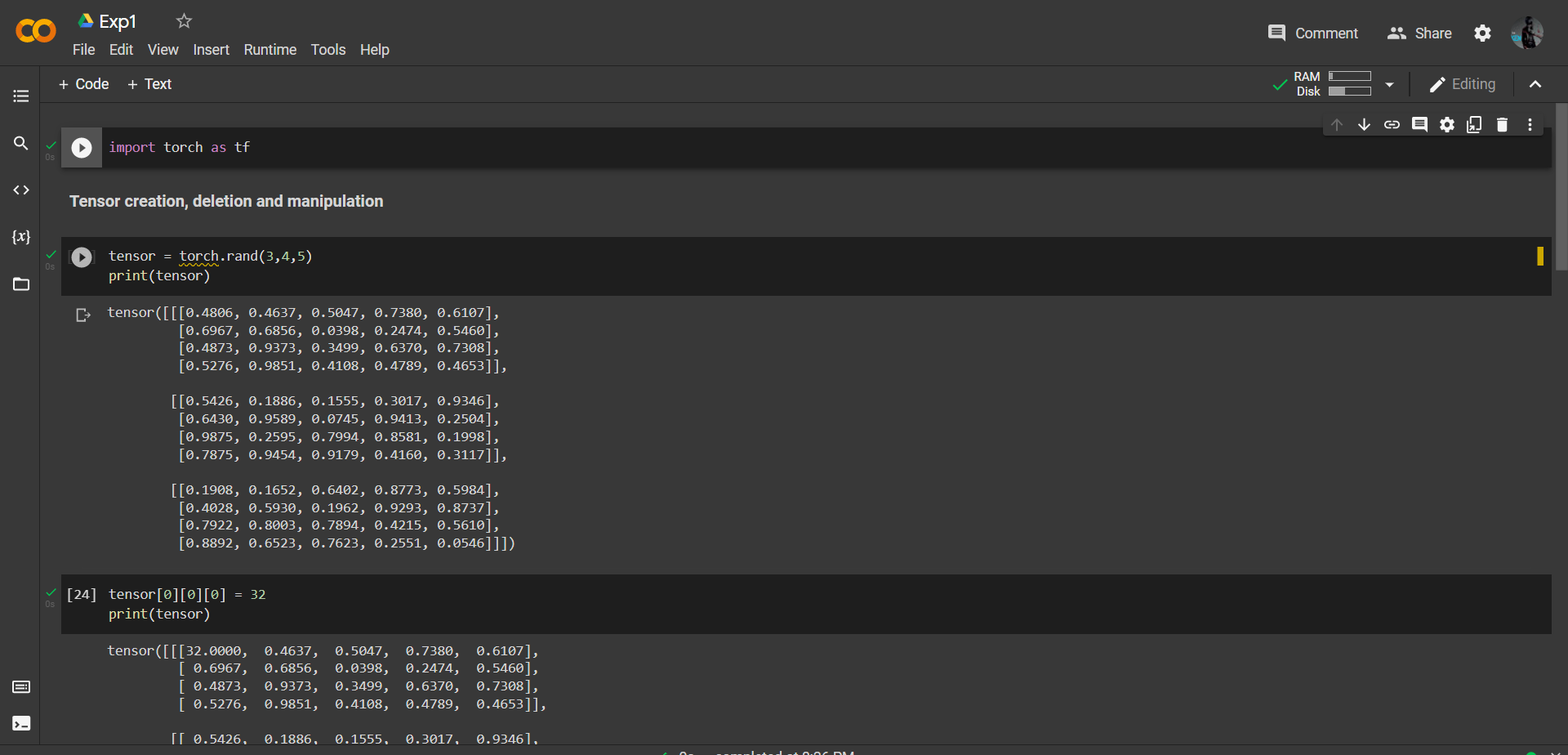
**ROHAN NYATI**

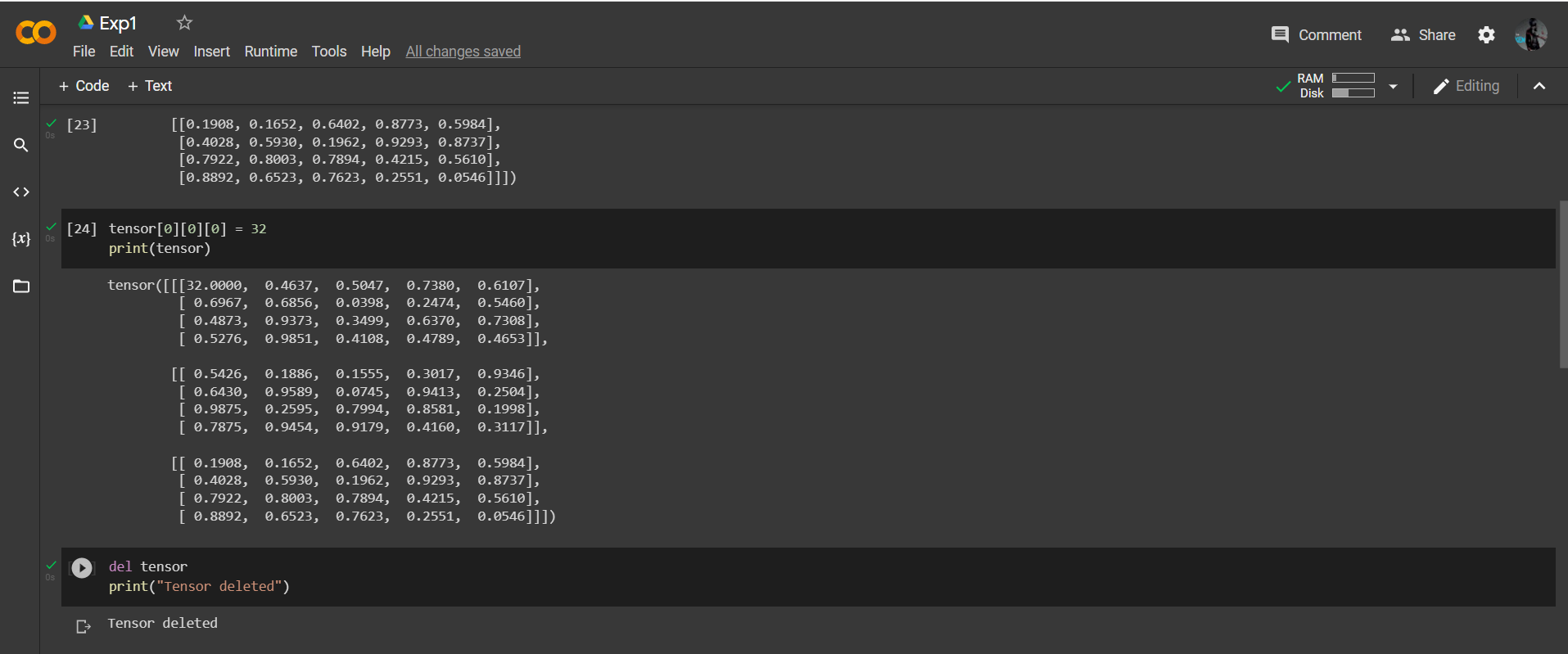
**500075940**

**R177219148**

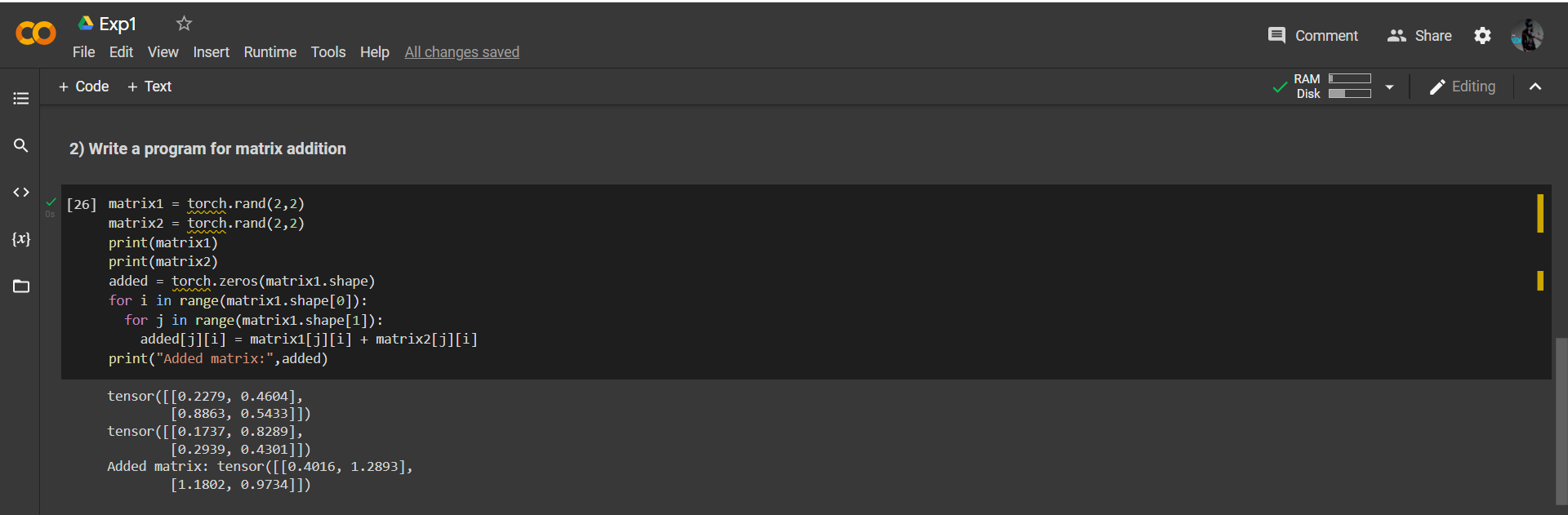
**BATCH – 5 (Ai & Ml )**

**Experiment -1**

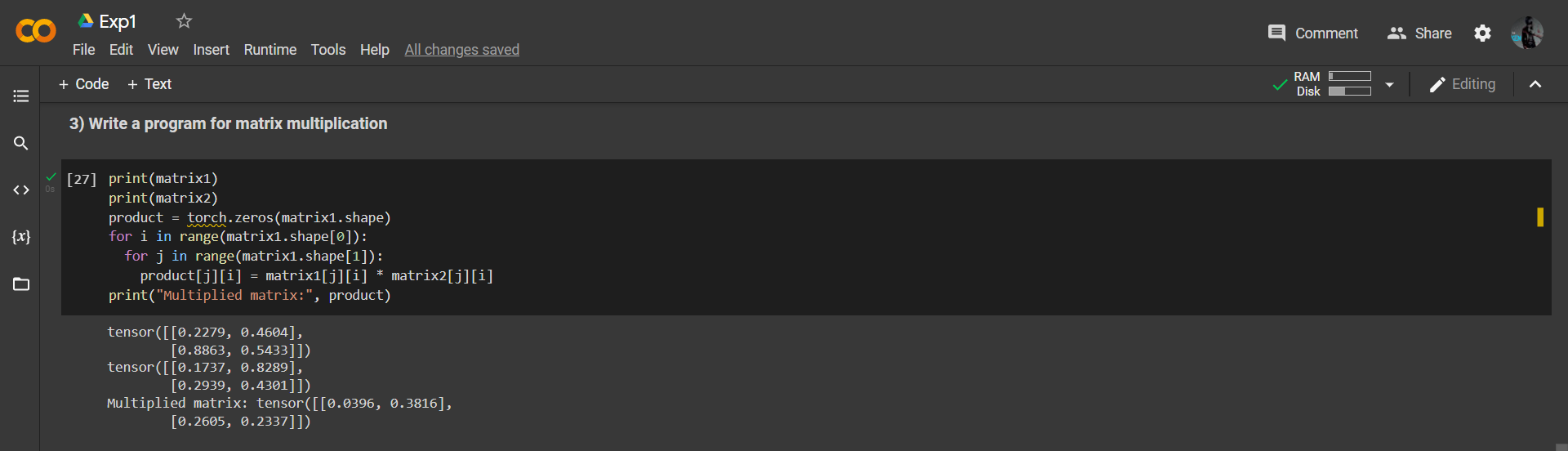




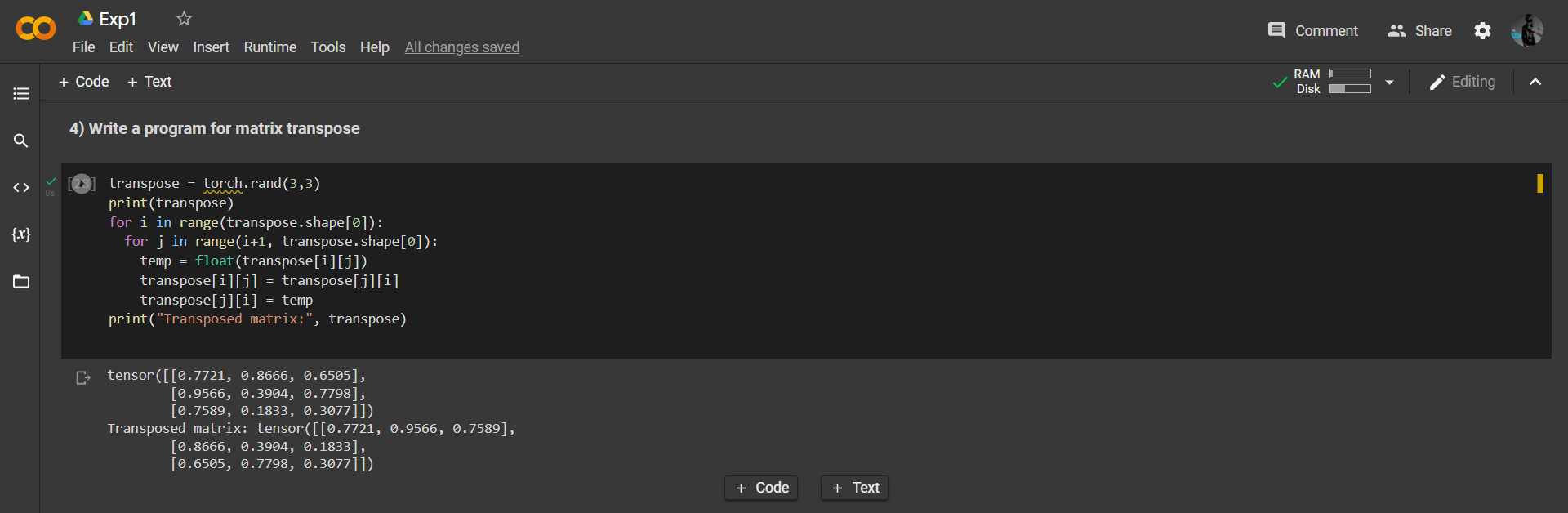
**2) Write a Program for Matrix Addition**



**3) Write a Program for Matrix Multiplication**



**4) Write a Program for Matrix