Team Id	**PNT2022TMID39413**
Project Name	**A Novel Method For Handwritten DigitRecognition System.**

A Novel Method For Handwritten Digit Recognition System

Model Building:

This activity includes the following steps

- Initializing the model
- Adding CNN Layers
- Training and testing the model
- Saving the model.

Add CNN Layer

Creating the model and adding the input, hidden, and output layers to it.

Creating the Model

```python
# create model
model = Sequential()
# adding model layer
model.add(Conv2D(64, (3, 3), input_shape=(28, 28, 1), activation='relu'))
model.add(Conv2D(32, (3, 3), activation='relu'))
#model.add(Conv2D(32, (3, 3), activation='relu'))
#flatten the dimension of the image
model.add(Flatten())
#output layer with 10 neurons
model.add(Dense(number_of_classes, activation='softmax'))
```

The Sequential model is a linear stack of layers. You can create a Sequential model by passing a list of layer instances to the constructor:

URL: https://youtu.be/FmpDIaiMIeA

To know more about layers watch the below video

Compiling The Model

With both the training data defined and model defined, it's time to configure the learning process. This is accomplished with a call to the compile () method of the Sequential model class. Compilation requires 3 arguments: an optimizer, a loss function, and a list of metrics.

Compiling the model

```
# Compile model
model.compile(loss='categorical_crossentropy', optimizer="Adam", metrics=['accuracy'])
```

Note: In our project, we have 2 classes in the output, so the loss is binary_crossentropy. If you have more than two classes in output put “loss = categorical_cross entropy”.

Train The Model

Now, let us train our model with our image dataset.
Fit: functions used to train a deep learning neural network.

Fitting the model

```
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=5, batch_size=32)
```

```
Epoch 1/5
1875/1875 [==============================] - 184s 98ms/step - loss: 0.2451 - accuracy: 0.9497 - val_loss: 0.0966 - val_accuracy: 0.9
715
Epoch 2/5
1875/1875 [==============================] - 183s 98ms/step - loss: 0.0694 - accuracy: 0.9785 - val_loss: 0.0971 - val_accuracy: 0.9
714
Epoch 3/5
1875/1875 [==============================] - 183s 98ms/step - loss: 0.0487 - accuracy: 0.9850 - val_loss: 0.0829 - val_accuracy: 0.9
782
Epoch 4/5
1875/1875 [==============================] - 177s 94ms/step - loss: 0.0382 - accuracy: 0.9877 - val_loss: 0.0881 - val_accuracy: 0.9
769
```

Arguments:
steps_per_epoch : it specifies the total number of steps taken from the generator as soon as one epoch is finished and the next epoch has started. We can calculate the value of steps_per_epoch as the total number of samples in your dataset divided by the batch size.

Epochs: an integer and number of epochs we want to train our model for.

Validation_data :

- an inputs and targets list
- a generator
- inputs, targets, and sample_weights list which can be used to evaluate the loss and metrics for any model after any epoch has ended.

validation_steps:
only if the validation_data is a generator then only this argument can be used. It specifies the total number of steps taken from the generator before it is stopped at every epoch and its value is calculated as the total number of validation data points in your dataset divided by the validation batch size.

Observing The Metrics

Observing the metrics

```
# Final evaluation of the model
metrics = model.evaluate(X_test, y_test, verbose=0)
print("Metrics(Test loss & Test Accuracy): ")
print(metrics)
```

```
Metrics(Test loss & Test Accuracy):
[0.1097492054104805, 0.9753000140190125]
```

We here are printing the metrics which lists out the Test loss and Test accuracy

- Loss value implies how poorly or well a model behaves after each iteration of optimization.
- An accuracy metric is used to measure the algorithm's performance in an interpretable way.

Test The Model

Firstly we are slicing the x_test data until the first four images. In the next step we the printing the predicted output.

