Name: Nasin Ayenul Akash

Id:17-35060-2

Section: B

Comparing the accuracy using different optimizers (Adam, SGD, RMSprop) by implementing a CNN architecture to classify the MNIST handwritten dataset

Abstract: This report is about the implementation of a CNN and the first footstep on getting started with the deep learning concept, to claim an overview of the basics of neural networks and playing with the datasets. Getting started with a single perceptron and understanding it's inner operations, we moved toward the basic problems of image classification, where sometimes in a neural network, a single neuron can do the computations smoothly. But then there comes a target of image processing and classification where for a single image classification we need a primary dataset by which we want to train our program and predict the actual outcome. In the Mid Term project, we're using the MNIST handwritten dataset to get started, using it to train our model and then test. During the whole task, we had to give the compile instructions before training where we had to set the optimizer. There're a lot of optimizers and these can be used to get more efficiency in case of training the dataset. Implementing different optimizers, we can improve the accuracy and manipulate our model. Here, for this particular project, three optimizers Adam, SGD and RMSprop is shown applied for MNIST dataset and given a comparison among these three generating the highest accuracy.

Introduction:

The CNN (Convolutional Neural Network) is on trend at recent times as getting deeper and deeper. To dive in, we must first know about the terms thoroughly. Basically the purpose of a convolutional layer is to detect features for example the edges, lines, colors and other elements which can be detected as a unique feature. CNN works with complex image classification tasks and processing that is basically evaluated based on pixels.

For implementing a CNN architecture, the primary need is a dataset which can help us to train a model. This training is needed because machines are not capable like human brains. Human Brain is a taugh word to use comparing with machines, but for going to that direction the sector neural network is introduced. To implement an image classifier model and for recognizing, a dataset is required. In this model, the MNIST handwritten dataset is used.

MNIST digit classification dataset is a well-known dataset where there are basically 10 numbers (0-1). This dataset contains 60,000 training images. These are all 28x28 gray scale images of the 10 digits along with the corresponding labels. There're 10,000 test images. For implementing and testing the model, the dataset is taken from Keras API under the tensorflow.

I've done the implementation in seven parts as following:

1. Importing the MNIST dataset.
2. Image preprocessing:
3. Building three different models for Adam, SGD and RSMProp
4. Compile
5. Training
6. Plotting the accuracy and loss for each optimizer.
7. Using the model of highest accuracy among the three to test.

The fisrt task was importing the MNIST dataset, I've loaded this data into two tuples defined as: `(X_train, Y_train), (X_test, Y_test).` where the first one is the 60K training images along with the label and the next tuple is the test images with the label. Then a checking has to be done if the dataset is okay or not. The shape of the train dataset was as expected (60000, 28, 28) and also the test dataset (10000, 28, 28).

When the dataset was imported successfully, the second step was Image Preprocessing. I've converted all the matrices data info 'float32' and then divided them by 255. It will take all the pixel values between range 0-1. I kept them 2-D as they were.

After that, it was the time to build the model. There are basically two models:

1. Sequential Model
2. Functional Model

I've designed a sequential model here. As for comparing among the optimizers, built three models one by one. But kept the internal layering were same as It will be easy to monitor the outputs. There is one input layer where the shape is (28,28). Then the hidden layers come. The first hidden layer is for flattening the values as they can adjust to the next Dense layer.

Then in the next hidden layer I gave 64 units of neurons for processing. On the next two layers the units are 32 and 16 respectively and for each of these hidden layers I've used ReLU (Reflected Linear Units) which is the basic ReLU activation function. And for the last layer, which is the output layer, I've given 10 units, as there are 10 labels for the dataset and used 'Softmax' activation function.

Model Summary:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_5 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_6 (Dense)              (None, 16)                528
_________________________________________________________________
dense_7 (Dense)              (None, 10)                170
=================================================================
Total params: 53,018
Trainable params: 53,018
Non-trainable params: 0
_________________________________________________________________

 -------------x-------------
```

Here, we have 53,018 params in total to play with.

After then, had to invoke the compile method before starting training. I've given three comile methods using three different optimizers.