Transpose**

****

**Experiment -2**

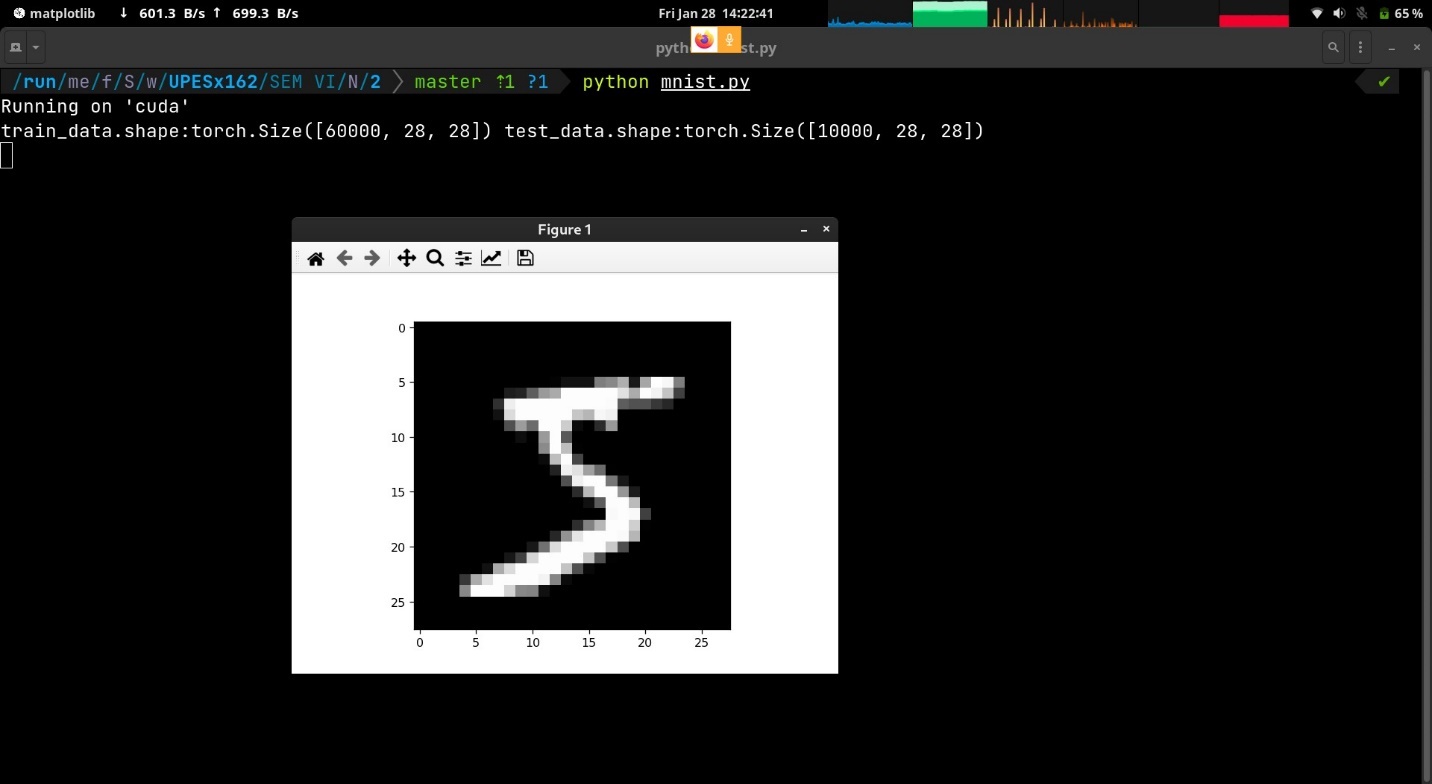
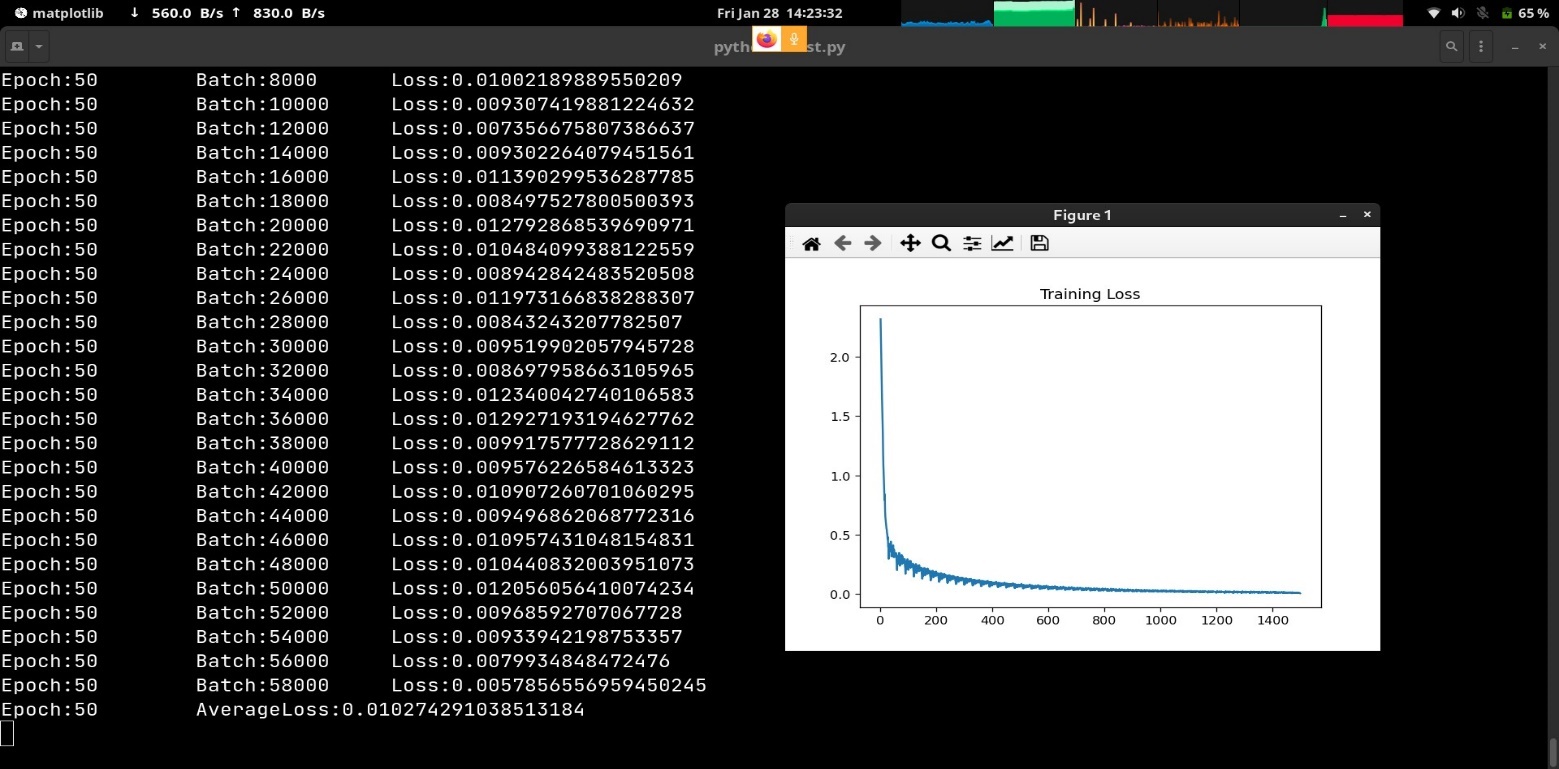
|  |
| --- |
| # Import torch |
|  |
| **import** torch |
| **import** torch.nn **as** nn |
| **import** torchvision |
| **import** matplotlib.pyplot **as** plt |
|  |
| # Device configuration |
| device = torch.device('cuda' **if** torch.cuda.is\_available() **else** 'cpu') |
| **print**(**f**"Running on '{device}'") |

