DEEP LEARNING (CS 6005)

MINI PROJECT ASSIGNMENT 1

CNN MODEL TO RECOGNIZE THE HAND WRITTEN DIGITS

BY

NAME: SRIDHAR.S

ROLL NO: 2018103068

BATCH: P

DATE: 20.04.2021

PROBLEM STATEMENT:

The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9. The **task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively.**

DATASET DESCRIPTION:

The MNIST database, an extension of the NIST database, is a low-complexity data collection of handwritten digits used to train and test various supervised machine learning algorithms. The database contains 70,000 28x28 black and white images representing the digits zero through nine. The data is split into two subsets, with 60,000 images belonging to the training set and 10,000 images belonging to the testing set. The separation of images ensures that given what an adequately trained model has learned previously, it can accurately classify relevant images not previously examined.

Purpose of MNIST Database and its Applications

In simple terms, MNIST can be thought of as the “Hello, World!” of machine learning. MNIST is primarily used to experiment with different machine learning algorithms and to compare their relative strengths. Yann LeCun, one of the three researchers behind the creation of MNIST, has devoted a portion of his research to using MNIST to experiment with cutting edge algorithms, which can be seen on his personal website yann.lecun.com. Many researchers, hobbyists, and students alike continue to use MNIST alongside their algorithmic implementations and other popular datasets as a way to solidify their understanding of the fundamental concepts in machine learning and to compare their new algorithms against existing cutting edge research.

MNIST Dataset File Formats

The data is stored in a very simple file format designed for storing vectors and multidimensional matrices. General info on this format is given at the end of this page, but you don't need to read that to use the data files.

All the integers in the files are stored in the MSB first (high endian) format used by most non-Intel processors. Users of Intel processors and other low-endian machines must flip the bytes of the header.

There are 4 files:

- train-images-idx3-ubyte: training set images
- train-labels-idx1-ubyte: training set labels
- t10k-images-idx3-ubyte: test set images
- t10k-labels-idx1-ubyte: test set labels

The training set contains 60000 examples, and the test set 10000

MODULES:

Module 1: Importing Necessary libraries

The necessary libraries which are used to build a CNN model like Keras, Sequential, Dense, Dropout, Flatten, Conv2D, MaxPooling2D etc are imported

Module 2: Data Preparation

In this module the data is processed the various steps involved in processing dataset include:

- Loading data from **KERAS**
- Reshaping the data (convert data into **0's and 1's**)
- Label encoding (Convert the target label **to_categorical**)
- Split training and validiation set

Module 3: Building a Convolutional Neural Network model

The CNN model is build using various layers like:

- **Conv2D**: The convolutional (Conv2D) layer. It is like a set of learnable filters. I choosed to set **32 filters** for the **first conv2D** layers and **64 filters** for the **second** layers. Each filter transforms a part of the image (defined by the kernel size) using the kernel filter. The kernel filter matrix is applied on the whole image. Filters can be seen as a transformation of the image. The features will be "extracted" from the image.

- **MaxPooling2D**: It acts as a **downsampling filter**. It looks at the 2 neighboring pixels and picks the maximal value. These are used to reduce computational cost, and to some extent also reduce overfitting. We have to choose the pooling size more the pooling dimension is high, more the downsampling is important. The **images get half sized.**

- **Flatten**: Transforms the format of the images from a 2d-array to a 1d-array

- **Regularization function**: I used **Dropout** technique to regularize which specifies the percentage of neurons to be droped at each iteration.

- **Activation function** used:

  - Hidden layer - **Relu**: given a value x, returns max(x, 0).
  - Output layer - **Softmax:** 10 neurons, probability that the image belongs to one of the classes.

The model built is compiled is compiled with parameters such as:

- **Optimizer**: **adam** = RMSProp + Momentum.
  - Momentum = takes into account past gradient to have a better update.
  - RMSProp = exponentially weighted average of the squares of past gradients.
- **Loss function**: I used **categorical_crossentropy** for classification, each images belongs to one class only

Module 4: Model Evaluation

- The model is evaluated by constructing **Accuracy and loss curves**
- The model is also evaluated by constructing **Confusion Matrix**

Module 5: Prediction

- The model predicts the target label for the test sample
- The output is checked for sample input

PROGRAM SNAPSHOTS:

MODULE 1:

#Importing libraries

```
import keras

from keras import backend as k
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
import matplotlib.pyplot as plt

from sklearn.metrics import confusion_matrix
import numpy as np
import seaborn as sns
```

MODULE 2:

