CS 512 Assignment 4: Report

Karan Bhatiya

A20424290

Deliverable 1: Custom CNN

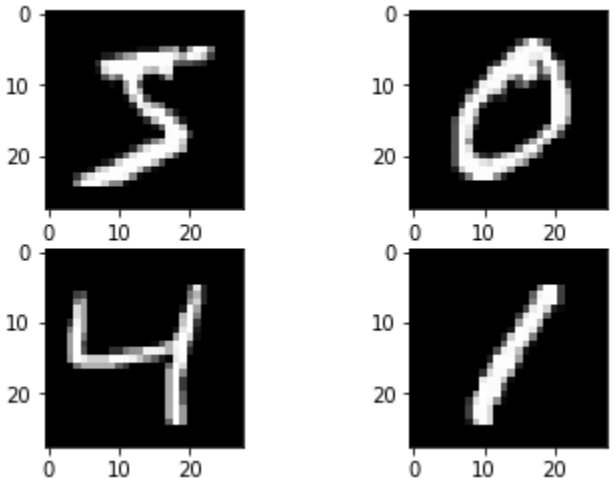
Model Details:

Input Parameters:

```
img_rows = 28  #dimensions of  image
img_cols = 28  #dimensions  of image
epochs = 5  #It shows number of times training dataset trained in the model
batch_size = 100  #It shows that, 100 images taken one at a time at training
drop_rate = 0.4  #drop rate shows how much amount of pixel values dropped
learn_rate = 0.001  #learning rate of optimizer

#Classification of images into two labels even(0) and odd(1)
y_train = np_utils.to_categorical(y_train%2!=0).astype(int)
y_test = np_utils.to_categorical(y_test%2!=0).astype(int)

#Layers applied on CNN model:
model.add(Conv2D(32,kernel_size=(5,5),input_shape=(1, img_rows,img_cols),activation = "relu")) #2D Convolution with 32 filters  and kernel size of 5
 model.add(MaxPooling2D(pool_size=(2,2)))  #Downsampling  by factor of 2
 model.add(Conv2D(64,kernel_size=(5,5),activation="relu"))  #2D Convolution with 64 filters and kernel size of 5 and applied 'relu' activation function which converts negative pixel values to 0
 model.add(MaxPooling2D(pool_size=(2,2)))  #Downsampling  by factor of 2
 model.add(Dropout(drop_rate))  #applied drop rate function on layer
 model.add(Flatten())  #convert the 2d matrix into 1d vector
 model.add(Dense(64,activation="relu"))
 model.add(Dense(20,activation="relu"))
model.add(Dense(number_of_classes,activation="softmax"))
```

Results:

First 4 images of training dataset:



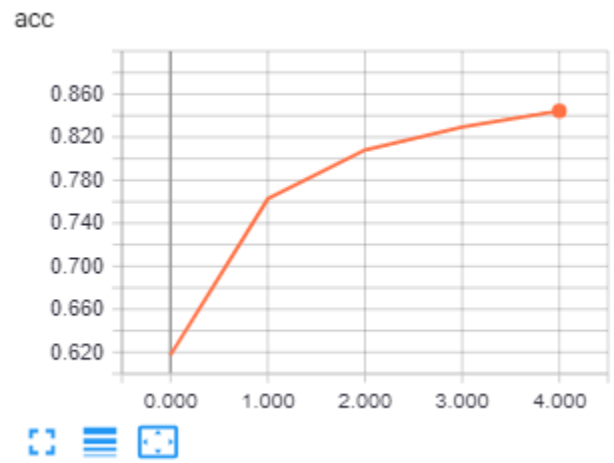
Accuracy and Loss on each iteration of training:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 156s 3ms/step
- loss: 0.6677 - acc: 0.6174 - precision: 0.4992 - recall:
0.9983 - val_loss: 0.5946 - val_acc: 0.7746 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 2/5
60000/60000 [==============================] - 163s 3ms/step
- loss: 0.5519 - acc: 0.7625 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.4326 - val_acc: 0.8278 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 3/5
60000/60000 [==============================] - 178s 3ms/step
- loss: 0.4406 - acc: 0.8082 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3565 - val_acc: 0.8469 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 4/5
60000/60000 [==============================] - 166s 3ms/step
- loss: 0.3913 - acc: 0.8295 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3238 - val_acc: 0.8613 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 5/5
60000/60000 [==============================] - 161s 3ms/step
- loss: 0.3595 - acc: 0.8444 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.2995 - val_acc: 0.8754 - val_precision:
0.5000 - val_recall: 1.0000
```

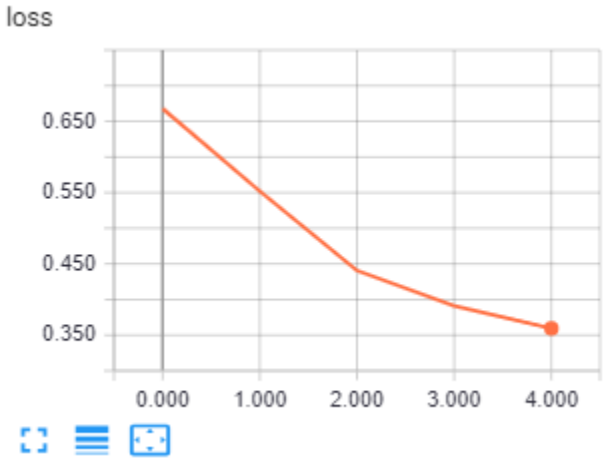
Total Accuracy, Loss, Precision and Recall at the end of the last epoch cycle:

```
Total Loss of the model: 0.30617202181021375
Total Accuracy of the model: 0.8729333333333333
Total Precision of the model: 0.5
Total Recall of the model: 1.0
```

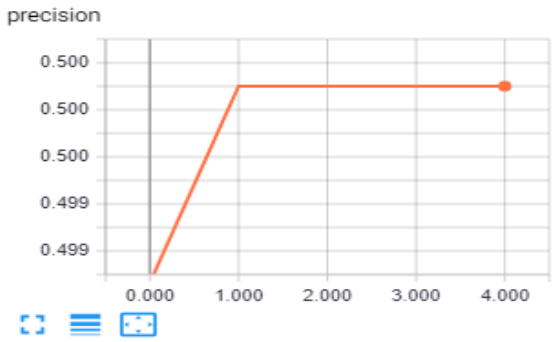
Accuracy of Training dataset:



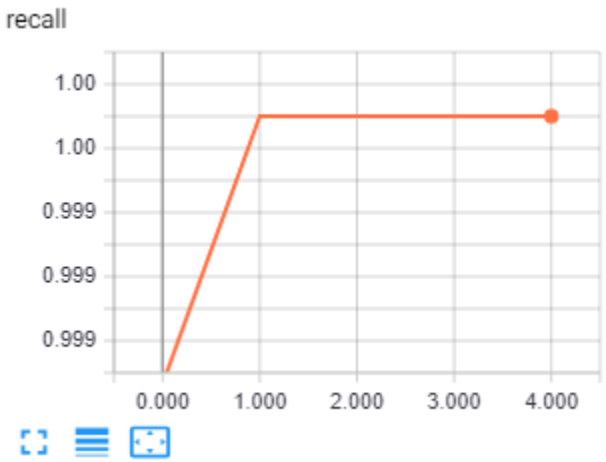
Loss of Training Dataset:



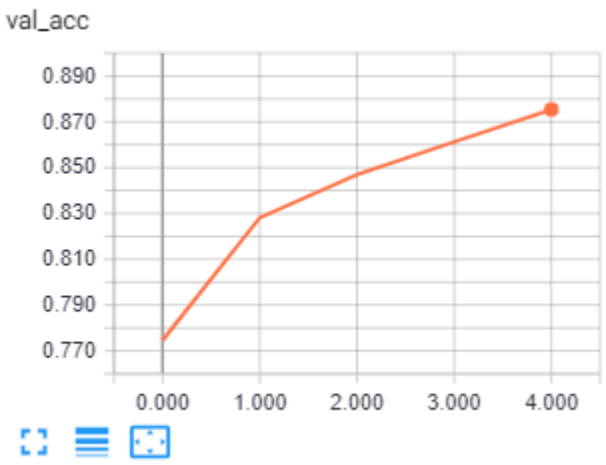
Precision of Training Dataset:



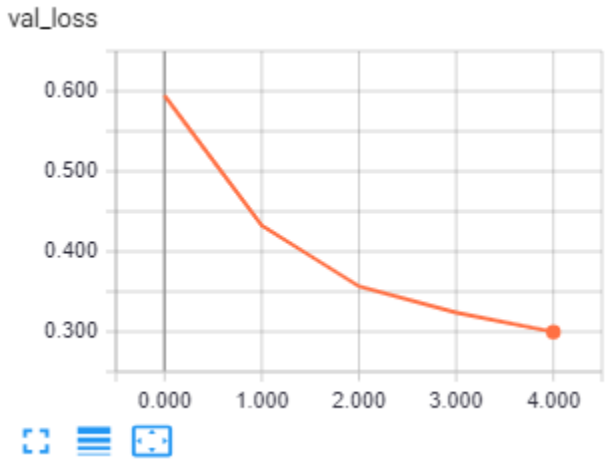
Recall of Training Dataset:



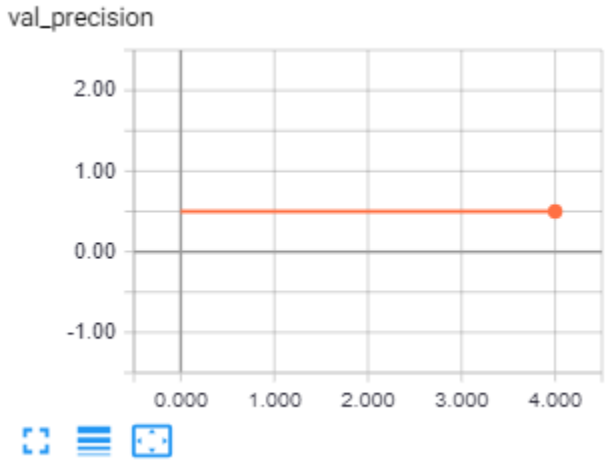
Testing Dataset Accuracy:



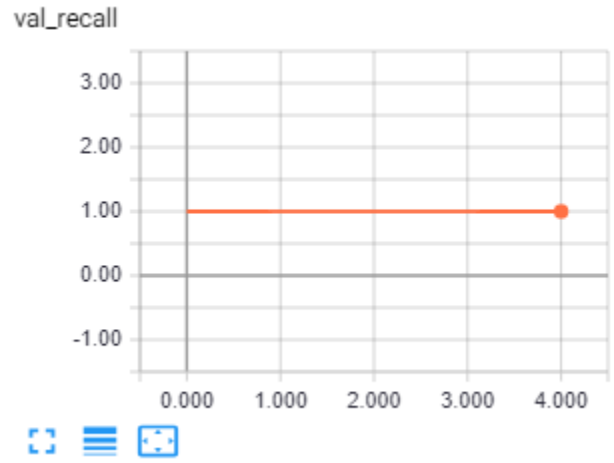
Testing Dataset Loss:



Testing Dataset Precision:



Testing Dataset Recall:



Discussion:

- As we trained the model, the **accuracy of the model goes on increasing** while the **loss of the model goes on decreasing.**
- You will get to know that from above structure of the model the **accuracy we get 87.29%** and the **loss is 30.61%.** You will get more accuracy if you increase the number of epoch cycles.
- Above model Contains, 2 2D convolution layer, 2 downsampling layer, then we apply flatten layer to represent the 2d matrix into 1d vector, as well as we have layer of dense and drop rate. We used categorical radiant optimizer function and 'relu' activation function.

Deliverable 2: Parameter Tuning

1. Changing the network architecture:

