# Homework1 Report **MNIST Classification using Deep Learning**

## Problem Description

MNIST is a well-known dataset generally used for educational purposes in classification. In this report, I would describe the procedure of training several DNNs with different parameters. At the end I will discuss the results conducted and try to draw a conclusion on the effect of changing parameters on classification problems.

## Installing requirements

In this homework, I used Miniconda as my virtual environment to install python and its packages and libraries.

### Making a virtual environment and installing dependencies

After installing Conda according to online resources, we build a environment and activate it using following commands.

conda create -n Mnist python=3.8.0

conda activate Mnist

Using these commands the Mnist environment would be activated for usage. After that we installed the TensorFlow using the following command:

conda install conda-forge::tensorflow

Now we can use TensorFlow and we install other libraries such as NumPy, matlibplot, and SciPy with:

pip install library\_name

## Importing required tools from different libraries

To use each function (e.g. optimizer, loss functions, and so on) we import them at the top of code, as follows.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import os

import random

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.optimizers import Adam, SGD, RMSprop, Adamax, Adadelta , Adagrad

from tensorflow.keras.layers import Dense, Dropout, Activation, BatchNormalization, concatenate, Input, Conv2D, Flatten, Lambda, MaxPooling2D

from utils import save\_results\_to\_excel

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.metrics import BinaryAccuracy

## Load dataset

To use data for training procedures, we load it with the following command and the result of this command would be:

(trainX, trainY), (testX, testY) = mnist.load\_data()

# summarize loaded dataset

print('Train: X=%s, y=%s' % (trainX.shape, trainY.shape))

print('Test: X=%s, y=%s' % (testX.shape, testY.shape))

Train: X=(60000, 28, 28), y=(60000,)

Test: X=(10000, 28, 28), y=(10000,)

## Data Visualization

In order to have an insight into input data, we use the following code to visualize a different group of them and it visualizes different one each time by using a random number.

a = int(random.random() \* 5000)

plotting = False

if plotting ==True:

    # Assuming trainX contains the image data

    for i in range(9):

        # Define subplot for a 4x4 grid

        plt.subplot(3, 3, i + 1)

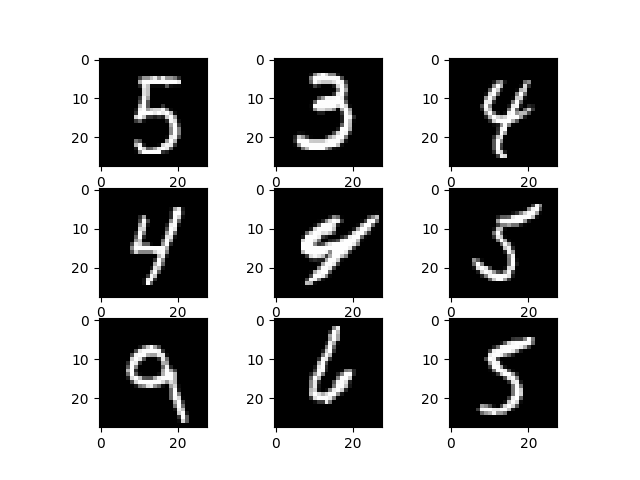
        # Plot raw pixel data

        plt.imshow(trainX[a+i], cmap='gray')

        # Show the figure

        # plt.tight\_layout()

    plt.show()

the output of this part of code would be like this

## Preprocessing Data

For preprocessing the data, we first add a third dimension to it. Then using to\_categorical function, it would be prepared for usage as output. Finally, we change data type to fload32 and it would be normalized by dividing to 255.

# reshape dataset to have a single channel

trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))

testX = testX.reshape((testX.shape[0], 28, 28, 1))

trainY = to\_categorical(trainY)

testY = to\_categorical(testY)

trainX = trainX.astype('float32')

testX = testX.astype('float32')

# normalize to range 0-1

trainX /= 255.0

testX /= 255.0

## Create and train model

In this section, we use a function which gives the ability to change parameters of DNN to find the best configurations.

