Implementing CNN architecture to classify the MNIST handwritten dataset.

SAHA, ANKUR

American International University, Bangladesh

CSE Department

18-37913-2@student.aiub.edu

***Abstract*— This report is based on the differences between various types optimizers which will change the weights and learning rate while training the dataset in order to decrease the losses. The MNIST handwritten dataset from tensorflow library was used for training and the CNN (Convolution Neural Network) model architecture was designed in such way that the accuracy reaches over 98%. Optimizer named Adam, SGD, RMSProp were used to get the visual graph of accuracy and the loss for the trained dataset to determine which provides the best accuracy and lowest loss.**

Keywords—CNN, neural network, MNIST handwritten dataset, Adam, SGD, RMSProp.

I. Introduction

In the domains of AI, machine learning, and deep learning, neural networks mimic the function of the human brain, allowing computer programs to spot patterns and solve common problems. It is like a computer system made up of linked nodes that function similarly to neurons in the brain. They can discover hidden patterns and correlations in raw data using various types of algorithms, cluster and categorize it, and learn and improve over time by training. One commonly used neural network, CNN (Convolution Neural Network) is known for investigation of visual images. Multilayer perceptrons are regularized variants of CNNs. Multilayer perceptrons are typically completely connected networks, meaning that each neuron in one layer is linked to all neurons in the following layer. These networks' "complete connectedness" makes them vulnerable to data overfitting. Regularization, or preventing overfitting, can be accomplished in a variety of methods, including punishing parameters during training (such as weight loss) or reducing connectivity (skipped connections, dropout, etc.) CNNs use a different approach to regularization: they take advantage of the hierarchical pattern in data and use smaller and simpler patterns imprinted in their filters to construct patterns of increasing complexity. There are variations of datasets available for image processing to train the computer which then be able to predict the similar images by partitioning them into categories. MNIST is one of the most widely used deep learning datasets. It's a dataset of handwritten digits with 60,000 examples in the training set and 10,000 examples in the test set. It's an useful database for experimenting with deep learning algorithms and patterns on real world data while spending as little time and effort as possible on data preparation. In this project, we are going to use convolution neural network on MNIST dataset for classification. The target is to reach over 98% accuracy by modifying the model architecture by increasing the hidden layers and the neurons in them. At last, we will try different optimizers to get an idea on which one provides the lowest loss and highest accuracy.

II. Methodolgy & Result

We started the implementation using the jupyter notebook and python 3.8 where we first activated the environment which contains the tensorflow package. Then at the beginning, we imported all the library such as tensorflow, numpy and matplotlib required to process and for plotting the dataset.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Then we downloaded the dataset from the tensorflow library and loaded them into 2 tuples. The first tuple represents the train images and their corresponding labels and the 2nd tuple contains the test images and the labels.

```
(X_train, Y_train), (X_test, Y_test) = tf.keras.datasets.mnist.load_data()
```

```
print (X_train.shape)
print (Y_train.shape)
print (X_test.shape)
print (Y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

As the images are greyscale so the values that represents the color matrix varies from 0 to 255. So, to provide a new shape to the numpy array without changing the actual data. This process is known as data preprocessing which converts the color matrix value range from 0 to 1. This will result a huge increase in the training accuracy and it has also a direct impact on the success rate of the project.

```
X_train, X_test = X_train.astype('float32')/255, X_test.astype ('float32')/255
X_train = X_train.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)
```

Then we used the sequential type to build the model in keras. It allows us to build the model layer by layer. We declared the input shape and defined the hidden layers of the model using the Conv2D () function which contains the number of the filters, filter size and the activation function relu. After using the filter when we get the scalar matrix, then we use the maxpool2d () method to compress the size of the matrix by selecting the maximum values. At last, to define the output we used the dense method and the activation function as softmax.

```
model = tf.keras.Sequential ([
    tf.keras.Input(shape=(28,28,1)),

    tf.keras.layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2)),

    tf.keras.layers.Conv2D(filters=64, kernel_size=(5,5), activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2)),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation='relu'),
    tf.keras.layers.Dense(units=10, activation='softmax')
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 9, 9, 64)          51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 128)               131200
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 184,074
Trainable params: 184,074
Non-trainable params: 0
_________________________________________________________________
```

After that, we compiled the model using the model several times using different optimizers such as Adam, SGD & RMSProp and also calculated the loss using the sparse categorical crossentropy. The matrices were defined by the accuracy.

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

```
model_2.compile(
    optimizer=tf.keras.optimizers.SGD(),
    loss= tf.keras.losses.sparse_categorical_crossentropy,
    metrics=['accuracy']
)
```

```
model_3.compile(
    optimizer=tf.keras.optimizers.RMSprop(),
    loss= tf.keras.losses.sparse_categorical_crossentropy,
    metrics=['accuracy']
)
```

Then we started training the model using the fit () method and the iteration was done by 10 times. We also used the validation split for our training set to generate multiple splits of train and validation sets. Batch size were also defined to refer to the number of training examples utilized in one iteration.