```
img_rows = 28
img_cols = 28
epochs = 5
batch_size = 100
drop_rate = 0.4
learn_rate = 0.001
model = Sequential()
   model.add(Conv2D(32,kernel_size=(5,5),input_shape=(1, img_rows,img_cols),activation = "relu"))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Conv2D(64,kernel_size=(5,5),activation="relu"))
   model.add(Dropout(drop_rate))
   model.add(Dropout(drop_rate))
   model.add(Flatten())#convert the 2d matrix into 1d vector
   model.add(Dense(64,activation="relu"))
   model.add(Dense(number_of_classes,activation="softmax"))
```

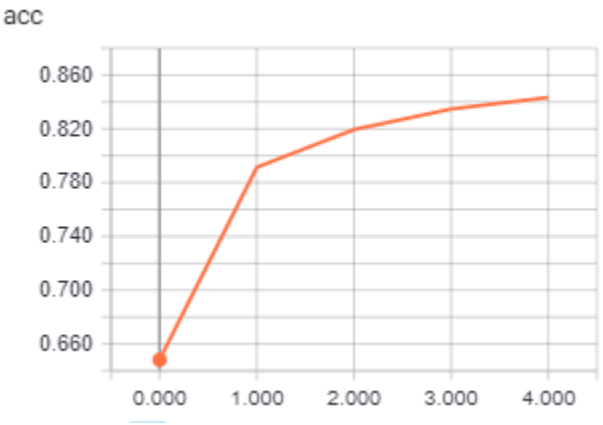
Result:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 159s 3ms/step
- loss: 0.6408 - acc: 0.6482 - precision: 0.4992 - recall:
0.9983 - val_loss: 0.5280 - val_acc: 0.8166 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 2/5
60000/60000 [==============================] - 155s 3ms/step
- loss: 0.4871 - acc: 0.7913 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3854 - val_acc: 0.8361 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 3/5
60000/60000 [==============================] - 153s 3ms/step
- loss: 0.4073 - acc: 0.8195 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3481 - val_acc: 0.8485 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 4/5
60000/60000 [==============================] - 154s 3ms/step
- loss: 0.3749 - acc: 0.8346 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3281 - val_acc: 0.8578 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 5/5
60000/60000 [==============================] - 152s 3ms/step
- loss: 0.3577 - acc: 0.8432 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3125 - val_acc: 0.8658 - val_precision:
0.5000 - val_recall: 1.0000
```

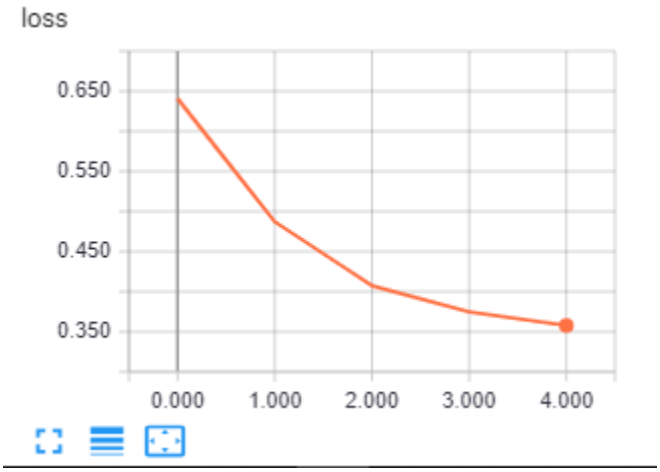
Accuracy, Loss, Precision and Recall:

```
Total Loss of the model: 0.3187321617841721
Total Accuracy of the model: 0.8641
Total Precision of the model: 0.5
Total Recall of the model: 1.0
```

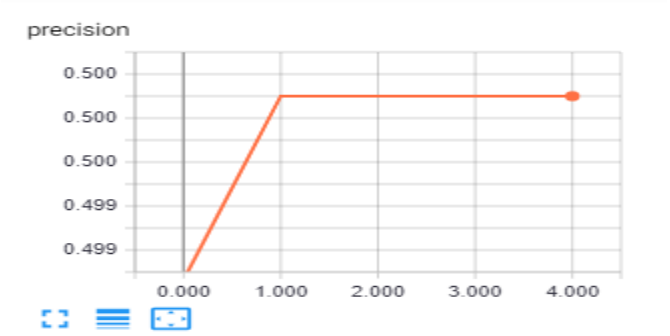
Accuracy of the training dataset;



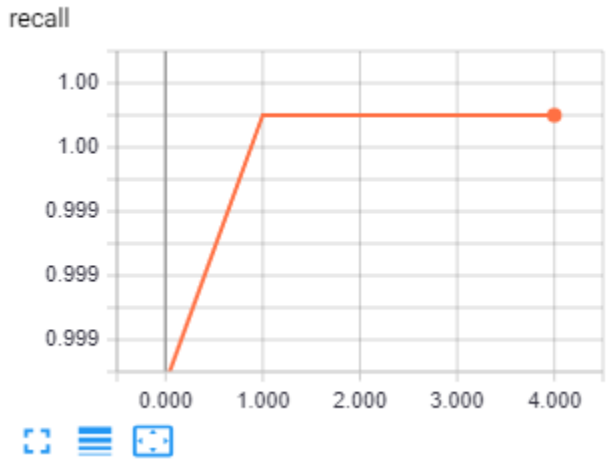
Loss of the training dataset:



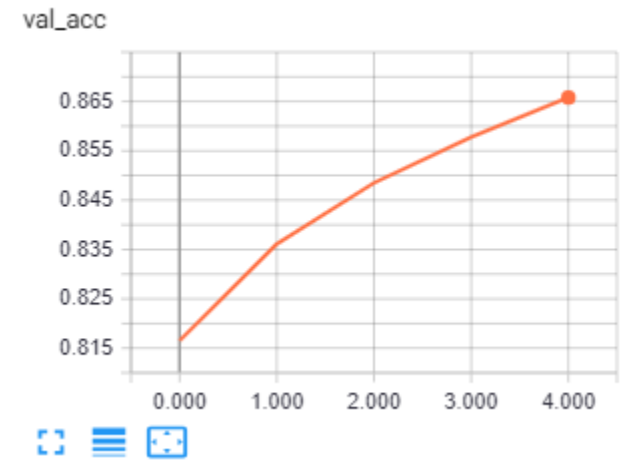
Precision of training dataset:



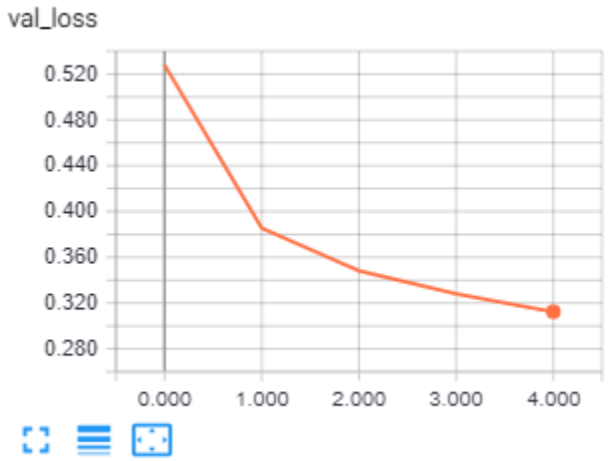
Recall of training dataset:



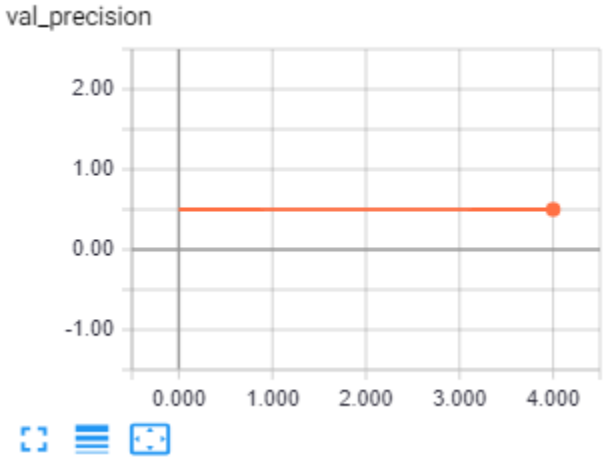
Accuracy of the Validation Test data:



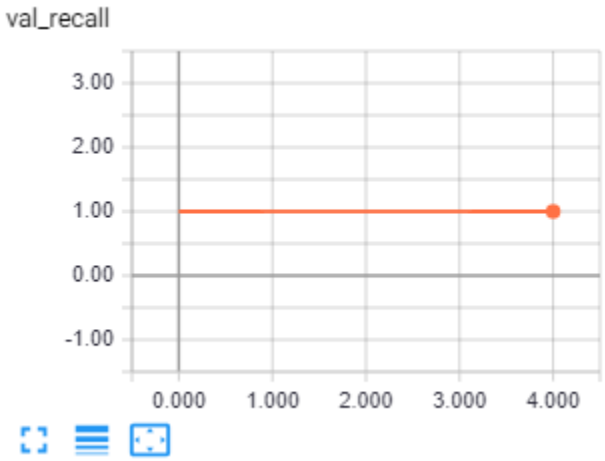
Loss of the Validation Test data:



Precision of the Validation Test data:



Recall of the Validation Test data:



Performance:

- **Accurcay is 86.41%**
- **Loss is 31.87%**

Comparison of Results:

- **Drop Layer has been added and one Dense Layer has been removed**, due to which **Accuracy** comes out to be **86%** which is lesser than the original CNN model.

Conclusion:

- Hence, we can conclude that by **increasing or decreasing layers Accuracy got affected**.

2. Stride parameter

```
img_rows = 28
img_cols = 28
epochs = 5
batch_size = 100
drop_rate = 0.4
learn_rate = 0.001

model = Sequential()
   model.add(Conv2D(32,kernel_size=(5,5),input_shape=(1,img_rows,img_cols),activation=
"relu",strides=(1,1)))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Conv2D(64,kernel_size=(5,5),activation="relu",strides=(1,1)))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Dropout(drop_rate))
   model.add(Flatten())#convert the 2d matrix into 1d vector
   model.add(Dense(64,activation="relu"))
   model.add(Dense(20,activation ="relu"))
   model.add(Dense(number_of_classes,activation="softmax"))
```

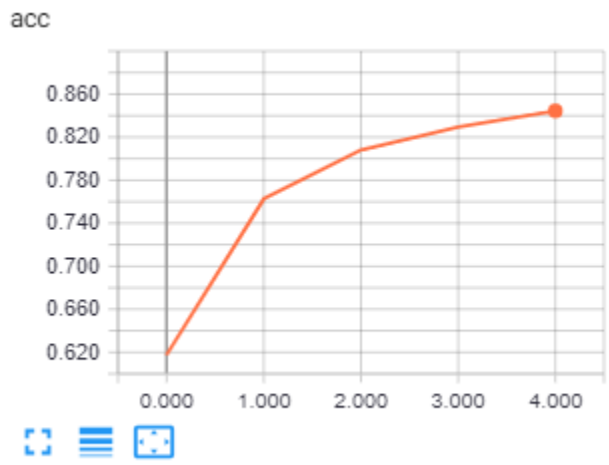
Result:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 158s 3ms/step
- loss: 0.6677 - acc: 0.6175 - precision: 0.4992 - recall:
0.9983 - val_loss: 0.5946 - val_acc: 0.7747 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 2/5
60000/60000 [==============================] - 155s 3ms/step
- loss: 0.5519 - acc: 0.7625 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.4326 - val_acc: 0.8280 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 3/5
60000/60000 [==============================] - 155s 3ms/step
- loss: 0.4406 - acc: 0.8082 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3565 - val_acc: 0.8469 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 4/5
60000/60000 [==============================] - 156s 3ms/step
- loss: 0.3913 - acc: 0.8295 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.3238 - val_acc: 0.8611 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 5/5
60000/60000 [==============================] - 149s 2ms/step
- loss: 0.3595 - acc: 0.8444 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.2995 - val_acc: 0.8754 - val_precision:
0.5000 - val_recall: 1.0000
```

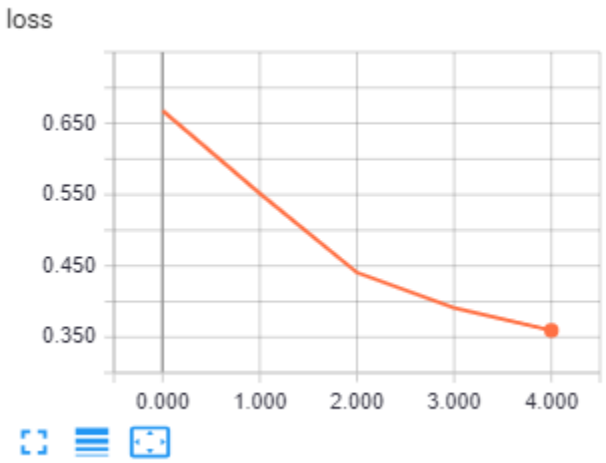
Accuracy, Loss, Precision and Recall of the model:

```
Total Loss of the model: 0.3061698676347733
Total Accuracy of the model: 0.8729833333333333
Total Precision of the model: 0.5
Total Recall of the model: 1.0
```

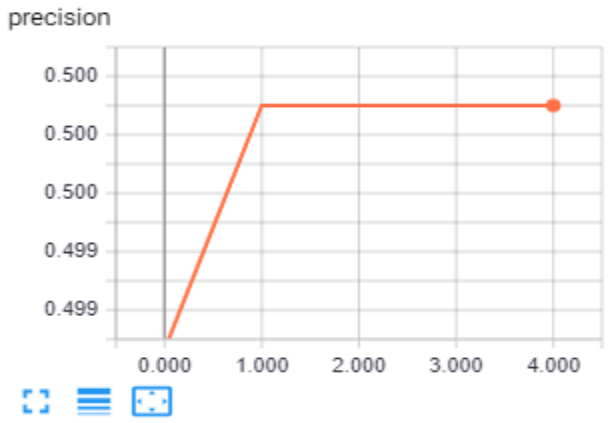
Accuracy of the Training Dataset:



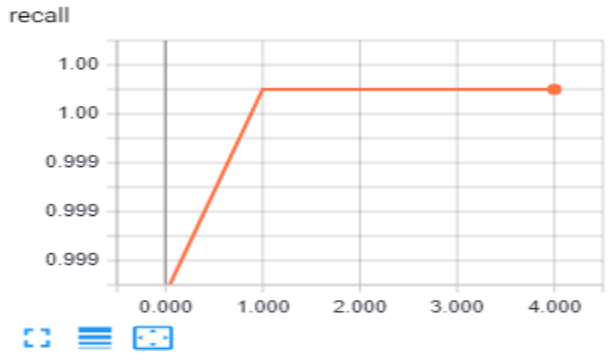
Loss of the Training Dataset:



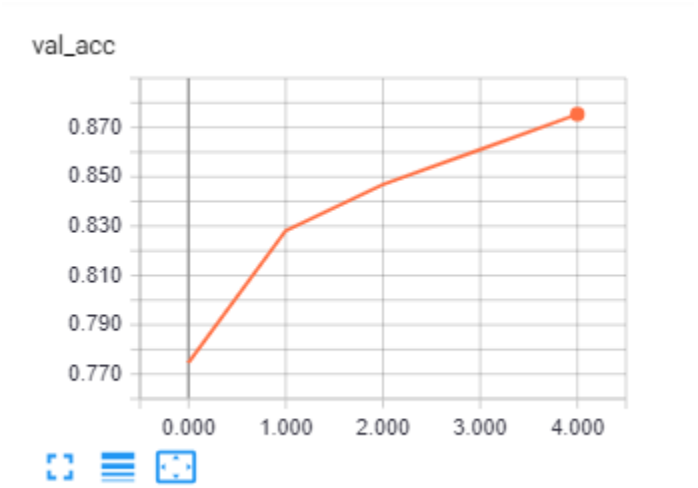
Precision of the Training Dataset:



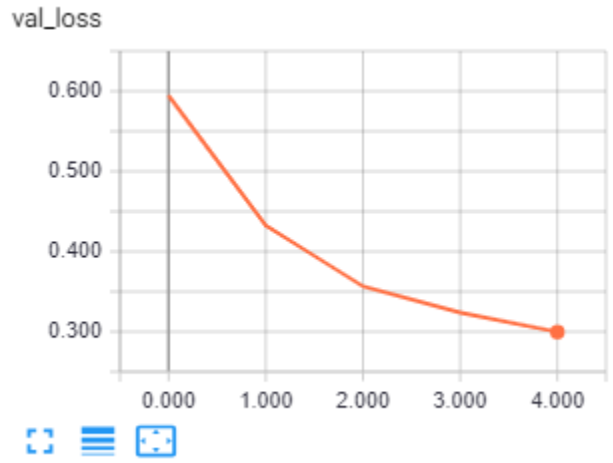
Recall of the Training Dataset:



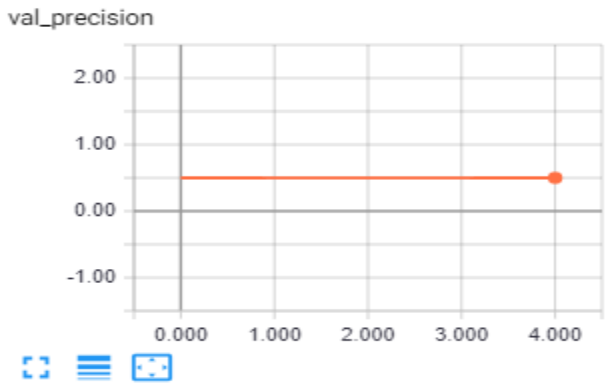
Accuracy of the Validation Test data:



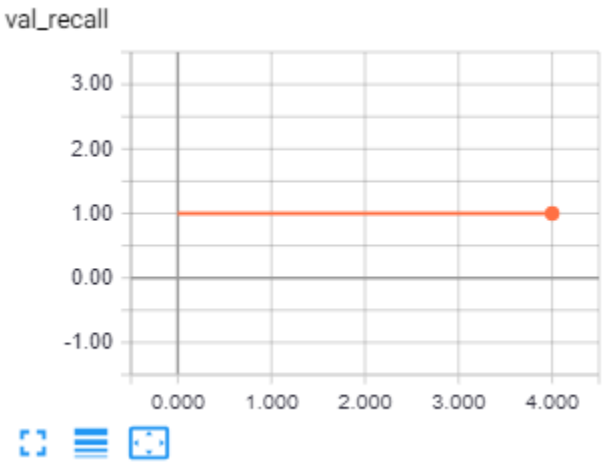
Loss of the validation test data:



Precision of the validation test data:



Recall of the validation test data:



Performance:

- **Accuracy is 87.29%**
- **Loss is 30.61%**

Comparison of Results:

- **Stride has been included**, due to which **Accuracy** got weaken as compare to original CNN model.

Conclusion:

- Hence, we can conclude that by adding Stride parameter **Accuracy got affected**.

3. Changing various parameters like epochs, drop rate, learning rate

img_rows = 28
img_cols = 28
epochs = 10
batch_size = 500
drop_rate = 0.2
learn_rate = 0.00001

```
model = Sequential()
   model.add(Conv2D(16,kernel_size=(5,5),input_shape=(1, img_rows,img_cols),activation = "relu"))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Conv2D(32,kernel_size=(5,5),activation="relu"))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Dropout(drop_rate))
   model.add(Flatten())#convert the 2d matrix into 1d vector
   model.add(Dense(64,activation="relu"))
   model.add(Dense(20,activation ="relu"))
   model.add(Dense(number_of_classes,activation="softmax"))
```

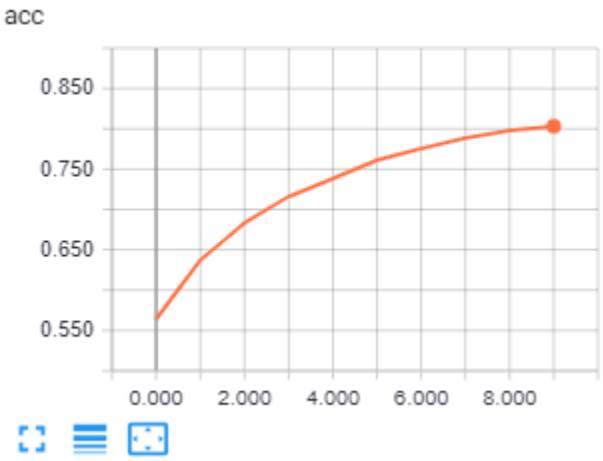
Results:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 77s 1ms/step - loss: 0.6865 - acc: 0.5638 - precision: 0.4958 - recall: 0.9917 - val_loss: 0.6735 - val_acc: 0.6703 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 2/10
60000/60000 [==============================] - 77s 1ms/step - loss: 0.6708 - acc: 0.6372 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.6573 - val_acc: 0.7239 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 3/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.6552 - acc: 0.6834 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.6384 - val_acc: 0.7541 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 4/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.6376 - acc: 0.7160 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.6159 - val_acc: 0.7780 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 5/10
60000/60000 [==============================] - 77s 1ms/step - loss: 0.6174 - acc: 0.7383 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.5900 - val_acc: 0.7937 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 6/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.5932 - acc: 0.7610 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.5611 - val_acc: 0.8095 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 7/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.5683 - acc: 0.7755 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.5304 - val_acc: 0.8191 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 8/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.5395 - acc: 0.7884 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.4992 - val_acc: 0.8252 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 9/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.5124 - acc: 0.7980 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.4692 - val_acc: 0.8328 -
val_precision: 0.5000 - val_recall: 1.0000
Epoch 10/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.4881 - acc: 0.8032 - precision: 0.5000 - recall: 1.0000 - val_loss: 0.4417 - val_acc: 0.8375 -
val_precision: 0.5000 - val_recall: 1.0000
```

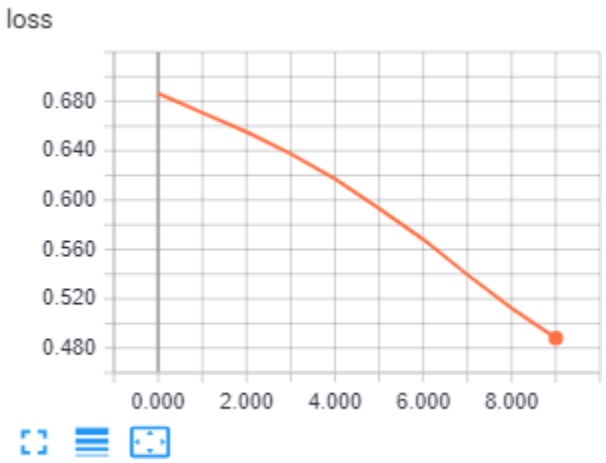
Accuracy, Loss, Precision and Loss :

```
Total Loss of the model: 0.44988732193311054
Total Accuracy of the model: 0.83405
Total Precision of the model: 0.5
Total Recall of the model: 1.0
```

