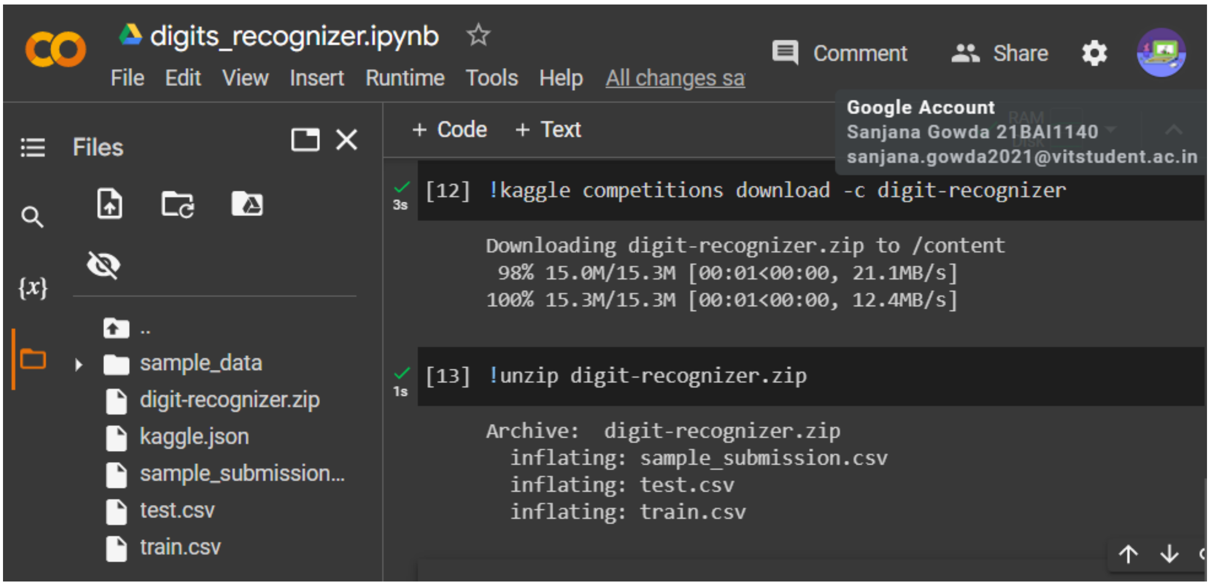
Build a learning model – DIGITS RECOGNITION: CNN

Public dataset: https://www.kaggle.com/competitions/digit-recognizer/data

Deep Learning (DL) Algorithm is adopted. – Deep Neural network the Keras way.

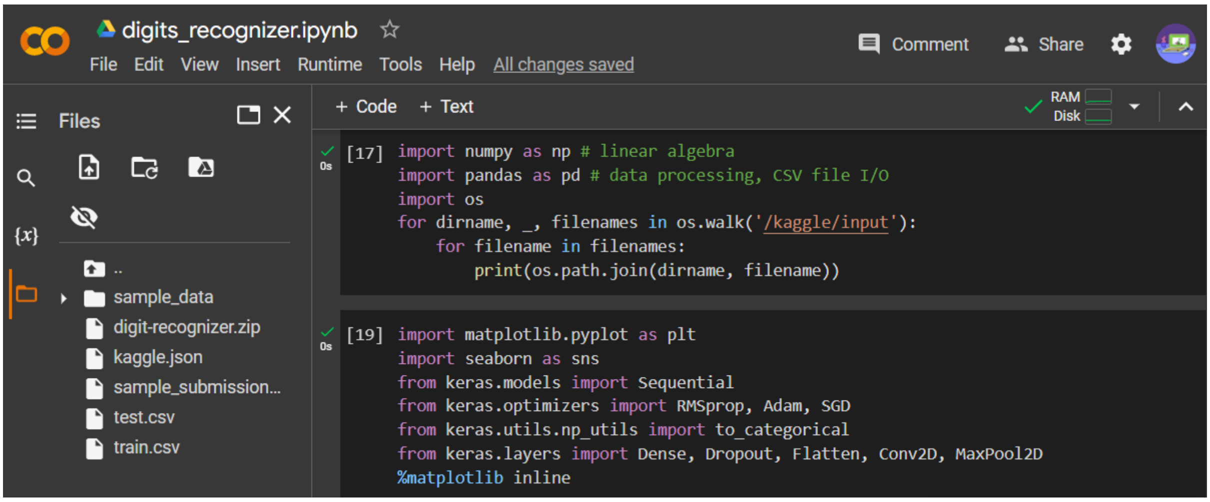
3 Different optimizers are used for tuning – Adam, RMSprop, SGD