```
h=model.fit(x=X_train, y=Y_train, epochs=10, validation_split= 0.2, batch_size=32)
```

```
Epoch 1/10
1500/1500 [==============================] - 20s 8ms/step - loss: 0.1426 - accuracy: 0.9549 - val_loss: 0.0597 - val_accuracy:
0.9827
Epoch 2/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0444 - accuracy: 0.9858 - val_loss: 0.0418 - val_accuracy:
0.9877
Epoch 3/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0285 - accuracy: 0.9906 - val_loss: 0.0367 - val_accuracy:
0.9893
Epoch 4/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0207 - accuracy: 0.9933 - val_loss: 0.0435 - val_accuracy:
0.9876
Epoch 5/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0158 - accuracy: 0.9947 - val_loss: 0.0524 - val_accuracy:
0.9865
Epoch 6/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0121 - accuracy: 0.9957 - val_loss: 0.0425 - val_accuracy:
0.9875
Epoch 7/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0090 - accuracy: 0.9970 - val_loss: 0.0350 - val_accuracy:
0.9915
Epoch 8/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0098 - accuracy: 0.9967 - val_loss: 0.0482 - val_accuracy:
0.9879
Epoch 9/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0065 - accuracy: 0.9979 - val_loss: 0.0402 - val_accuracy:
0.9915
Epoch 10/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0062 - accuracy: 0.9980 - val_loss: 0.0424 - val_accuracy:
0.9909
```

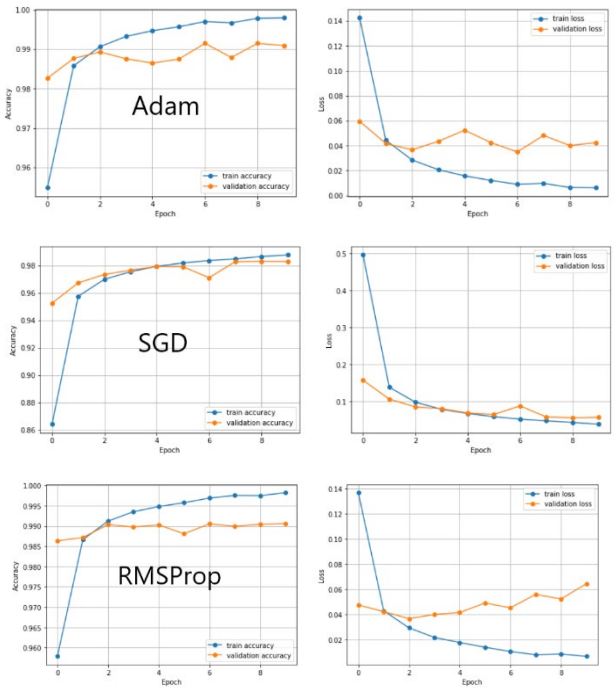
```
h_2=model_2.fit(x=X_train, y=Y_train, epochs=10, validation_split= 0.2, batch_size=32)
```

```
Epoch 1/10
1500/1500 [==============================] - 13s 9ms/step - loss: 0.4964 - accuracy: 0.8644 - val_loss: 0.1579 - val_accuracy:
0.9527
Epoch 2/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.1389 - accuracy: 0.9576 - val_loss: 0.1069 - val_accuracy:
0.9674
Epoch 3/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0985 - accuracy: 0.9700 - val_loss: 0.0858 - val_accuracy:
0.9735
Epoch 4/10
1500/1500 [==============================] - 13s 8ms/step - loss: 0.0792 - accuracy: 0.9755 - val_loss: 0.0808 - val_accuracy:
0.9767
Epoch 5/10
1500/1500 [==============================] - 13s 9ms/step - loss: 0.0679 - accuracy: 0.9795 - val_loss: 0.0693 - val_accuracy:
0.9793
Epoch 6/10
1500/1500 [==============================] - 13s 9ms/step - loss: 0.0591 - accuracy: 0.9819 - val_loss: 0.0654 - val_accuracy:
0.9792
Epoch 7/10
1500/1500 [==============================] - 13s 9ms/step - loss: 0.0527 - accuracy: 0.9837 - val_loss: 0.0885 - val_accuracy:
0.9712
Epoch 8/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.0481 - accuracy: 0.9849 - val_loss: 0.0585 - val_accuracy:
0.9828
Epoch 9/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.0438 - accuracy: 0.9866 - val_loss: 0.0563 - val_accuracy:
0.9832
Epoch 10/10
1500/1500 [==============================] - 14s 9ms/step - loss: 0.0389 - accuracy: 0.9878 - val_loss: 0.0575 - val_accuracy:
0.9830
```