### 7.1. Creating DNN model

For building DNN we define Create\_CNN\_model. This function allows us to pass different parameters of the model as input. For example, we pass learning rate, optimizer function, loss function, number of channels in Conv2D layers, number of Con2D layers, and number of Dense layers.

In this function we first add a Conv2D as input layer. Other layers would be added according to input parameters. After Conv2D layers, we use Flatten to be able to use dense layers and dense layers would be added to our model. After compiling the model, the model would be returned as output of this function.

# define cnn model

def Create\_CNN\_model(lr, optimizer, loss, num\_channels, num\_conv, num\_dense):

  model = Sequential()

  model.add(Conv2D(num\_channels, (3, 3), activation='relu', padding='same', input\_shape=(28, 28, 1)))

  # Add increasing layers

  for i in range(num\_conv):

    model.add(Conv2D(num\_channels\* (2 \*\* (i + 1)), (3, 3), activation='relu',padding='same'))

    model.add(MaxPooling2D((2, 2)))

  model.add(Flatten())

  size = model.layers[-1].output\_shape[1]

  print("Size is ",size)

  for i in range(num\_dense):

    model.add(Dense(2000\*(num\_dense-i)/(num\_dense+1), activation='relu'))

  model.add(Dense(10, activation='softmax'))

  # compile model

  model.compile(optimizer=optimizer(learning\_rate=lr), loss=loss, metrics=['accuracy'])

  model.summary()

  return model

## 7.2. Training procedure with different configurations

To investigate the effect of changing different configurations on performance of DNN, we train models with a bunch of configurations. Each model results would be finally in the result.xlsx file.

In this section we define different parameters to train network with them.

# Define the parameter combinations

optimizers = [Adam, SGD, RMSprop]

loss\_functions = [ 'mse', 'mae']

learning\_rates = [0.001, 0.01, 0.1]

num\_channels = [32, 64]

num\_convs = [2, 3]

num\_denses = [1, 3, 5]

After that, using nested loops we use different configurations for training DNN. The training would be done with model.fit command.

results = []

# Test different combinations of optimizers, loss functions, and learning rates

for optimizer in optimizers:

    for loss in loss\_functions:

        for lr in learning\_rates:

            for num\_channel in num\_channels:

                for num\_conv in num\_convs:

                  for num\_dense in num\_denses:

                    try:

                        print(f'Testing {optimizer.\_\_name\_\_} with {loss} at learning rate {lr}')

                        # Create and compile the model

                        model = Create\_CNN\_model(lr, optimizer, loss, num\_channel, num\_conv, num\_dense)

                        # Train the model

                        history = model.fit(trainX, trainY,

                                validation\_data=(testX, testY),

                                epochs=20, batch\_size=64)

                                # callbacks=[model\_checkpoint])  # Adjust epochs as needed

                                # callbacks=[lr\_scheduler, model\_checkpoint])  # Adjust epochs as needed

After the training procedure is completed, the evaluation will be done on training and testing data using model.evaluate command. We calculate other metrics such as accuracy, mean absolute error, and mean square error.

                        # Evaluate the model

                        loss\_value, mse\_value = model.evaluate(trainX, trainY, verbose=0)

                        loss\_value\_test, mse\_value\_test = model.evaluate(testX, testY, verbose=0)

                        y\_train\_pred = model.predict(trainX)

                        y\_test\_pred = model.predict(testX)

                        # Calculate metrics

                        # Assume y\_pred\_train and y\_train are your model predictions and true labels for training

                        # Similarly, y\_pred\_test and y\_test are for testing

                        accuracy\_metric.update\_state(trainY, y\_train\_pred)

                        train\_accuracy = accuracy\_metric.result().numpy()  # Training accuracy

                        accuracy\_metric.reset\_states()  # Reset the metric state

                        accuracy\_metric.update\_state(testY, y\_test\_pred)