- Downloading data from Kaggle and importing it to google colab.



Once we get our data files unzipped and ready, we are going to train our model with this train.csv file and then we will predict the values of the images present in test.csv file.
We will finally save all the output data in submission.csv file

- Importing libraries



Importing numpy and pandas for linear algebra and data processing respectively, then importing matplotlib and seaborn for data visualization. From keras models, importing the sequential model, from keras optimizers, we will work on 3 optimisers hence those are chosen. From keras.utils.np_utils we import categorial which will be useful for data pre-processing while training, and from layers, we are importing dense, dropout flatten conv2D MaxPool2D.

DATA PREPARATION

- Load the training data

digits_recognizer.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
Google Account
Sanjana Gowda 21BAI1140
sanjana.gowda2021@vitstudent.ac.in
+ Code + Text

```
[18] # Load the data
     train = pd.read_csv("/content/train.csv")
     train.head()
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 785 columns

```
[19] train.shape

(42000, 785)
```

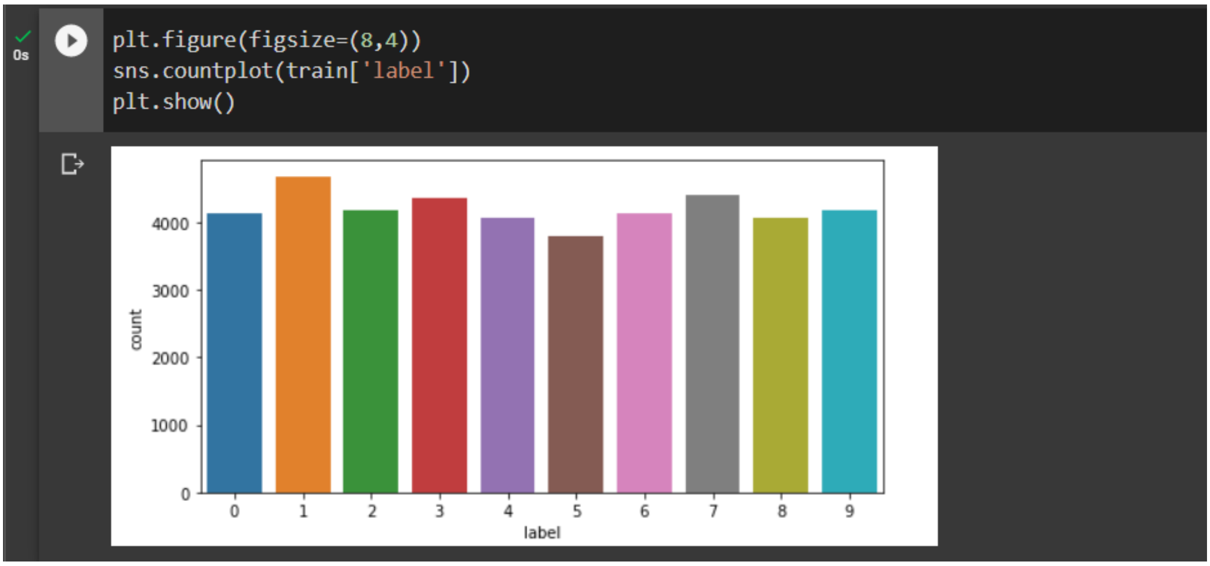
First, we are loading the data by creating training data frame using pd.read.csv and then run the training data frame head. We get 785 columns. These columns are nothing but the pixel values. When we further display the shape of the training data frame, it actually has 42000 rows and 785 columns totally.

Then, to check digits present we check the number of counts of the digits, as shown below and the digit ‘1’ appears the most, whereas the digit ‘5’ appears the least. The digits range from 0-9.

