Implementing CNN architecture to classify the MNIST Handwritten dataset

Ahmmed, Ehsanul
American International University Bangladesh
Dept:CSE
18-37857-2@student.aiub.edu

***Abstract*—It is known to all that A convolutional neural network (CNN) is a form of artificial neural network that is especially intended to analyze pixel input and is used in image recognition and processing. There is some models in architecture. These models are used to train the dataset. This project report is based on the differences between various types of optimizers which will change the weights and learning rates while training the dataset in order to decrease the losses. As there is a lot dataset in tensorflow library we have chosen the MNIST Handwritten dataset. The main goal is to reach over 98% accuracy. In this project we have used Adam, SGD and RMSprop optimizers.**

Keywords— convolutional, neural, pixel, optimizers, MNIST , Adam, SGD, RMSprop

I. Introduction

A neural network is a set of algorithms that attempts to detect underlying relationships in a batch of data using a method that mimics how the human brain works. Neural networks, in this context, refer to systems of neurons that can be biological or artificial in nature. Because neural networks can adapt to changing input, they can produce the best possible outcome without requiring the output criteria to be redesigned. Convolutional Neural Network is a type of neural network that is used for processing pixel data. 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. A convolutional layer's principal function is to recognize characteristics such as edges, lines, color blobs, and other visual aspects. These characteristics are detectable by the filters. The more filters a convolutional layer receives, the more characteristics it can identify. A square-shaped device that scans the image is known as a filter. Individual grid pixels can be represented by a grid. The convolutional layer may be seen of as a smaller grid that sweeps across each row of the picture from left to right. There are different kinds of datasets available in Tensorflow library for image processing to train the machine which will be able to predict the similar types of images from the classification of images. MNIST handwritten dataset is a dataset which is consist of 50000 train images and 10000 test images. It is an useful database for training the datasets with those algorithms and different types of filters to increase the accuracy of 98%. In this project we will use a lot hidden layers which will increase the accuracy.

II. Methodology and Result

At the start of the process, we activated the tensorflow environment which contains the tensorflow packages. Then we imported the library which will be needed to run the process. The libraries are tensorflow, matplotlib, numpy. Numpy and matplotlib will be used for calculation and plotting.

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

Then the dataset was downloaded from the tensorflow library and loaded then as 2 tuples. The first tuple holds the train images and the other one holds 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 which represents the color matrix varies from 0 to 255. So we have to take into new shape to the numpy array without changing the actual data. This process is called data preprocessing which takes the value of color range from 0 to 1. This process will help to gain the faster calculation and get the more accurate result.

```
X_train, X_test =X_train.astype('float32')/255, X_test.astype('float32')/255
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
```

Sequential type was used to build the model in keras, so that the model will be build sequentially. In this model we declared the input shape and defined the hidden layer as convo2D() function which contains

the number of filters, filter size and the activation function ‘relu’. When we acquire the scalar matrix after applying the filter, we use the maxpool2d () function to reduce the size of the matrix by picking the maximum values. Finally, we utilized the dense technique to define the output, with softmax as the activation function.

```
In [9]: model=tf.keras.Sequential(
        [
            tf.keras.Input(shape=(28,28,1)),
            tf.keras.layers.Conv2D(filters=64,kernel_size=(5,5), activation='relu'),
            tf.keras.layers.MaxPool2D(pool_size=(2,2)),
            tf.keras.layers.Conv2D(filters=128,kernel_size=(3,3), activation='relu'),
            tf.keras.layers.MaxPool2D(pool_size=(2,2)),
            tf.keras.layers.Conv2D(filters=128,kernel_size=(3,3), 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, 24, 24, 64)	1664
max_pooling2d (MaxPooling2D)	(None, 12, 12, 64)	0
conv2d_1 (Conv2D)	(None, 10, 10, 128)	73856
max_pooling2d_1 (MaxPooling2	(None, 5, 5, 128)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	147584
max_pooling2d_2 (MaxPooling2	(None, 1, 1, 128)	0
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 128)	16512

Then the model was compiled several times using different optimizers such as Adam, SGD and RMSprop and also calculated the loss using the sparse categorial crossentropy. This matrices were defined by the accuracy.

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

```
model2.compile(
    optimizer=tf.keras.optimizers.SGD(),
    loss=tf.keras.losses.sparse_categorical_crossentropy,
    metrics=['accuracy']
)
```

