**INDEX**

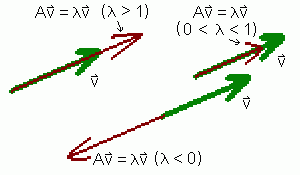
|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.**  **No.** | **Title** | **Date** | **Page No.** |
| 1 | Matrix Multiplication, Eigen Vectors, Eigenvalues Computation using TensorFlow |  |  |
| 2 | Deep Forward Network for XOR |  |  |
| 3a | Classification using DNN |  |  |
| 3b | Binary Classification using MLP |  |  |
| 3c | Convolutional Neural Network |  |  |
| 4 | Predicting the Probability of the class |  |  |
| 5a | CNN for CIFAR10 Images |  |  |
| 5b | Image Classification |  |  |
| 5c | Data Augmentation |  |  |
| 6 | Building RNN using Single Neuron |  |  |
| 7 | NLP Data Handling   1. Study of various Corpus and Corpora 2. Create and use your own corpora (plaintext categorical) |  |  |
| 8 | Text Pre-processing using NLTK   1. Tokenization 2. Lowering the case 3. Stemming 4. Lemmatization 5. Removing stop words |  |  |
| 9 | Conversion of Text to Vectors   1. One-hot Encoding 2. Bag of Words 3. N-grams 4. TF-IDF |  |  |
| 10 | Word Embedding   1. Word2Vec – CBOW & Skip Gram 2. Glove   https://www.analyticsvidhya.com/blog/2021/06/practical-guide-to-word-embedding-system/ |  |  |

**PRACTICAL 1**

**MATRIX MULTIPLICATION, EIGEN VECTORS, EIGENVALUE COMPUTATION USING TENSORFLOW**

Eigenvalues and eigenvectors play a prominent role in the study of ordinary differential equations and in many applications in the physical sciences. Expect to see them come up in a variety of contexts!

**Definitions**



Let AA be an n×nn×n matrix. The number λλ is an **eigenvalue** of AA if there exists a non-zero vector vv such that

Av=λv.Av=λv.

In this case, vector vv is called an **eigenvector** of AA corresponding to λλ.

**CODE:**

import tensorflow as tf

print("Matrix Multiplication Demo")

x=tf.constant([1,2,3,4,5,6],shape=[2,3])

print(x)

y=tf.constant([7,8,9,10,11,12],shape=[3,2])

print(y)

z=tf.matmul(x,y)

print("Product:",z)

e\_matrix\_A=tf.random.uniform([2,2],minval=3,maxval=10,dtype=tf.float32,name="matrixA")

print("Matrix A:\n{}\n\n".format(e\_matrix\_A))

eigen\_values\_A,eigen\_vectors\_A=tf.linalg.eigh(e\_matrix\_A)

print("Eigen Vectors:\n{}\n\nEigen values:\n{}\n".format(eigen\_vectors\_A, eigen\_values\_A))

**OUTPUT:**



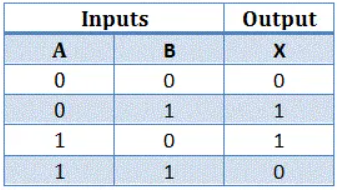
**PRACTICAL 2**

**DEEP FORWARD NETWORK FOR XOR**

**Deep feedforward networks**, also often called **feedforward neural networks**, or **multilayer perceptron’s** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f\*. For example, for a classiﬁer, y = f\*(**x**) maps an input **x** to a category **y**. A feedforward network deﬁnes a mapping **y**= f (**x**; **θ**) and learns the value of the parameters **θ**that result in the best function approximation.

These models are called feedforward because information ﬂows through the function being evaluated from **x**, through the intermediate computations used to deﬁne f, and ﬁnally to the output y. There are no feedback connections in which outputs of the model are fed back into itself.

XOR Truth Table:



**CODE:**

import numpy as np

from keras.layers import Dense

from keras.models import Sequential

model=Sequential()

model.add(Dense(units=2,activation='relu',input\_dim=2))

model.add(Dense(units=1,activation='sigmoid'))

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())

print(model.get\_weights())

X=np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])