```
train['label'].value_counts()

1    4684
7    4401
3    4351
9    4188
2    4177
6    4137
0    4132
4    4072
8    4063
5    3795
Name: label, dtype: int64
```

- Plotting the label column of training set as a bar graph to visualise it better



- Creation of X train and Y train

We are dropping the label column and saving the train data frame in x_train and the label column alone is saved in the y_train.

DATA PRE-PROCESSING

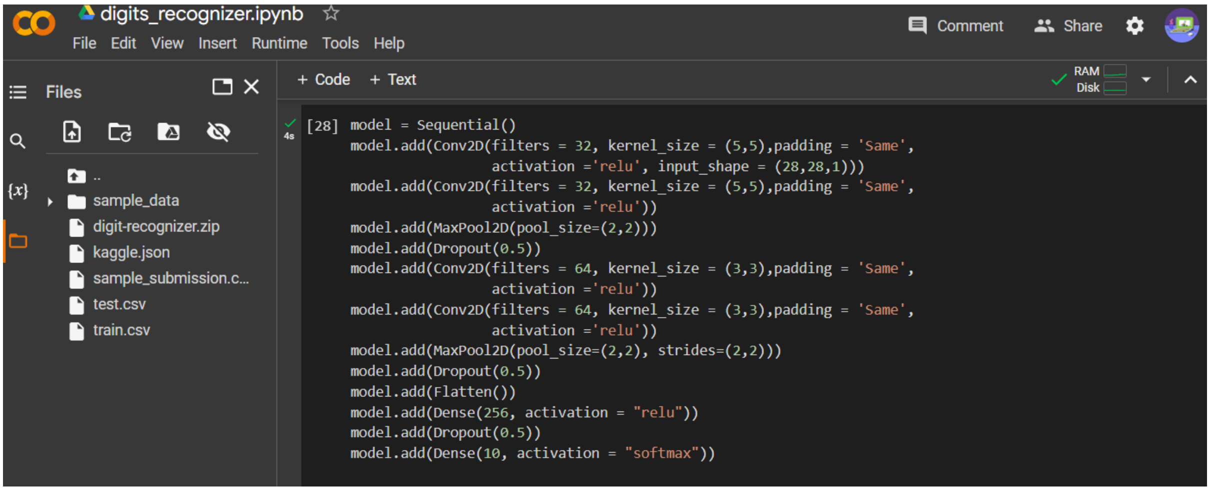
- Re-shaping the extra pixel values into 28x28 pixels as this will be good for CNN.

Converting all these pixel values to range 0 to 1, which can be done by dividing it by 255, to get a small value. Usage of to_categorical on the y_train data frame and assigning num_classes as 10.

We will just check the data at the 6th row of x_train, and call it by matplotlib to show the image present at fifth index which is a *seven*.

MODEL BUILDING

- Setting the parameters



Since we are using sequential model, we are adding convolutional 2D layer and in first layer we are adding input shape as 28x28 pixels and this remains constant, the kernel size is (5,5) and (3,3), number of filters 32 and 64, padding is going to remain same from all the sides activation function is 'relu' which is called rectified linear unit. Adding the MaxPool2D layer which with the pool size of 2x2 and using dropout layer of .5. Adding flatten which is basically to convert your image into one dimensional array and then using Dense then again ending the model with a dense layer with 10 classes because there are only digits from 0 to 9 that gives only 10 outputs possible and activation is soft max.

- Compiling the model by choosing an optimizer

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

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        832

 conv2d_1 (Conv2D)           (None, 28, 28, 32)        25632

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 dropout (Dropout)           (None, 14, 14, 32)        0

 conv2d_2 (Conv2D)           (None, 14, 14, 64)        18496

 conv2d_3 (Conv2D)           (None, 14, 14, 64)        36928

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 7, 7, 64)          0

 flatten (Flatten)           (None, 3136)              0

 dense (Dense)               (None, 256)               803072

 dropout_2 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 887,530
Trainable params: 887,530
Non-trainable params: 0
```