|  |
| --- |
|  |
| # Downloading dataset |
|  |
| **from** torchvision **import** datasets |
|  |
| train\_data = datasets.MNIST( |
| *root* = 'data', |
| *train* = True, |
| *download* = True, |
| ) |
|  |
| test\_data = datasets.MNIST( |
| *root* = 'data', |
| *train* = False, |
| *download* = True, |
| ) |
|  |
| # Loading dataset |
|  |
| train\_labels = train\_data.targets |
| test\_labels = test\_data.targets |
| train\_data = train\_data.data |
| test\_data = test\_data.data |
|  |
| **print**(**f**"train\_data.shape:{train\_data.shape} test\_data.shape:{test\_data.shape}") |
|  |
| # Exploring data |
|  |
| plt.imshow(train\_data[0], *cmap*='gray') |
| plt.show() |
|  |
| # Set hyperparameters |
|  |
| n\_train\_sample = train\_data.shape[0] |
| n\_test\_samples = test\_data.shape[0] |
|  |
| image\_size = train\_data.shape[-1] |
| input\_size = image\_size \* image\_size |
| hidden\_layer\_size = 512 |
| n\_classes = 10 |
|  |
| n\_iters = 50 |
| batch\_size = 2000 |
| lr = 0.001 |

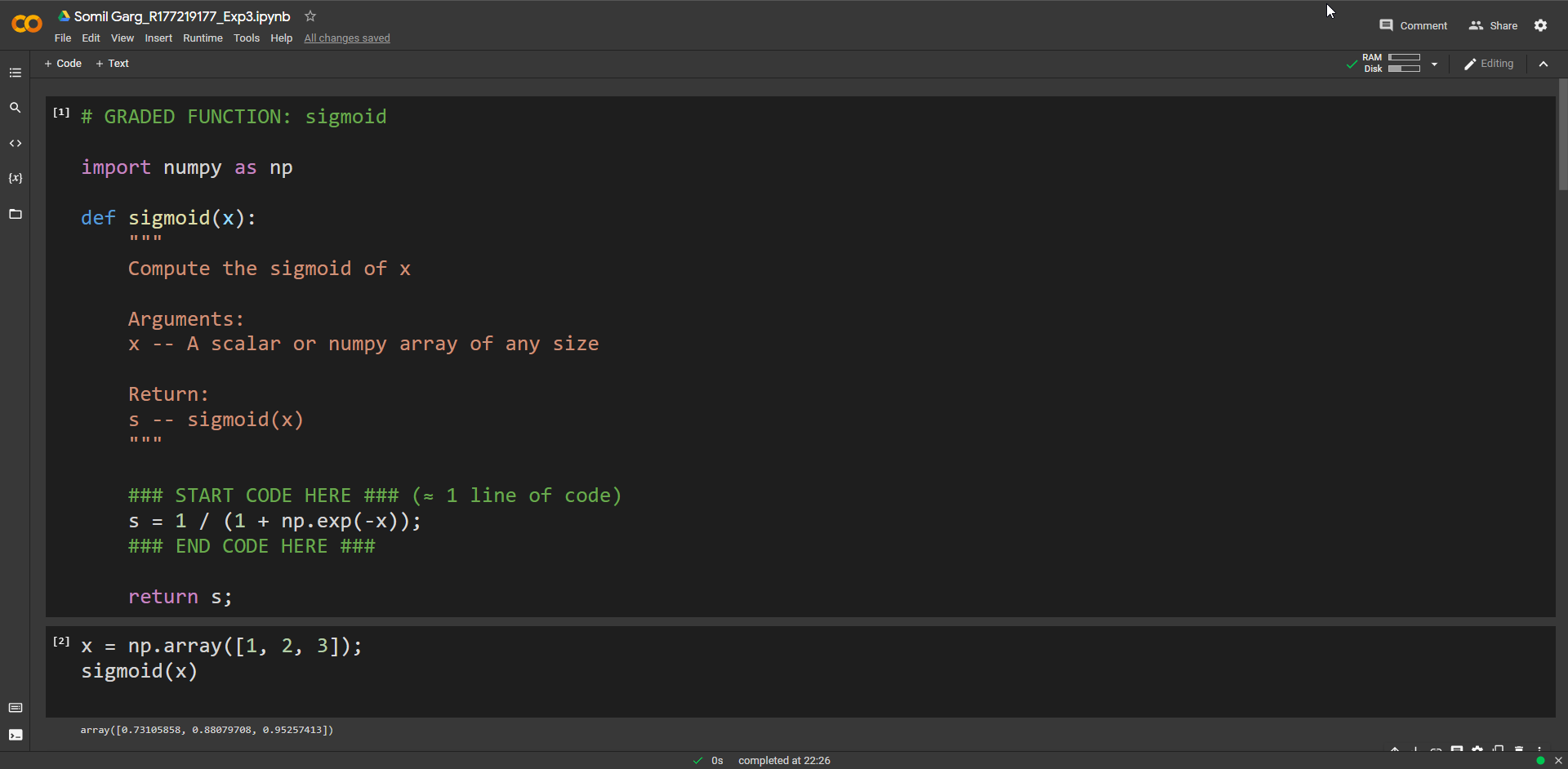
|  |
| --- |
|  |
| # Transform data |
|  |
| train\_data = train\_data.to(torch.float32)/255. |
| test\_data = test\_data.to(torch.float32)/255. |
| train\_data = train\_data.reshape(-1, input\_size) |
| test\_data = test\_data.reshape(-1, input\_size) |
|  |
| # Store all data on gpu |
|  |
| train\_data = train\_data.to(device) |
| test\_data = test\_data.to(device) |
| train\_labels = train\_labels.to(device) |
| test\_labels = test\_labels.to(device) |
|  |
| # Making model |
|  |
| **class FFNN**(***nn***.***Module***): |
|  |
| **def \_\_init\_\_**(*self*, *input\_size*, *hidden\_layer\_size*, *n\_classes*): |
| ***super***(FFNN, self).**\_\_init\_\_**() |
| self.input\_size = input\_size |
| self.hidden\_layer\_size = hidden\_layer\_size |
| self.n\_classes = n\_classes |
| # Adding layers |
| self.l1 = nn.Linear(input\_size, hidden\_layer\_size) |
| self.l1\_activation = nn.ReLU() |
| self.l2 = nn.Linear(hidden\_layer\_size, n\_classes) |
|  |
| **def forward**(*self*, *x*): |
| z = self.l1(x) |
| a = self.l1\_activation(z) |
| z2 = self.l2(a) |
| **return** z2 |
|  |
| # Making model and training |
|  |
| ffnn = FFNN(input\_size, hidden\_layer\_size, n\_classes) |
| ffnn.to(device) |
|  |
| # Loss and Optimizer |
|  |
| criterion = nn.CrossEntropyLoss() |
| optimizer = torch.optim.Adam(ffnn.parameters(), *lr*=lr) |