                        test\_accuracy = accuracy\_metric.result().numpy()  # Testing accuracy

                        mae\_value\_train = mean\_absolute\_error(trainY, y\_train\_pred)

                        mae\_value\_test = mean\_absolute\_error(testY, y\_test\_pred)

                        mse\_value = mean\_squared\_error(trainY, y\_train\_pred)

                        mse\_value\_test = mean\_squared\_error(testY, y\_test\_pred)

                        model\_name = f"{optimizer.\_\_name\_\_}\_{loss}\_lr{lr:.0e}\_num\_channel{num\_channel}\_num\_conv{num\_conv}\_num\_dense{num\_dense}.h5"

                        model\_path = os.path.join("models", model\_name)

                        model.save(model\_path)

                        # Find the epoch that corresponds to the best validation loss

                        best\_epoch = np.argmin(history.history['val\_loss']) + 1  # Adding 1 to account for 0-based index

                        print(f"The best model was saved at epoch: {best\_epoch}")

                        trainable\_params = sum(tf.size(variable).numpy() for variable in model.trainable\_variables)

Finally, we append all this information to the results.

                        # Store the metrics in your results

                        results.append({

                            'optimizer': optimizer.\_\_name\_\_,

                            'loss\_func': loss,

                            'learning\_rate': lr,

                            'loss': loss\_value,

                            'mse': mse\_value,

                            'mae': mae\_value\_train,

                            'accuracy': train\_accuracy,

                            'loss\_test': loss\_value\_test,

                            'mse\_test': mse\_value\_test,

                            'mae\_test': mae\_value\_test,

                            'accuracy\_test': test\_accuracy,

                            'model\_path': model\_path,

                            'best\_epoch': best\_epoch,

                            'num\_parameters': trainable\_params

                        })

                    except Exception as e:

                        print(f"Error with configuration {optimizer.\_\_name\_\_}, {loss}, {lr}: {e}. Skipping...")

                        # Skip to the next iteration

                        continue

At the end of the process, the results variable would be stored in an Excel file using save\_results\_to\_excel function. This function is in utils.py

import pandas as pd

def save\_results\_to\_excel(results, excel\_file='results.xlsx'):

    # Convert results list to a pandas DataFrame

    df = pd.DataFrame(results)

    # Save to Excel file (write if it doesn't exist, append otherwise)

    try:

        # If the file already exists, append without overwriting

        with pd.ExcelWriter(excel\_file, mode='a', engine='openpyxl', if\_sheet\_exists='replace') as writer:

            df.to\_excel(writer, index=False, sheet\_name='Results')

    except FileNotFoundError:

        # If the file doesn't exist, create a new one

        df.to\_excel(excel\_file, index=False, sheet\_name='Results')

the saving result variable would be done using the

save\_results\_to\_excel(results)

