Computer Vision And Pattern Recognition [B]

Mid Project Report

Id: 18-39135-3

Table of content	

Title: Evaluation of proposed CNN model to classify the MNIST handwritten dataset

Abstract

Implementing the CNN architecture to classify the MNIST dataset. In conventional artificial neural network we get 97% accuracy for MNIST dataset. In artificial neural network the duration of the network is unknown, and also the result we get from the ANN is not optimum always. So to get more accuracy and more optimum result for MNIST dataset we will using CNN architecture to classify the dataset.

Introduction

A CNN is also a class of artificial neural network but the difference is CNN is mostly use for visualizing images. A convolutional Neural network taken data, train them self to recognize the pattern in this data and predict the output for a new set of similar data. We are using MNIST dataset this dataset consists of handwritten digits from 0 to 9 and it provides training and testing images for processing.

STEPS:

Importing dataset:

Before starting the further process we import the MNIST dataset using keras. From **keras.dataset** module we import the MNIST function which contains the dataset. After importing the dataset we load the whole dataset into the variable data using **mnist.load_data()**.

Split the dataset into test and train:

After load the dataset we directly split the whole dataset into training and testing set. We are initializes those data using four variables **X_train, Y_train, X_test and Y_test**. MNIST dataset contains the handwritten digit images and the shape of each image is 28x28 in size means image has 28pixels x 28pixels. Reshaping the images in such way so that we can access every pixel of the image. The reason for accessing every pixel is that so that we can apply deep learning ideas and can assign color code to every pixel. Each of the pixel has its unique color and also it has maximum value 255. So for converting all the values from 0 to 255 for every pixel to a range

of values from 0 to 1, dividing all the value of every pixel by 255 to get the values in the range of 0 to 1.

```
X_train = X_train.reshape((60000,28,28))
X_train= X_train.astype('float32') / 255

 X_test = X_test.reshape((10000,28,28))
 X_test= X_test.astype('float32') / 255
```

Creating the model:

Importing the Sequential and Dense function to performs the deep learning and which are available under the keras library.

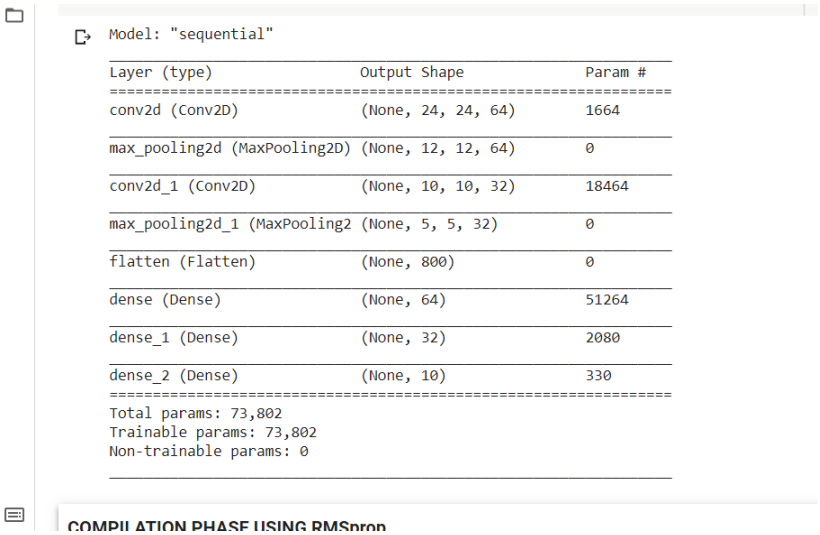
Storing the Sequential function into the model variable so that it became easier to access.

We start with the input layer which is defined as 28, 28.

We start with two convolutional layer with small kernel size (5, 5) and (3, 3) and number of filters (64) and (32) followed by max pooling layer (2,2). Conv2D layer consist set of filter and each filter can detect specific pattern. Using maxpooling (2,2) to reduce the input and maxpooling (2,2) means selects the max value from each 2x2 so that output will became half. We are using **relu** as our activation function. Using the flatten function converting the shape into 1D.

We need an output layer with 10 nodes in order to predict the probability of an image belonging to each of the 10 classes and this will use **softmax** as activation function. Between the feature layer and the output layer, we add a dense layer to interpret the features, in this case with 64 nodes and use **relu** as activation.

We use **model.summary()** which provides brief details about our model:



```
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, 32)        18464
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 800)               0
_________________________________________________________________
dense (Dense)                (None, 64)                51264
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 10)                330
=================================================================
Total params: 73,802
Trainable params: 73,802
Non-trainable params: 0
_________________________________________________________________
```

COMPILATION PHASE USING RMSprop

COMPILATION PHASE:

After getting the brief detail about the model we compile the entire model and for compiling the model we use different optimizer such as **RMSprop(), Adam()** and **SGD.** The uses of optimizer is that optimizer will defined how the models update itself. We use **categorical_crossentropy** as our loss function which is measured the performance on the train data. And use **accuracy** as our matrics to evaluate our model.

