Homework1 Report

MNIST Classification using Deep Learning

1. 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.

2. Installing requirements

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

2.1. 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
```

3. 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
```

4. 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,)

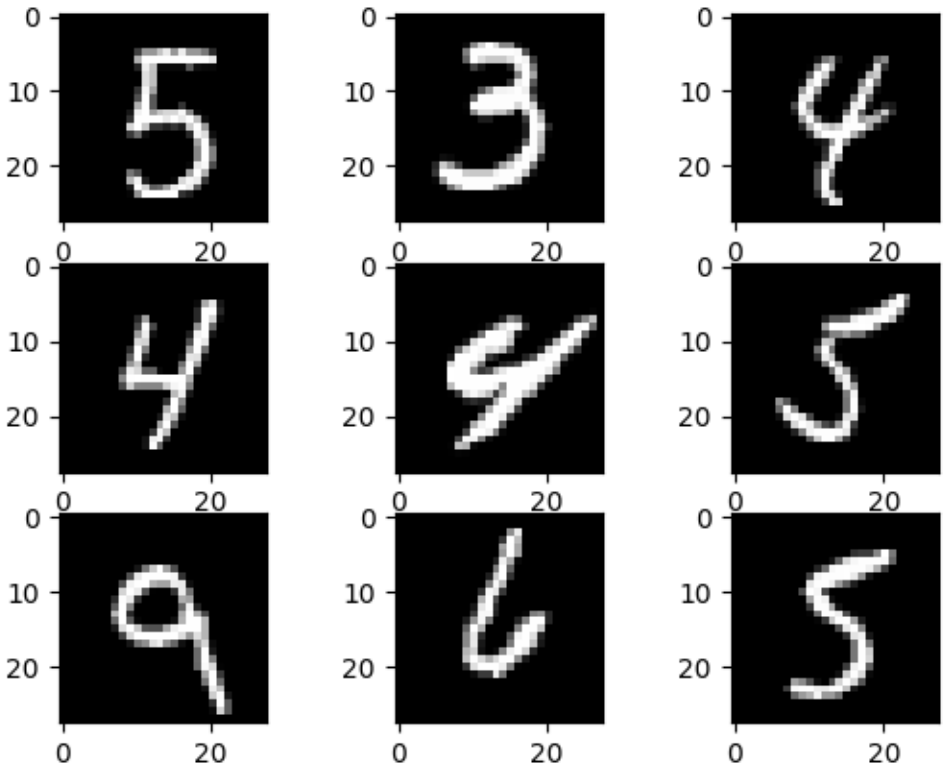
5. Data Visualization

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



6. 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
```

7. 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_co
nv}_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)
```