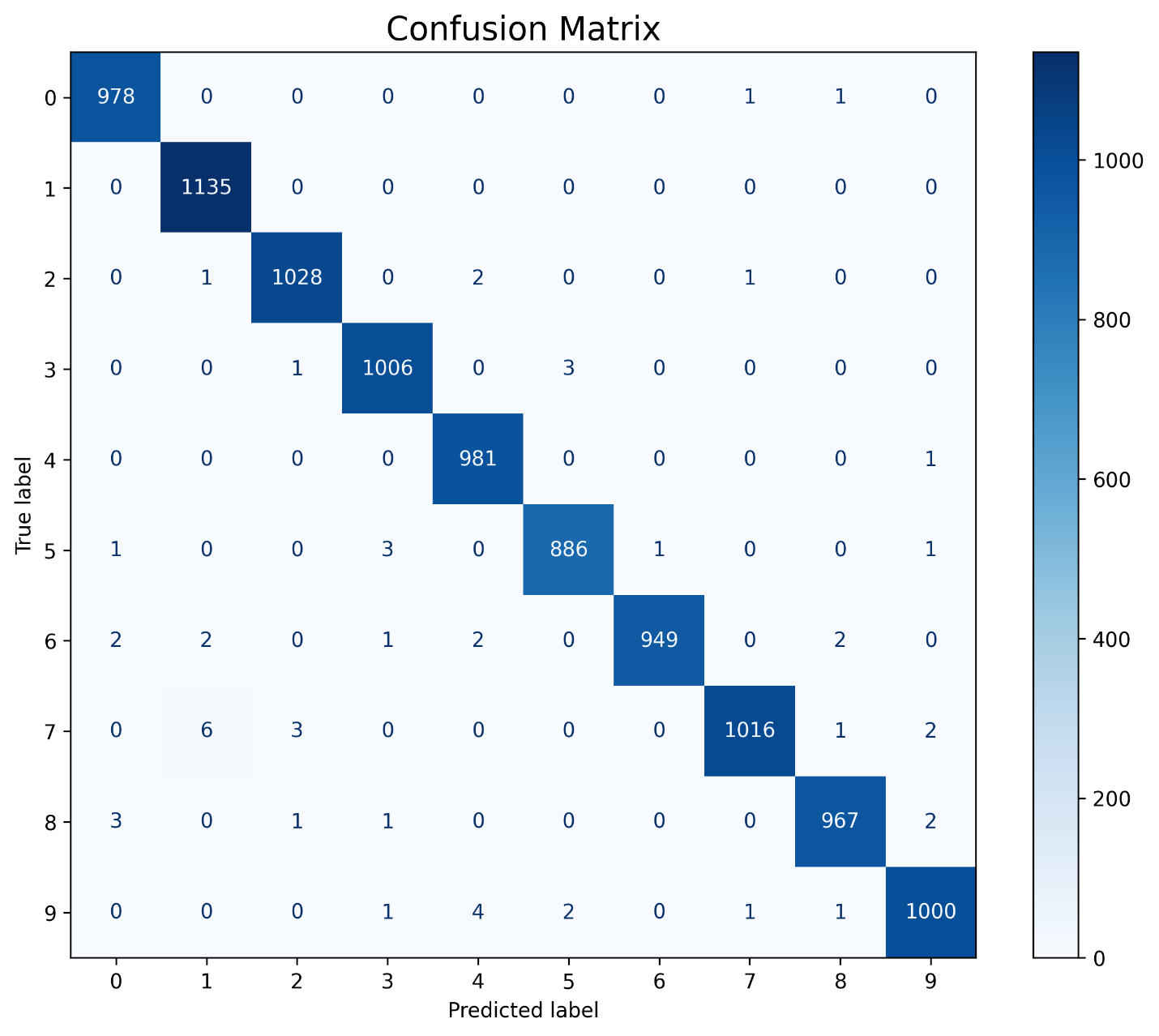
## Comparing different DNN

We trained different models. You can see the best models in the following table.