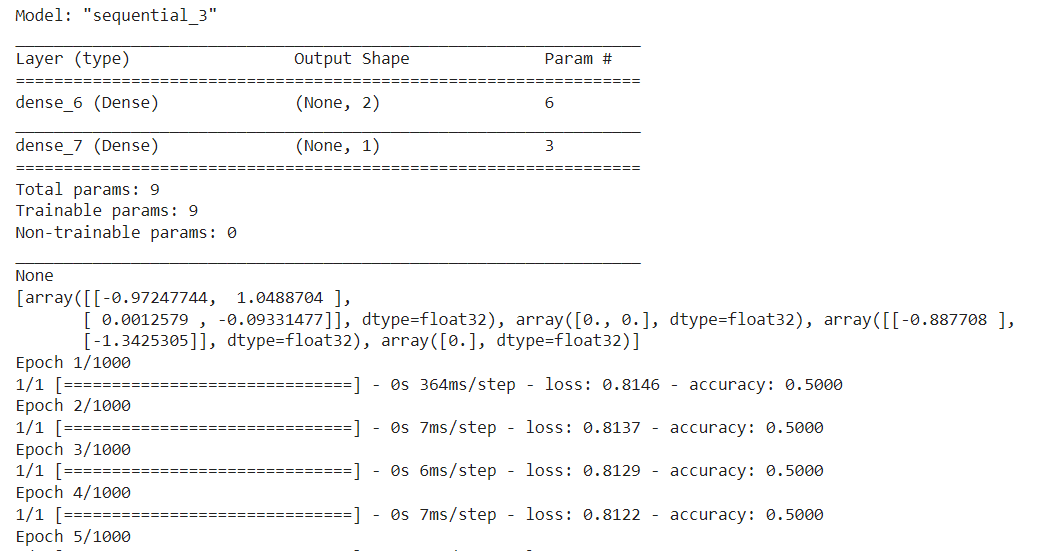
Y=np.array([0.,1.,1.,0.])

model.fit(X,Y,epochs=1000,batch\_size=4)

print(model.get\_weights())

print(model.predict(X,batch\_size=4))

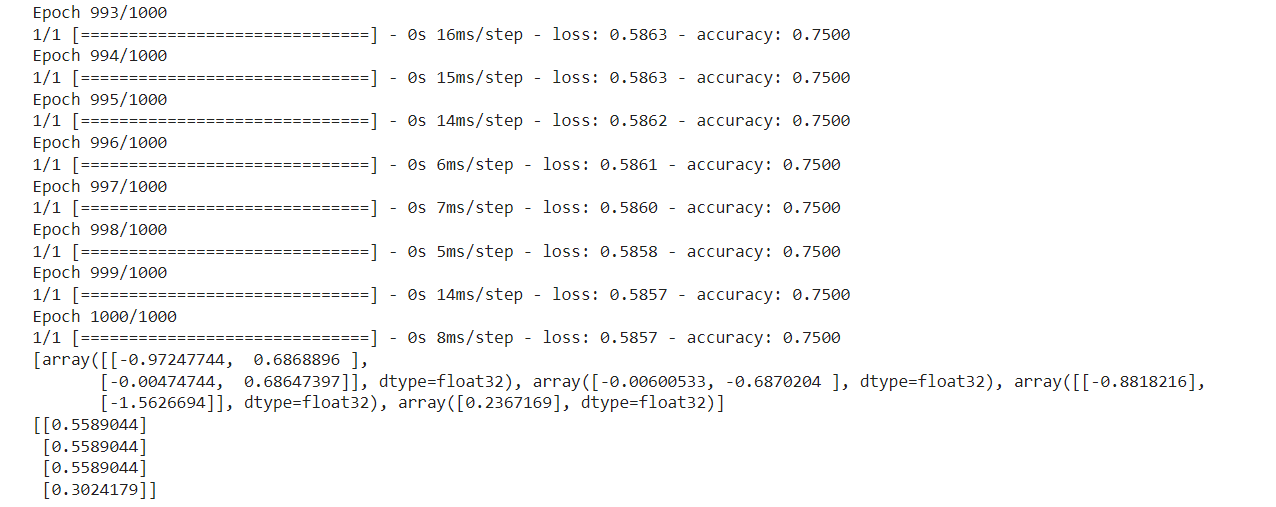
**OUTPUT:**



.

.

.



**PRACTICAL 3A**

**CLASSIFICATION USING DNN**

Classification neural networks used for feature categorization are **very similar to fault-diagnosis networks**, except that they only allow one output response for any input pattern, instead of allowing multiple faults to occur for a given set of operating conditions.

The classification network selects the category based on which output response has the highest output value.

**Problem statement:**

#The given dataset comprises health information about diabetic women patients. We need to create a deep feed forward network that will classify women suffering from diabetes mellitus as 1.

**CODE:**

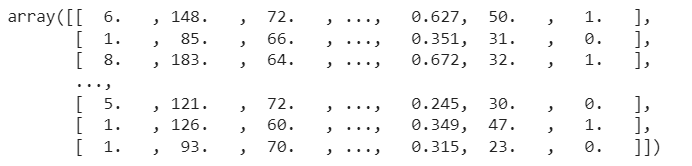
from numpy import loadtxt

from keras.models import Sequential

from keras.layers import Dense

dataset=loadtxt('/content/sample\_data/pima-indians-diabetes.csv',delimiter=',')