Now we compile the model with the first optimizer as *Adam,* loss as categorical loss and entropy metrics as accuracy as we are looking for accuracy at every epoch. Summary of the model is also shown above, this is all the information about my layers that are added output shape parameters and these contain the layers, convolutional, MaxPool2D, dense & dropout.

- Check model fit for every epoch

```
epochs=20
batch_size=64
model.fit(x_train,y_train,epochs=epochs,batch_size=batch_size)
```

```
Epoch 1/20
657/657 [==============================] - 16s 7ms/step - loss: 0.2947 - accuracy: 0.9046
Epoch 2/20
657/657 [==============================] - 5s 7ms/step - loss: 0.1027 - accuracy: 0.9696
Epoch 3/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0763 - accuracy: 0.9765
Epoch 4/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0631 - accuracy: 0.9807
Epoch 5/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0560 - accuracy: 0.9827
Epoch 6/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0513 - accuracy: 0.9842
Epoch 7/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0474 - accuracy: 0.9855
Epoch 8/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0431 - accuracy: 0.9871
Epoch 9/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0407 - accuracy: 0.9871
Epoch 10/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0402 - accuracy: 0.9875
Epoch 11/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0382 - accuracy: 0.9874
Epoch 12/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0340 - accuracy: 0.9901
Epoch 13/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0337 - accuracy: 0.9900
Epoch 14/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0352 - accuracy: 0.9891
Epoch 15/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0313 - accuracy: 0.9901
Epoch 16/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0294 - accuracy: 0.9907
Epoch 17/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0309 - accuracy: 0.9901
Epoch 18/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0287 - accuracy: 0.9915
Epoch 19/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0272 - accuracy: 0.9914
Epoch 20/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0290 - accuracy: 0.9907
<keras.callbacks.History at 0x7f06dfc448e0>
```

We can modify this data as we wish to obtain the best accuracy, hence first we take epochs as 20 and batch size as 64. We now let it run 20 times through the data and we obtain the best accuracy as 99.15%

To obtain a better accuracy, we now take epochs as 35 and batch size as 50. We now let it run 35 times through the data and we obtain the best accuracy as 99.29%, though this takes a higher run time than previous values.

```
epochs=35
batch_size=50
model.fit(x_train,y_train,epochs=epochs,batch_size=batch_size)
```

```
Epoch 1/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0291 - accuracy: 0.9908
Epoch 2/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0308 - accuracy: 0.9901
Epoch 3/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0313 - accuracy: 0.9902
Epoch 4/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0277 - accuracy: 0.9915
Epoch 5/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0286 - accuracy: 0.9915
Epoch 6/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0301 - accuracy: 0.9903
Epoch 7/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0249 - accuracy: 0.9919
Epoch 8/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0247 - accuracy: 0.9927
Epoch 9/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0273 - accuracy: 0.9921
Epoch 10/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0240 - accuracy: 0.9924
Epoch 11/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0270 - accuracy: 0.9917
Epoch 12/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0270 - accuracy: 0.9918
Epoch 13/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0264 - accuracy: 0.9918
Epoch 14/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0233 - accuracy: 0.9928
Epoch 15/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0242 - accuracy: 0.9926
Epoch 16/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0274 - accuracy: 0.9918
Epoch 17/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0262 - accuracy: 0.9917
Epoch 18/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0258 - ac
Epoch 19/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0258 - accuracy: 0.9929
Epoch 20/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0240 - accuracy: 0.9929
Epoch 21/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0279 - accuracy: 0.9921
Epoch 22/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0237 - accuracy: 0.9929
Epoch 23/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0219 - accuracy: 0.9932
Epoch 24/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0268 - accuracy: 0.9921
Epoch 25/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0249 - accuracy: 0.9926
Epoch 26/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0243 - accuracy: 0.9927
Epoch 27/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0260 - accuracy: 0.9925
Epoch 28/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0255 - accuracy: 0.9926
Epoch 29/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0265 - accuracy: 0.9926
Epoch 30/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0243 - accuracy: 0.9929
Epoch 31/35
840/840 [==============================] - 6s 7ms/step - loss: 0.0240 - accuracy: 0.9928
Epoch 32/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0252 - accuracy: 0.9926
Epoch 33/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0244 - accuracy: 0.9926
Epoch 34/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0260 - accuracy: 0.9925
Epoch 35/35
840/840 [==============================] - 5s 6ms/step - loss: 0.0250 - accuracy: 0.9924
<keras.callbacks.History at 0x7f06dd5a9f40>
```