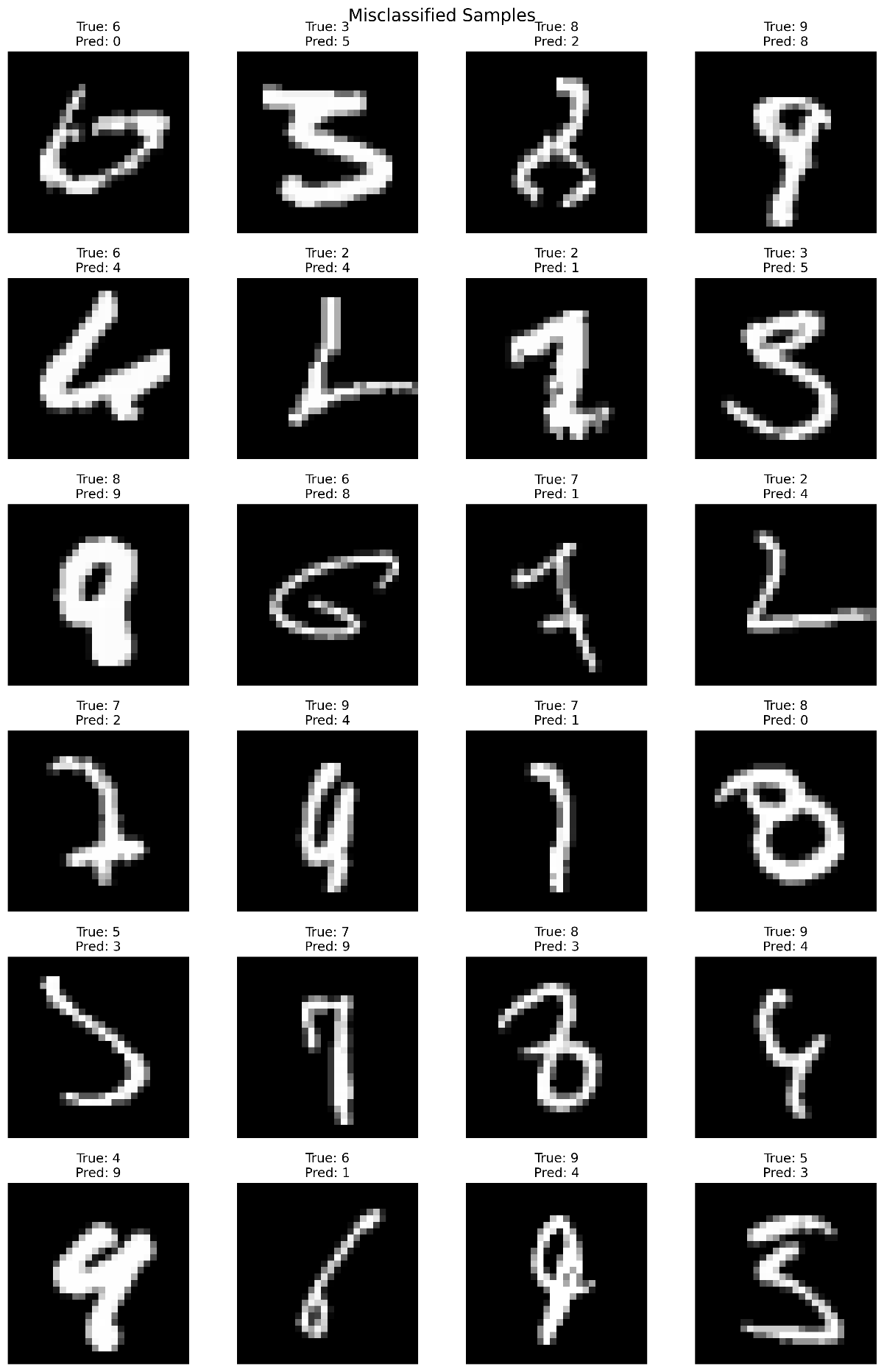
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| optimizer | Loss function | Learning rate | Train | | | | Test | | | | Number of Channels | Number of Conv2D | Number of Dense | Number of parameters |
| loss | mse | mae | accuracy | Loss | mse | mae | accuracy |
| RMSprop | mse | 0.001 | 0.00018 | 0.0002 | 0.0003 | 10469834 | 0.00093 | 0.00093 | 0.00136 | 0.9989 | 64 | 3 | 3 | 10469834 |
| RMSprop | mse | 0.001 | 0.0001 | 0.0001 | 0.0002 | 12924674 | 0.00089 | 0.00089 | 0.00135 | 0.9988 | 64 | 2 | 1 | 12924674 |
| RMSprop | mse | 0.001 | 0.00013 | 0.0001 | 0.0002 | 6168834 | 0.00099 | 0.00099 | 0.00134 | 0.9987 | 64 | 3 | 1 | 6168834 |
| RMSprop | mse | 0.001 | 0.00017 | 0.0002 | 0.0003 | 14991718 | 0.00106 | 0.00106 | 0.00138 | 0.9987 | 32 | 2 | 5 | 14991718 |
| RMSprop | mse | 0.001 | 0.0001 | 0.0001 | 0.0001 | 21193674 | 0.00100 | 0.00100 | 0.00139 | 0.9987 | 64 | 2 | 3 | 21193674 |
| RMSprop | mse | 0.001 | 0.00022 | 0.0002 | 0.0004 | 25717862 | 0.00102 | 0.00102 | 0.00143 | 0.9987 | 64 | 2 | 5 | 25717862 |
| RMSprop | mse | 0.001 | 0.00011 | 0.0001 | 0.0002 | 6375682 | 0.00103 | 0.00103 | 0.00156 | 0.9986 | 32 | 2 | 1 | 6375682 |
| RMSprop | mse | 0.001 | 0.00023 | 0.0002 | 0.0004 | 5851850 | 0.00107 | 0.00107 | 0.00140 | 0.9986 | 32 | 3 | 3 | 5851850 |