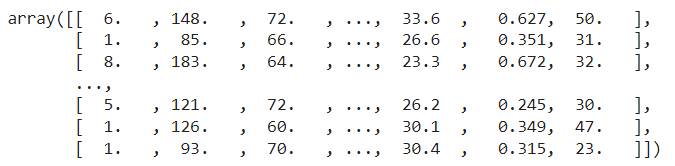
dataset



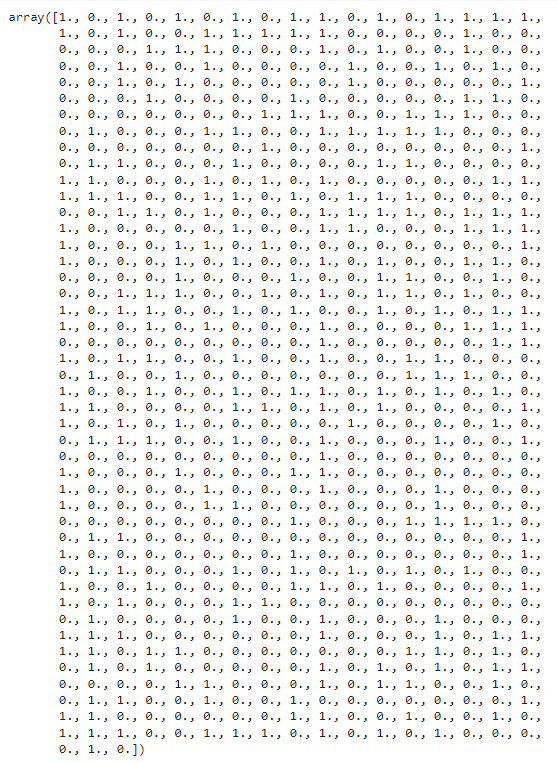
X=dataset[:,0:8]

Y=dataset[:,8]

X



Y



#Creating model

model = Sequential()

model.add(Dense(12, input\_dim=8, activation='relu'))

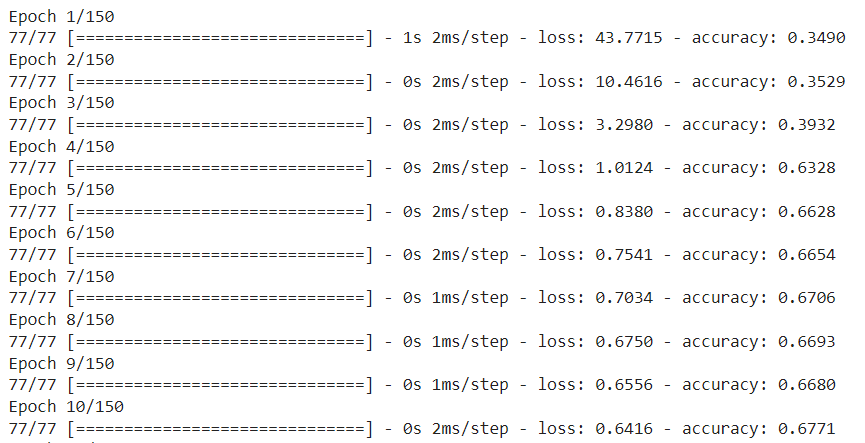
model.add(Dense(8, activation='relu'))

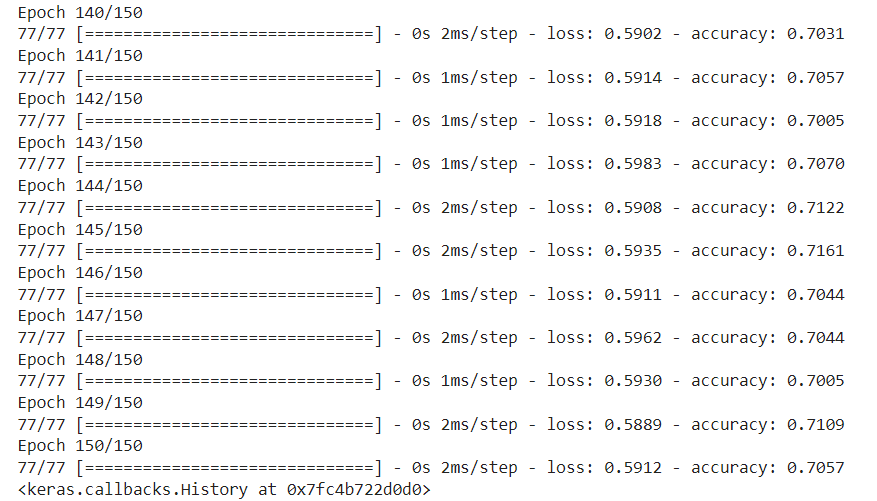
model.add(Dense(1, activation='sigmoid'))

#Compiling and fitting model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(X, Y, epochs=150, batch\_size=10)





model.fit(X, Y, epochs=150, batch\_size=10)