```
model3.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 with 20 epochs and we used validation split for our training dataset for generating multiple splits and validation sets. To define the number of training examples utilized in one epoch we used Batch size.

```
h=model.fit(x=X_train, y=Y_train, epochs=20, validation_split=0.3, batch_size=64)
```

```
Epoch 1/20
657/657 [==============================] - 14s 19ms/step - loss: 0.2294 - accuracy: 0.9292 - val_loss: 0.0818 - val_a
ccuracy: 0.9748
Epoch 2/20
657/657 [==============================] - 11s 17ms/step - loss: 0.0641 - accuracy: 0.9807 - val_loss: 0.0702 - val_a
ccuracy: 0.9794
Epoch 3/20
657/657 [==============================] - 12s 18ms/step - loss: 0.0456 - accuracy: 0.9854 - val_loss: 0.0540 - val_a
ccuracy: 0.9842
Epoch 4/20
657/657 [==============================] - 12s 19ms/step - loss: 0.0344 - accuracy: 0.9891 - val_loss: 0.0549 - val_a
ccuracy: 0.9830
Epoch 5/20
657/657 [==============================] - 12s 19ms/step - loss: 0.0254 - accuracy: 0.9916 - val_loss: 0.0437 - val_a
ccuracy: 0.9877
Epoch 6/20
657/657 [==============================] - 12s 18ms/step - loss: 0.0211 - accuracy: 0.9931 - val_loss: 0.0411 - val_a
ccuracy: 0.9887
Epoch 7/20
657/657 [==============================] - 12s 18ms/step - loss: 0.0176 - accuracy: 0.9944 - val_loss: 0.0408 - val_a
ccuracy: 0.9891
```

```
h2=model2.fit(x=X_train, y=Y_train, epochs=20, validation_split=0.3, batch_size=64)
```

```
Epoch 1/20
657/657 [==============================] - 54s 82ms/step - loss: 1.4511 - accuracy: 0.6167 - val_loss: 1.2675 - val_a
ccuracy: 0.6163
Epoch 2/20
657/657 [==============================] - 53s 81ms/step - loss: 0.3331 - accuracy: 0.8994 - val_loss: 0.2389 - val_a
ccuracy: 0.9261
Epoch 3/20
657/657 [==============================] - 53s 81ms/step - loss: 0.2054 - accuracy: 0.9378 - val_loss: 0.1927 - val_a
ccuracy: 0.9406
Epoch 4/20
657/657 [==============================] - 53s 81ms/step - loss: 0.1602 - accuracy: 0.9510 - val_loss: 0.1324 - val_a
ccuracy: 0.9601
Epoch 5/20
657/657 [==============================] - 53s 81ms/step - loss: 0.1298 - accuracy: 0.9612 - val_loss: 0.1238 - val_a
ccuracy: 0.9620
```

```
h3=model3.fit(x=X_train, y=Y_train, epochs=20, validation_split=0.3, batch_size=64)
```

```
Epoch 1/20
657/657 [==============================] - 56s 84ms/step - loss: 0.2342 - accuracy: 0.9268 - val_loss: 0.0792 - val_a
ccuracy: 0.9769
Epoch 2/20
657/657 [==============================] - 55s 83ms/step - loss: 0.0594 - accuracy: 0.9817 - val_loss: 0.0543 - val_a
ccuracy: 0.9833
Epoch 3/20
657/657 [==============================] - 55s 83ms/step - loss: 0.0404 - accuracy: 0.9878 - val_loss: 0.0490 - val_a
ccuracy: 0.9861
Epoch 4/20
657/657 [==============================] - 55s 83ms/step - loss: 0.0287 - accuracy: 0.9910 - val_loss: 0.0453 - val_a
ccuracy: 0.9876
Epoch 5/20
657/657 [==============================] - 55s 83ms/step - loss: 0.0232 - accuracy: 0.9929 - val_loss: 0.0775 - val_a
ccuracy: 0.9823
Epoch 6/20
657/657 [==============================] - 55s 83ms/step - loss: 0.0179 - accuracy: 0.9945 - val_loss: 0.0598 - val_a
ccuracy: 0.9857
```

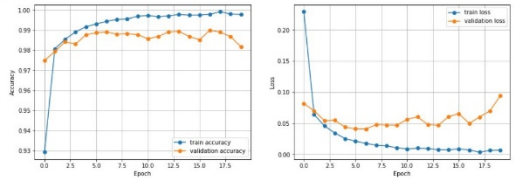
Next, we calculated the accuracy, validation accuracy, training loss and validation loss and plotted them in a graph.

```
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(h.history['accuracy'], 'o-', label='train accuracy')
plt.plot(h.history['val_accuracy'], 'o-', label = 'validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend(loc='lower right')

plt.subplot(1,2,2)
plt.plot(h.history['loss'], 'o-', label='train loss')
plt.plot(h.history['val_loss'], 'o-', label='validation loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.legend(loc='upper right')

plt.show()
```