- Compiling the model and choosing the SGD optimizer we get the following results.

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

```
epochs=20
batch_size=64
model.fit(x_train,y_train,epochs=epochs,batch_size=batch_size)
```

```
Epoch 1/20
657/657 [==============================] - 6s 7ms/step - loss: 0.0217 - accuracy: 0.9931
Epoch 2/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0184 - accuracy: 0.9943
Epoch 3/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0170 - accuracy: 0.9950
Epoch 4/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0176 - accuracy: 0.9944
Epoch 5/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0155 - accuracy: 0.9952
Epoch 6/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0135 - accuracy: 0.9955
Epoch 7/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0137 - accuracy: 0.9957
Epoch 8/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0142 - accuracy: 0.9954
Epoch 9/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0144 - accuracy: 0.9957
Epoch 10/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0140 - accuracy: 0.9955
Epoch 11/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0116 - accuracy: 0.9961
Epoch 12/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0143 - accuracy: 0.9959
Epoch 13/20
657/657 [==============================] - 6s 9ms/step - loss: 0.0134 - accuracy: 0.9955
Epoch 14/20
657/657 [==============================] - 6s 8ms/step - loss: 0.0149 - accuracy: 0.9954
Epoch 15/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0153 - accuracy: 0.9950
Epoch 16/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0121 - accuracy: 0.9962
Epoch 17/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0118 - accuracy: 0.9962
Epoch 18/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0135 - accuracy: 0.9958
Epoch 19/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0121 - accuracy: 0.9961
Epoch 20/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0122 - accuracy: 0.9965
<keras.callbacks.History at 0x7f06df40afa0>
```

By choosing the SGD optimizer, this time we see a better accuracy in the results for epochs taken as 20 and batch size taken as 64, the best accuracy being 99.65%

- Compiling the model and choosing the RMSprop optimizer we get the following results.

By choosing the RMSprop optimizer, this time we see similar accuracy in the results for epochs taken as 20 and batch size taken as 64, the best accuracy being 99.65% same as what we got with SGD optimizer. Hence, we can conclude any of SGD or RMSprop can be chosen over Adam for this model. This concludes the training part of the model.

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

```
epochs=20
batch_size=64
model.fit(x_train,y_train,epochs=epochs,batch_size=batch_size)
```