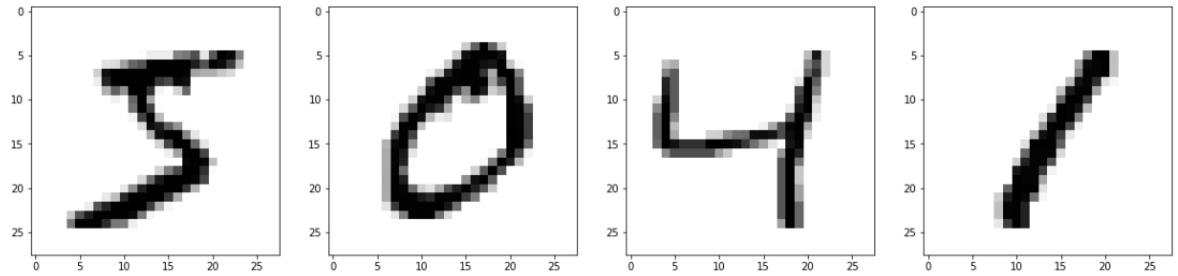
#Loading and viewing data:

```
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
X_train_ = X_train.reshape(X_train.shape[0], 28, 28)

fig, axis = plt.subplots(1, 4, figsize=(20, 10))
for i, ax in enumerate(axis.flat):
    ax.imshow(X_train_[i], cmap='binary')
```



#Viewing Shape of data:

```
print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", y_test.shape)
```

```
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

```
img_rows , img_cols = 28, 28
```

```
num_category = 10
```

#Reshaping data:

```
if k.image_data_format() == 'channels_first':
    X_train = X_train.reshape(X_train.shape[0], 1, img_rows, img_cols)
    X_test = X_test.reshape(X_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
    X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

```
X_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

#One hot encoding target labels:

```
y_train = keras.utils.to_categorical(y_train, num_category)
y_test = keras.utils.to_categorical(y_test, num_category)
y_train[0]
```

```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

MODULE 3:

#Building CNN Model:

```
model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(num_category, activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
```

#Fitting the model:

```
batch_size = 128
num_epoch = 15
model_log =model.fit(X_train, y_train, batch_size=128, epochs=10, validation_split = 0.20)
```

```
Epoch 1/10
375/375 [==============================] - 41s 109ms/step - loss: 0.3554 - accuracy: 0.8899 - val_loss: 0.0811 - val_accuracy:
0.9743
Epoch 2/10
375/375 [==============================] - 41s 110ms/step - loss: 0.1191 - accuracy: 0.9639 - val_loss: 0.0552 - val_accuracy:
0.9834
Epoch 3/10
375/375 [==============================] - 42s 111ms/step - loss: 0.0887 - accuracy: 0.9738 - val_loss: 0.0497 - val_accuracy:
0.9853
Epoch 4/10
375/375 [==============================] - 41s 108ms/step - loss: 0.0717 - accuracy: 0.9783 - val_loss: 0.0431 - val_accuracy:
0.9876
Epoch 5/10
375/375 [==============================] - 42s 113ms/step - loss: 0.0648 - accuracy: 0.9808 - val_loss: 0.0381 - val_accuracy:
0.9881
Epoch 6/10
375/375 [==============================] - 38s 102ms/step - loss: 0.0553 - accuracy: 0.9830 - val_loss: 0.0380 - val_accuracy:
0.9896
Epoch 7/10
375/375 [==============================] - 41s 110ms/step - loss: 0.0502 - accuracy: 0.9846 - val_loss: 0.0360 - val_accuracy:
0.9902
Epoch 8/10
375/375 [==============================] - 41s 110ms/step - loss: 0.0446 - accuracy: 0.9862 - val_loss: 0.0398 - val_accuracy:
0.9886
Epoch 9/10
375/375 [==============================] - 41s 108ms/step - loss: 0.0424 - accuracy: 0.9868 - val_loss: 0.0337 - val_accuracy:
0.9906
Epoch 10/10
375/375 [==============================] - 41s 108ms/step - loss: 0.0387 - accuracy: 0.9870 - val_loss: 0.0361 - val_accuracy:
0.9899
```

MODULE 4:

#Evaluating the model & Printing Test loss & Test Accuracy:

```
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.026397110894322395
Test accuracy: 0.9919000267982483
```

