dataset of hand written digits is a dataset that consists of 70,000 grayscale images of 28×28 pixels, that represent a digit (from 0 to 9). 60,000 of these images are part of the training set, while 10,000 of these images are part of the test set. Each pixel has a value between 0 - 255 that represents the grayscale value of the pixel (with 0 being black and 255 being white). Therefore, each image is represented by a 28×28 matrix.

For simplicity's sake, we will import the dataset from keras.

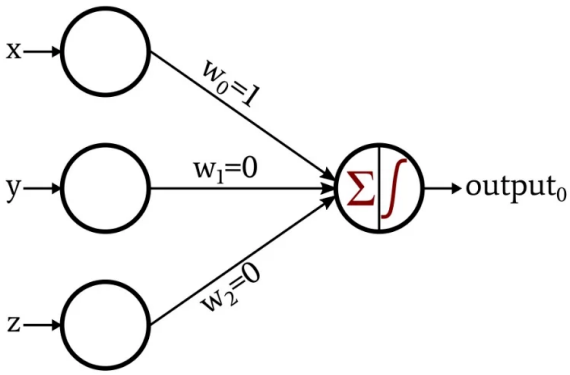
```
from keras.datasets import mnist
```

Neural Network

Artificial neural networks, usually simply called neural networks, are computing systems inspired by the biological neural networks that constitute biological brains. An Artificial Neural Network is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. In our neural network, the "neurons" are based on the mathematical model called ***Perceptron***.

Perceptron

The perceptron is a mathematical model of a biological neuron. While in actual neurons the dendrite receives electrical signals from the axons of other neurons, in the perceptron these electrical signals are represented as numerical values. At the synapses between the dendrite and axons, electrical signals are modulated in various amounts. This is also modeled in the perceptron by multiplying each input value by a value called the weight. An actual neuron fires an output signal only when the total strength of the input signals exceed a certain threshold. We model this phenomenon in a perceptron by calculating the weighted sum of the inputs to represent the total strength of the input signals, and applying a step function on the sum to determine its output. As in biological neural networks, this output is fed to other perceptrons.



Network Architecture

Let's try to define the architecture of our network. To be able to classify digits, we must end up with the probabilities of an image belonging to a certain class. The class with the highest probability will be the network's prediction. If we compare the probabilities given as output by the network with the correct label of a digit, then we can quantify how well our neural network is able to classify the digits. This will be helpful when we want to train the network, because the network will use it's wrong predictions and "*learn from its mistakes"*.

- ***Input Layer:*** In this layer we input our dataset, consisting of 28x28 images. Therefore, we flatten these images into one array (or vector) of 784 elements, one for each pixel of our picture. Therefore, our input layer will have 784 nodes.
- ***Hidden Layers:*** Our network consists of two hidden layers, one with 256 and one with 128 neurons. These hidden layers are fully connected to one another, as well as to the input and output layers. Each of the hidden layers has an Activation function in order to pass its the output $(Z = X \times W^T + B)$ through this function, so that we can solve the non-linear problem of the handwritten digit classification. In our case, the Activation function is ***ReLU***.
- ***Output Layer:*** The output layer consists of 10 neurons, each representing one class of our classification problem (digits 0-9). The output of this layer is again passed through an Activation function, this time through the ***Softmax*** function, that produces numbers between 0 and 1, in order to represent the probability of each class. The neuron with the highest value (therefore the class with the highest probability) is the networks prediction.
- ***Activation Funcions:*** As mentioned above, the two activation functions for our network will be *ReLU* for the hidden layers and *Softmax* for the output layer.
- ***Loss Function:*** The loss function we are going to use is ***Categorical Cross Entropy***.

Imports

We start by importing some standard libraries that we will need. We import numpy, so that we can use it for storing our data in np.arrays (matrices and vectors) and other linear algebra calculations. We import pyPlot, that we are going to use for creating our charts. Last but not least, we import the mnist dataset from keras.datasets, which holds 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.
We also import some utility libraries like *time* and *random*.

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
import time
import random

%matplotlib inline
```

Loading the data

As in the Intermediary project, again we load the data from the `mnist.dataset`, using `mnist.load_data()` and store each NumPy array in the appropriate NumPy array. For example, we have `X_train` containing the images of the training set and `y_train` containing the labels of the training set.

We then reshape the `X_train` and `X_test` NumPy arrays from $(60000, 28, 28)$ and $(10000, 28, 28)$ to $(60000, (28 \times 28))$ and $(10000, (28 \times 28))$ respectively, basically transforming each image's 28×28 pixel matrix to a a vector.
We also keep the original dataset in the `X_train_og` and `X_test_og` NumPy arrays, in order to use them later when we want to plot the (28×28) images.