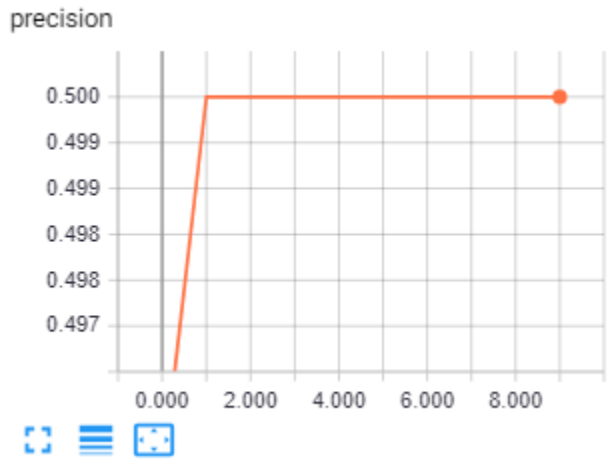
Accuracy of Training Dataset:



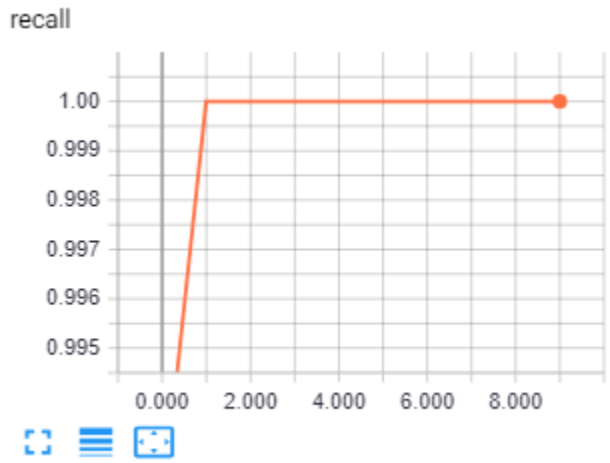
Loss of the Training Dataset:



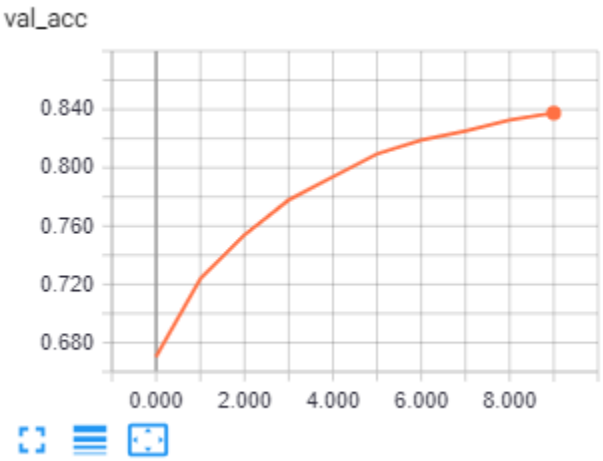
Precision of the Training Dataset:



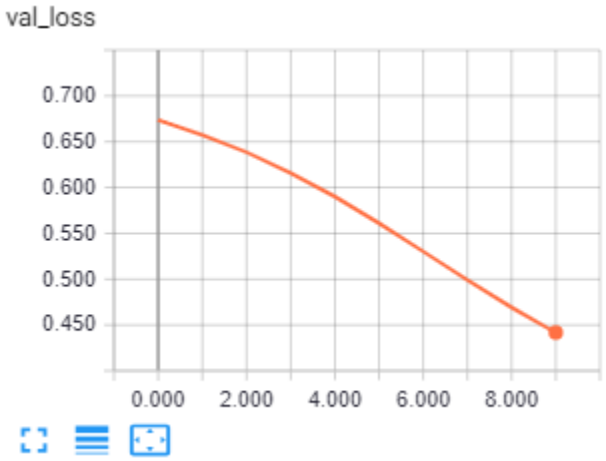
Recall of the Training Dataset:



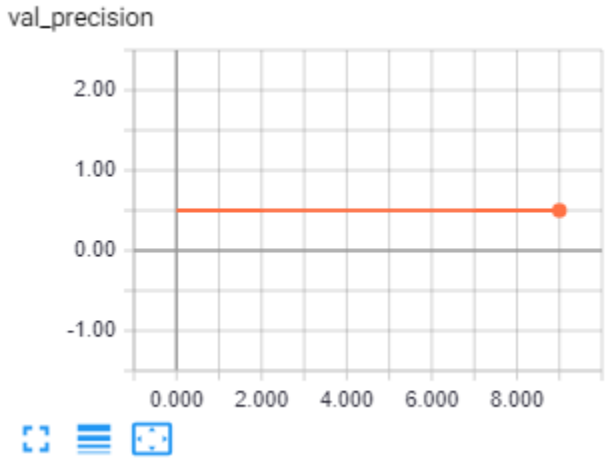
Accuracy of the validation Test data:



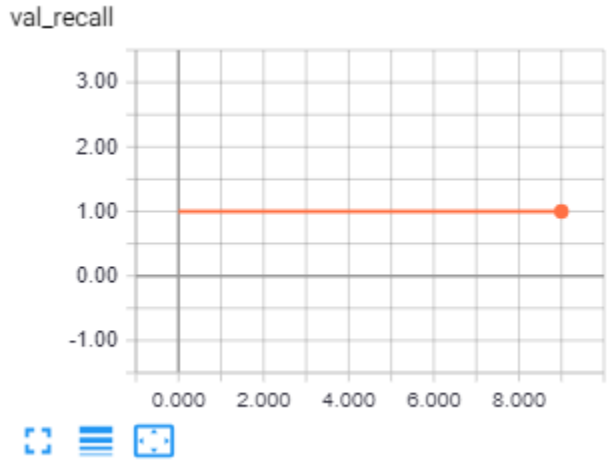
Loss of the validation Test data:



Precision of the validation test data:



Recall of the validation test data:



Performance:

- **Accuracy is 83.405%**
- **Loss is 44.98%**

Comparison of Results:

- **By increasing Batch Size from 100 to 500, increasing epochs from 5 to 10, decreasing drop rate from 0.4 to 0.2, decreasing learn rate from 0.001 to 0.00001**, **Accuracy** comes out to be **83.405%** which is quite lesser than the original CNN model.

Conclusion:

- Hence, we can conclude that by **changing various parameters layers Accuracy got affected**.

4. Changing Optimizer

```
#Input parameter
img_rows = 28
img_cols = 28
epochs = 5
batch_size = 100
drop_rate = 0.4
learn_rate = 0.001

#cnn model
def model_cnn():
   model = Sequential()
   model.add(Conv2D(32,kernel_size=(5,5),input_shape=(1, img_rows,img_cols),activation = "relu"))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Conv2D(64,kernel_size=(5,5),activation="relu"))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Dropout(drop_rate))
   model.add(Flatten())#convert the 2d matrix into 1d vector
   model.add(Dense(64,activation="relu"))
   model.add(Dense(20,activation ="relu"))
   model.add(Dense(number_of_classes,activation="softmax"))
   return(model)
#compiler function where I have used Mean_squred_error
model.compile(loss="mean_squared_error",optimizer='adam',metrics=["accuracy",precision,recall])
```