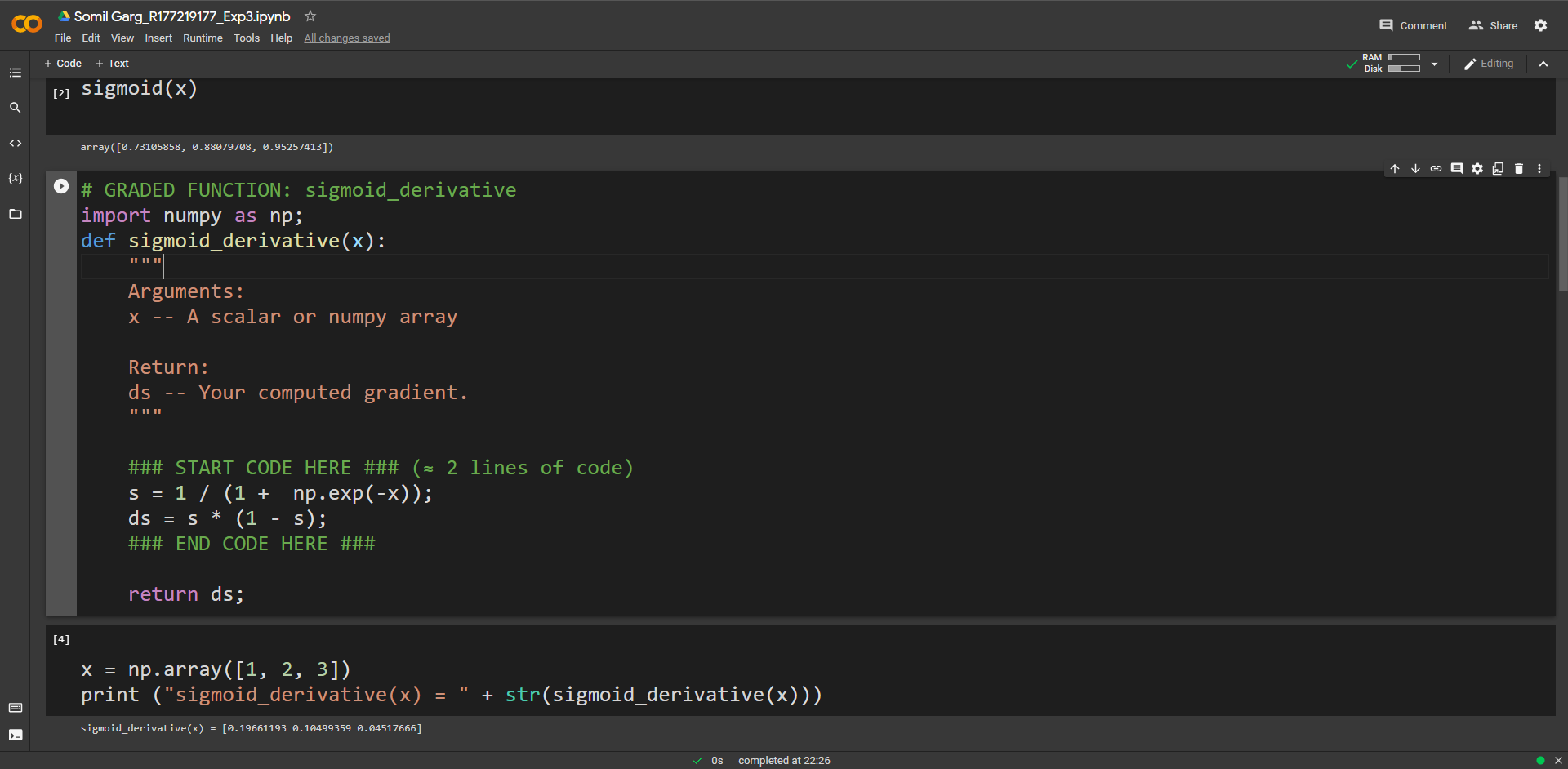
|  |
| --- |
|  |
| # Training |
|  |
| losses = [] |
| test\_losses = [] |
|  |
| **for** epoch **in range**(n\_iters): |
| avg, c = 0, 0 |
| **for** idx **in range**(0, n\_train\_sample, batch\_size): |
| # Get images and labels for current epoch |
| batch\_x = train\_data[idx:idx+batch\_size] |
| batch\_labels = train\_labels[idx:idx+batch\_size] |
| # Forward pass |
| predictions = ffnn.forward(batch\_x) |
| # Compute Loss |
| loss = criterion(predictions, batch\_labels) |
| losses.append(***float***(loss)) |
| avg, c = avg+loss, c+1 |
| # Backprop |
| optimizer.zero\_grad() # Reset gradients |
| loss.backward() # Recompute gradients |
| optimizer.step() # Update weights |
| **print**(**f**"Epoch:{epoch+1}\tBatch:{idx}\tLoss:{loss}") |
| # Output loss every 100 epochs |
| **if** ((epoch + 1) % 5) == 0: |
| **print**(**f**"Epoch:{epoch+1}\tAverageLoss:{(avg/c)}") |
| # Compute loss on test set |
| predictions = ffnn.forward(test\_data) |
| test\_loss = criterion(predictions, test\_labels) |
| test\_losses.append(***float***(test\_loss)) |
|  |
| # Plot losses |
|  |
| plt.plot(losses) |
| plt.title("Training Loss") |
| plt.show() |
|  |
| # Plot test losses |
|  |
| plt.plot(test\_losses) |
| plt.title("Test Losses") |
| plt.show() |
|  |
| # Finally compute loss on both sets once more |
|  |
| predictions = ffnn.forward(train\_data) |
| train\_loss = criterion(predictions, train\_labels) |
| **print**(**f**"Loss on train set: {train\_loss}") |
|  |
| predictions = ffnn.forward(test\_data) |
| test\_loss = criterion(predictions, test\_labels) |
| **print**(**f**"Loss on test set: {test\_loss}") |
|  |
| # Save model for future use |
|  |
| torch.save(ffnn.state\_dict(), "./ffnn\_mnist.torch") |

