

Name: Atanu Chatterjee

Roll Number: 222CS010

Course: Deep Learning (CS737)

Assignment Number: 01

Experiment Title: Hand Written Digit Recognition using
Artificial Neural Network (ANN)

Date of the Experiment: 14/11/2022



Department of Computer Science and Engineering
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA (NITK)
SURATHKAL, MANGALORE - 575 025
November 15, 2022

Contents

1	Introduction	1
2	About the Data Set	1
3	Preprocessing of Input & Output Data	2
3.1	Flattening the Input Images	2
3.2	One-hot encoding of Output values	2
4	Experiment Steps	3
4.1	Defining the Network	3
4.2	Training the Network	3
4.3	Testing the Network	4
5	Different Experiment Models	5
5.1	Model 1	5
5.1.1	Design	5
5.1.2	Result	5
5.2	Model 2	6
5.2.1	Design	6
5.2.2	Result	6
5.3	Model 3	6
5.3.1	Design	6
5.3.2	Result	7
5.4	Model 4	7
5.4.1	Design	7
5.4.2	Result	8
5.5	Model 5	8
5.5.1	Design	8
5.5.2	Result	9
6	Conclusion	10

1 Introduction

In this assignment we will do handwritten digit recognition using MNIST dataset with the help of Artificial Neural Network(ANN). First we will train the neural network with different image samples with resolution of 28 x 28. Then the network's performance will be evaluated in terms of test accuracy with respect to different network architecture with single or multiple layers of neurons with some optimization and regularization techniques.

2 About the Data Set

To do hand-written digit recognition we need a data set and for that we are importing MNIST data sets from python's keras datasets. We are getting 60,000 image samples for training and 10,000 samples for testing. The resolution of these sample images are 28 x 28 and pixel values of

```
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("number of training samples= ", len(X_train))
print("number of testing samples= ", len(y_test))

#normalizing pixel values
X_train = X_train / 255
X_test = X_test / 255
```

number of training samples= 60000
number of testing samples= 10000

Figure 1: data set detail

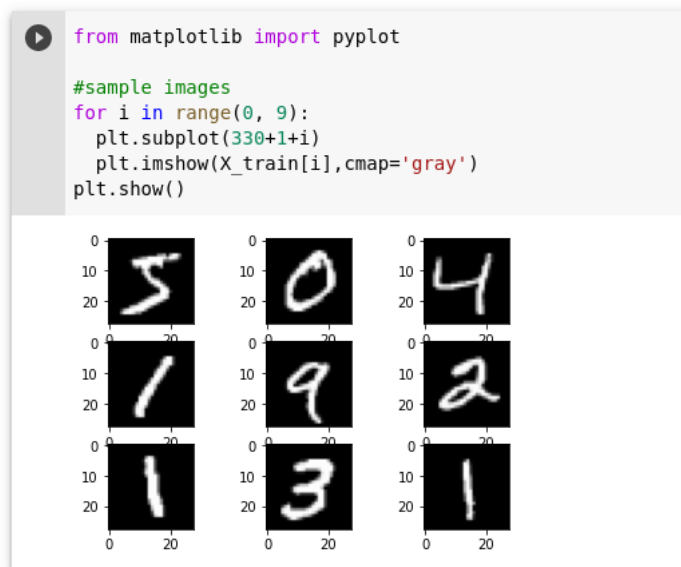


Figure 2: some image samples

these images are in the range of 0 to 255. For normalising these pixel values I am dividing it by 255 so that the pixels range can be reduced to 0 to 1 which will help the model for better convergence.