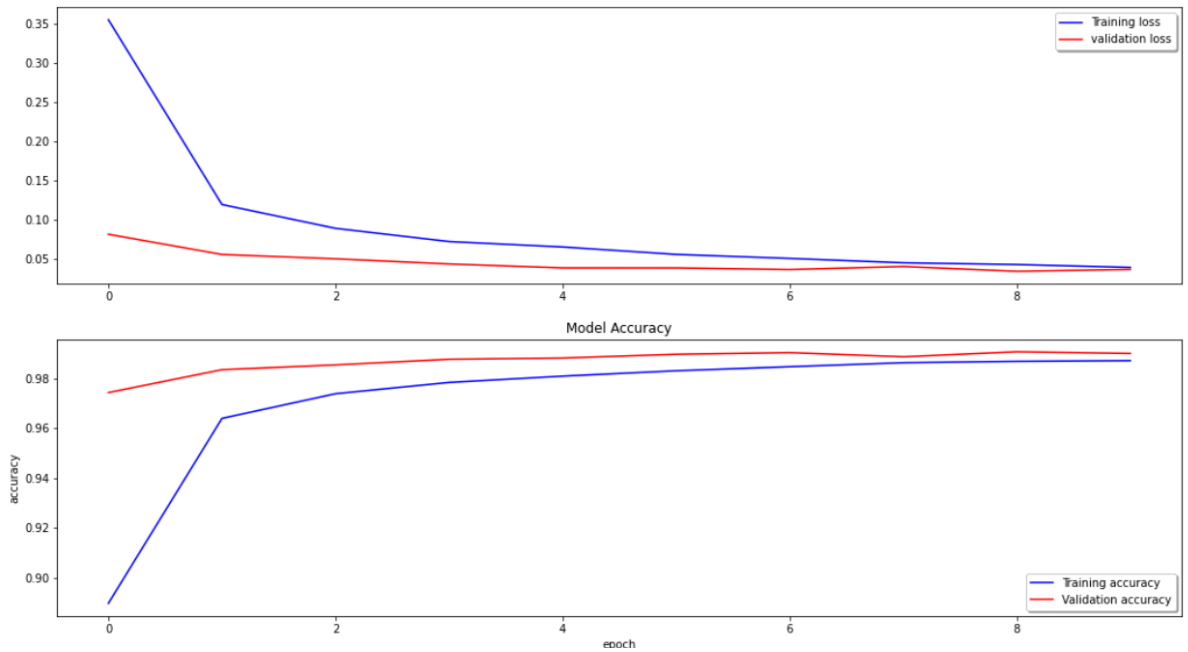
#Printing Accuracy and Loss Curves:

```
fig, ax = plt.subplots(2,1, figsize=(18, 10))

ax[0].plot(model_log.history['loss'], color='b', label="Training loss")
ax[0].plot(model_log.history['val_loss'], color='r', label="validation loss")
plt.title('Model Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(model_log.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(model_log.history['val_accuracy'], color='r',label="Validation accuracy")
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')

legend = ax[1].legend(loc='best', shadow=True)
```



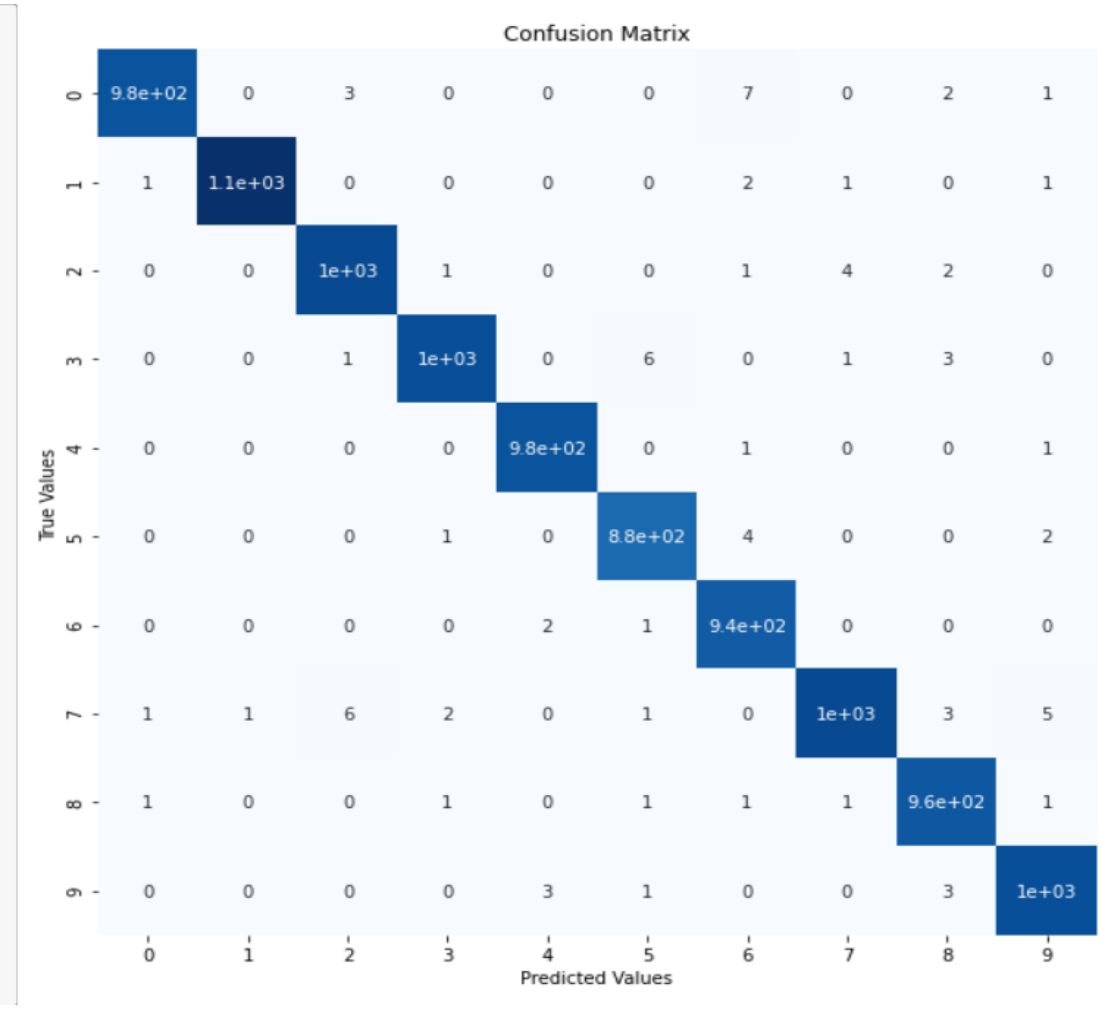
#Printing Confusion Matrix:

```
fig = plt.figure(figsize=(10, 10))

y_pred = model.predict(X_test)
Y_pred = np.argmax(y_pred, 1)
Y_test = np.argmax(y_test, 1)

mat = confusion_matrix(Y_test, Y_pred)

sns.heatmap(mat.T, square=True, annot=True, cbar=False, cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Values')
plt.ylabel('True Values');
plt.show();
```