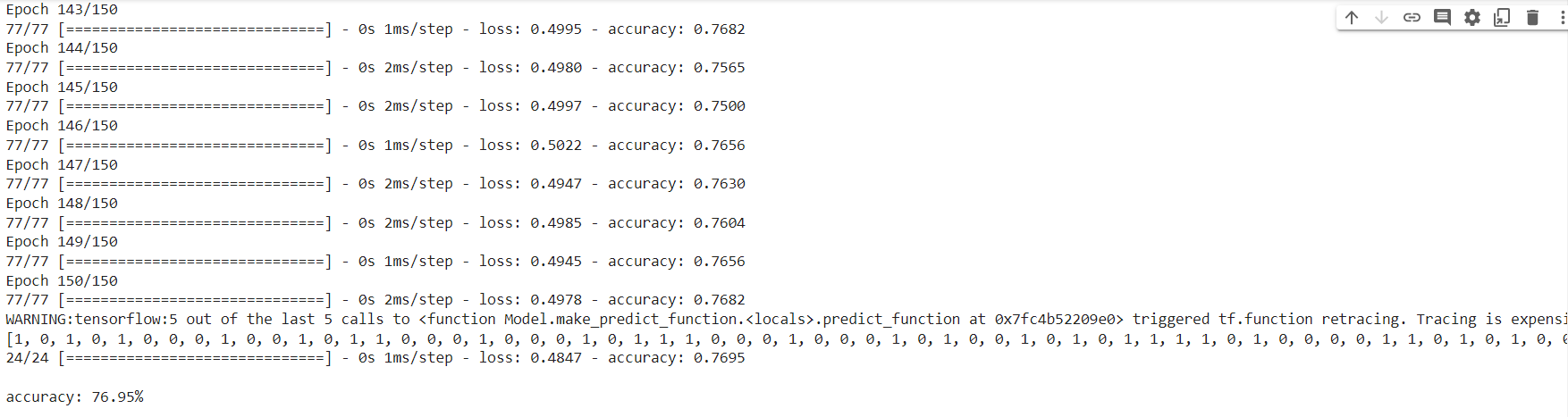
predictions = model.predict(X)

rounded = [round(x[0]) for x in predictions]

print(rounded)

scores = model.evaluate(X, Y)

print("\n%s: %.2f%%" % (model.metrics\_names[1], scores[1]\*100))



**PRACTICAL 3B**

**BINARY CLASSIFICATION USING MLP**

Multilayer Perceptron falls under the category of [feedforward algorithms](https://en.wikipedia.org/wiki/Feedforward_neural_network), because inputs are combined with the initial weights in a weighted sum and subjected to the activation function, just like in the Perceptron. But the difference is that each linear combination is propagated to the next layer.

Each layer is feeding the next one with the result of their computation, their internal representation of the data. This goes all the way through the hidden layers to the output layer.

Binary classification, which looks at an input and predicts which of two possible classes it belongs to. Practical uses include [sentiment analysis](https://www.wintellect.com/binary-classification-sentiment-analysis/), [spam detection](https://www.wintellect.com/binary-classification-spam-filtering/), and [credit-card fraud detection](https://www.wintellect.com/pca-based-anomaly-detection/). Such models are trained with datasets labelled with 1s and 0s representing the two classes, employ popular learning algorithms such as [logistic regression](https://en.wikipedia.org/wiki/Logistic_regression) and [Naïve Bayes](https://en.wikipedia.org/wiki/Naive_Bayes_classifier), and are frequently built with libraries such as Scikit-learn.

**CODE:**

# mlp for binary classification

from pandas import read\_csv

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from tensorflow.keras import Sequential

from tensorflow.keras.layers import Dense

# load the dataset

path = '/content/sample\_data/lonospear.csv'

df = read\_csv(path, header=None)

# split into input and output columns

X, y = df.values[:, :-1], df.values[:, -1]

# ensure all data are floating point values

X = X.astype('float32')

# encode strings to integer

y = LabelEncoder().fit\_transform(y)

# split into train and test datasets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33)

print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape)

# determine the number of input features

n\_features = X\_train.shape[1]

# define model

model = Sequential()

model.add(Dense(10, activation='relu', kernel\_initializer='he\_normal', input\_shape=(n\_features,)))

model.add(Dense(8, activation='relu', kernel\_initializer='he\_normal'))

model.add(Dense(1, activation='sigmoid'))

# compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# fit the model

model.fit(X\_train, y\_train, epochs=150, batch\_size=32, verbose=0)

# evaluate the model

loss, acc = model.evaluate(X\_test, y\_test, verbose=0)

print('Test Accuracy: %.3f' % acc)

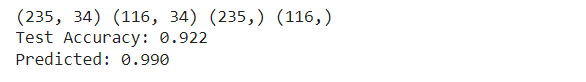
# make a prediction

row = [1,0,0.99539,-0.05889,0.85243,0.02306,0.83398,-0.37708,1,0.03760,0.85243,-0.17755,0.59755,-0.44945,0.60536,-0.38223,0.84356,-0.38542,0.58212,-0.32192,0.56971,-0.29674,0.36946,-0.47357,0.56811,-0.51171,0.41078,-0.46168,0.21266,-0.34090,0.42267,-0.54487,0.18641,-0.45300]

yhat = model.predict([row])

print('Predicted: %.3f' % yhat)

**OUTPUT:**



**PRACTICAL 4A**

**FEED FORWARD NN**

**Feedforward neural networks** are [artificial neural networks](https://brilliant.org/wiki/artificial-neural-network/) where the connections between units do not form a [cycle](https://brilliant.org/wiki/graphs/##graphs-basic). Feedforward neural networks were the first type of artificial neural network invented and are simpler than their counterpart, [recurrent neural networks](https://brilliant.org/wiki/recurrent-neural-network/). They are called feedforward because information only travels forward in the network (no loops), first through the input nodes, then through the [hidden nodes](https://brilliant.org/wiki/artificial-neural-network/#putting-it-all-together) (if present), and finally through the output nodes.