3 Preprocessing of Input & Output Data

3.1 Flattening the Input Images

Since the size of the images are 28 x 28 so we have to flatten this images to an 1D array of size 784 as we are using dense layer in this network so we will be providing this 1D array as input. The other reason for flattening is multi-Dimensional arrays take more amount of memory while 1D arrays take less memory

```
#flattening 28 x 28 input images to 784 x 1
print(X_train[0].shape)
X_trn = []
for i in X_train:
    X_trn.append(i.flatten())

X_tst = []
for i in X_test:
    X_tst.append(i.flatten())

print(X_trn[0].shape)

(28, 28)
(784,)
```

Figure 3: Flattening the input images

3.2 One-hot encoding of Output values

As we will be using `categorical_crossentropy` as loss function in this model we have to do one-hot encoding of output values. But this step can be avoided if we would have used `sparse_categorical_crossentropy`.

```
from tensorflow.keras.utils import to_categorical

#one hot encoding of output data
print(y_train[0])
train_labels = to_categorical(y_train)
test_labels = to_categorical(y_test)
print(train_labels[0])

5
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Figure 4: one-hot encoding of output values

4 Experiment Steps

First we will take a simple single layer neural network with 10 neurons. We are taking 10 neurons because we have 10 classes to predict. We will name this simple single layer model to Model 0.

4.1 Defining the Network

We are using the sequential model since in we need plain stack of layers where each layer's neuron has exactly one input and one output. For that importing *model* from *keras* library then we are initializing the netork by *model.Sequential()* function.

we have added 1 dense layer with 10 neurons. Activation function used *Softmax* and the input size used defined for this layer is 784 x 1. Total number of parameters in this network is 7850.

```
# simple single layer network with 10 neurons
from keras import models
from keras import layers

network = models.Sequential()

network.add(layers.Dense(10, activation='softmax', input_shape=(784,)))

network.build()
```

Figure 5: Model 0 network design

```
network.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 10)	7850

=====
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0

Figure 6: Model 0 network summary

4.2 Training the Network

In this step we will configure and train the model using *network.compile()* and *network.fit()* respectively. To configure the network we can pass some arguments like optimizer, loss function, matrices to be evaluated while training, loss weights etc.

For this architecture we are not using any optimizer. we are using *categorical_crossentropy* loss function and providing 'accuracy' argument so that model calculate the training and validation accuracy for us.

For training we will run this model for 25 epochs and the batch size we are taking is 5000. We also specified the the validation split of 0.2 that means 20% of training data will be kept for validation by the model. We are passing these parameters to *network.fit()* function along with training data and training labels.

For each epoch the model will perform training in batches for the fixed number of epochs specified by us. After training it returns a History object. Its *History.history* attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values.

```
[17] import tensorflow as tf
import numpy as np

network.compile(loss='categorical_crossentropy', metrics=['accuracy'])

hist = network.fit(tf.convert_to_tensor(X_trn), train_labels, epochs=25, batch_size=5000, validation_split=0.2)

model_data = hist.history

print('Train Accuracy: ', np.mean(model_data['accuracy']), '\nTrain Loss: ', np.mean(model_data['loss']))
print('Validation Accuracy: ', np.mean(model_data['val_accuracy']), '\nValidation Loss: ', np.mean(model_data['val_loss']))
```

```
Epoch 1/25
10/10 [=====] - 1s 40ms/step - loss: 1.9524 - accuracy: 0.3857 - val_loss: 1.6207 - val_accuracy: 0.6296
Epoch 2/25
10/10 [=====] - 0s 24ms/step - loss: 1.5019 - accuracy: 0.6690 - val_loss: 1.3123 - val_accuracy: 0.7455
Epoch 3/25
10/10 [=====] - 0s 24ms/step - loss: 1.2487 - accuracy: 0.7485 - val_loss: 1.1022 - val_accuracy: 0.7999
Epoch 4/25
10/10 [=====] - 0s 24ms/step - loss: 1.0684 - accuracy: 0.7873 - val_loss: 0.9474 - val_accuracy: 0.8238
Epoch 5/25
10/10 [=====] - 0s 24ms/step - loss: 0.9325 - accuracy: 0.8082 - val_loss: 0.8294 - val_accuracy: 0.8368
Epoch 6/25
10/10 [=====] - 0s 26ms/step - loss: 0.8276 - accuracy: 0.8237 - val_loss: 0.7376 - val_accuracy: 0.8490
Epoch 7/25
10/10 [=====] - 0s 25ms/step - loss: 0.7445 - accuracy: 0.8356 - val_loss: 0.6648 - val_accuracy: 0.8554
Epoch 8/25
10/10 [=====] - 0s 24ms/step - loss: 0.6776 - accuracy: 0.8452 - val_loss: 0.6058 - val_accuracy: 0.8652
Epoch 9/25
10/10 [=====] - 0s 24ms/step - loss: 0.6233 - accuracy: 0.8531 - val_loss: 0.5586 - val_accuracy: 0.8717
```

Figure 7: Model 0 training steps

In this step finally we have calculated the the average accuracy and loss values of training and validation datas.