```
[9]  #compile:

     modeladam.compile(
         optimizer=tf.keras.optimizers.Adam(),
         loss=tf.losses.sparse_categorical_crossentropy,
         metrics=['accuracy']
     )

     modelsgd.compile(
         optimizer=tf.keras.optimizers.SGD(momentum=0.9,learning_rate=0.01),
         loss=tf.losses.sparse_categorical_crossentropy,
         metrics=['accuracy']
     )

     modelrmsprop.compile(
         optimizer=tf.keras.optimizers.RMSprop(),
         loss=tf.losses.sparse_categorical_crossentropy,
         metrics=['accuracy']
     )
```

Now, I've invoked the fit method for the training.

```
#training:

h1 = modeladam.fit(x=X_train,y=Y_train, epochs=10 , validation_split=0.2 , batch_size=64)
print("\n -------------x-------------\n")
h2 = modelsgd.fit(x=X_train,y=Y_train, epochs=10 , validation_split=0.2 , batch_size=64)
print("\n -------------x-------------\n")
h3 = modelrmsprop.fit(x=X_train,y=Y_train, epochs=10 , validation_split=0.2 , batch_size=64)
print("\n -------------x-------------\n")
```

Results:

Training using Adam Optimizer:

```
Epoch 1/10
750/750 [==============================] - 3s 3ms/step - loss: 0.4131 - accuracy: 0.8764 - val_loss: 0.2322 - val_accuracy: 0.9299
Epoch 2/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1763 - accuracy: 0.9482 - val_loss: 0.1570 - val_accuracy: 0.9541
Epoch 3/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1291 - accuracy: 0.9614 - val_loss: 0.1331 - val_accuracy: 0.9592
Epoch 4/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1046 - accuracy: 0.9676 - val_loss: 0.1187 - val_accuracy: 0.9626
Epoch 5/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0855 - accuracy: 0.9743 - val_loss: 0.1203 - val_accuracy: 0.9638
Epoch 6/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0706 - accuracy: 0.9779 - val_loss: 0.1342 - val_accuracy: 0.9621
Epoch 7/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0602 - accuracy: 0.9809 - val_loss: 0.1115 - val_accuracy: 0.9680
Epoch 8/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0522 - accuracy: 0.9832 - val_loss: 0.1032 - val_accuracy: 0.9702
Epoch 9/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0461 - accuracy: 0.9854 - val_loss: 0.1129 - val_accuracy: 0.9687
Epoch 10/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0409 - accuracy: 0.9870 - val_loss: 0.1222 - val_accuracy: 0.9669

 -------------x-------------
```

Training using SGD Optimizer:

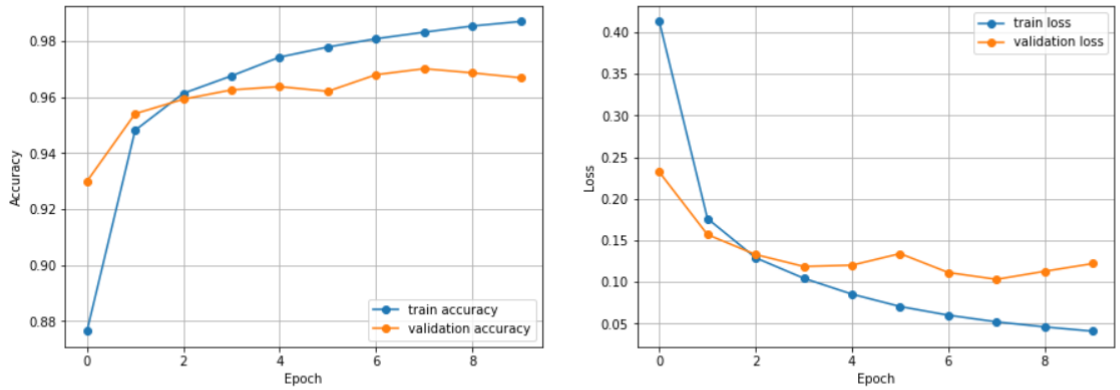
```
 -------------x-------------