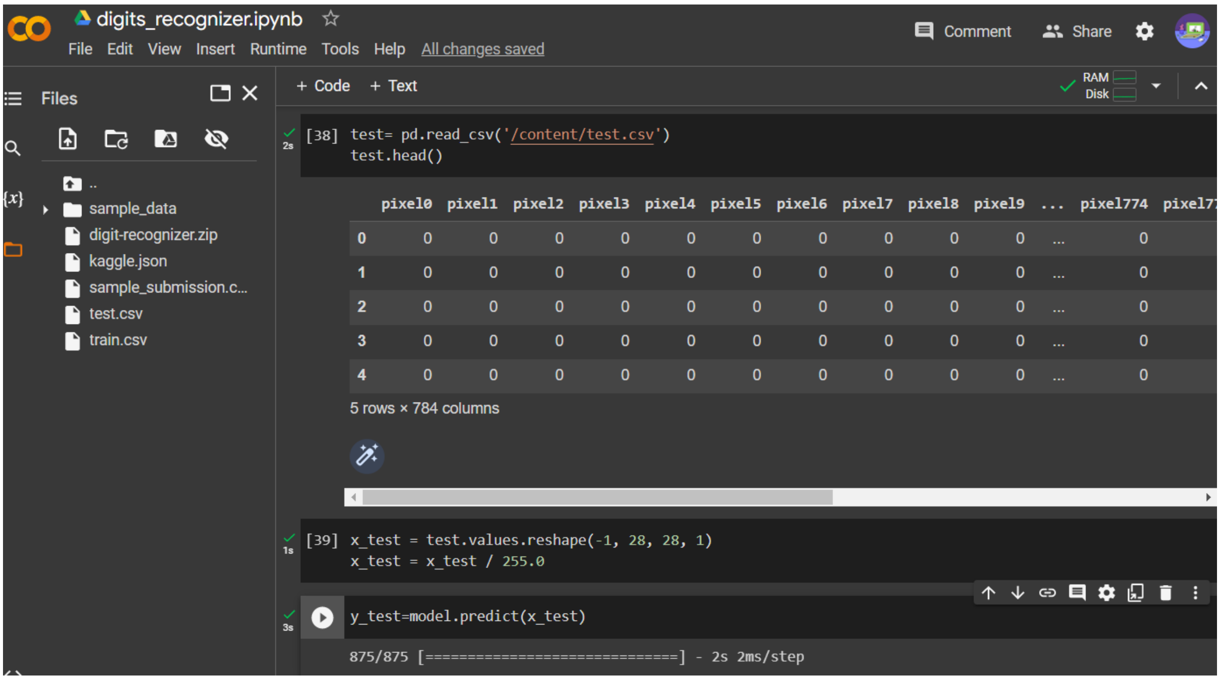
```
Epoch 1/20
657/657 [==============================] - 7s 7ms/step - loss: 0.0138 - accuracy: 0.9960
Epoch 2/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0167 - accuracy: 0.9955
Epoch 3/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0186 - accuracy: 0.9947
Epoch 4/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0201 - accuracy: 0.9949
Epoch 5/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0183 - accuracy: 0.9956
Epoch 6/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0178 - accuracy: 0.9957
Epoch 7/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0189 - accuracy: 0.9953
Epoch 8/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0177 - accuracy: 0.9955
Epoch 9/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0188 - accuracy: 0.9951
Epoch 10/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0196 - accuracy: 0.9951
Epoch 11/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0159 - accuracy: 0.9963
Epoch 12/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0183 - accuracy: 0.9950
Epoch 13/20
657/657 [==============================] - 5s 8ms/step - loss: 0.0178 - accuracy: 0.9952
Epoch 14/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0198 - accuracy: 0.9954
Epoch 15/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0157 - accuracy: 0.9959
Epoch 16/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0168 - accuracy: 0.9958
Epoch 17/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0147 - accuracy: 0.9960
Epoch 18/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0157 - accuracy: 0.9961
Epoch 19/20
657/657 [==============================] - 5s 7ms/step - loss: 0.0134 - accuracy: 0.9965
Epoch 20/20
657/657 [==============================] - 4s 7ms/step - loss: 0.0177 - accuracy: 0.9959
<keras.callbacks.History at 0x7f06a06bc430>
```

- Testing the data frame

Now we need to test the data, so we create a test data frame and check the head of this test data frame. We notice there is no label column since we need to predict the values of these pixels.

We will then do the same as previous train data frame, we will reshape my pixel values of test data frame into 28x28 and dividing it by 255 so that it gets converted to the range 0 to 1.