```
def load_data():
  (X_train, y_train), (X_test, y_test) = mnist.load_data()
  X_train_og = X_train
  X_test_og = X_test

  X_train = X_train.reshape(60000, (28*28))
  X_test = X_test.reshape(10000, (28*28))

  return X_train, y_train, X_test, y_test, X_train_og, X_test_og
```

Utillity Functions

Below we define two utillity functions that will help us with one-hot encoding and batches.

The `to_categorical()` function

This function is used to convert the label arrays *y* in to *one hot encoding* ***labels*** *(e.g., 4 = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0])*. It takes as as parameters a numpy array `x` and the `number_of_columns` that each label will have (10). The function creates a numpy array of zeros (`np.zeros((x.shape[0], number_of_columns`) of shape [`x.shape()` , `number_of_columns`] and adds a 1 to the numpy array cell that is appropriate to create the `one_hot` label.

The `batch_loader()` fucntion

This function generates batches for the training of the Neural Network. It creates a numpy array with a size of `batch_size`, which by default is 64. Every batch of 64 samples is yielded, untill the whole of the dataset has been loaded in batches.
Batches are used in the training of the Neural Network, so that we do not calculate the gradient for each data sample, since this would take a lot of time.

```
def to_categorical(x, number_of_columns=None):
    if not number_of_columns:
        number_of_columns = np.amax(x)+1

    one_hot = np.zeros((x.shape[0], number_of_columns ))
    one_hot[np.arange(x.shape[0]), x] = 1
    return one_hot

def get_accuracy(y, pred):
    return np.sum(y == pred, axis = 0) / len(y)

def batch_loader(X, y = None, batch_size=64):
  """ Generates batches for training"""
  samples = X.shape[0]
  for i in np.arange(0, samples, batch_size):
    begin, end = i, min(i + batch_size, samples)
    if y is not None:
      yield X[begin:end], y[begin: end]
    else:
      yield X[begin:end]
```

Preprocessing the Data

We use the function defined above to load our data. We then preprocess our data, using the `to_categorical()` function defined above for our labels and dividing our data samples by 255, in order to normalize their values between 0 and 1.

```
X_train, y_train, X_test, y_test, X_train_og, X_test_og = load_data()
y_train, y_test = to_categorical(y_train.astype("int")), to_categorical(y_test.astype("int"))
X_train = X_train / 255
X_test = X_test / 255
```

Showing images of our data

*The function below is used to show an image of our data, given an **index** (which sample we want to show) and the **X** and **Y** NumPy arrays of the data set*. We must use the **X_train_og** and **X_test_og** arrays, since we kept them in dimensions of *(60000, (28 * 28))* and *(10000, (28 * 28))* respectively. We use the *matplotlib library* in order to plot the images of our data samples.

```
def image_show(index, X, Y):
    image = X[index]
    fig = plt.figure
    plt.imshow(image, cmap='gray')
    print("Showing sample data "  + str(index) +  " with label: "+ str(np.argmax(Y[index]))),
    plt.show()
```

Printing the predictions

We have defined a method below, that prints the predictions of our Neural Network. The function takes as parameters the predictions array, which is the predictions of the neural network we want to print, the X_test_og array, in order to plot the image of the samples if we want, the y_test which is the array that holds the labels of our dataset, and two boolean variables called print_wrong_predictions and print_correct_predictions, so that we can specify what kind of predictions (or both) we want to see. It also takes a number_of_predictions parameter, that specifies how many predictions we want to see. This number must be between 1 and 10,000, although setting it to a large number might not be the optimal choice.

In this function, we use a pseudo-random number generator to generate a number between 0 and 10,000, that is the index of the data sample we are going to print, in order to show a number of different predictions each time and not all of them, since that will take some time and the output will not be very readable.

When an index is generated, we check if the `predictions[index]` is equal to the `y[index]` in order to determine if the prediction was correct or not. This is not only done so that we can accompany the image of our data sample with the appropriate label (i.e., the prediction was CORRECT!), but also to allow the user to only print correct or wrong predictions if he wishes.

This works better if you run the python file and not the notebook