Epoch 1/10
750/750 [==============================] - 2s 3ms/step - loss: 0.4856 - accuracy: 0.8496 - val_loss: 0.2237 - val_accuracy: 0.9337
Epoch 2/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1816 - accuracy: 0.9466 - val_loss: 0.1609 - val_accuracy: 0.9527
Epoch 3/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1337 - accuracy: 0.9603 - val_loss: 0.1398 - val_accuracy: 0.9573
Epoch 4/10
750/750 [==============================] - 2s 2ms/step - loss: 0.1065 - accuracy: 0.9679 - val_loss: 0.1213 - val_accuracy: 0.9639
Epoch 5/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0893 - accuracy: 0.9725 - val_loss: 0.1244 - val_accuracy: 0.9634
Epoch 6/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0767 - accuracy: 0.9767 - val_loss: 0.1233 - val_accuracy: 0.9664
Epoch 7/10
750/750 [==============================] - 2s 2ms/step - loss: 0.0662 - accuracy: 0.9793 - val_loss: 0.1185 - val_accuracy: 0.9678
Epoch 8/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0584 - accuracy: 0.9817 - val_loss: 0.1124 - val_accuracy: 0.9687
Epoch 9/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0506 - accuracy: 0.9840 - val_loss: 0.1135 - val_accuracy: 0.9682
Epoch 10/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0474 - accuracy: 0.9850 - val_loss: 0.1172 - val_accuracy: 0.9669

 -------------x-------------
```

Training using RMSprop Optimizer:

```
-------------x-------------

Epoch 1/10
750/750 [==============================] - 3s 3ms/step - loss: 0.3952 - accuracy: 0.8873 - val_loss: 0.2003 - val_accuracy: 0.9436
Epoch 2/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1798 - accuracy: 0.9455 - val_loss: 0.1725 - val_accuracy: 0.9506
Epoch 3/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1348 - accuracy: 0.9603 - val_loss: 0.1419 - val_accuracy: 0.9580
Epoch 4/10
750/750 [==============================] - 2s 3ms/step - loss: 0.1091 - accuracy: 0.9677 - val_loss: 0.1194 - val_accuracy: 0.9657
Epoch 5/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0912 - accuracy: 0.9730 - val_loss: 0.1187 - val_accuracy: 0.9659
Epoch 6/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0791 - accuracy: 0.9769 - val_loss: 0.1206 - val_accuracy: 0.9677
Epoch 7/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0707 - accuracy: 0.9791 - val_loss: 0.1276 - val_accuracy: 0.9653
Epoch 8/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0624 - accuracy: 0.9812 - val_loss: 0.1125 - val_accuracy: 0.9703
Epoch 9/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0554 - accuracy: 0.9835 - val_loss: 0.1152 - val_accuracy: 0.9695
Epoch 10/10
750/750 [==============================] - 2s 3ms/step - loss: 0.0504 - accuracy: 0.9847 - val_loss: 0.1155 - val_accuracy: 0.9702

 -------------x-------------
```

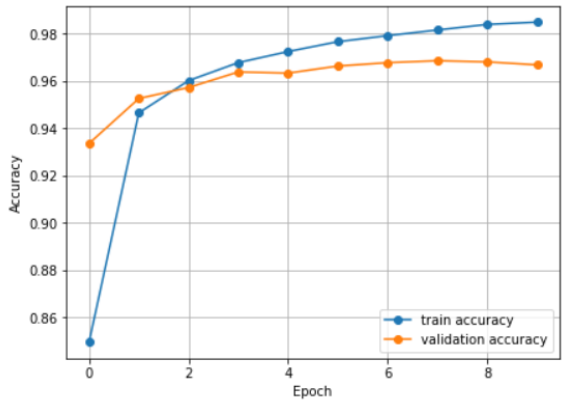
Validation & Losses:

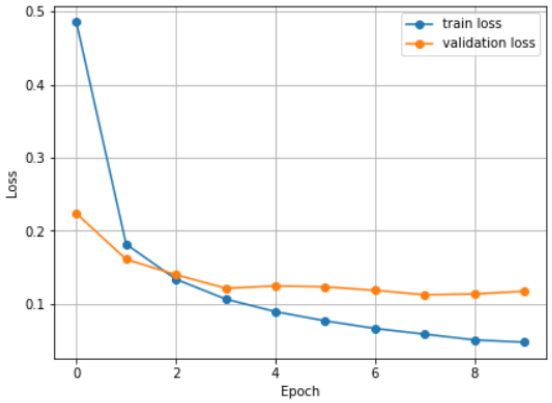
1. Adam Optimization Function