Adam



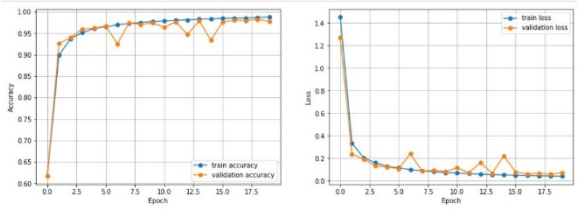

```
test_loss,test_acc = model.evaluate (X_test, Y_test)
```

```
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(h2.history['accuracy'], 'o-', label='train accuracy')
plt.plot(h2.history['val_accuracy'], 'o-', label = 'validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend(loc='lower right')

plt.subplot(1,2,2)
plt.plot(h2.history['loss'], 'o-', label='train loss')
plt.plot(h2.history['val_loss'], 'o-', label='validation loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.legend(loc='upper right')

plt.show()
```

SGD

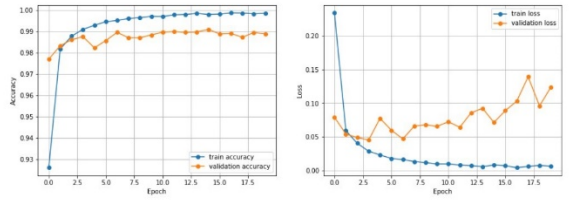


```
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(h3.history['accuracy'], 'o-', label='train accuracy')
plt.plot(h3.history['val_accuracy'], 'o-', label = 'validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend(loc='lower right')

plt.subplot(1,2,2)
plt.plot(h3.history['loss'], 'o-', label='train loss')
plt.plot(h3.history['val_loss'], 'o-', label='validation loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.legend(loc='upper right')

plt.show()
```

RMSprop



Then we tested the models with the test datasets and tried to find the best accuracy and less error.

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

```
313/313 [==============================] - 5s 15ms/step - loss: 0.0975 - accuracy: 0.9804

Test Accuracy: 0.980400025844574

Test Loss: 0.0975005179643631
```

Adam=
Test Accuracy: 0.980400025844574
Test Loss: 0.0975005179643631

```
test_loss,test_acc = model2.evaluate (X_test, Y_test)
print('\nTest Accuracy:', test_acc)
print('\nTest Loss:', test_loss)
```

```
313/313 [==============================] - 5s 15ms/step - loss: 0.0622 - accuracy: 0.9816

Test Accuracy: 0.9815999865531921

Test Loss: 0.06224692240357399
```

SGD=
Test Accuracy: 0.9815999865531921
Test Loss: 0.06224692240357399

```
test_loss,test_acc = model3.evaluate (X_test, Y_test)
print('\nTest Accuracy:', test_acc)
print('\nTest Loss:', test_loss)
```

```
313/313 [==============================] - 5s 15ms/step - loss: 0.0929 - accuracy: 0.9916

Test Accuracy: 0.991599977016449

Test Loss: 0.09294538944959640
```

RMSprop=
Test Accuracy: 0.991599977016449
Test Loss: 0.09294538944959640

III. Discussion

In this project while we were implementing the CNN model architecture on the MNIST dataset and targeted the accuracy of 98%. As we have used 3 different types of optimizers those give us different types of accuracy , validation accuracy and loss. Then we try to test it in test data set. And try to get the accuracy and loss.

For Adam, The accuracy for training dataset is 99.78% and loss is 0.0070. Using this model for testing dataset we have got the accuracy of 98.04% and loss of .0975.

For SGD, The accuracy for training dataset is 98.77% and loss is 0.0718. Using this model for testing dataset we have got the accuracy of 98.15% and loss of .062247.

For RMSprop, The accuracy for training dataset is 99.85% and loss is 0.1237. Using this model for testing dataset we have got the accuracy of 99.15 % and loss of .09294.

As we can see, for the Adam we got a good accuracy on training and testing dataset both also get less validation loss. For the SGD, we have got lowest accuracy rate for both training and testing and got a decent loss. For RMSprop, we have got the highest accuracy rate but not a good loss rate.

So overall, according to accuracy and loss adam optimizer provides the best result.