| RMSprop | mse | 0.001 | 0.00027 | 0.0003 | 0.0004 | 13676646 | 0.00121 | 0.00121 | 0.00158 | 0.9986 | 32 | 3 | 5 | 13676646 |
| RMSprop | mse | 0.001 | 0.00013 | 0.0001 | 0.0002 | 11508682 | 0.00116 | 0.00116 | 0.00152 | 0.9985 | 32 | 2 | 3 | 11508682 |
| RMSprop | mse | 0.001 | 0.00066 | 0.0007 | 0.0009 | 8676198 | 0.00140 | 0.00140 | 0.00182 | 0.9983 | 32 | 3 | 5 | 8676198 |
| RMSprop | mae | 0.001 | 0.00189 | 0.0019 | 0.0019 | 2702850 | 0.00171 | 0.00168 | 0.00171 | 0.9982 | 32 | 3 | 1 | 2702850 |
| RMSprop | mse | 0.001 | 0.00018 | 0.0002 | 0.0003 | 2702850 | 0.00134 | 0.00134 | 0.00188 | 0.9982 | 32 | 3 | 1 | 2702850 |
| SGD | mse | 0.1 | 0.00082 | 0.0008 | 0.0023 | 13676646 | 0.00155 | 0.00155 | 0.00337 | 0.9979 | 64 | 3 | 5 | 13676646 |
| SGD | mse | 0.1 | 0.00099 | 0.001 | 0.0027 | 6168834 | 0.00166 | 0.00166 | 0.00355 | 0.9978 | 64 | 3 | 1 | 6168834 |
| SGD | mse | 0.1 | 0.00081 | 0.0008 | 0.0022 | 8676198 | 0.00168 | 0.00168 | 0.00330 | 0.9978 | 32 | 3 | 5 | 8676198 |
| SGD | mse | 0.1 | 0.00097 | 0.001 | 0.0028 | 21193674 | 0.00180 | 0.00180 | 0.00395 | 0.9977 | 64 | 2 | 3 | 21193674 |
| SGD | mse | 0.1 | 0.00096 | 0.001 | 0.0029 | 14991718 | 0.00181 | 0.00181 | 0.00406 | 0.9976 | 32 | 2 | 5 | 14991718 |