```
def print_predictions(predictions, X, y, print_wrong_predictions, print_correct_predictions, number_of_predictions_to_print):
  for i in range(0, number_of_predictions_to_print):
    index = random.randrange(0,10000)
    if(np.argmax(predictions[index]) != np.argmax(y[index])):
      if(print_wrong_predictions):
        image_show(index,X, y)
        print("Neural Network's prediction: " , np.argmax(predictions[index]))
        print("The prediction was WRONG!")
        print("The correct label is: ", np.argmax(y_test[index]))
        print("\n")
    else:
      if(print_correct_predictions):
        image_show(index,X, y)
        print("Neural Network's prediction: " , np.argmax(predictions[index]))
        print("The prediction was CORRECT!")
        print("\n")
```

Getting the wrong predictions

The function below is used to get the wrong predictions and the number of wrong predictions that the neural network has made. It takes as parameters the predictions of the network and the y (e.g., y_test) array, which holds the labels for our test data. We iterate through the predictions array `for i in range(1, 10000)` and check if the `predictions[i]` is not equal to the `y[i]`. If yes, then the `wrong_predictions_total` is incremented and that prediction is put in the wrong_predictions array. We finally return the `wrong_predictions` array and the `wrong_predictions_total`.

```
def get_wrong_predictions(predictions, y, set_size=10000):
  wrong_predictions = []
  wrong_predictions_total = 0
  for i in range(1, set_size):
    if((np.argmax(predictions[i])) != np.argmax(y[i])):
      wrong_predictions_total += 1
      wrong_predictions.append(i)
  return wrong_predictions, wrong_predictions_total
```

Neural Network Utillity functions

Finally, we will talk about utillity functions that we have defined to help us with the training and testing of our Neural Network, such as the loss and activation functions. Afterwards, we will discuss about the code for our Neural Network.

Loss function

We are using Categorical Cross Entropy for our loss function. we create a `CrossEntropy()` class, that we will put in our neural network structure and use it to calculate the loss of our neural network. This class has two methods, apart from the constructor. the `loss()` method, is used to calculate the loss of the neural network based on the Cross Entropy formula. It takes the labels array y and the prediction array as parameters and returns the loss for those predictions :

$$Loss = -y \times \log(\hat{y}) - (1-y) \times \log(1-\hat{y})$$

The other method that we have in the `CrossEntropy` class, is the `gradient()` function, which calculates the derivative of the loss (derivative of the Categorical Cross Entropy function):

$$\frac{dE}{dy} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

```
#loss and activation functions
class CrossEntropy():
    def __init__(self):
        pass

    def loss(self, y, prediction):
        prediction = np.clip(prediction, 1e-15, 1- 1e-15)
        return -y*np.log(prediction) - (1 - y) * np.log(1 - prediction)

    def gradient(self, y, prediction):
        prediction = np.clip(prediction, 1e-15, 1- 1e-15)
        return -(y/prediction) + (1-y)/(1-prediction)
```

Activation Functions

For our Activation functions, we will use the ReLU function for our hidden layers and the SoftMax function for the output layer.

ReLU

The ReLU or Rectified Linear Unit function, is used to pass the output of a neuron (or layer of neurons) through an activation function to give the ability to our Neural Network to solve non linear problems, like that of the classification problem with 10 classes. We have created the `ReLU()` class, that has the `activation()` and `derivative()` methods.
The `activation()` method takes the output of a layer `x` and implements the ReLU function, by returning the maximum number between 0 and x (`np.maximum(0,x)`), since ReLU is 0 for negative values of x and equal to x for positive values of x.
The `derivative()` method, calculates the derivative of the ReLU activation function with respect to the output of the layer `x`.
The `derivative()` method returns 1 for positive values of x and 0 for negative values of x (`np.where(x >= 0, 1, 0`)

SoftMax

The Softmmax function is used in the output layer, in order to put the values of the outcome of our Neural Network, or predictions if you will, in range of 0 - 1, signifying the probability that each class has to be the correct class for the prediction. Like the `ReLU()` class, we have two methods, the `activation()` and the `derivative()` methods.
The `activation()` method takes the output of the output layer `x` and implements the softmax function: $\frac{e^x}{\sum e^x}$
The `derivative()` method calculates the derivative of the softmax function: $\frac{e^x}{\sum e^x} \times \left(1 - \frac{e^x}{\sum e^x}\right)$