Train Accuracy: 0.8393400037288665

Train Loss: 0.6553118538856506

Validation Accuracy: 0.8656500029563904

Validation Loss: 0.5858187258243561

```
Epoch 25/25
10/10 [=====] - 0s 23ms/step - loss: 0.3435 - accuracy: 0.9059 - val_loss: 0.3239 - val_accuracy: 0.9115
Train Accuracy: 0.8393400037288665
Train Loss: 0.6553118538856506
Validation Accuracy: 0.8656500029563904
Validation Loss: 0.5858187258243561
```

Figure 8: Model 0 training result

4.3 Testing the Network

Once our model is trained using 60,000 training samples now we will test the model using 10,000 samples. For testing we have used *network.evaluate()* function which return testing accuracy and testing loss. We need to pass testing input data and testing labels as input parameters for this function.

```

test_loss, test_acc = network.evaluate(tf.convert_to_tensor(X_tst), test_labels)
print('Test Accuracy: ', test_acc, '\nTest Loss: ', test_loss)

313/313 [=====] - 1s 2ms/step - loss: 0.3256 - accuracy: 0.9106
Test Accuracy: 0.9106000065803528
Test Loss: 0.32556846737861633

```

Figure 9: Model 0 testing result

5 Different Experiment Models

In last section we have taken single layer model with 10 neurons. There we see the performance of that model in terms of loss and accuracy of training data, validation data and testing data.

In this section we will try to improve the accuracies of our model

5.1 Model 1

5.1.1 Design

We will improve the last model performance by training the model for 100 epochs and reduce the batch size to 1000 and also we will use *Adam* optimizer.

```

network.add(layers.Dense(10, activation='softmax', input_shape=(784,)))

```

Figure 10: Model 1 architecture

```

network.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = network.fit(tf.convert_to_tensor(X_trn), train_labels, epochs=100, batch_size=1000, validation_split=0.2)

```

Figure 11: Model 1 configuration

5.1.2 Result

The performance of this model in training and testing phase given in Fig 12 & Fig 13 respectively.

```

Train Accuracy: 0.9297029155492783
Train Loss: 0.2531313669681549
Validation Accuracy: 0.9288333332538605
Validation Loss: 0.2638649976253509

```

Figure 12: Model 1 Training result

```

313/313 [=====] .
Test Accuracy: 0.927799997138977
Test Loss: 0.26487982273101807

```

Figure 13: Model 1 Test result

5.2 Model 2

5.2.1 Design

Compared to model 1 in this model we added 1 layer at the beginning. So this is the model consisting of 2 layers.

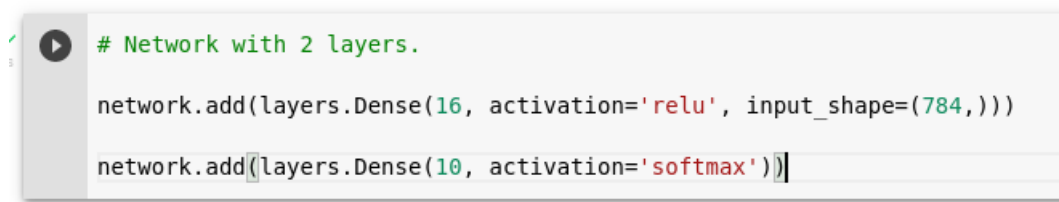
Number of neurons in 1st layer = 16

Activation Function at 1st layer = *Relu*

Number of neurons in 2nd layer = 10

Activation Function at 1st layer = *Softmax*

Except this everything else are left same as Model 1.