Feedforward neural networks are primarily used for [supervised learning](https://brilliant.org/wiki/supervised-learning/) in cases where the data to be learned is neither sequential nor time-dependent. That is, feedforward neural networks compute a function ff on fixed size input xx such that f(x) \approx. yf(x)≈y for training pairs (x, y) (x, y). On the other hand, recurrent neural networks learn sequential data, computing gg on variable length input X\_k = \{x\_1, \dots, x\_k\}Xk​={x1​,…,xk​} such that g(X\_k) \approxy\_kg(Xk​)≈yk​ for training pairs (X\_n, Y\_n)(Xn​,Yn​) for all 1 \le k \le n1≤k≤n.



**CODE:**

from keras.models import Sequential

from keras.layers import Dense

from sklearn.datasets import make\_blobs

from sklearn.preprocessing import MinMaxScaler

import numpy as np

X,Y=make\_blobs(n\_samples=100,centers=2,n\_features=2,random\_state=1)

scalar=MinMaxScaler()

scalar.fit(X)

X=scalar.transform(X)

model=Sequential()

model.add(Dense(4,input\_dim=2,activation='relu'))

model.add(Dense(4,activation='relu'))

model.add(Dense(1,activation='sigmoid'))

model.compile(loss='binary\_crossentropy',optimizer='adam')

model.fit(X,Y,epochs=500)



Xnew,Yreal=make\_blobs(n\_samples=5,centers=2,n\_features=2,random\_state=1)

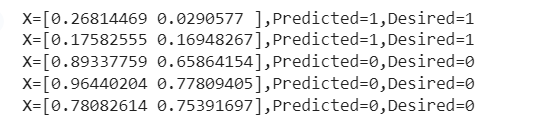
Xnew=scalar.transform(Xnew)

Ynew=model.predict(Xnew)

Ynew=np.round(Ynew).astype(int)

for i in range(len(Xnew)):

  print("X=%s,Predicted=%d,Desired=%s"%(Xnew[i],Ynew[i],Yreal[i]))



**PRACTICAL 4B**

**PREDICTING THE PROBABILITY OF THE CLASS**

**CODE:**

from keras.models import Sequential

from keras.layers import Dense

from sklearn.datasets import make\_blobs

from sklearn.preprocessing import MinMaxScaler

X,Y=make\_blobs(n\_samples=100,centers=2,n\_features=2,random\_state=1)

scalar=MinMaxScaler()

scalar.fit(X)

X=scalar.transform(X)

model=Sequential()

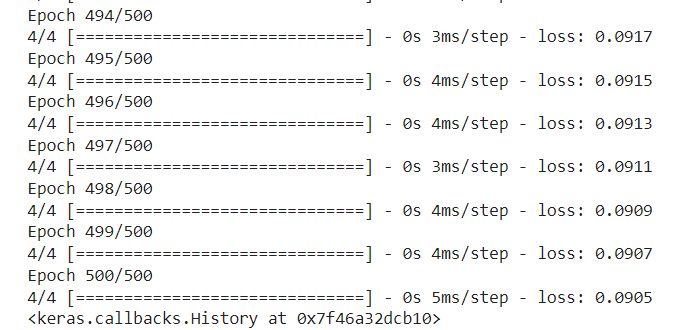
model.add(Dense(4,input\_dim=2,activation='relu'))

model.add(Dense(4,activation='relu'))

model.add(Dense(1,activation='sigmoid'))

model.compile(loss='binary\_crossentropy',optimizer='adam')

model.fit(X,Y,epochs=500)



Xnew,Yreal=make\_blobs(n\_samples=3,centers=2,n\_features=2,random\_state=1)

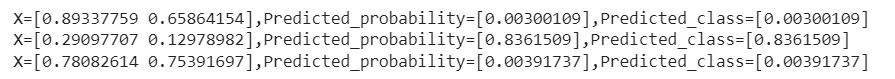
Xnew=scalar.transform(Xnew)

Yclass=model.predict(Xnew)

Ynew=model.predict(Xnew)

for i in range(len(Xnew)):

  print("X=%s,Predicted\_probability=%s,Predicted\_class=%s"%(Xnew[i],Ynew[i],Yclass[i]))



**PRACTICAL 5A**

**CNN FOR CIFAR10 IMAGES**

A [**neural network**](https://developers.google.com/machine-learning/glossary/#neural_network) in which at least one layer is a [**convolutional layer**](https://developers.google.com/machine-learning/glossary/#convolutional_layer). A typical convolutional neural network consists of some combination of the following layers:

* [**convolutional layers**](https://developers.google.com/machine-learning/glossary/#convolutional_layer)
* [**pooling layers**](https://developers.google.com/machine-learning/glossary/#pooling)
* [**dense layers**](https://developers.google.com/machine-learning/glossary/#dense_layer)

Convolutional neural networks have had great success in certain kinds of problems, such as image recognition.