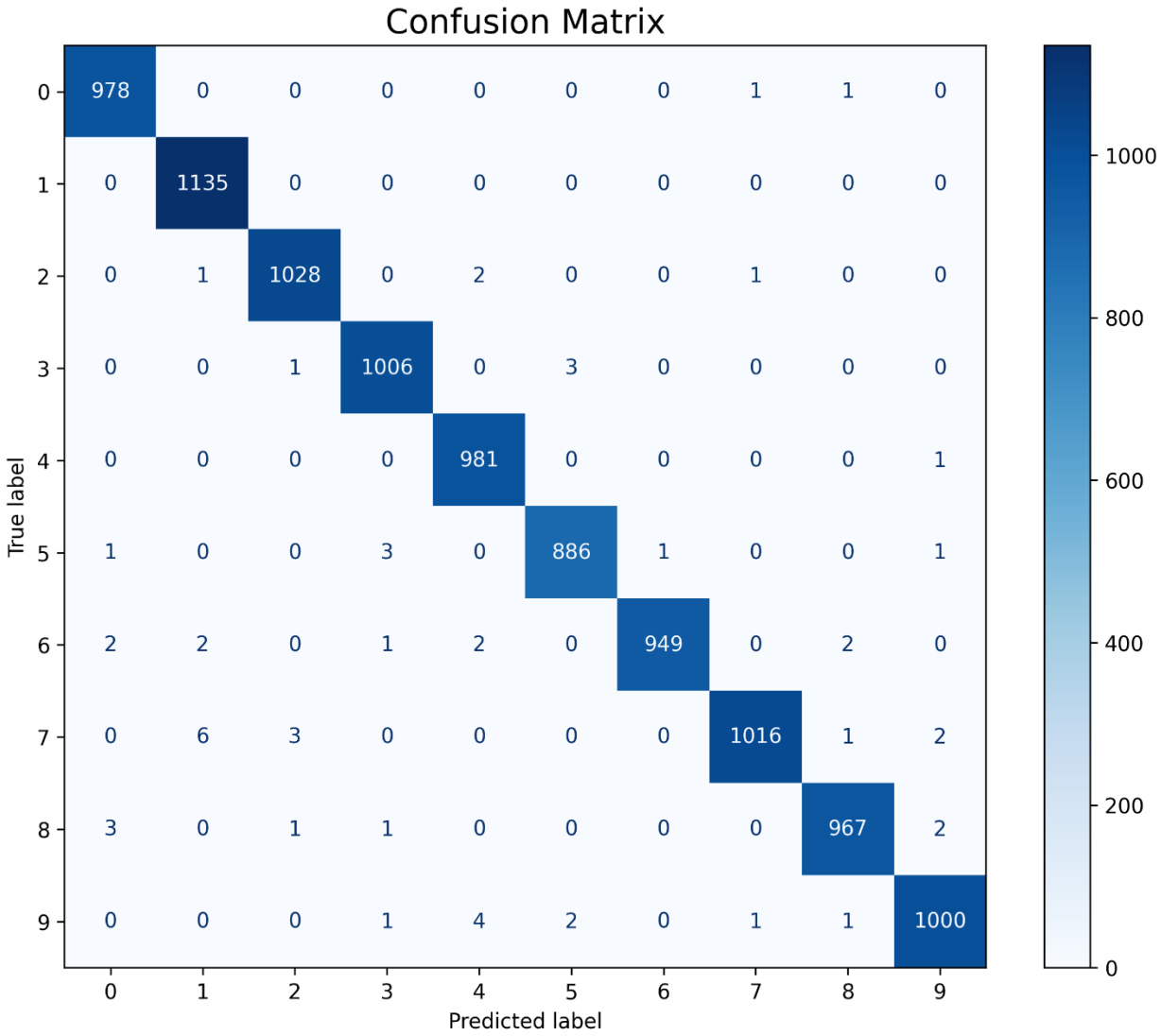
8. Comparing different DNN

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

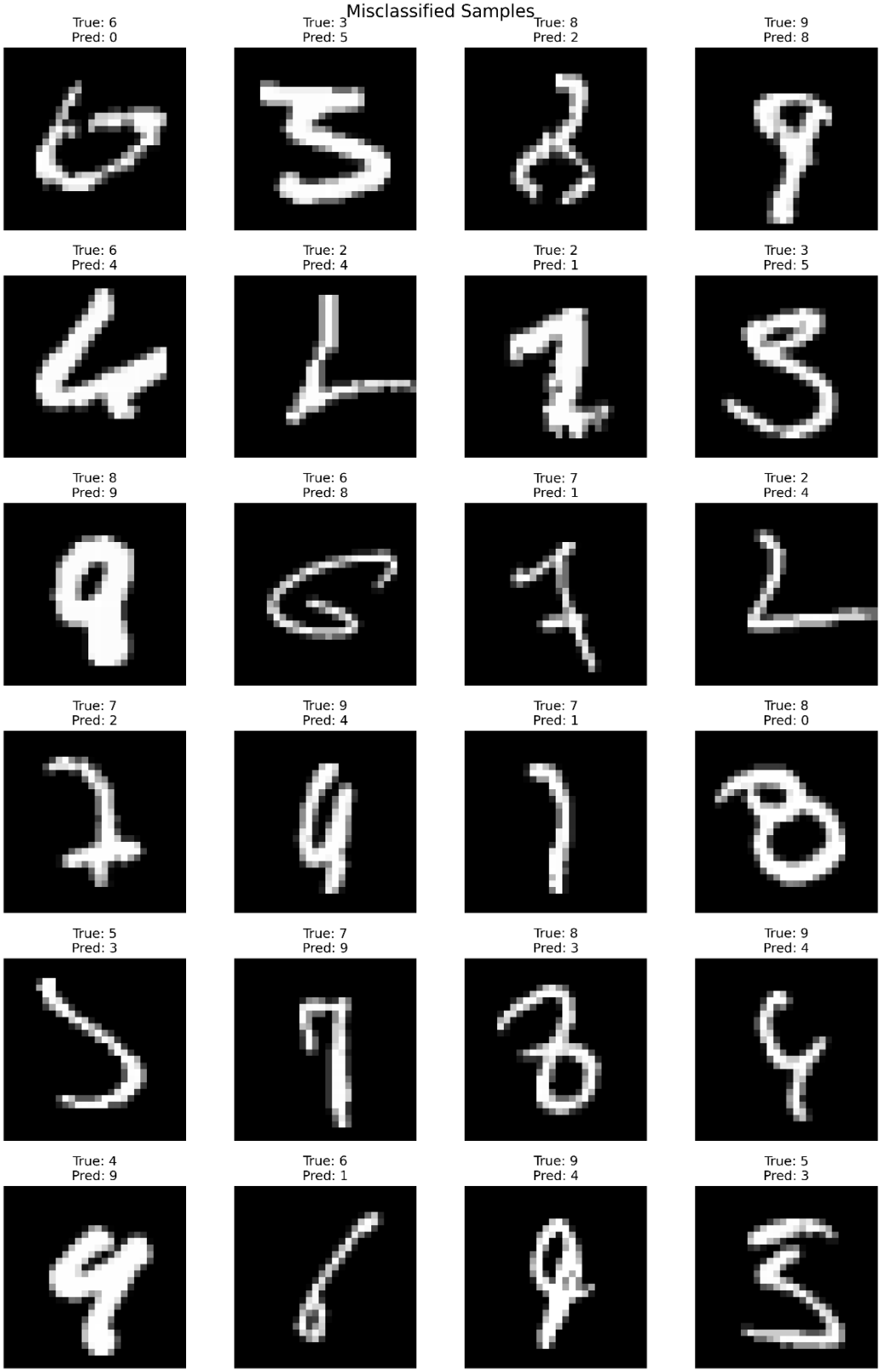
8.1. Performance of best Model