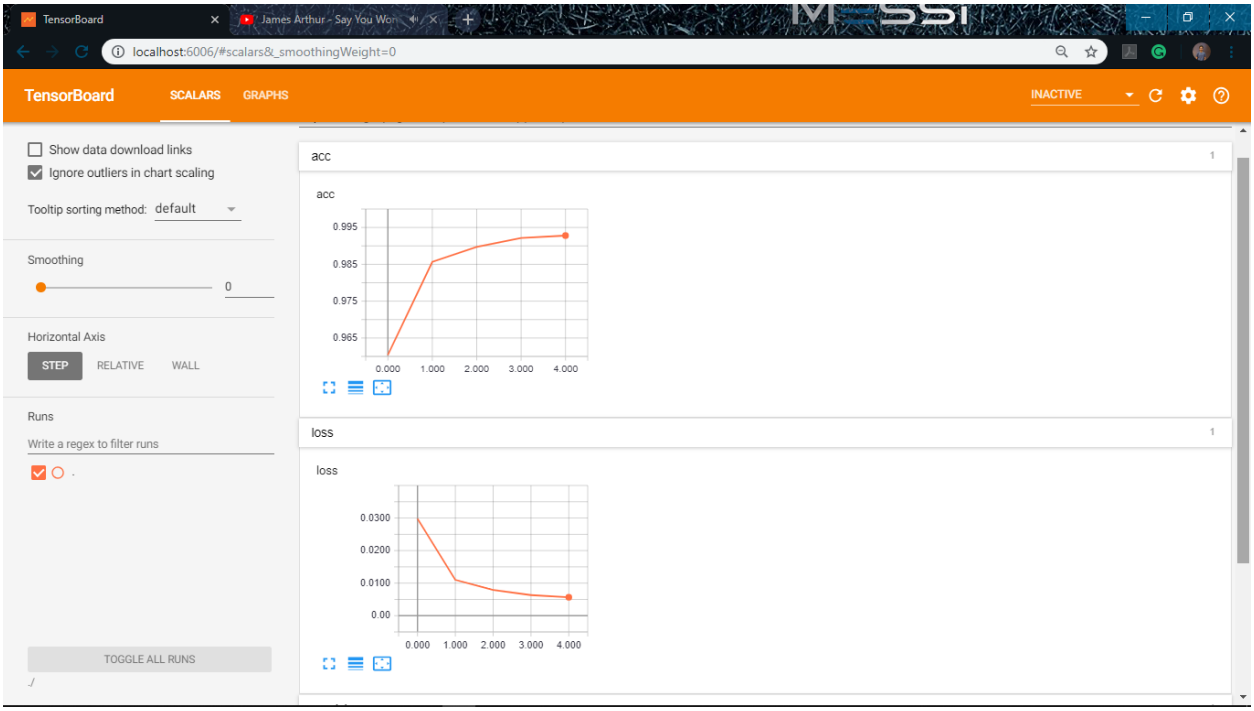
Results:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 154s 3ms/step
- loss: 0.0298 - acc: 0.9604 - precision: 0.4992 - recall:
0.9983 - val_loss: 0.0086 - val_acc: 0.9888 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 2/5
60000/60000 [==============================] - 153s 3ms/step
- loss: 0.0110 - acc: 0.9857 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0054 - val_acc: 0.9931 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 3/5
60000/60000 [==============================] - 154s 3ms/step
- loss: 0.0079 - acc: 0.9897 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0040 - val_acc: 0.9954 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 4/5
60000/60000 [==============================] - 151s 3ms/step
- loss: 0.0063 - acc: 0.9922 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0041 - val_acc: 0.9949 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 5/5
60000/60000 [==============================] - 151s 3ms/step
- loss: 0.0057 - acc: 0.9928 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0044 - val_acc: 0.9943 - val_precision:
0.5000 - val_recall: 1.0000
```

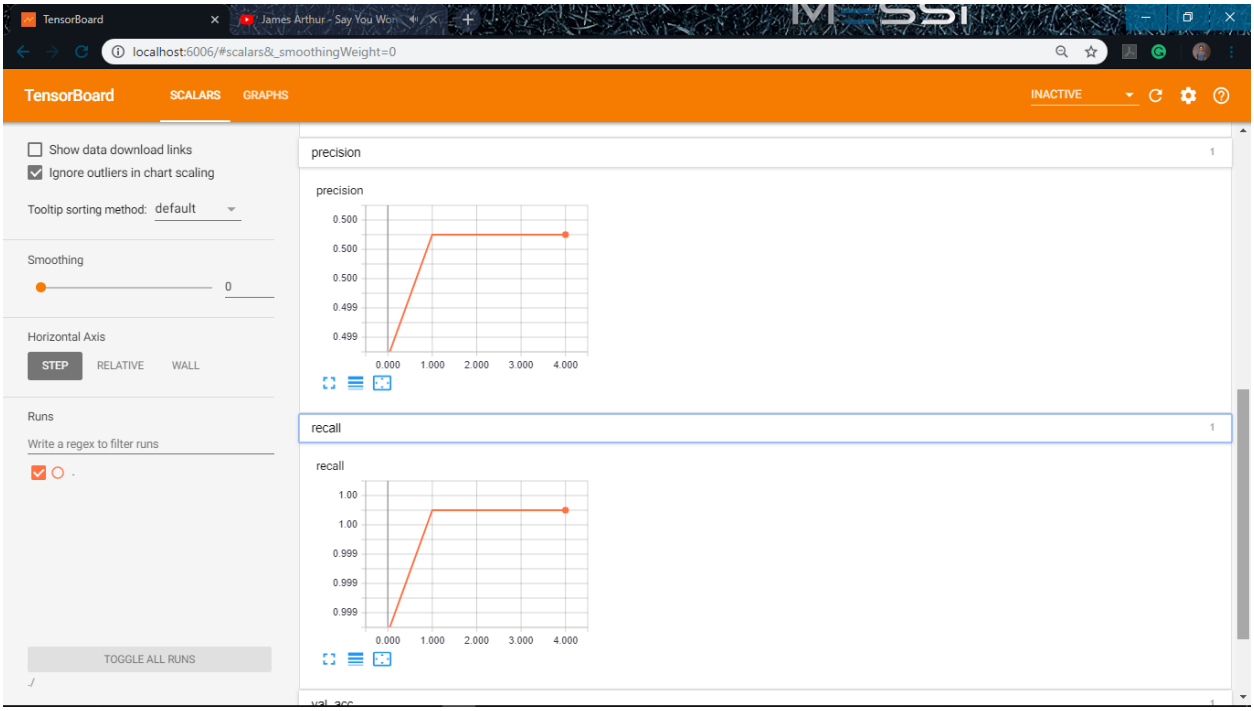
Accuracy, Loss, Precision and Recall:

```
Total Loss of the model: 0.0030786980311946727
Total Accuracy of the model: 0.9962333333333333
Total Precision of the model: 0.5
Total Recall of the model: 1.0
```

Accuracy and Loss of the Training dataset:



Precision and Recall of the Training Dataset:



Accuracy and Loss of validation Test data:

Precision and Recall of Validation test data:

Performance:

- **Accuracy is 99.62%**
- **Loss is 00.30%**

Comparison of Results:

- **By adding Adam Optimizer and using loss function mean square error, Accuracy** comes out to be **99.62%** which is extremely higher than the original CNN model which uses SGD optimizer and all other models which we implemented.

Conclusion:

- Hence, we can conclude that **Adam learns the fastest**.
- **Adam is more stable than the other optimizers; it doesn't suffer any major decreases in accuracy.**

5. Weight Initializer

#Input Parameter

```
img_rows = 28
img_cols = 28
epochs = 5
batch_size = 100
drop_rate = 0.4
learn_rate = 0.001
#CNN model
def model_cnn():
   model = Sequential()
   model.add(Conv2D(32,kernel_size=(5,5),input_shape=(1, img_rows,img_cols),activation =
"relu",kernel_initializer = 'he_normal'))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Conv2D(64,kernel_size=(5,5),activation="relu",kernel_initializer = 'he_normal'))
   model.add(MaxPooling2D(pool_size=(2,2)))
   model.add(Dropout(drop_rate))
   model.add(Flatten())#convert the 2d matrix into 1d vector
   model.add(Dense(64,activation="relu"))
   model.add(Dense(20,activation ="relu"))
   model.add(Dense(number_of_classes,activation="softmax"))
   return(model)
```

#Compiler

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

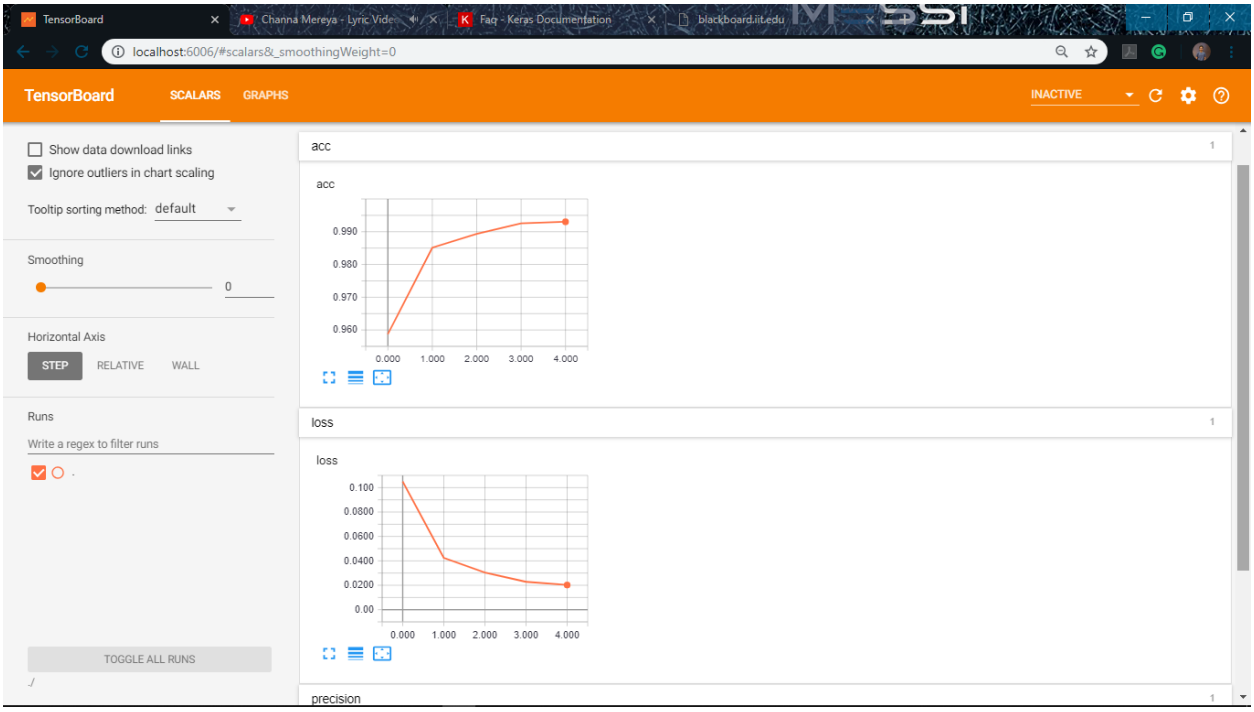
Result:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 183s 3ms/step
- loss: 0.1050 - acc: 0.9588 - precision: 0.4992 - recall:
0.9983 - val_loss: 0.0315 - val_acc: 0.9890 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 2/5
60000/60000 [==============================] - 190s 3ms/step
- loss: 0.0425 - acc: 0.9851 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0221 - val_acc: 0.9922 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 3/5
60000/60000 [==============================] - 159s 3ms/step
- loss: 0.0305 - acc: 0.9893 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0214 - val_acc: 0.9926 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 4/5
60000/60000 [==============================] - 152s 3ms/step
- loss: 0.0228 - acc: 0.9925 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0208 - val_acc: 0.9930 - val_precision:
0.5000 - val_recall: 1.0000
Epoch 5/5
60000/60000 [==============================] - 150s 2ms/step
- loss: 0.0202 - acc: 0.9930 - precision: 0.5000 - recall:
1.0000 - val_loss: 0.0163 - val_acc: 0.9947 - val_precision:
0.5000 - val_recall: 1.0000
```