```
class ReLU():
    def __init__(self):
        pass
    def __call__(self, x):
        return self.activation(x)
    def activation(self, x):
        return np.maximum(0,x)
    def derivative(self, x):
        return np.where(x >= 0, 1, 0)

class SoftMax():
    def __init__(self): pass

    def __call__(self, x):
        return self.activation(x)

    def activation(self, x):
        e_x = np.exp(x - np.max(x, axis = -1, keepdims=True))
        return e_x / np.sum(e_x, axis=-1, keepdims = True)

    def derivative(self, x):
        p = self.activation(x)
        return p * (1 - p)
```

Neural Network

Layer Classes

We have two kinds of Layer classes. One is the Dense layer of our fully connected Neural Network and one is the Activation Layer, which represents the activation functions of our neurons. Both of them have the same methods for `forward_pass()` and `back_propagation()`, but their function is quite different.

Activation Layer

The activation layer has several fields. An activation field, which holds one of the activation functions defined above (ReLU or Softmax), and the gradient (derivative of the error with respect to the output)
During the `forward_pass()`, the input from the `Dense` layer is passed through the activation function (`self.activation(x)`) and the output of that function is returned.

During `back_propagation()`, the gradient of the error is calculated (derivative of the error with respect to the output), `self.gradient(self.input) * output_error`. This gradient is returned to the previous `Dense` Layer of our Neural Network, in order to calculate the weights and biases gradient.

```
class Activation():
    def __init__(self, activation, name="activation"):
        self.activation = activation
        self.gradient = activation.derivative
        self.input = None
        self.output = None
        self.name = name

    def forward_pass(self, x):
        self.input = x
        self.output = self.activation(x)
        return self.output

    def back_propagation(self, output_error, learning_rate = 0.01):
        return self.gradient(self.input) * output_error

    def __call__(self, x):
        return self.forward_pass(x)
```

Dense Layer

The dense layer has fields for the parameters of our Neural Network (Weights and Biases). The weights and biases are initialized in the constructor of this class, where the weights are initialized randomly using `np.random` and the biases are initialized as 0, using `np.zeros`. During the `forward_pass()`, the input, which is either the input from the dataset for the input layer, or the output from the previous `Activation` layer (*meaning, the output of the previous (dense) layer passed through the activation function*) is used to calculate the output of this (dense) layer (`self.output = np.dot(self.input, self.weights) + self.biases`) based on the Z = W * X + B formula.

During `back_propagation()` the gradient that is passed from the next `Activation` layer as output_error is used to calculate the input, weights and biases gradients(biases gradient is the same as output_gradient). We use the `weights_gradient` and `output_error` (biases gradient) to update the parameters

```
    self.weights -= learning_rate * weights_gradient
    self.biases -= learning_rate * np.mean(output_error)
```

and then we return the input gradient (`input_error`) to the "previous" layer, so that back propagation can continue (use that output_error to calculate that layer's gradients and update the parameters).

Neural Network Class

Below, we create the `Neural_network()` class, which will be the mode of our neural network. The most important field that this class has, is the `layers` field, that is initialized in the constructor. The `layers` field is a list of all the layers in our Neural Network. During the initialization of the `layers` we create objects from the classes defined above as `Dense_layer` and `Activation`, passing the dimensions we want for the layers (and their output, which needs to be the same as the next layer's dimensions) as parameters. For this Network, we created 4 layers, one input layer with 784 neurons, two hidden layers with 256 and 128 neurons respectively, and an output layer with 10 neurons, one for each of the classes we want to classify our input as.
We can experiment with our network, by changing the number of hidden layers, or even the number of neurons in the hidden layers. One other thing we could do is implement a sigmoid activation function and use it as the `Activation` in our network.
We also pass the `learning_rate` as a parameter in the initialization of our network.
For the `Neural_network()` class, we define a `forward_pass()` and a `back_propagation()` method, as well as a `train()` method, where both of the previous ones are called from.

The `forward_pass()` method simply calls each layer (for the layer class `__call__` method, the `forward_pass()` method is called) with `x` being the batch of data we are passing through our neural network.

The `back_propagation()` method calls the `back_propagation()` method for each layer (in reverse order) passing the `learning_rate` and `loss_gradient` as a parameter. The `loss_gradient` is updated after each call to the layer's `back_propagation()` method and continues to be passed to the previous one.

The `train()` method, is called from the driver program and expects the number of epochs as a parameter. During training, we use the batch_loader to load batches in the `x_batch` and `y_batch` arrays. We calculate the output of our network for the batch, using the `forward_pass()`, and then we calculate the loss, accuracy and error of our network for that batch, using the `get_accuracy()`, `loss_function.loss()` and `loss_function.gradient()` methods, before we call on the `back_propagation()` method. This process is happening as long as the `batch_loader()` function yields batches and it is repeated for the number of epochs. After everything is done, we return the mean_loss, mean_accuracy and delta_time (time it took for the training process) to the driver program.