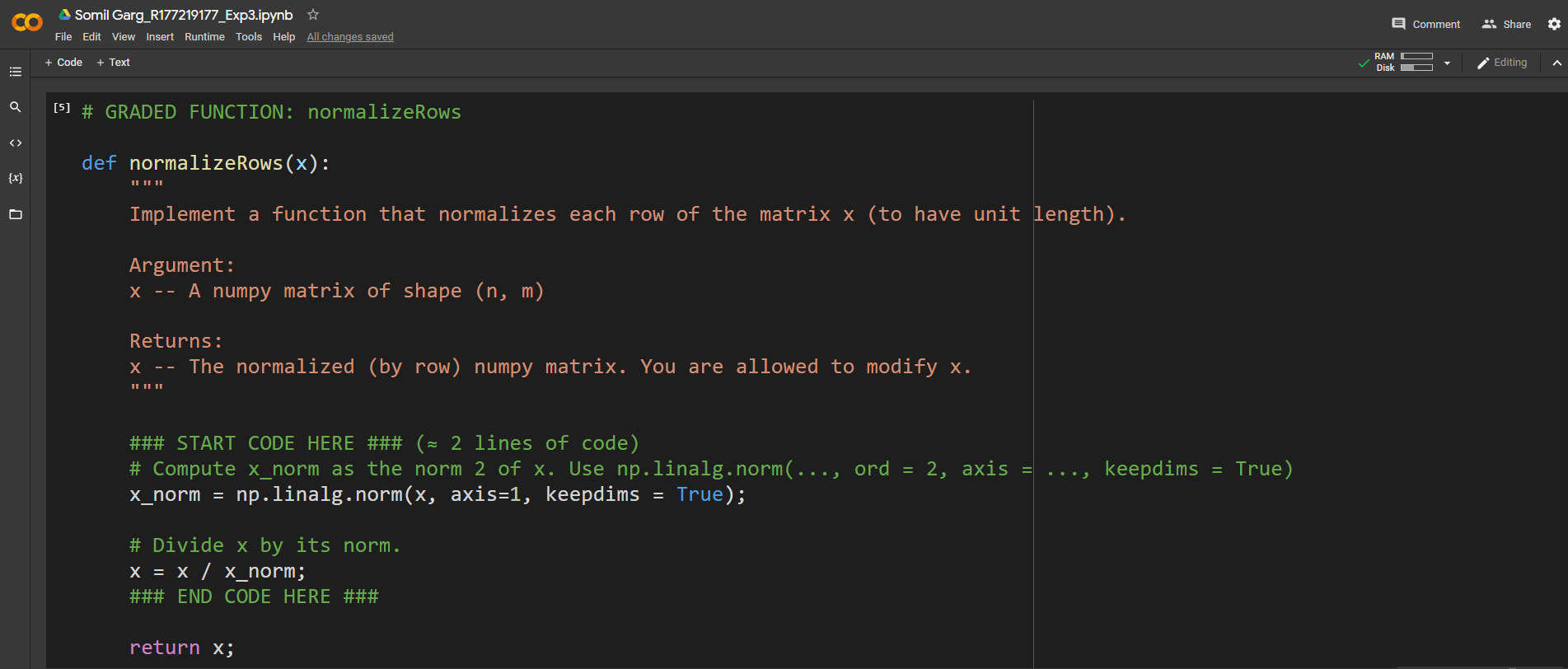
# OUTPUT AND SCREENSHOTS



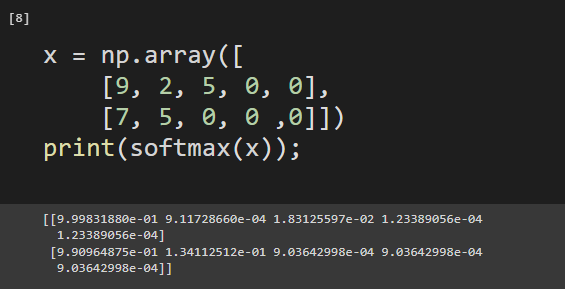
**Experiment -> 3**

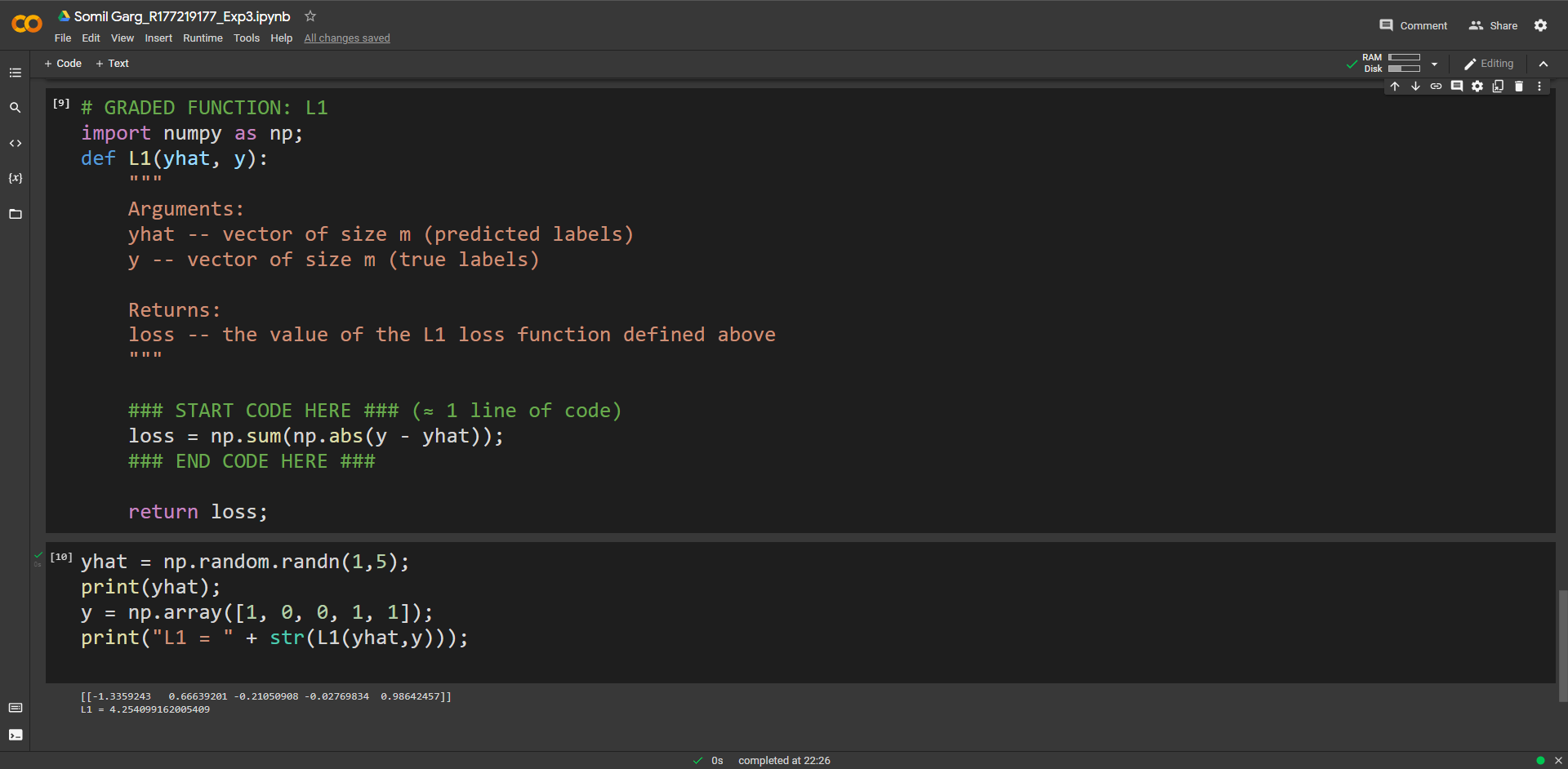




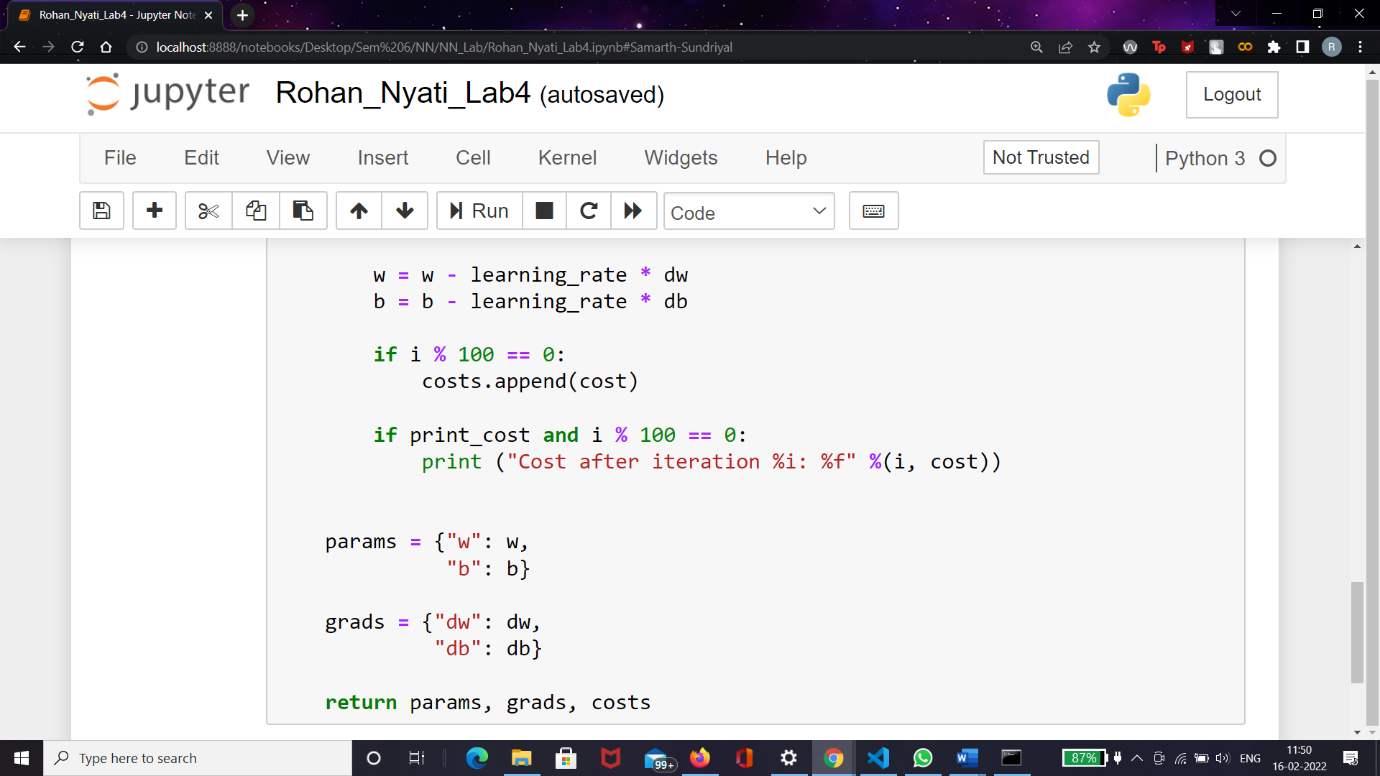