**CODE:**

import tensorflow as tf

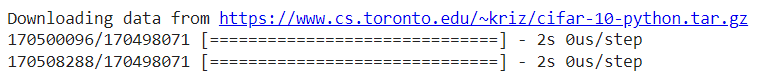
from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt

(train\_images, train\_labels), (test\_images, test\_labels) = datasets.cifar10.load\_data()

# Normalize pixel values to be between 0 and 1

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0



class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',

               'dog', 'frog', 'horse', 'ship', 'truck']

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

for i in range(25):

    plt.subplot(5,5,i+1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

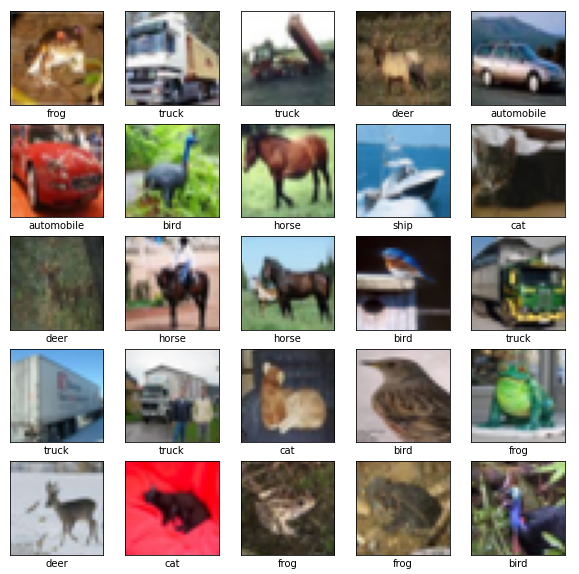
    plt.imshow(train\_images[i])

    # The CIFAR labels happen to be arrays,

    # which is why you need the extra index

    plt.xlabel(class\_names[train\_labels[i][0]])

plt.show()



model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)))

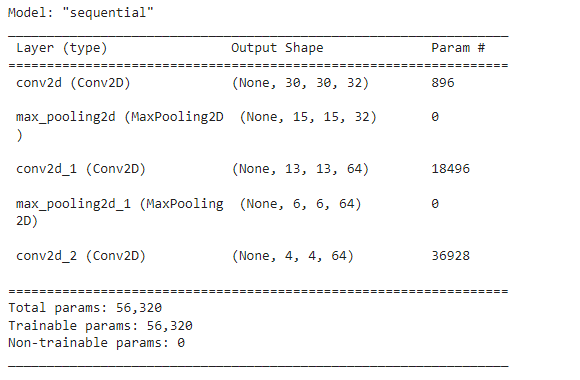
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.summary()

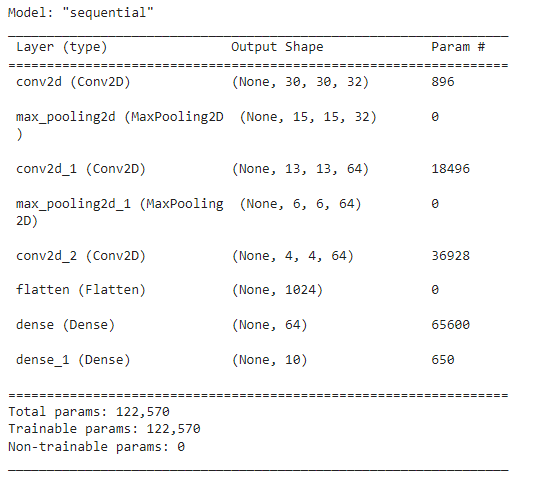


model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

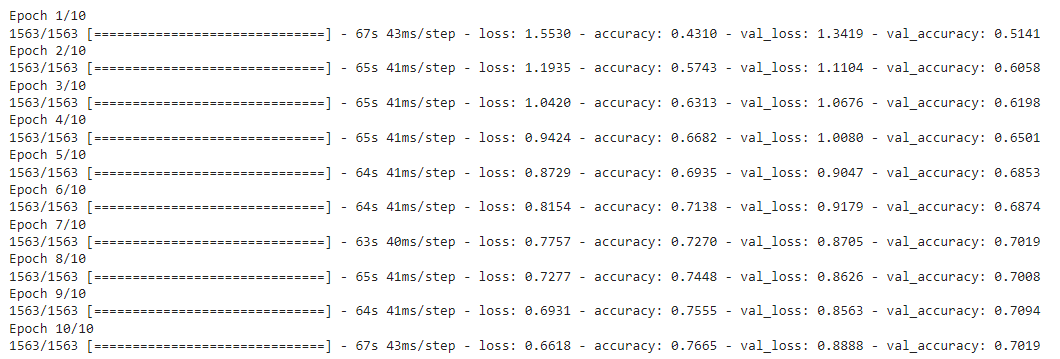
model.add(layers.Dense(10))

model.summary()