```
class Neural_network():
    def __init__(self, input_dimensions, output_dimensions, loss_function,  learning_rate = 0.01):
        #input dimensions is 784 and output dimensions is 10 for MNIST classification
        self.layers = [Dense_layer(input_dimensions, 256, name = "input"),
                       Activation(ReLU(), name = "relu"),
                       Dense_layer(256, 128, name = "hidden"),
                       Activation(ReLU(), name = "relu2"),
                       Dense_layer(128, output_dimensions, name = "output"),
                       Activation(SoftMax(), name = "softmax")
                       ]
        self.learning_rate = learning_rate
        self.loss_function = loss_function

    def forward_pass(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def back_propagation(self, loss_gradient):
        for layer in reversed(self.layers):
            loss_gradient = layer.back_propagation(loss_gradient, self.learning_rate)

    def __call__(self, x):
        return self.forward_pass(x)

    def train(self, epochs):

        mean_loss = []
        mean_accuracy = []
        for epoch in range(epochs):
            loss = []
            accuracy = []
            for x_batch, y_batch in batch_loader(X_train, y_train):
                out = self(x_batch) #Forward Pass
                loss.append(np.mean(self.loss_function.loss(y_batch, out))) #calculate loss (with cross entropy class)
                # print(get_accuracy(y_batch, out))
                accuracy.append(get_accuracy(np.argmax(y_batch, axis=1), np.argmax(out, axis=1))) #Network's accuracy
                error = self.loss_function.gradient(y_batch, out) #calculate gradient of loss (with cross entropy class)
                self.back_propagation(error) #Back Propagation


            # print(delta_time)
            mean_loss.append(np.mean(loss))
            mean_accuracy.append(np.mean(accuracy))
        return mean_loss, mean_accuracy
```

Driver Program

Below we will create an object of our Neural Network class, train it and test it, as well as print some of it's predictions and some charts regarding it's accuracy, time, and loss.

We start by initializing the values of some of our ***hyperparameters***, such as the number of `epochs`, `the loss_function` and the `learning_rate`. Those are all parameters that can be tweaked, in order to test a slightly different model of our Neural Network.

```
input_dimensions = 784 # (28 x 28)
output_dimensions = 10 # 10 classes (10 digits)
```

```
loss_function = CrossEntropy()
learning_rate = 0.001
epochs = 10

neural_network = Neural_network(input_dimensions, output_dimensions, loss_function, learning_rate)
```

We use `neural_network.train()` to train the neural network and we keep track of how much time it takes for the training of the neural network. We then print the mean_loss and mean_accuracy for each epoch of the training.

```
start_time = time.process_time()

mean_loss, mean_accuracy = neural_network.train(epochs)

end_time = time.process_time()
delta_time = (end_time - start_time) * 1000
training_time = round(delta_time)

for epoch in range(0,epochs):
  print(f"Epoch {epoch + 1}, Loss: {mean_loss[epoch]}, Accuracy:  {mean_accuracy[epoch]}")
print(f"Time it took to train the model with 10 epochs: {training_time}ms")
```

We should get results that are similar to these:

```
Epoch 1, Loss: 0.09619462461323508, Accuracy:  0.8446994936034116
Epoch 2, Loss: 0.040641301596010065, Accuracy:  0.9320195895522388
Epoch 3, Loss: 0.029244475225455653, Accuracy:  0.9514259061833689
Epoch 4, Loss: 0.02268330526316628, Accuracy:  0.9624367004264393
Epoch 5, Loss: 0.01839219114484902, Accuracy:  0.9699826759061834
Epoch 6, Loss: 0.015295919136598536, Accuracy:  0.9755463752665245
Epoch 7, Loss: 0.012964618545043929, Accuracy:  0.9793610074626866
Epoch 8, Loss: 0.011067057443133906, Accuracy:  0.982459355010661
Epoch 9, Loss: 0.0095083369381383, Accuracy:  0.9855077292110874
Epoch 10, Loss: 0.008217997411096996, Accuracy:  0.9877065565031983
Time it took to train the model with 10 epochs: 180812ms
```

Before we test the neural network, we will create two more test sets. One is the test set, but reversed and the other one is a random portion of the test set we get by using two random indexes.

It doesn't make a difference, since the order of the data samples is not important during testing, which is what we want to showcase here.