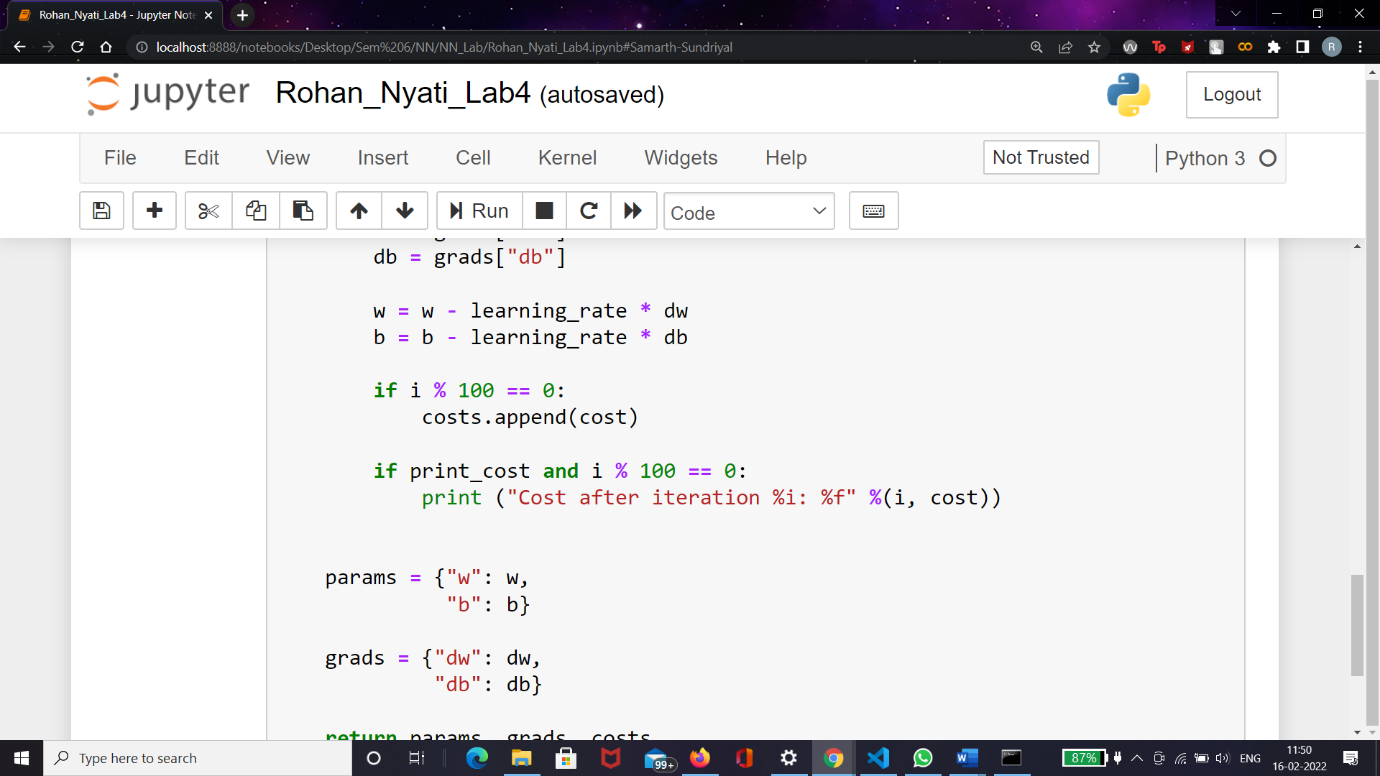
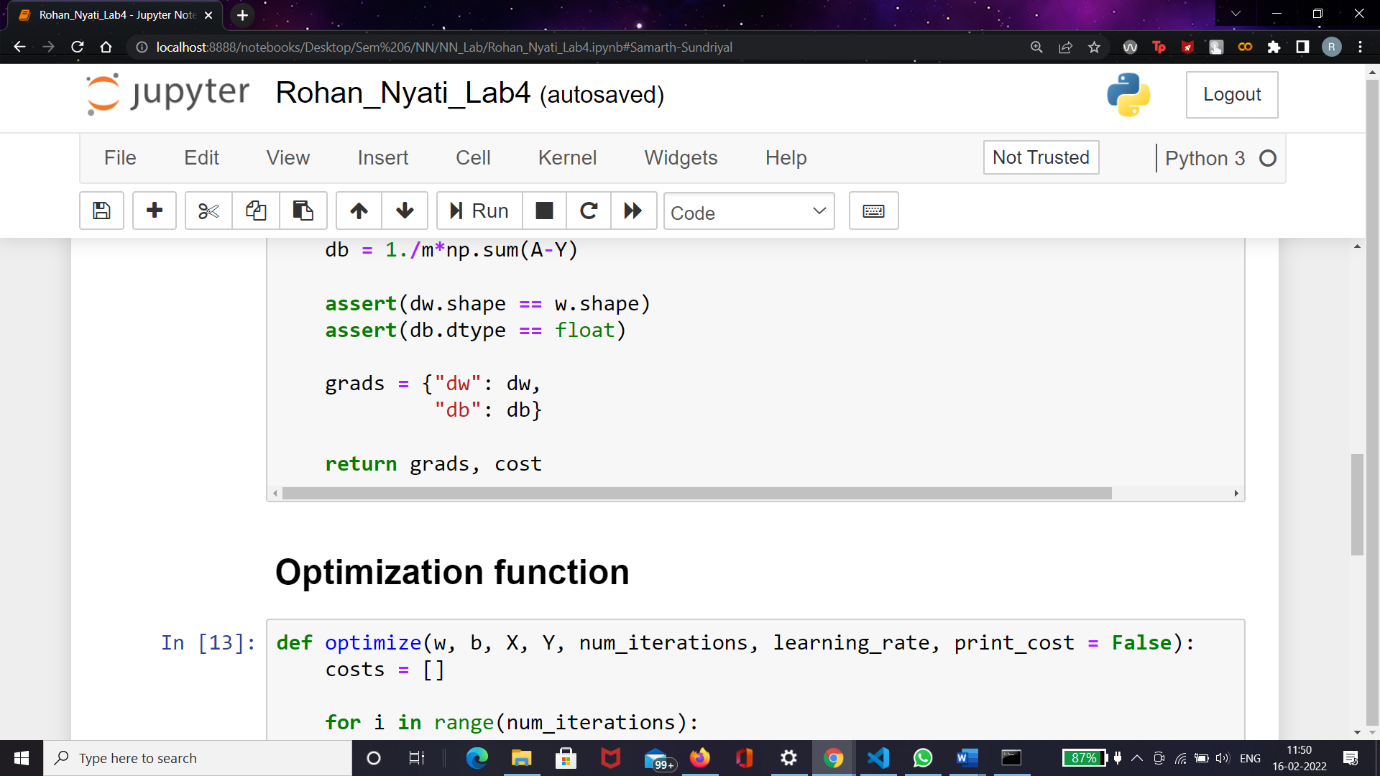
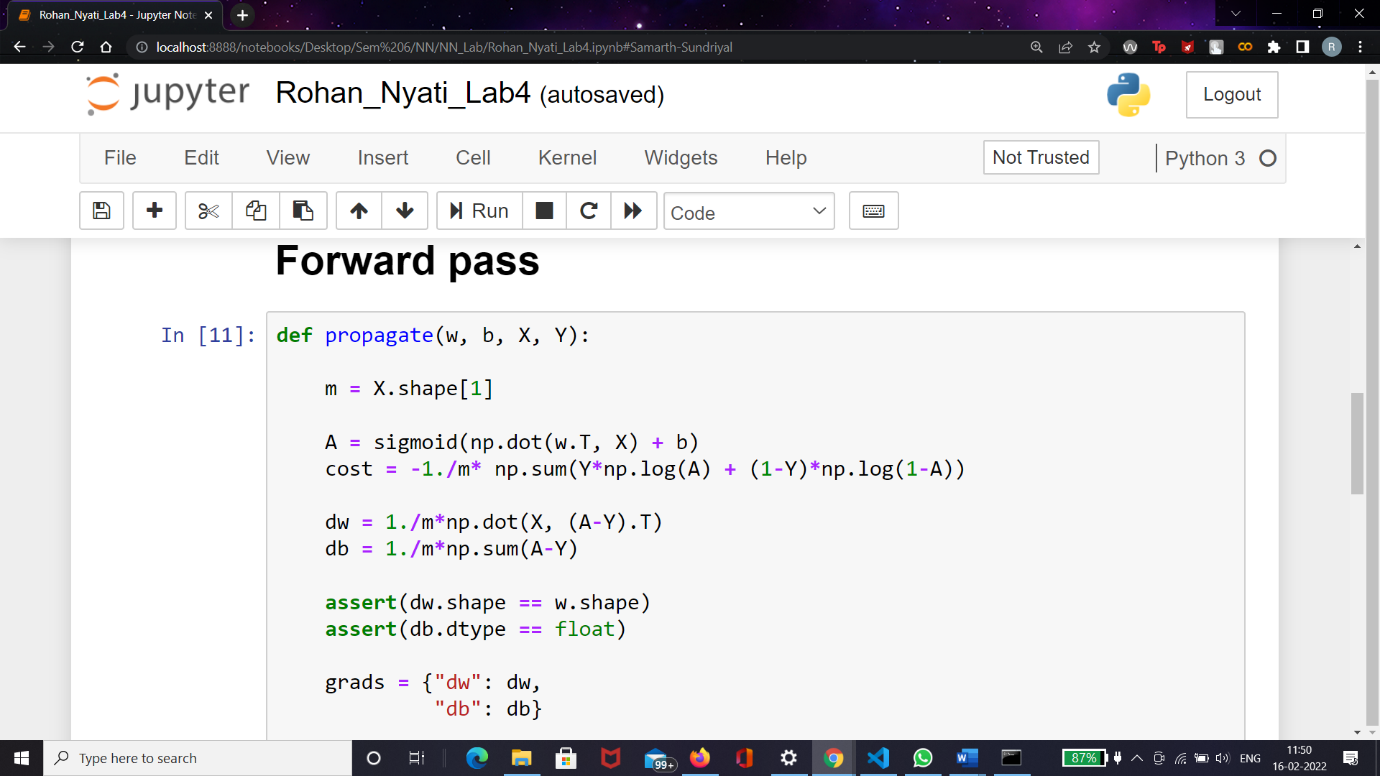
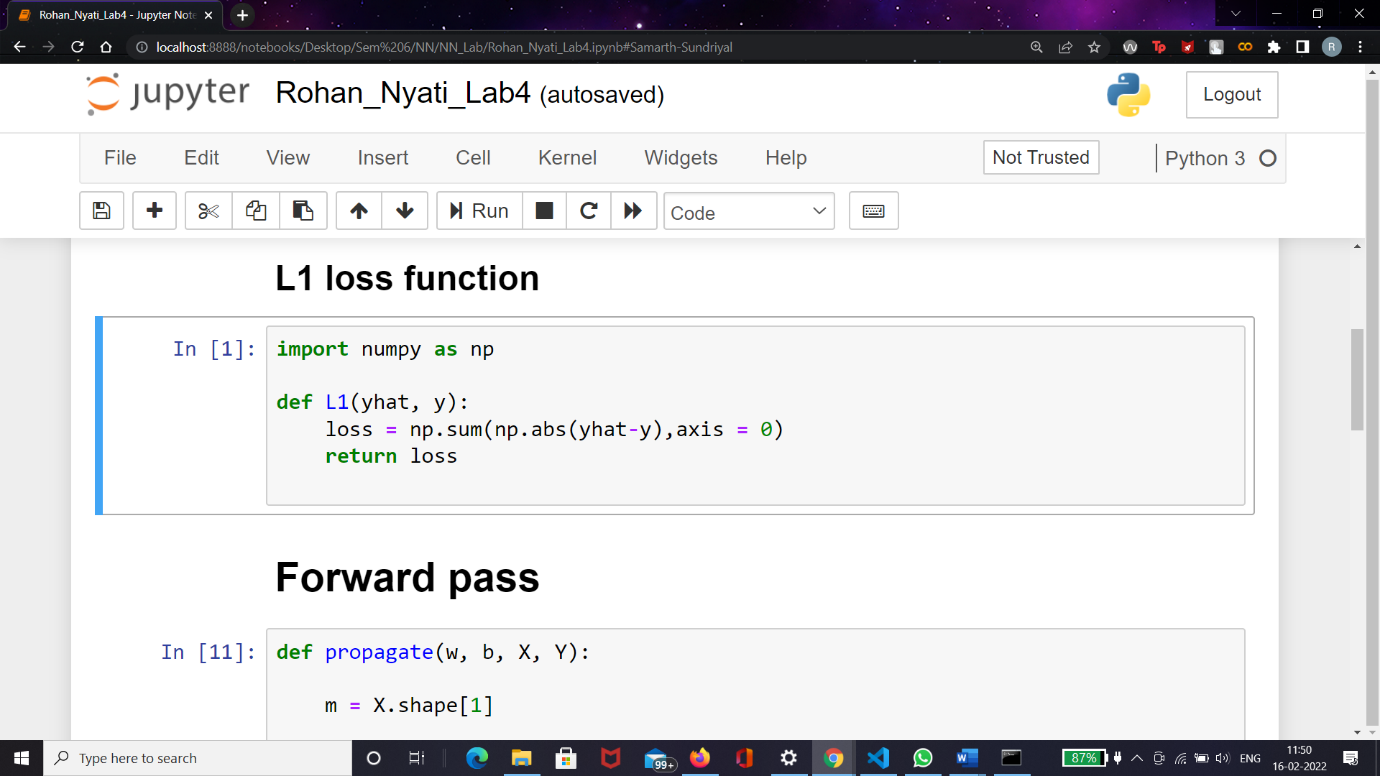
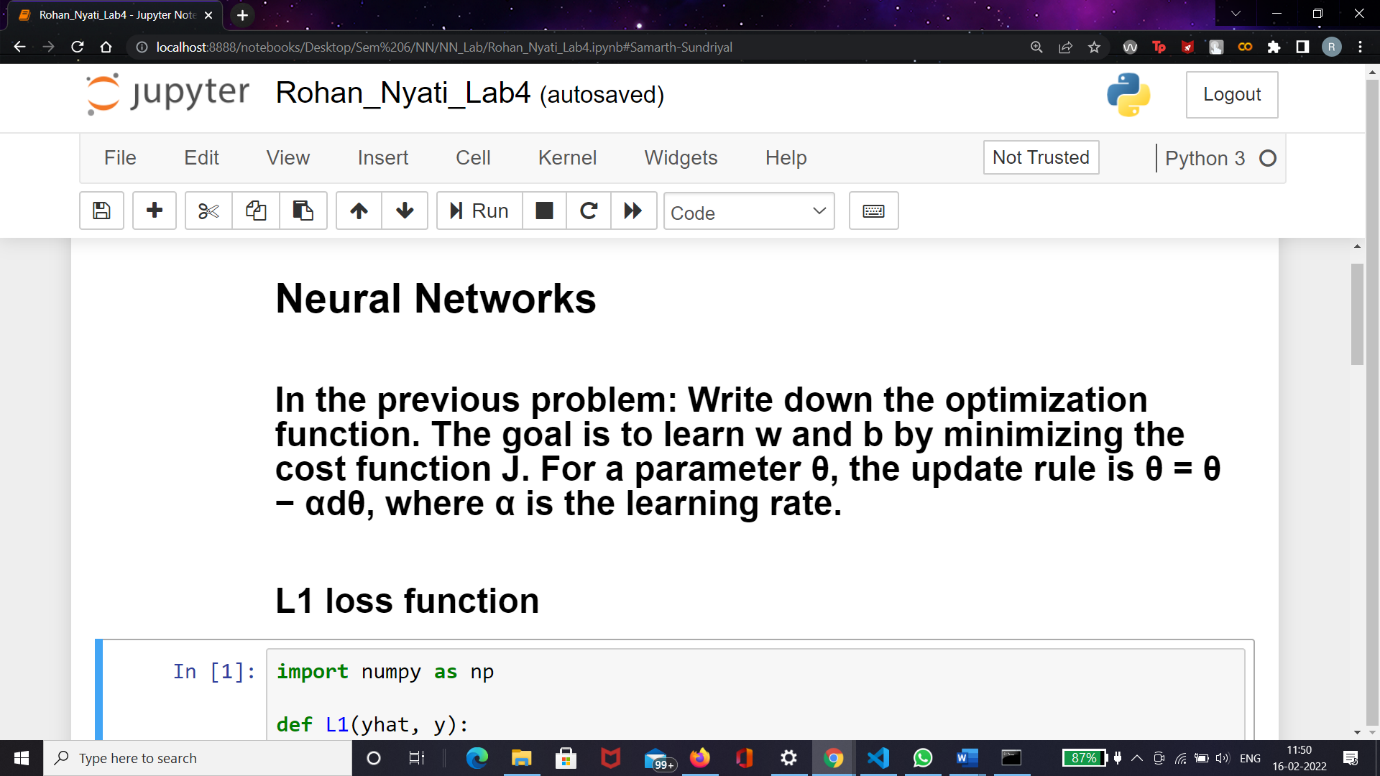