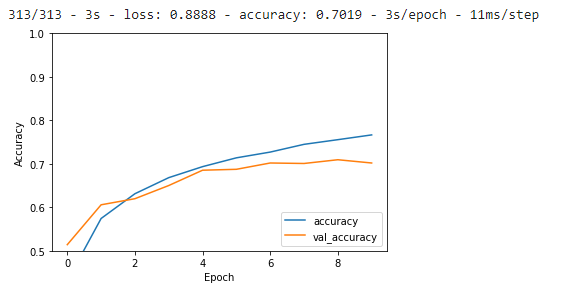


**this step will take time to execute –5 to 8 minutes**

model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),  
              metrics=['accuracy'])  
  
history = model.fit(train\_images, train\_labels, epochs=10,   
                    validation\_data=(test\_images, test\_labels))



plt.plot(history.history['accuracy'], label='accuracy')  
plt.plot(history.history['val\_accuracy'], label ='val\_accuracy')  
plt.xlabel('Epoch')  
plt.ylabel('Accuracy')  
plt.ylim([0.5,1])  
plt.legend(loc='lower right')  
  
test\_loss,test\_acc=model.evaluate(test\_images,  test\_labels, verbose=2)



print(test\_acc)



**PRACTICAL 5B**

**IMAGE CLASSIFICATION**

We will classify images of flowers. It creates an image classifier using a tf.keras.Sequential model, and loads data using tf.keras.utils.image\_dataset\_from\_directory. You will gain practical experience with the following concepts:

* Efficiently loading a dataset off disk.
* Identifying overfitting and applying techniques to mitigate it, including data augmentation and dropout.

1. Examine and understand data
2. Build an input pipeline
3. Build the model
4. Train the model
5. Test the model
6. Improve the model and repeat the process

**CODE:**

import matplotlib.pyplot as plt

import numpy as np

import os

import PIL

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

from tensorflow.keras.models import Sequential

This tutorial uses a dataset of about 3,700 photos of flowers. The dataset contains five sub-directories, one per class:

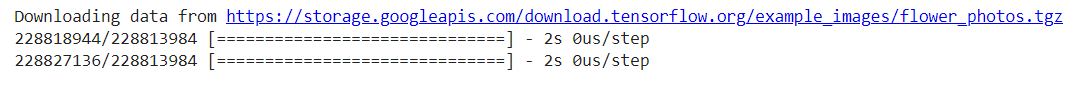
flower\_photo/  
  daisy/  
  dandelion/  
  roses/  
  sunflowers/  
  tulips/

import pathlib

dataset\_url = "https://storage.googleapis.com/download.tensorflow.org/example\_images/flower\_photos.tgz"

data\_dir = tf.keras.utils.get\_file('flower\_photos', origin=dataset\_url, untar=True)

data\_dir = pathlib.Path(data\_dir)



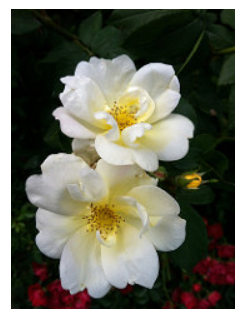
image\_count = len(list(data\_dir.glob('\*/\*.jpg')))

print(image\_count)



roses = list(data\_dir.glob('roses/\*'))

PIL.Image.open(str(roses[0]))



PIL.Image.open(str(roses[1]))



tulips = list(data\_dir.glob('tulips/\*'))



PIL.Image.open(str(tulips[0]))



batch\_size = 32

img\_height = 180

img\_width = 180

train\_ds = tf.keras.utils.image\_dataset\_from\_directory(

  data\_dir,

  validation\_split=0.2,

  subset="training",

  seed=123,

  image\_size=(img\_height, img\_width),

  batch\_size=batch\_size)



val\_ds = tf.keras.utils.image\_dataset\_from\_directory(

  data\_dir,

  validation\_split=0.2,

  subset="validation",

  seed=123,

  image\_size=(img\_height, img\_width),

  batch\_size=batch\_size)