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.

Misclassified Samples

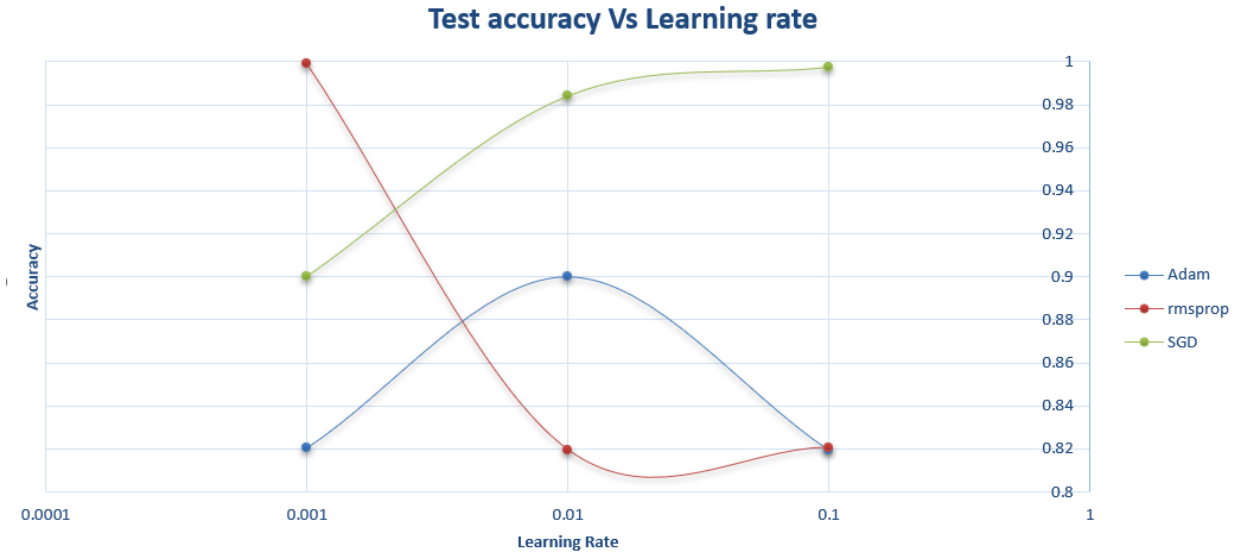


8.2. The effect of different Optimizer and Learning rate

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.