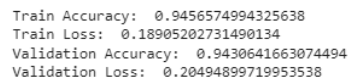
A code editor window showing the Keras model architecture for Model 2. The code defines a network with two layers: a first layer with 16 neurons using 'relu' activation, and a second layer with 10 neurons using 'softmax' activation. The input shape is (784,).

```
# Network with 2 layers.  
network.add(layers.Dense(16, activation='relu', input_shape=(784,)))  
network.add(layers.Dense(10, activation='softmax'))
```

Figure 14: Model 2 architecture

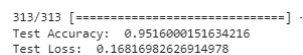
5.2.2 Result

The performance of this model is summarized in Fig 15 & 16.

A text box displaying the training results for Model 2, showing accuracy and loss for both training and validation datasets.

```
Train Accuracy: 0.9456574994325638  
Train Loss: 0.18905202731490134  
Validation Accuracy: 0.9430641663074494  
Validation Loss: 0.20494899719953538
```

Figure 15: Model 2 Training result

A text box displaying the test results for Model 2, showing accuracy and loss for the test dataset.

```
313/313 [=====] .  
Test Accuracy: 0.9516000151634216  
Test Loss: 0.16816982626914978
```

Figure 16: Model 2 Test result

5.3 Model 3

5.3.1 Design

We have kept the same design as Model 2. But at 1st layer we have used *tanh* activation function instead of *relu*.


```

# Network with 2 layers.

network.add(layers.Dense(16, activation='tanh', input_shape=(784,)))

network.add(layers.Dense(10, activation='softmax'))

```

Figure 17: Model 3 architecture

5.3.2 Result

Here we are using *tanh* activation function which takes more time to reduce the loss so with same number of epochs 100, the performance of this model is not that good but when are running this model for 300 epochs the model giving better result close to model 2. Fig 18 & 19 shows the training results of this model with 100 & 300 epochs respectively. Similarly Fig 20 & 21 shows the testing results of this model with 100 & 300 epochs. respectively.

Train Accuracy: 0.9440391653776169
 Train Loss: 0.213353306427598
 Validation Accuracy: 0.9358141666650772
 Validation Loss: 0.23653558656573295

Train Accuracy: 0.9682184733947118
 Train Loss: 0.12025808606296778
 Validation Accuracy: 0.9397933324178059
 Validation Loss: 0.22590411643187205

Figure 18: Model 3 train result with 100 epochs Figure 19: Model 3 train result with 300 epochs

313/313 [=====]
 Test Accuracy: 0.9456999897956848
 Test Loss: 0.18259499967098236

313/313 [=====]
 Test Accuracy: 0.9424999952316284
 Test Loss: 0.2525997757911682

Figure 20: Model 3 test result with 100 epochs Figure 21: Model 3 test result with 300 epochs

5.4 Model 4

5.4.1 Design

So far we have seen model with 2 layers. To improve performance in this model we have taken 3 layers.

Number of neurons in 1st layer = 50

Activation Function at 1st layer = Relu

Number of neurons in 2nd layer = 100

Activation Function at 2nd layer = Relu

Number of neurons in 2nd layer = 10

Activation Function at 3rd layer = Softmax

Number of epochs = 100

Batch size = 100

optimizer = Adam

```
# Network with 3 layers.

network.add(layers.Dense(50, activation='relu', input_shape=(784,)))
network.add(layers.Dense(100, activation='relu'))
network.add(layers.Dense(10, activation='softmax'))
```

Figure 22: Model 4 architecture

```
hist = network.fit(tf.convert_to_tensor(X_trn), train_labels, epochs=100, batch_size=100, validation_split=0.2)
```

Figure 23: Model 4 configuration

5.4.2 Result

The performance of this model is summarized below. If you see the 2 graphs you will observe that this model is over fitted because the training accuracy is close to 100%.

```
Train Accuracy: 0.9996302086114883
Train Loss: 0.0013792156848801262
Validation Accuracy: 0.9761100006103516
Validation Loss: 0.27336000770330426
```

Figure 24: Model 4 train and validation result

```
313/313 [=====]
Test Accuracy: 0.977699950408936
Test Loss: 0.2943854033946991
```