Train the model:

Using the **fit()** function train the model which takes the train set as input, set the epochs = 5 and 10 based on the different optimizer and set validation_split = 0.2 ans set the batch_size = 64.

Epoch means training the neural network with all the training data for one cycle. An epoch is for one forward pass and one backward pass for all the images. We send the model to train 10 times to get high accuracy. We could changes the number of epoch depending on the model performance.

Batch_size is the number of training examples in one forward and one backward pass. The higher the batch_size the more memory we will need. We send 64 images to train as a batch per iteration.

```
                            #which is the number of correctly labled images
)
```

```
n = model.fit(x = X_train, y = Y_train, epochs=5, validation_split=0.2, batch_size=64
```

Predicting accuracy:

To know how well the model works over the testing dataset use the **evaluate()** function which takes the test set as the input. This compute the loss and the accuracy of the model over the test set.

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

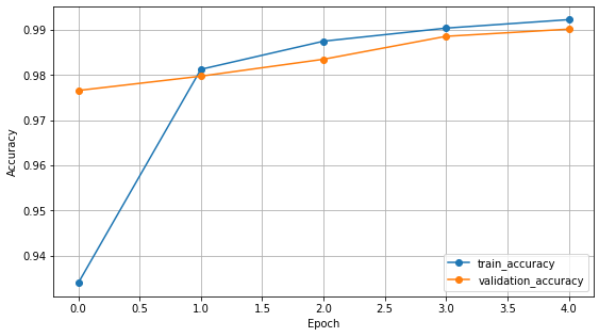
```
313/313 [==============================] - 1s 4ms/st
```

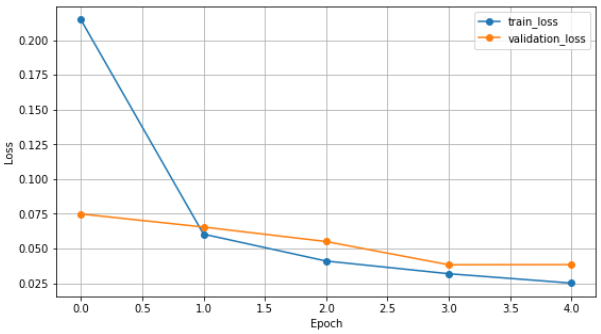
Result

When we were implement conventional artificial neural network we get 97% accuracy for MNIST dataset. But in CNN we get the higher accuracy for MNIST dataset.

Using RMSprop:

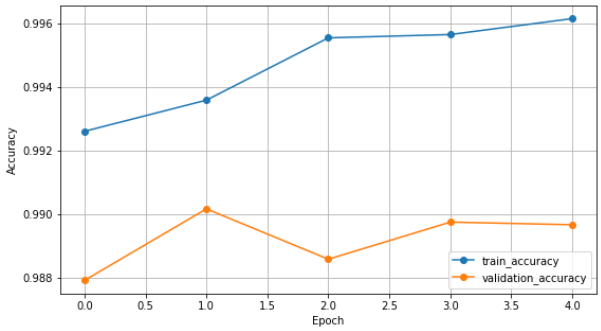
```
Epoch 1/5
750/750 [==============================] - 38s 10ms/step - loss: 0.2147 - accuracy: 0.9339 - val_loss: 0.0749 - val_accuracy: 0.9766
Epoch 2/5
750/750 [==============================] - 6s 9ms/step - loss: 0.0604 - accuracy: 0.9813 - val_loss: 0.0655 - val_accuracy: 0.9797
Epoch 3/5
750/750 [==============================] - 6s 9ms/step - loss: 0.0410 - accuracy: 0.9875 - val_loss: 0.0551 - val_accuracy: 0.9835
Epoch 4/5
750/750 [==============================] - 6s 9ms/step - loss: 0.0319 - accuracy: 0.9904 - val_loss: 0.0383 - val_accuracy: 0.9886
Epoch 5/5
750/750 [==============================] - 6s 9ms/step - loss: 0.0251 - accuracy: 0.9923 - val_loss: 0.0384 - val_accuracy: 0.9902
```

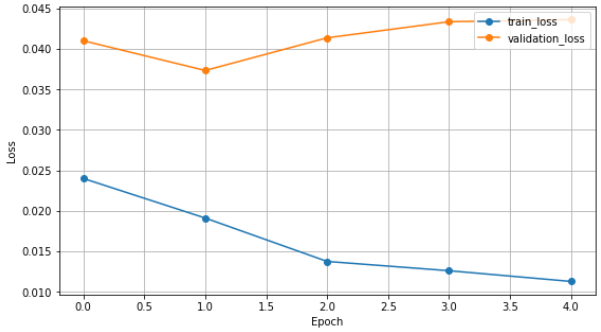




Using Adam:

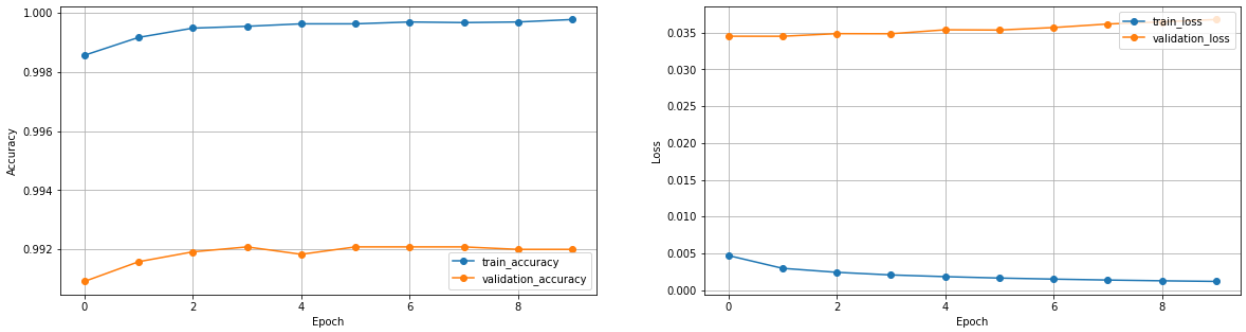
```
Epoch 1/5
750/750 [==============================] - 6s 8ms/step - loss: 0.0240 - accuracy: 0.9926 - val_loss: 0.0410 - val_accuracy: 0.9879
Epoch 2/5
750/750 [==============================] - 6s 8ms/step - loss: 0.0191 - accuracy: 0.9936 - val_loss: 0.0373 - val_accuracy: 0.9902
Epoch 3/5
750/750 [==============================] - 5s 7ms/step - loss: 0.0137 - accuracy: 0.9955 - val_loss: 0.0414 - val_accuracy: 0.9886
Epoch 4/5
750/750 [==============================] - 5s 7ms/step - loss: 0.0126 - accuracy: 0.9956 - val_loss: 0.0434 - val_accuracy: 0.9898
Epoch 5/5
750/750 [==============================] - 6s 8ms/step - loss: 0.0113 - accuracy: 0.9961 - val_loss: 0.0436 - val_accuracy: 0.9897
```





Using SGD:

```
Epoch 1/10
750/750 [==============================] - 6s 8ms/step - loss: 0.0047 - accuracy: 0.9986 - val_loss: 0.0345 - val_accuracy: 0.9909
Epoch 2/10
750/750 [==============================] - 6s 8ms/step - loss: 0.0030 - accuracy: 0.9992 - val_loss: 0.0345 - val_accuracy: 0.9916
Epoch 3/10
750/750 [==============================] - 5s 7ms/step - loss: 0.0024 - accuracy: 0.9995 - val_loss: 0.0349 - val_accuracy: 0.9919
Epoch 4/10
750/750 [==============================] - 5s 7ms/step - loss: 0.0021 - accuracy: 0.9995 - val_loss: 0.0349 - val_accuracy: 0.9921
Epoch 5/10
750/750 [==============================] - 6s 8ms/step - loss: 0.0018 - accuracy: 0.9996 - val_loss: 0.0354 - val_accuracy: 0.9918
Epoch 6/10
750/750 [==============================] - 6s 8ms/step - loss: 0.0016 - accuracy: 0.9996 - val_loss: 0.0354 - val_accuracy: 0.9921
Epoch 7/10
750/750 [==============================] - 5s 7ms/step - loss: 0.0015 - accuracy: 0.9997 - val_loss: 0.0357 - val_accuracy: 0.9921
Epoch 8/10
750/750 [==============================] - 6s 8ms/step - loss: 0.0014 - accuracy: 0.9997 - val_loss: 0.0362 - val_accuracy: 0.9921
Epoch 9/10
750/750 [==============================] - 5s 7ms/step - loss: 0.0013 - accuracy: 0.9997 - val_loss: 0.0365 - val_accuracy: 0.9920
Epoch 10/10
750/750 [==============================] - 5s 7ms/step - loss: 0.0012 - accuracy: 0.9998 - val_loss: 0.0368 - val_accuracy: 0.9920
```



From all the above graph we can see for all the cases where the model achieves perfect skill and the percentage is 98% to 99% which is really a good result.

Overview of the model performance:

Optimizer	Train Accuracy	Validation Accuracy	Test Accuracy
RMSprop	99%	99%	98%
Adam	99%	98%	99%
SGD	99%	99%	99%

Discussion

From the above result we can say that for different optimizers, model predict different accuracy and different loss.

If we talk about the higher train accuracy than SGD works much better as the accuracy is near to 100% but if we talk about higher validation accuracy than RMSprop works much better as the accuracy is near to 99%.

If we talk about the loss and loss should be idle close to 0 so for both the training and validation loss RMSprop works much better which is nearly close to 0.