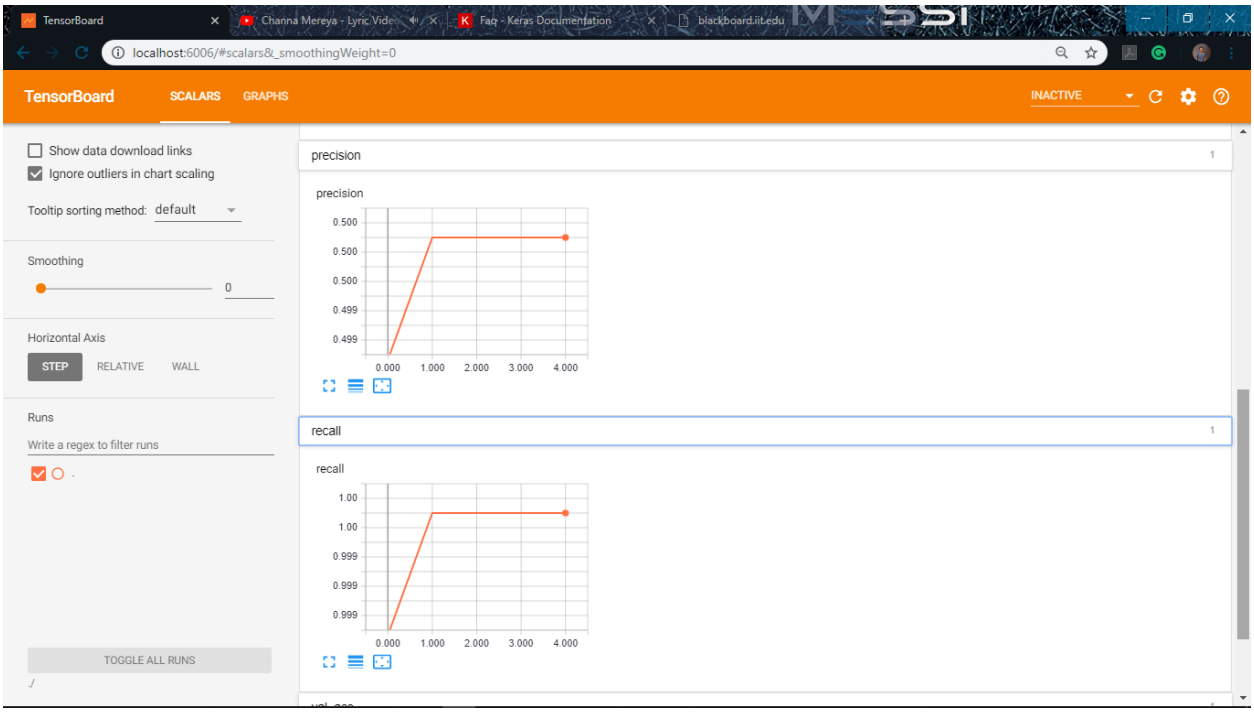
Accuracy, Loss, Precision and Recall:

```
Total Loss of the model: 0.009002393317682435
Total Accuracy of the model: 0.9971166666666667
Total Precision of the model: 0.5
Total Recall of the model: 1.0
```

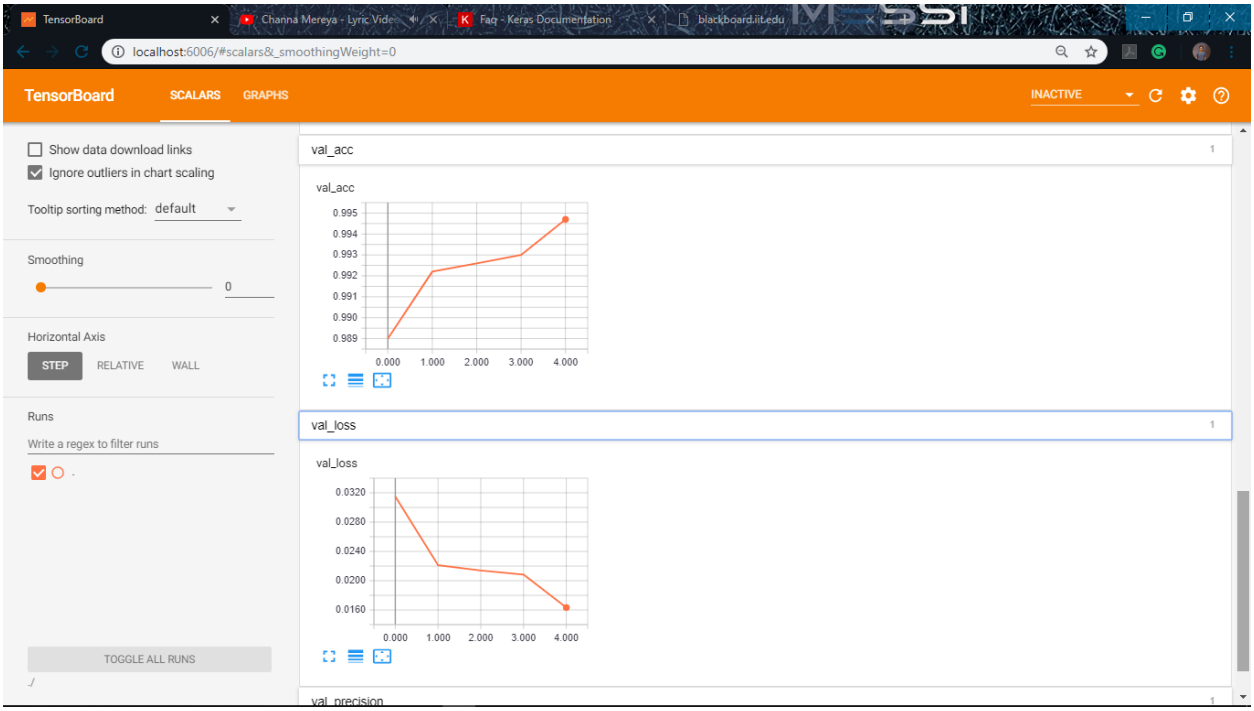
Accuracy and Loss of Training dataset:



Precision and Recall of Training dataset:



Accuracy and Loss of Training dataset:



Precision and Recall of Training dataset:

Performance:

- **Accuracy is 99.71%**
- **Loss is 00.90%**

Comparison of Results:

- **By adding Weight Initializer with Adam Optimizer Accuracy** comes out to be **99.71%** which is extremely higher than the original CNN model which uses SGD optimizer and little bit higher than the model which doesn't use weight initialize.