```
h_3=model_3.fit(x=X_train, y=Y_train, epochs=10, validation_split= 0.2, batch_size=32)
```

```
Epoch 1/10
1500/1500 [==============================] - 17s 11ms/step - loss: 0.1368 - accuracy: 0.9579 - val_loss: 0.0476 - val_accuracy:
0.9863
Epoch 2/10
1500/1500 [==============================] - 16s 11ms/step - loss: 0.0431 - accuracy: 0.9866 - val_loss: 0.0424 - val_accuracy:
0.9872
Epoch 3/10
1500/1500 [==============================] - 16s 11ms/step - loss: 0.0294 - accuracy: 0.9912 - val_loss: 0.0367 - val_accuracy:
0.9903
Epoch 4/10
1500/1500 [==============================] - 16s 11ms/step - loss: 0.0218 - accuracy: 0.9935 - val_loss: 0.0401 - val_accuracy:
0.9898
Epoch 5/10
1500/1500 [==============================] - 17s 11ms/step - loss: 0.0178 - accuracy: 0.9948 - val_loss: 0.0416 - val_accuracy:
0.9902
Epoch 6/10
1500/1500 [==============================] - 16s 11ms/step - loss: 0.0140 - accuracy: 0.9957 - val_loss: 0.0492 - val_accuracy:
0.9881
Epoch 7/10
1500/1500 [==============================] - 17s 11ms/step - loss: 0.0106 - accuracy: 0.9969 - val_loss: 0.0453 - val_accuracy:
0.9905
Epoch 8/10
1500/1500 [==============================] - 19s 13ms/step - loss: 0.0081 - accuracy: 0.9975 - val_loss: 0.0560 - val_accuracy:
0.9899
Epoch 9/10
1500/1500 [==============================] - 17s 12ms/step - loss: 0.0086 - accuracy: 0.9975 - val_loss: 0.0524 - val_accuracy:
0.9904
Epoch 10/10
1500/1500 [==============================] - 20s 13ms/step - loss: 0.0068 - accuracy: 0.9982 - val_loss: 0.0645 - val_accuracy:
0.9906
```

Then, we calculated the training accuracy, validation accuracy, training loss and the validation loss and plotted them in the graph.



Then we evaluated the model with test dataset and tried to find the best accuracy with less error.

```
test_loss,test_acc = model.evaluate (X_test, Y_test)
print('\nTest Accuracy:', test_acc)
print('\nTest Loss:', test_loss)
```

```
313/313 [==============================] - 2s 6ms/step - loss: 0.0331 - accuracy: 0.9918

Test Accuracy: 0.9918000102043152

Test Loss: 0.03313888609409332
```

Adam

```
test_loss,test_acc = model_2.evaluate (X_test, Y_test)
print('\nTest Accuracy:', test_acc)
print('\nTest Loss:', test_loss)
```

```
313/313 [==============================] - 2s 7ms/step - loss: 0.0488 - accuracy: 0.9831

Test Accuracy: 0.9830999970436096

Test Loss: 0.048768118023872375
```

SGD

```
test_loss,test_acc = model_3.evaluate (X_test, Y_test)
print('\nTest Accuracy:', test_acc)
print('\nTest Loss:', test_loss)
```

```
313/313 [==============================] - 2s 7ms/step - loss: 0.0592 - accuracy: 0.9914

Test Accuracy: 0.9914000034332275

Test Loss: 0.05916814133524895
```

RMSProp

III. DISCUSSION

Implementing the CNN model architecture on the MNIST dataset, we were able to achieve more than 98% accuracy in predicting the image pattern. We can say that the training was quite successful. In 3 different models with different optimizers we can clearly see from the plotting that if we want to rate the optimizers according to their performance, we can provide them by from their values. Using Adam optimizer, we got the train accuracy 99.80% and loss 0.0062. For the test dataset the accuracy was 99.18% and loss was 0.0331. For SGD, training set gave us accuracy of 98.78% and loss of 0.0389. Test images got 98.30% accuracy and 0.0487 loss. Finally, from RMSProp, we got the accuracy for training image 99.82% and loss 0.0068. For test images, the accuracy was 99.14% and loss was 0.0591. From the values, we can tell that, the Adam and RMSProp optimizer gave us the best accuracy and lowest loss. Between them Adam is the highest rated one. On the other hand, SGD also has done a nice job in training by getting over 98% accuracy. Finally we can come to a conclusion that Adam is providing the best accuracy in MNIST dataset implemented with CNN architecture.