```
x_reversed = X_test.tolist()
y_reversed = y_test.tolist()
x_reversed.reverse()
y_reversed.reverse()

x_reversed = np.array(x_reversed)
y_reversed = np.array(y_reversed)

index1 = random.randrange(0,5000)
index2 = random.randrange(index1, 10000)

x_test2 = X_test[index1:index2]
y_test2 = y_test[index1:index2]
```

Now, finally we can get to testing our Neural Network with the test set and the two other sets we created above. We are using `neural_network()` to get the neural_network's predictions, passing a test set as a parameter (i.e., `X_test`, `x_reversed`, `x_test2`). We also use `get_accuracy()` to get the accuracy of our model in its predictions and we also get the wrong predictions by using the appropriate function defined above. We will also call on the `print_predictions()` method for each of the tests, to see some of the neural networks predictions.

```
print_wrong_predictions = False
print_correct_predictions = False
number_of_predictions_to_print = 10
```

```
predictions = neural_network(X_test)
accuracy1 = get_accuracy(np.argmax(y_test, axis=1), np.argmax(predictions, axis=1))
accuracy1 = round(accuracy1*100, 2)
print(f"Accuracy when testing the on the test set: {accuracy1}%")
wrong_predictions1, wrong_predictions_total1 = get_wrong_predictions(predictions, y_test)
print(f"{wrong_predictions_total1} wrong predictions out of 10,000 predictions")
print_predictions(predictions, X_test_og, y_test, print_wrong_predictions, print_correct_predictions, number_of_predictions_to_print)
print("\n")

predictions2 = neural_network(x_reversed)
accuracy2 = get_accuracy(np.argmax(y_reversed, axis=1), np.argmax(predictions2, axis=1))
accuracy2 = round(accuracy2*100, 2)
print(f"Accuracy when testing the on the reversed test set: {accuracy2}%")
wrong_predictions2, wrong_predictions_total2 = get_wrong_predictions(predictions2, y_reversed)
print(f"{wrong_predictions_total2} wrong predictions out of 10,000 predictions")
print_predictions(predictions, X_test_og, y_test, print_wrong_predictions, print_correct_predictions, number_of_predictions_to_print)
print("\n")

predictions3 = neural_network(x_test2)
accuracy3 = get_accuracy(np.argmax(y_test2, axis=1), np.argmax(predictions3, axis=1))
accuracy3 = round(accuracy3*100, 2)
print(f"Accuracy when testing the on a random batch of the test set ({index1} - {index2}): {accuracy3}%")
wrong_predictions3, wrong_predictions_total3 = get_wrong_predictions(predictions3, y_test2, (index2-index1))
print(f"{wrong_predictions_total3} wrong predictions out of {index2 - index1} predictions")
print_predictions(predictions, X_test_og, y_test, print_wrong_predictions, print_correct_predictions, number_of_predictions_to_print)
print("\n")
```

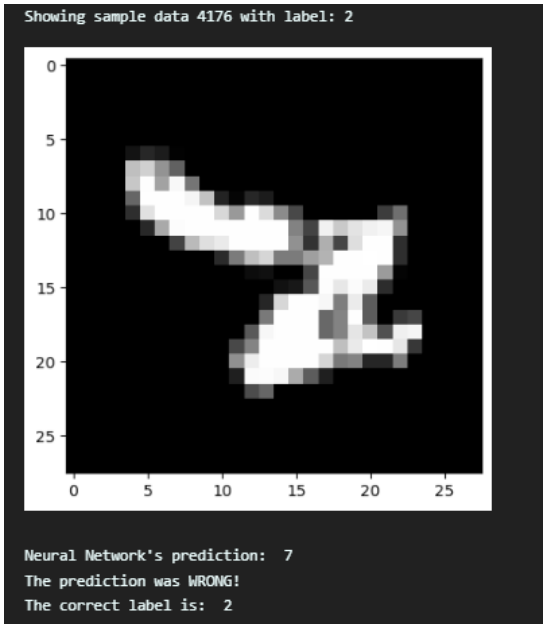
The results of the tests look something like this:

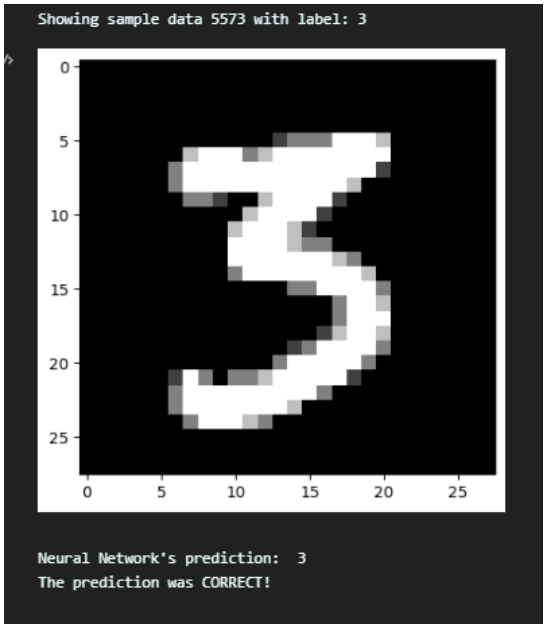
```
Accuracy when testing the on the test set: 97.64%
236 wrong predictions out of 10,000 predictions