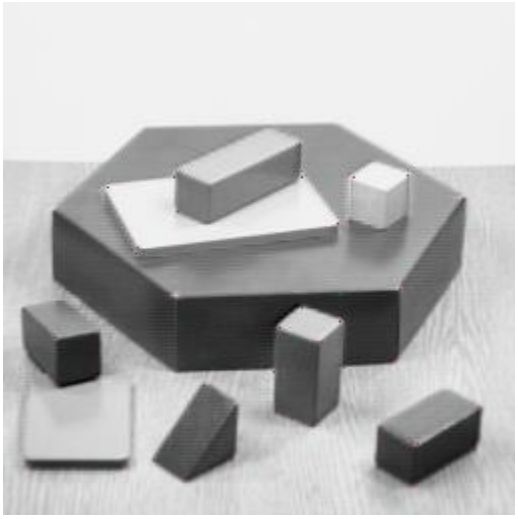
Conclusion:

- Hence, we can conclude that **Adam learns the fastest with weight Initializer**.

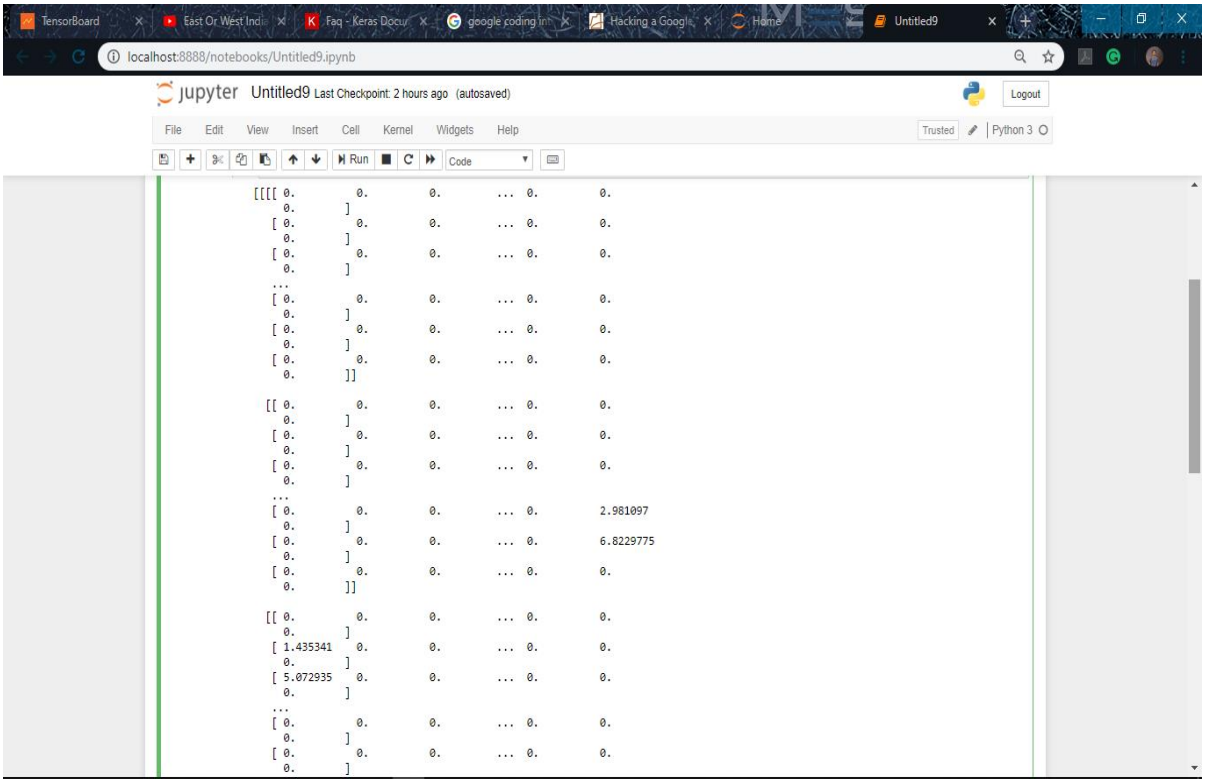
6. Using features from pretrained model VGG16

```
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

model = VGG16(weights='imagenet', include_top=False)

img_path = 'test2.png'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

features = model.predict(x)
print(features)
```

Input Image:



Output:


TensorBoard
East Or West Indi
Faq - Keras Docu
google coding in
Hacking a Google
Home
Untitled9
localhost:8888/notebooks/Untitled9.ipynb
jupyter Untitled9 Last Checkpoint: 2 hours ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
[[[[0. 0. 0. ... 0. 0.
0.]
[0. 0. 0. ... 0. 0.
0.]
[0. 0. 0. ... 0. 0.
0.]
...
[0. 0. 0. ... 0. 0.
0.]
[0. 0. 0. ... 0. 0.
0.]
[0. 0. 0. ... 0. 0.
0.]]
[[0. 0. 0. ... 0. 0.
0.]
[0. 0. 0. ... 0. 0.
0.]
[0. 0. 0. ... 0. 0.
0.]
...
[0. 0. 0. ... 0. 2.981097
0.]
[0. 0. 0. ... 0. 6.8229775
0.]
[0. 0. 0. ... 0. 0.
0.]]
[[0. 0. 0. ... 0. 0.
0.]
[1.435341 0. 0. ... 0. 0.
0.]
[5.072935 0. 0. ... 0. 0.
0.]
...
[0. 0. 0. ... 0. 0.
0.]
[0. 0. 0. ... 0. 0.
0.]

Deliverable 3: Application

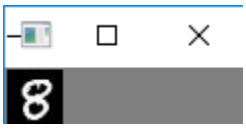
Algorithm:

1. First load the saved model
2. Now compile the model
3. Applied while loop to continuously take the input image path
4. Input the image path
5. Convert the image into Grayscale by using cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
6. Resize the grayscale image into 28*28 by using cv2.resize(gray,(28,28))
7. Now convert the grayscale image into binary image by using cv2.adaptiveThreshold(gray,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY,11,2)
8. Now reshape the image by using np.reshape(binary_image,[1,1,28,28])
9. Now pass the reshape binary image to predict function of the model i.e model.predict_classes(binary_image)
10. You get the result as ‘0’ if the number in the image is even, else ‘1’ if the number in the image is odd.
11. If you want to continue to testing the image press any key except q, else press q for exit the while loop

Performance:

Using SGD:

Image path: img_6.jpg
28
binary image
img
(1, 1, 28, 28)
'0' represent as even class and '1' represent as odd class
Output Class: [0]
If you wanna continue press any key except 'q'



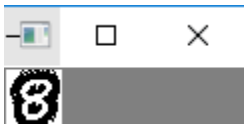


Image path: img_52.jpg
28
binary image
img
(1, 1, 28, 28)

'0' represent as even class and '1' represent as odd class

Output Class: [0]
If you wanna continue press any key except 'q'


Image path: images.jpg
28
binary image
img
(1, 1, 28, 28)
'0' represent as even class and '1' represent as odd clas
Output Class: [0]
If you wanna continue press any key except 'q'

Alternative Using Adam class:

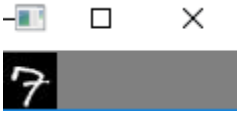
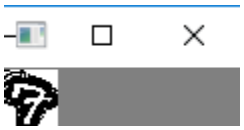



Image path: img_6.jpg
28
binary image
img
(1, 1, 28, 28)
'0' represent as even class and '1' represent as odd class
Output Class: [1]
If you wanna continue press any key except 'q'


Image path: img_52.jpg
28
binary image
img
(1, 1, 28, 28)
'0' represent as even class and '1' represent as odd class
Output Class: [0]
If you wanna continue press any key except 'q'

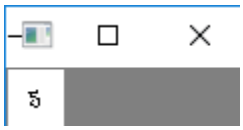


Image path: images.jpg
28
binary image
img
(1, 1, 28, 28)
'0' represent as even class and '1' represent as odd clas
Output Class: [0]
If you wanna continue press any key except ‘q’

Comparison and Result:

- If we use **SGD** as optimizer, the prediction rate of detecting images is not good.
- If we use **Adam** as an optimizer, the prediction rate of detecting images is very good.
- We can see above **result** which tell us difference of both optimizer and performance of **cnn** model.
- **But, if we tried for the our own handwritten digits we are not getting good result in both cases.**

Conclusion:

- Form the above discussion and result we can conclude that, the cnn model works good for mnist dataset but not giving good results for the own handwritten digits. The accuracy is very less for own handwritten digits compared to other images.
- The result is very good if we used **‘Adam’** optimizer as compared to **‘SGD’**.