****

****



**Experiment - 4**



**Experiment - 5**

# Import PyTorch

[2]:

**import**

**torch**

**import**

**torchvision**

**import**

**torchvision**

**.**

**datasets**

**as**

**datasets**

**import**

**torch**

**.**

**nn**

**as**

**nn**

**import**

**torchvision**

**.**

**transforms**

**as**

**transforms**

**import**

**matplotlib**

**.**

**pyplot**

**as**

**plt**

# Initialize Hyper-parameters

[3]:

input\_size

=

784

hidden\_size

=

150

num\_classes

=

15

num\_epochs

=

5

batch\_size

=

150

learning\_rate

=

0.001

# Build the Feedforward Neural Network

[139]:

**class**

**NeuralNet**

(

nn

.

Module):

**def**

\_\_init\_\_

(

self

, input\_size, hidden\_size, num\_classes):

super

(

NeuralNet,

self

)

.

\_\_init\_\_

()

self

.

l1

=

nn

.

Linear(input\_size, hidden\_size)

self

.

relu

=

nn

.

ReLU()

self

.

l2

=

nn

.

Linear(hidden\_size, num\_classes)

**def**

forward

(

self

, x):

out

=

self

.

l1(x)

out

=

self

.

relu(out)

out

=

self

.

l2(out)

**return**

out

# Instantiate the FNN

[140]: model = NeuralNet(input\_size, hidden\_size, num\_classes)

# Enable GPU

[141]: device= torch.device('cuda' **if** torch.cuda.is\_available() **else** 'cpu')

# Choose the Loss Function and Optimizer

[142]: criterion = nn.CrossEntropyLoss() optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate)

# Training the FNN Model

[143]:

n\_total\_steps

=

len

(

train\_loader

)

**for**

epoch

**in**

range

(

num\_epochs

):

**for**

i, (images, labels)

**in**

enumerate

(

train\_loader

):

images

=

images

.

reshape(

-

1

,

28

\*

28

)

.

to(device)

labels

=

labels

.

to(device)

outputs

=

model(images)

loss

=

criterion(outputs, labels)

optimizer

.

zero\_grad()

loss

.

backward()

optimizer

.

step()

**if**

(

i

+

1

)

%

100

==

0

:

print

(

f

'

Epoch [

**{**

epoch

+

1

**}**

/

**{**

num\_epochs

**}**

]

, step

[

**{**

i

+

1

**}**

/

*,*

→

**{**

n\_total\_steps

**}**

]

, loss

:

**{**

loss

.

item()

**:**

.4

f

**}**

'

)

Epoch [1/5], step[100/400], loss: 0.3850

Epoch [1/5], step[200/400], loss: 0.3349

Epoch [1/5], step[300/400], loss: 0.1991

Epoch [1/5], step[400/400], loss: 0.2551

Epoch [2/5], step[100/400], loss: 0.2451

Epoch [2/5], step[200/400], loss: 0.2422

Epoch [2/5], step[300/400], loss: 0.2784

Epoch [2/5], step[400/400], loss: 0.2823

Epoch [3/5], step[100/400], loss: 0.1294

Epoch [3/5], step[200/400], loss: 0.1487

Epoch [3/5], step[300/400], loss: 0.0640

Epoch [3/5], step[400/400], loss: 0.1173

Epoch [4/5], step[100/400], loss: 0.1329

Epoch [4/5], step[200/400], loss: 0.1056

Epoch [4/5], step[300/400], loss: 0.1117

Epoch [4/5], step[400/400], loss: 0.1124

Epoch [5/5], step[100/400], loss: 0.1078

Epoch [5/5], step[200/400], loss: 0.0448

Epoch [5/5], step[300/400], loss: 0.0944

Epoch [5/5], step[400/400], loss: 0.0841

# Testing the FNN Model

[144]:

**with**

torch

.

no\_grad():

n\_correct

=

0

n\_samples

=

0

**for**

images, labels

**in**

test\_loader:

images

=

images

.

reshape(

-

1

,

28

\*

28

)

.

to(device)

labels

=

labels

.

to(device)

outputs

=

model(images)

\_, predicted

=

torch

.

max(outputs

.

data,

1

)

n\_samples

+

=

labels

.

size(

0

)

n\_correct

+

=

(

predicted

==

labels)

.

sum()

.

item()

[ ]:

**Experiment - 6**

**import tensorflow as tf**

x = tf.constant([[1., 2., 3.],

[4., 5., 6.]])

print(x)

print(x.shape) print(x.dtype)

In [1]:

In [2]:

x + x

In [3]:

5 \* x

In [4]:

x @ tf.transpose(x)

In [5]:

tf.concat([x, x, x], axis=0)

In [7]:

tf.reduce\_sum(x)

In [8]:

**if** tf.config.list\_physical\_devices('GPU'): print("TensorFlow \*\*IS\*\* using the GPU")

**else**:

print("TensorFlow \*\*IS NOT\*\* using the GPU")

In [9]:

var = tf.Variable([0.0, 0.0, 0.0])

In [10]:

var.assign([1, 2, 3])

In [11]:

var.assign\_add([1, 1, 1])

x = tf.Variable(1.0)

**def** f(x):

y = x\*\*2 + 2\*x - 5

**return** y

In [12]:

In [13]:

f(x)

In [14]:

**with** tf.GradientTape() **as** tape: y = f(x)

g\_x = tape.gradient(y, x) *# g(x) = dy/dx*

g\_x

In [15]:

@tf.function

**def** my\_func(x):

print('Tracing.**\n**')

**return** tf.reduce\_sum(x)

In [16]:

x = tf.constant([1, 2, 3]) my\_func(x)

In [17]:

x = tf.constant([10, 9, 8]) my\_func(x)

In [18]:

x = tf.constant([10.0, 9.1, 8.2], dtype=tf.float32) my\_func(x)

In [19]:

**class MyModule**(tf.Module): **def** init (self, value):

self.weight = tf.Variable(value)

@tf.function

**def** multiply(self, x):

**return** x \* self.weight

mod = MyModule(3)

mod.multiply(tf.constant([1, 2, 3]))

In [20]:

In [21]:

save\_path = './saved'

tf.saved\_model.save(mod, save\_path)

reloaded = tf.saved\_model.load(save\_path) reloaded.multiply(tf.constant([1, 2, 3]))

In [22]:

**import matplotlib**

**from matplotlib import** pyplot **as** plt

matplotlib.rcParams['figure.figsize'] = [9, 6]

In [23]:

In [24]:

x = tf.linspace(-2, 2, 201) x = tf.cast(x, tf.float32)

**def** f(x):

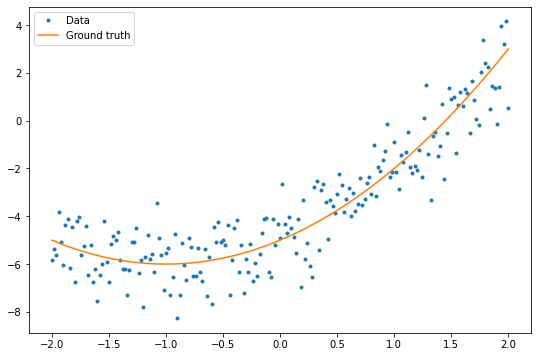
y = x\*\*2 + 2\*x - 5

**return** y

y = f(x) + tf.random.normal(shape=[201])

plt.plot(x.numpy(), y.numpy(), '.', label='Data') plt.plot(x, f(x), label='Ground truth')

plt.legend();



In [25]:

**class Model**(tf.keras.Model): **def** init (self, units):

super(). init ()

self.dense1 = tf.keras.layers.Dense(units=units,

activation=tf.nn.relu,

kernel\_initializer=tf.random.normal, bias\_initializer=tf.random.normal)

self.dense2 = tf.keras.layers.Dense(1)

**def** call(self, x, training=**True**): x = x[:, tf.newaxis]

x = self.dense1(x) x = self.dense2(x)

**return** tf.squeeze(x, axis=1)

In [26]:

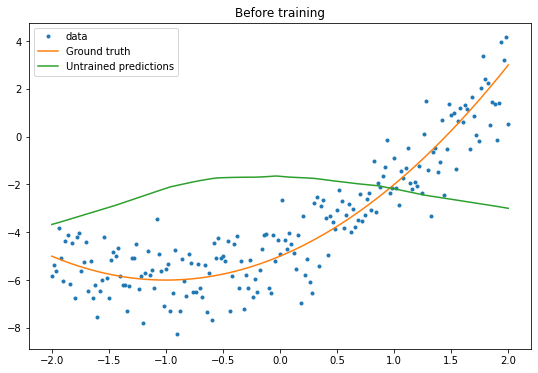
model = Model(64)

In [27]:

plt.plot(x.numpy(), y.numpy(), '.', label='data') plt.plot(x, f(x), label='Ground truth')

plt.plot(x, model(x), label='Untrained predictions') plt.title('Before training')

plt.legend();



In [28]:

variables = model.variables

optimizer = tf.optimizers.SGD(learning\_rate=0.01)

**for** step **in** range(1000):

**with** tf.GradientTape() **as** tape: prediction = model(x)

error = (y-prediction)\*\*2

mean\_error = tf.reduce\_mean(error)

gradient = tape.gradient(mean\_error, variables)

optimizer.apply\_gradients(zip(gradient, variables))

**if** step % 100 == 0:

print(f'Mean squared error: {mean\_error.numpy():0.3f}')

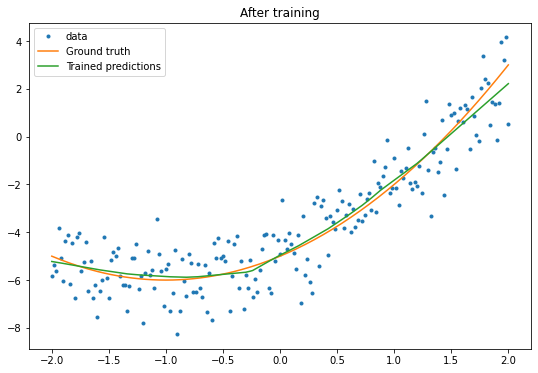
Mean squared error: 10.085 Mean squared error: 1.083 Mean squared error: 1.077 Mean squared error: 1.074 Mean squared error: 1.071 Mean squared error: 1.068 Mean squared error: 1.066 Mean squared error: 1.065 Mean squared error: 1.064 Mean squared error: 1.062

In [29]:

plt.plot(x.numpy(),y.numpy(), '.', label="data") plt.plot(x, f(x), label='Ground truth')

plt.plot(x, model(x), label='Trained predictions') plt.title('After training')

plt.legend();



In [30]

new\_model = Model(64)

**Experiment – 7**

**Business Proposal**

My proposal is for opening a Ice cream parlor . For this the factors that I need to consider for finding the right longitude and latitude for finding the right land to buy for our parlor, we will conduct a survey form having following questions :

* House owners name
* Number of people in the house
* Number of elderly people
* Number of children
* Number of earning people in the house
* Children’s school/college name
* Number of motorized vehicles owned
* Number of Ac’s in the house
* Number of electronic gadgets owned

On the basis of these factors we are going to try to find:

* The working capital of that particular locality
* The number of children in that particular locality
* The number of elderly people in that particular locality

We find the working capital of each house hold on the basis of following features :

* Number of earning people in the house
* Children’s school/college name
* Number of motorized vehicles owned
* Number of Ac’s in the house
* Number of electronic gadgets owned

Then on the basis of this we try to find whether that locality belong to which segment of society rich or middle class. Then after finding this we decide which type of Parlor we want to open .