Predicting the output

```
prediction=model.predict(X_test[:4])
print(prediction)
```

```
[[5.50544734e-15 7.41999492e-20 5.00876077e-12 1.26642463e-09
  3.52252804e-21 1.54133163e-17 3.15550259e-21 1.00000000e+00
  1.32678888e-13 6.44072333e-14]
 [1.51885260e-08 8.02883537e-09 1.00000000e+00 6.44802788e-13
  6.37117113e-16 3.40490114e-15 2.15804121e-08 2.18907611e-19
  3.38496564e-10 2.07915498e-20]
 [3.14093924e-08 9.99941349e-01 2.01593957e-06 1.45100779e-10
  5.25237965e-06 1.59223120e-07 3.15299786e-08 1.53995302e-07
  5.09846941e-05 1.14552066e-07]
 [1.00000000e+00 1.35018288e-14 2.28308122e-10 1.79766094e-16
  1.28767550e-14 7.12401882e-12 2.92727509e-11 3.52439052e-13
  2.56207252e-12 2.32345068e-12]]
```

```
import numpy as np
print(np.argmax(prediction,axis=1)) #printing our labels from first 4 images
print(y_test[:4]) #printing the actual labels
```

```
[7 2 1 0]
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

As we already predicted the input from the x_test. According to that by using argmax function here we are printing the labels with high prediction values.

Observing The Metrics

Observing the metrics

```
# Final evaluation of the model
metrics = model.evaluate(X_test, y_test, verbose=0)
print("Metrics(Test loss & Test Accuracy): ")
print(metrics)
```

```
Metrics(Test loss & Test Accuracy):
[0.1097492054104805, 0.9753000140190125]
```

We here are printing the metrics which lists out the Test loss and Test accuracy

- Loss value implies how poorly or well a model behaves after each iteration of optimization.
- An accuracy metric is used to measure the algorithm's performance in an interpretable way.

Test The Model

Firstly we are slicing the x_test data until the first four images. In the next step we the printing the predicted output.

Predicting the output

```
prediction=model.predict(X_test[:4])
print(prediction)
```

```
[[5.50544734e-15 7.41999492e-20 5.00876077e-12 1.26642463e-09
  3.52252804e-21 1.54133163e-17 3.15550259e-21 1.00000000e+00
  1.32678888e-13 6.44072333e-14]
 [1.51885260e-08 8.02883537e-09 1.00000000e+00 6.44802788e-13
  6.37117113e-16 3.40490114e-15 2.15804121e-08 2.18907611e-19
  3.38496564e-10 2.07915498e-20]
 [3.14093924e-08 9.99941349e-01 2.01593957e-06 1.45100779e-10
  5.25237965e-06 1.59223120e-07 3.15299786e-08 1.53995302e-07
  5.09846941e-05 1.14552066e-07]
 [1.00000000e+00 1.35018288e-14 2.28308122e-10 1.79766094e-16
  1.28767550e-14 7.12401882e-12 2.92727509e-11 3.52439052e-13
  2.56207252e-12 2.32345068e-12]]
```

```
import numpy as np
print(np.argmax(prediction,axis=1)) #printing our labels from first 4 images
print(y_test[:4]) #printing the actual labels
```

```
[7 2 1 0]
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

As we already predicted the input from the x_test. According to that by using argmax function here we are printing the labels with high prediction values

Save The Model

Your model is to be saved for future purposes. This saved model can also be integrated with an android application or web application in order to predict something.

Saving the model

```
# Save the model
model.save('models/mnistCNN.h5')
```

The model is saved with .h5 extension as follows:
An H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data.

Test With Saved Model

Now open another jupyter file and write the below code

Taking images as input and checking results

```
# Importing the Keras libraries and packages
from tensorflow.keras.models import load_model
model = load_model(r'C:/Users/DELL/Hand written recognition System/models/mnistCNN.h5')
from PIL import Image#used for manipulating image uploaded by the user.
import numpy as np#used for numerical analysis
for index in range(4):
    img = Image.open('data/' + str(index) + '.png').convert("L")# convert image to monochrome
    img = img.resize((28,28))# resizing of input image
    im2arr = np.array(img) #converting to image
    im2arr = im2arr.reshape(1,28,28,1) #reshaping according to our requirement
    # Predicting the Test set results
    y_pred = model.predict(im2arr) #predicting the results
    print(y_pred)

[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
[[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
[[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]]
```

Firstly we are loading the model which was built. Then we are applying for a loop for the first four images and converting the image to the required format. Then we are resizing the input image, converting the image as per the CNN model and we are reshaping it according to the requirement. At last, we are predicting the result.

You can use predict_classes for just predicting the class of an image