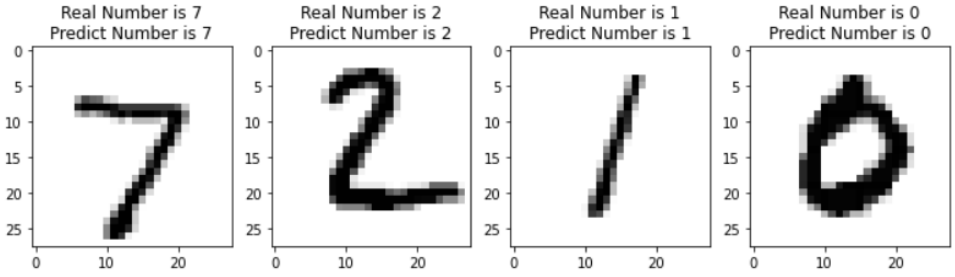


MODULE 5:

#Predicting output for test sample

```
y_pred = model.predict(X_test)
X_test_ = X_test.reshape(X_test.shape[0], 28, 28)

fig, axis = plt.subplots(1, 4, figsize=(12, 14))
for i, ax in enumerate(axis.flat):
    ax.imshow(X_test_[i], cmap='binary')
    ax.set(title = f"Real Number is {y_test[i].argmax()}\nPredict Number is {y_pred[i].argmax()}");
```

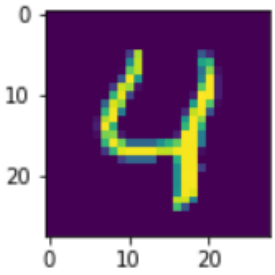


#Predict for test sample:

```
classes =['zero','one','two','three','four','five','six','seven','eight','nine']
```

```
def plot_sample(X,y,index):
    plt.figure(figsize=(15,2))
    plt.imshow(X[index])
    plt.xlabel(classes[y[index]])
```

```
plot_sample(X_test, y_test, 4)
```



```
y_classes=[np.argmax(element) for element in y_pred]
classes[y_classes[4]]
```

```
'four'
```

#Printing model summary:

```
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, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
dropout (Dropout)            (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 128)               204928
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 225,034
Trainable params: 225,034
Non-trainable params: 0
_________________________________________________________________
```

RESULT:

Thus the CNN model for classifying the hand written digits works efficiently with an accuracy of around 99%

CONCLUSION:

CNN models are best for classifying image dataset with good accuracy. Computational power for classifying image dataset is much less when compared to other learning models because it uses advanced feature extraction technique which uses filters, kernel, pooling etc to extract features from images.

REFERENCE:

- https://towardsdatascience.com/a-simple-2d-cnn-for-mnist-digit-recognition-a998dbc1e79a
- https://www.kaggle.com/raoulma/mnist-image-class-tensorflow-cnn-99-51-test-acc