Accuracy when testing the on the reversed test set: 97.64%
236 wrong predictions out of 10,000 predictions


Accuracy when testing the on a random batch of the test set (2221 - 7548): 97.67%
124 wrong predictions out of 5327 predictions
```

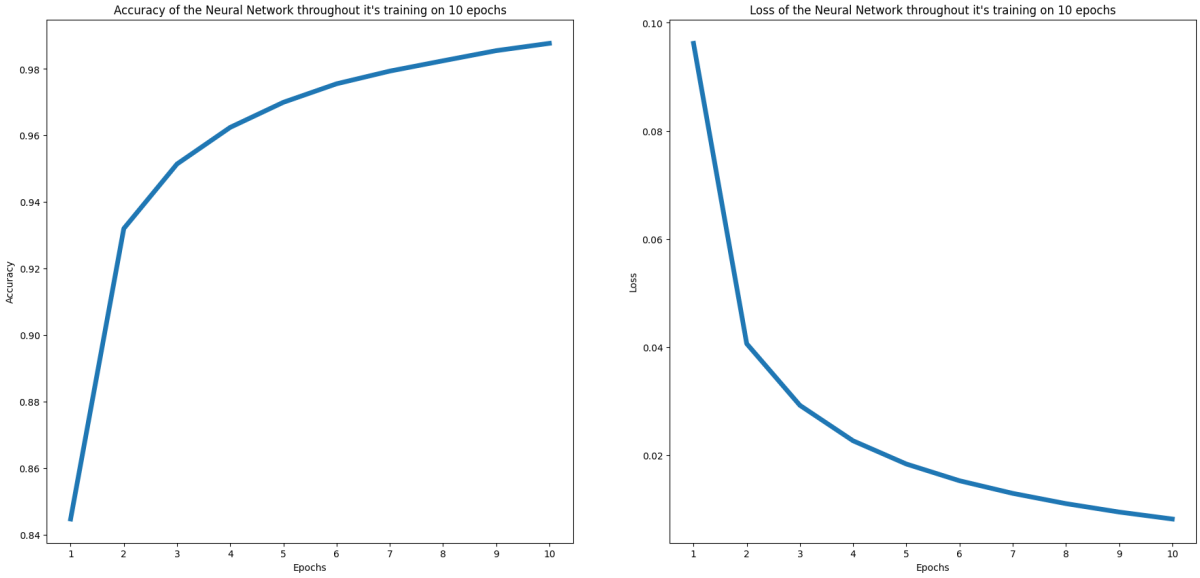
Some of the network's predictions





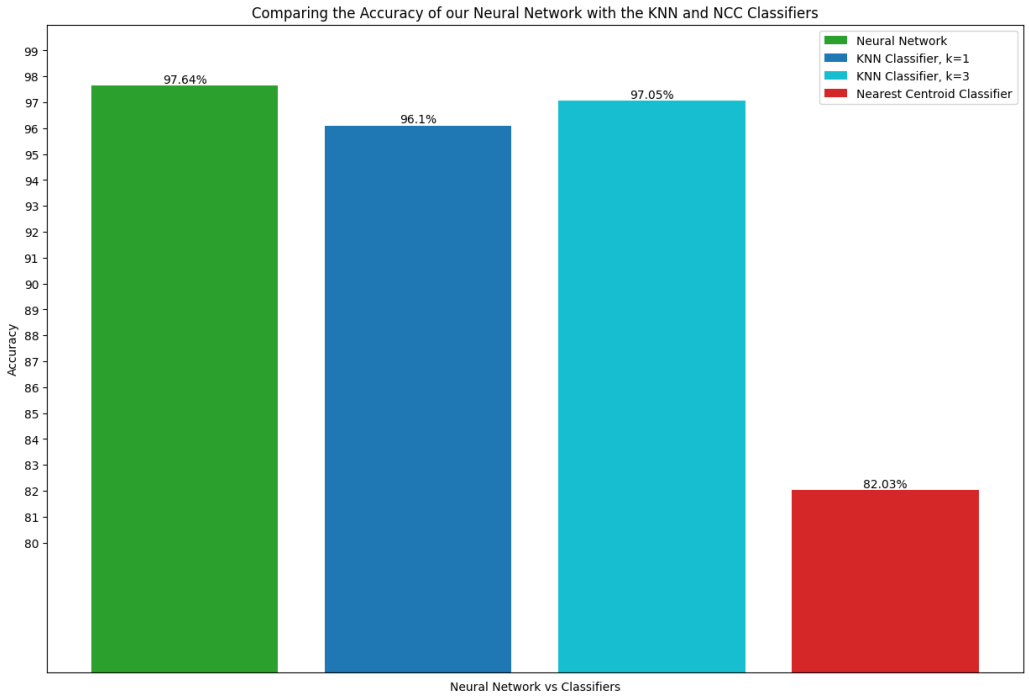
Results

Our Neural Network's Accuracy and Loss during the training of the network

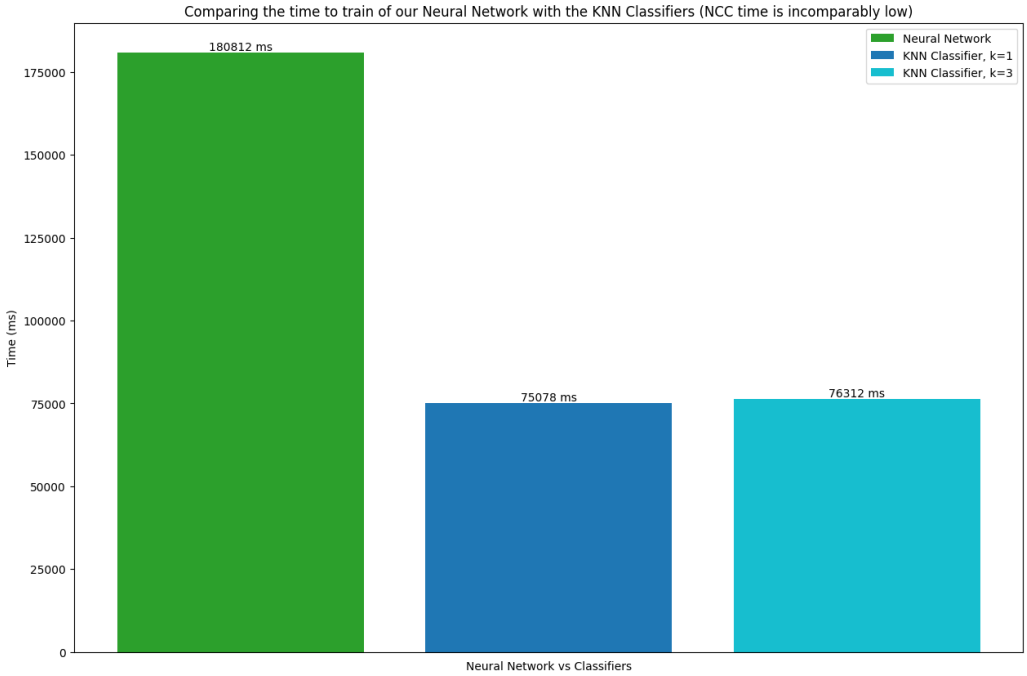


As we can see, training the network and using *Stochastic Gradient Descent*, we can minimize our loss significally in only 10 epochs, bringing it down to only 0.82%. Similar success can be observed with the Accuracy metric. It rises significally and we reach 98.77% accuracy in 10 epochs.

Comparing our Neural Network with the Machine Learning classifiers from the intermediary project



As we can see above, the neural network slightly outperforms the K Nearest Neighbor classifiers (*~2-0.5%*), while the Nearest Centroid Classifier is lacking behind (*~15%*).



The time it takes for our Neural Network to be trained is more than double the time that the two KNN classifiers needed to make their predictions. Although, the Neural Network only needs to be trained once and is a lot faster when predicting than the KNN classifiers.

Refferences

- Simon Haykin - Neural Networks and Learning Machines, Third Edition
- 3Blue1Brown - But what is a neural network? | Chapter 1, Deep learning
- 3Blue1Brown - Gradient descent, how neural networks learn | Chapter 2, Deep learning
- 3Blue1Brown - What is backpropagation really doing? | Chapter 3, Deep learning