Figure 25: Model 4 testing result

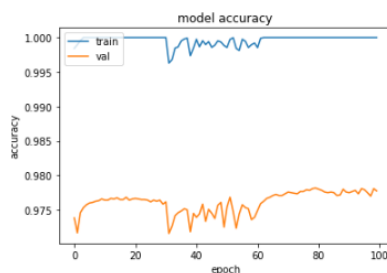


Figure 26: Model 4 train accuracy vs validation accuracy

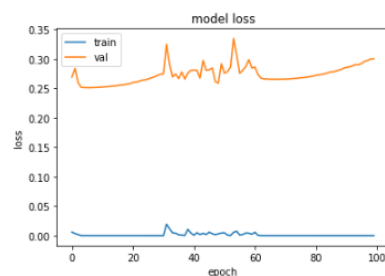


Figure 27: Model 4 train loss vs validation loss

5.5 Model 5

5.5.1 Design

We have already seen model 4 which is a 3 layer architecture and over fitted. Now to avoiding the over fitting of that model we will add some dropouts in hidden layers and except that all other configuration is exactly same as model 4.

Dropout applied between 1st and 2nd layer = 0.25

Dropout applied between 2nd and 3rd layer = 0.35

```
# Network with 3 layers.

network.add(layers.Dense(50, activation='relu', input_shape=(784,)))
network.add(layers.Dropout(0.25))
network.add(layers.Dense(100, activation='relu'))
network.add(layers.Dropout(0.35))
network.add(layers.Dense(10, activation='softmax'))
```

Figure 28: Model 5 architecture

5.5.2 Result

The performance of this model is summarized below. If you see the 2 graphs you will observe that this model is not over fitted because the training accuracy and validation loss are close to each other.

```
Train Accuracy: 0.96594124853611
Train Loss: 0.10895954765379429
Validation Accuracy: 0.9703224992752075
Validation Loss: 0.10917579926550389
```

Figure 29: Model 5 train and validation result

```
313/313 [=====] .
Test Accuracy: 0.972100019454956
Test Loss: 0.11764665693044662
```

Figure 30: Model 5 testing result

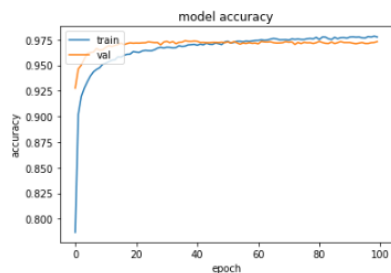


Figure 31: Model 5 train accuracy vs validation accuracy

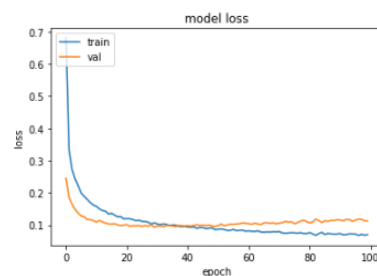


Figure 32: Model 5 train loss vs validation loss

6 Conclusion

In this experiment of hand written digit recognition using Artificial Neural Network(ANN) we presented how are taking sample image data from MNIST data set present under Keras library. Then we training our model with those sample data that we have received and also testing the model how accurate it is. To do the experiment we have suggested different layers of architecture from a single layer of 10 neurons to 3 layered dense network. We have used differed types of activation function across different layers based on our requirement. We have used `categorical_crossentropy` loss function for all the models which we have evaluated. First we started with a single layer network with 10 neurons and presented its training and testing accuracy then we improve this single layer network using an optimizer called *Adam*. Subsequently we moved to doubled layered network and compared the performance when using *tanh* or *relu* activation function at hidden layer and we found out that the model with *relu* is giving more promising result compared to the other one because using *tanh* it takes more time to reduce the loss value because the gradient part reduces exponentially and as a result the weights and biases will not be updated effectively. Then we moved to 3 layered architecture and found out the model was performing very well on the training data compared to the validation data so the model was over fitted. To reduce the over fitting of the model we introduced some dropouts at hidden layers and successfully achieved a test accuracy of 97.21%.