### 8.1. Performance of best Model

The best model has been chosen based on the best accuracy over the test data. The performance of this model on test data in shown on the following confusion matrix. Only 54 misclassified data are among test data.



Some of misclassified data are plotted in following Image.

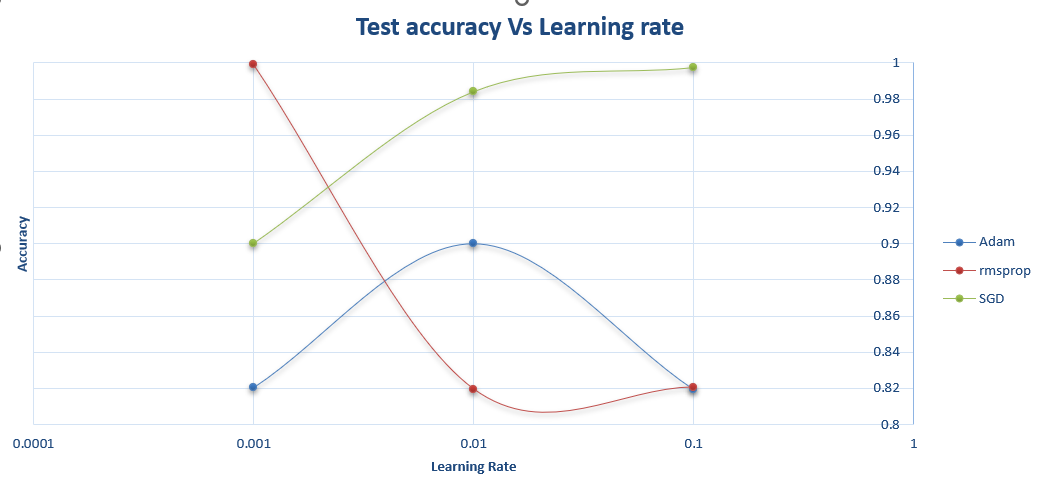


### 8.2. The effect of different Optimizer and Learning rate

We trained different models. You can see the best models in the following table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| optimizer | Loss function | Learning rate | Train | | | | Test | | | | Number of Channels | Number of Conv2D | Number of Dense | Number of parameters |
| loss | mse | mae | accuracy | Loss | mse | mae | accuracy |
| Adam | mse | 0.001 | 0.17912 | 0.1791 | 0.1791 | 0.8208 | 0.17944 | 0.17944 | 0.179439 | 0.820559 | 64 | 3 | 3 | 10469834 |
| Adam | mse | 0.01 | 0.08999 | 0.09 | 0.1799 | 0.8885 | 0.08998 | 0.08998 | 0.179931 | 0.899999 | 64 | 3 | 3 | 10469834 |
| Adam | mse | 0.1 | 0.18053 | 0.1805 | 0.1805 | 0.8195 | 0.18036 | 0.18036 | 0.180359 | 0.819639 | 64 | 3 | 3 | 10469834 |
| RMSprop | mse | 0.001 | 0.00018 | 0.0002 | 0.0003 | 0.9997 | 0.00093 | 0.00093 | 0.001360 | 0.998939 | 64 | 3 | 3 | 10469834 |
| RMSprop | mse | 0.01 | 0.18026 | 0.1803 | 0.1803 | 0.8199 | 0.18040 | 0.1804 | 0.1804 | 0.819599 | 64 | 3 | 3 | 10469834 |
| RMSprop | mse | 0.1 | 0.18014 | 0.1801 | 0.1801 | 0.8196 | 0.17936 | 0.17936 | 0.17936 | 0.820640 | 64 | 3 | 3 | 10469834 |
| SGD | mse | 0.001 | 0.08982 | 0.0898 | 0.1798 | 0.9 | 0.08981 | 0.08981 | 0.179816 | 0.899999 | 64 | 3 | 3 | 10469834 |
| SGD | mse | 0.01 | 0.01335 | 0.0133 | 0.033 | 0.9844 | 0.01259 | 0.01259 | 0.031251 | 0.984189 | 64 | 3 | 3 | 10469834 |
| SGD | mse | 0.1 | 0.00131 | 0.0013 | 0.0032 | 0.9984 | 0.00194 | 0.00194 | 0.0040232 | 0.99763 | 64 | 3 | 3 | 10469834 |

Changing learning rate and optimizer on accuracy has been demonstrated on following image.



### 8.3. The effect of Network size on accuracy

As you can see by increasing the size of the network, always the accuracy of model would not increase.