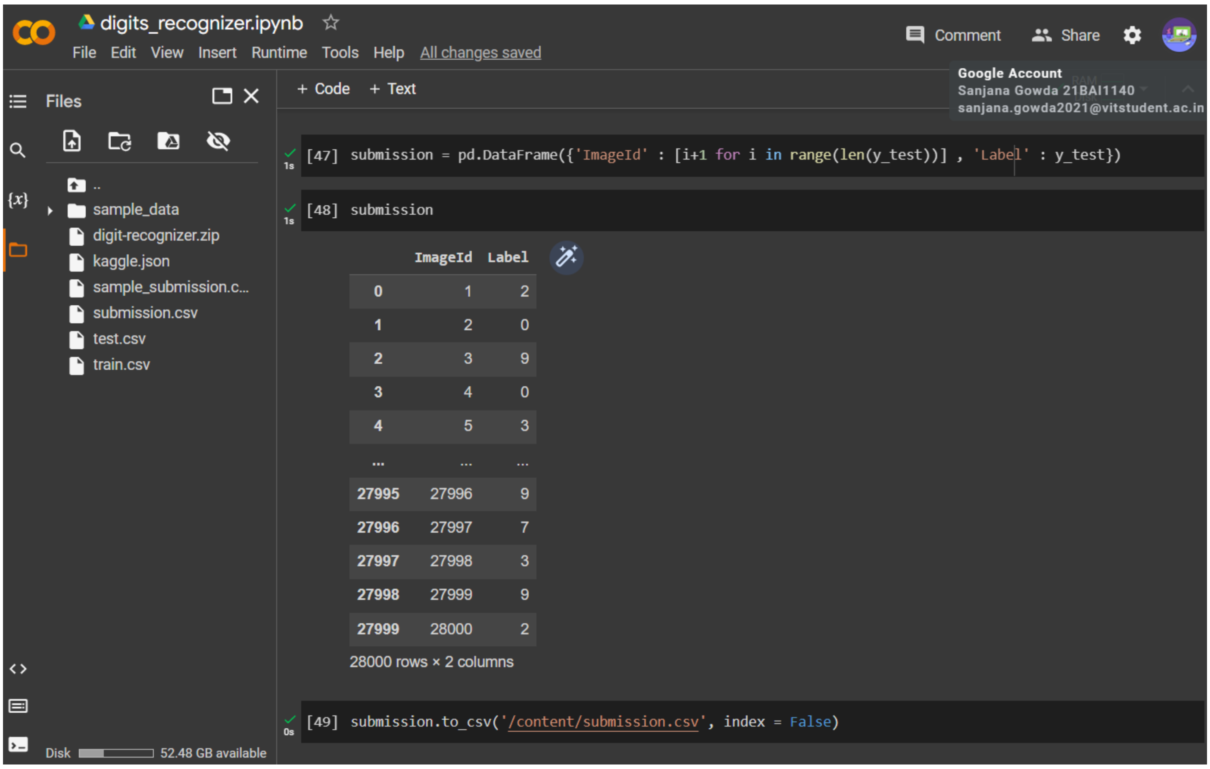
Then we will predict the x test and those values will be contained by y test.



- Checking the digit present in the 4th index and cross verifying it.

We first check the digit present in the 4th index and we can see its an image of '3'. To verify this we check y_test[4] and get values which is incomprehensible hence we have to see which is the maximum argument present in the array so that value will be our digit only so if we use this function that is *np.argmax(y_test1) for y_test1 in y_test* which gives us the answer only after running y_test[4] again. We get the value 3.

- Creating the submissions and saving it to a submission.csv file



As we got desired outputs successfully, we can now proceed to submit these predictions in a new submissions file.

Inference:

We learnt how to clean and visualise any given dataset using libraries.
We learnt how to pre process the data and tune different parameter during model building to see the various type of results that we might get.
We learnt the importance of an optimiser and how we can check accuracy with multiple optimisers and compare for the best results.
After training the model we also learnt to test it, by using Deep learning algorithm.