```
313/313 [==============================] - 0s 1ms/step - loss: 0.1192 - accuracy: 0.9673

Test loss: 0.11921301484107971 Test Accuracy: 0.9672999978065491
```

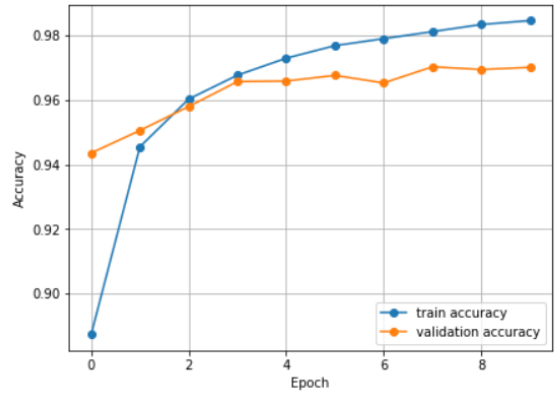
2. SGD Optimization Function

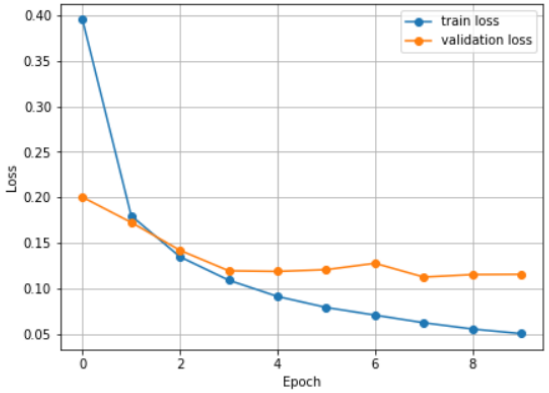





```
313/313 [==============================] - 0s 1ms/step - loss: 0.1128 - accuracy: 0.9678

Test loss: 0.11277458816766739 Test Accuracy: 0.9678000211715698
```

3. RMSprop Optimization Function





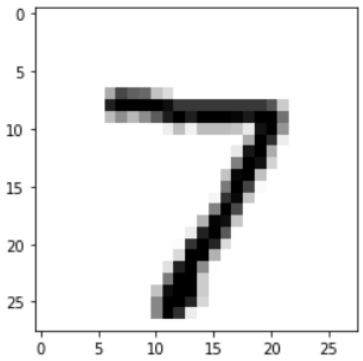

```
313/313 [==============================] - 0s 1ms/step - loss: 0.1113 - accuracy: 0.9717

Test loss: 0.11125949770212173 Test Accuracy: 0.9717000126838684
```

Testing:

```
[21] predictions=modelrmsprop.predict(X_test) #using RMSprop activation function
     predictions[0]
```

```
array([1.3597429e-05, 1.2180487e-07, 7.8232057e-05, 3.2171196e-05,
       2.3163015e-10, 1.5118824e-09, 4.5473951e-11, 9.9984694e-01,
       3.1076670e-08, 2.8949122e-05], dtype=float32)
```

```
[23] plt.figure()
     plt.imshow(X_test[0], cmap=plt.cm.gray_r)
     plt.show()
```




```
[22] np.argmax(predictions[0])
```

```
7
```

Discussion: This CNN implementation has a lot of parameters that can push the accuracy higher and higher. Such as optimizers, activation functions, filtering, pooling necessary times, cross validation and many more. Plotting the validation and loss it is seen that, although the layers are similar, the inner workings are almost same, just setting different optimizers the accuracy increased a bit. Although it increased by rarely one percent we can say, but these parameters can increase more. For RMSprop, got the accuracy most and small loss comparably.

Though we're comparing between the optimizers, for which the change is seen, but all the accuracy rate touches around 98%.