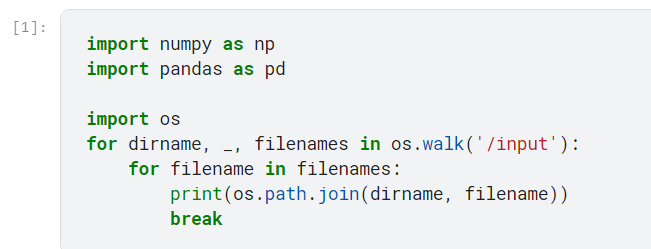
If the society belong to rich class segment than its of no use to buy a land over their because if its from rich class then we won’t have high sales because most of them won’t have time to come to parlor again and again for ice cream , even if they come it will be a weekly visit only and their children also tend to stay at home and that’s no what we want .

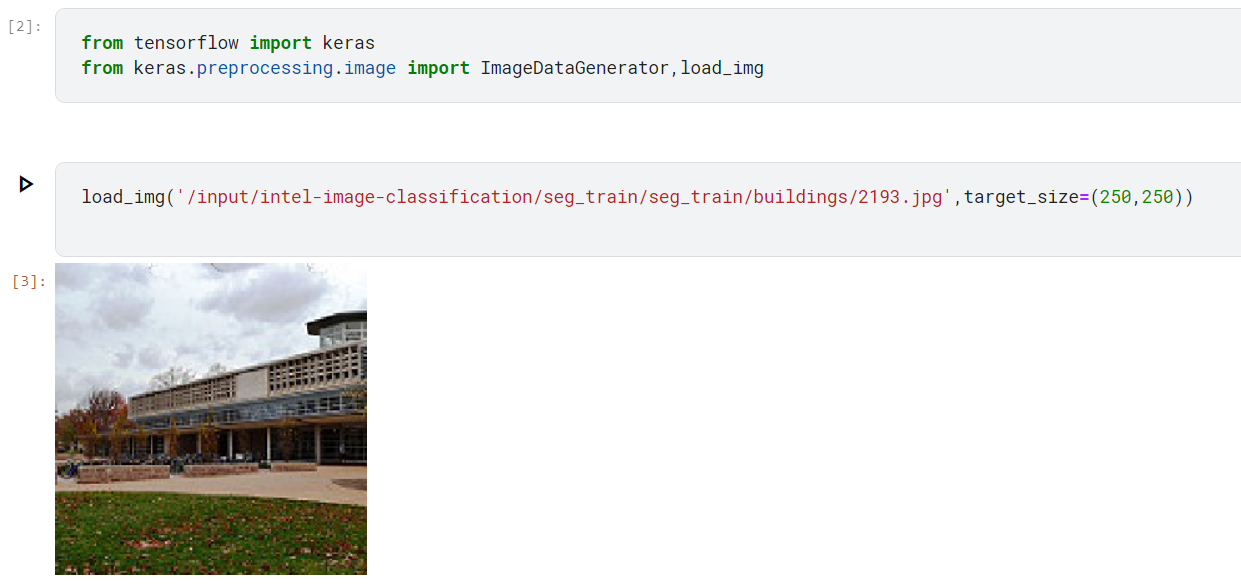
So our target audience over here is Middle class people because we all know that it’s the middle class people who are tend to spend more time with family and they are more tend to eat ice cream everyday with their families in summer as a refreshment and fun.

For more understanding and for getting more idea about that locality , we can ask the company for providing us with the data of early and monthly sales of nearby shops where same companies ice creams are sold , apart from that we can get the same data from different companies for a better comparison .

On the basis of all this we can select the best locality for opening our Ice Cream Parlor and on the basis of this we can buy our desired land in that particular locality/location.

**Experiment – 8**

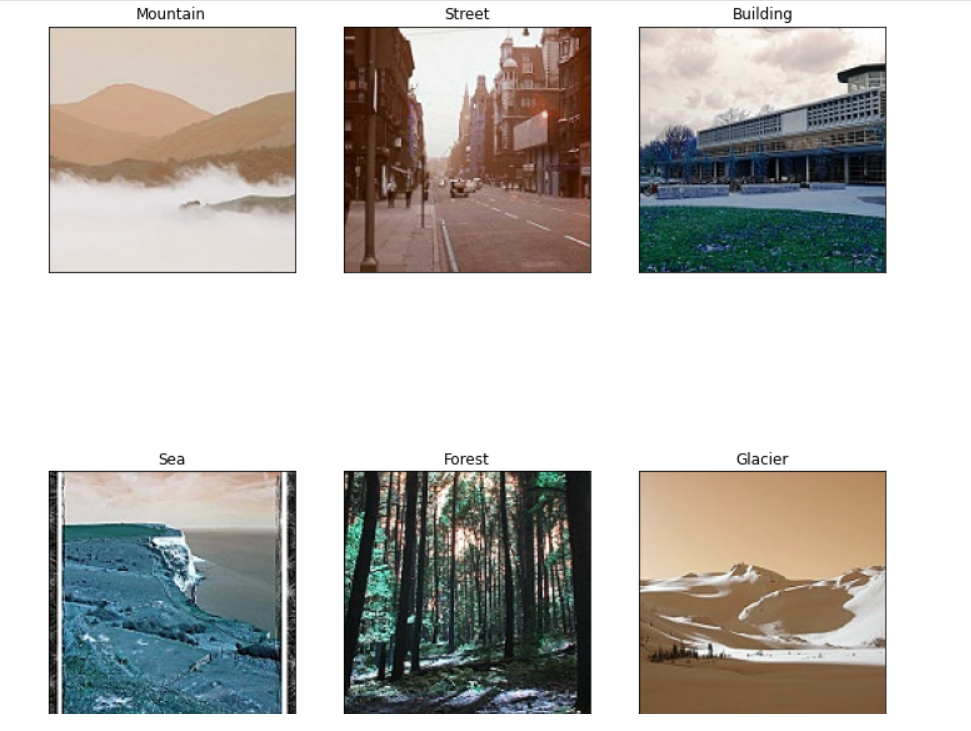
****

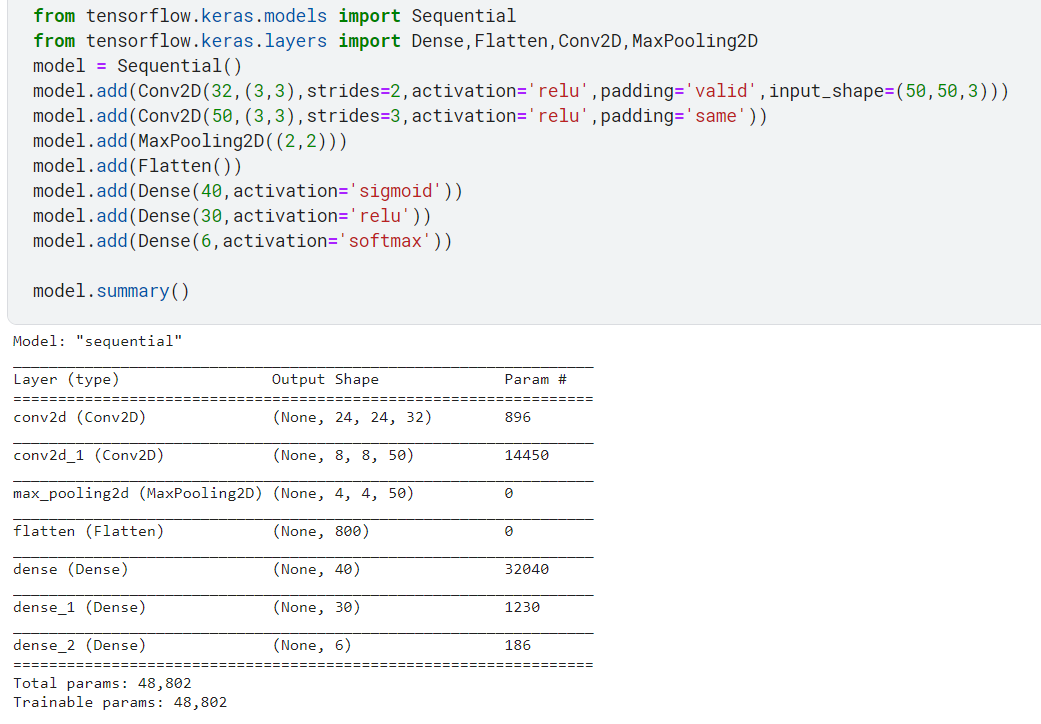
****

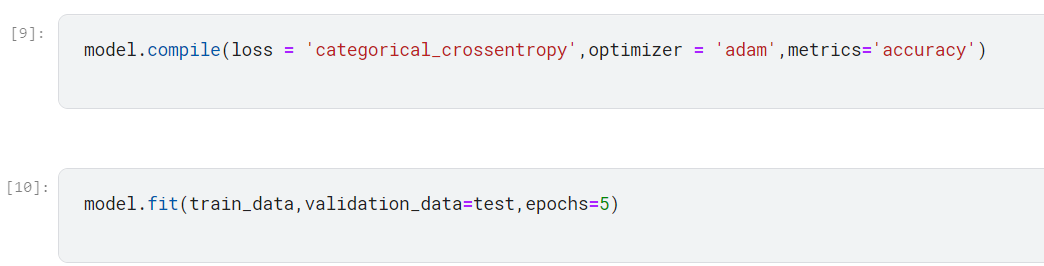
****

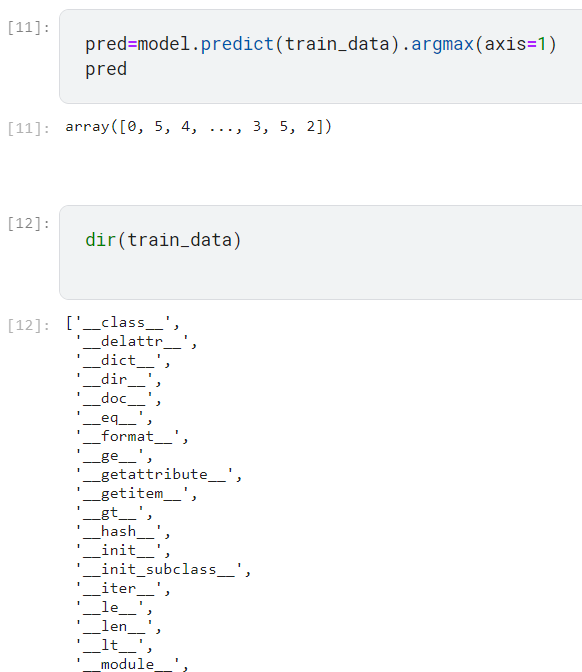
****

****

****

****

****

****

****

****

****