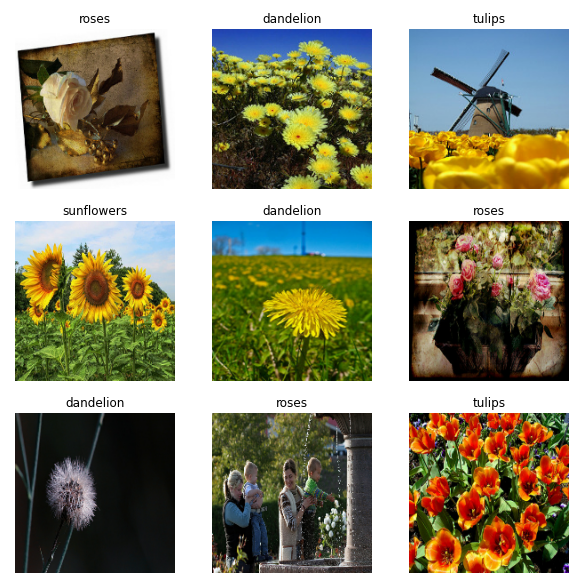


class\_names = train\_ds.class\_names

print(class\_names)



importmatplotlib.pyplotasplt  
  
plt.figure(figsize=(10,10))  
for images, labels intrain\_ds.take(1):  
  foriin range(9):  
    ax=plt.subplot(3,3,i+1)  
    plt.imshow(images[i].numpy().astype("uint8"))  
    plt.title(class\_names[labels[i]])  
    plt.axis("off")



forimage\_batch,labels\_batchintrain\_ds:  
  print(image\_batch.shape)  
  print(labels\_batch.shape)  
  break



AUTOTUNE = tf.data.AUTOTUNE

train\_ds = train\_ds.cache().shuffle(1000).prefetch(buffer\_size=AUTOTUNE)

val\_ds = val\_ds.cache().prefetch(buffer\_size=AUTOTUNE)

normalization\_layer = layers.Rescaling(1./255)

normalized\_ds = train\_ds.map(lambda x, y: (normalization\_layer(x), y))

image\_batch, labels\_batch = next(iter(normalized\_ds))

first\_image = image\_batch[0]

# Notice the pixel values are now in `[0,1]`.

print(np.min(first\_image), np.max(first\_image))



num\_classes = 5

model = Sequential([

  layers.Rescaling(1./255, input\_shape=(img\_height, img\_width, 3)),

  layers.Conv2D(16, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Conv2D(32, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Conv2D(64, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Flatten(),

  layers.Dense(128, activation='relu'),

  layers.Dense(num\_classes)

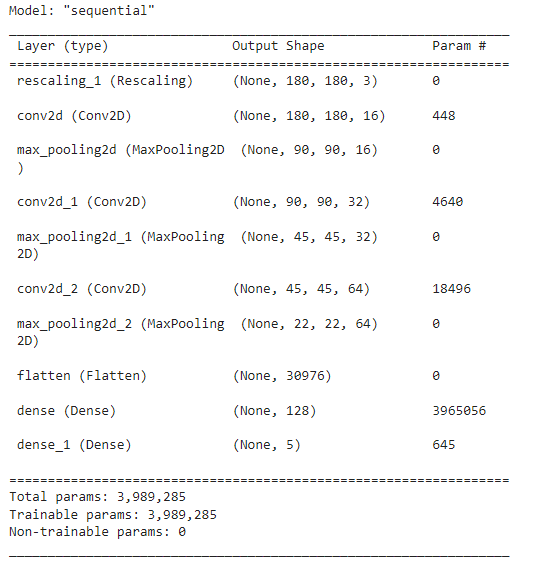
])

model.compile(optimizer='adam',

              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

              metrics=['accuracy'])

model.summary()



epochs=10

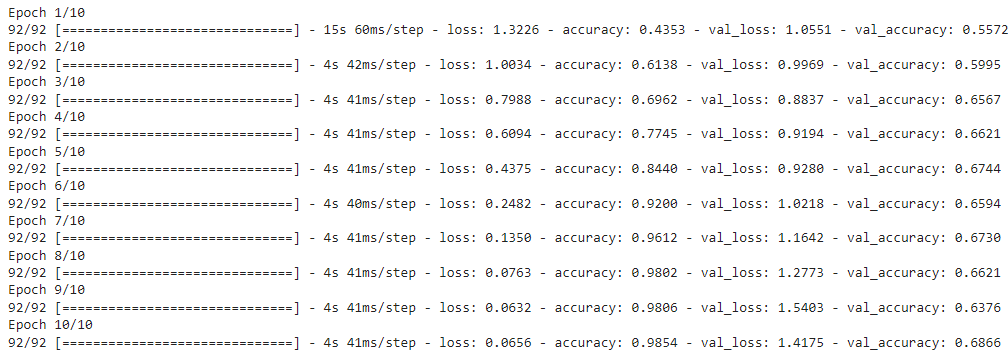
history = model.fit(

  train\_ds,

  validation\_data=val\_ds,

  epochs=epochs

)



acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs\_range = range(epochs)

plt.figure(figsize=(8, 8))

plt.subplot(1, 2, 1)

plt.plot(epochs\_range, acc, label='Training Accuracy')

plt.plot(epochs\_range, val\_acc, label='Validation Accuracy')

plt.legend(loc='lower right')

plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)

plt.plot(epochs\_range, loss, label='Training Loss')

plt.plot(epochs\_range, val\_loss, label='Validation Loss')

plt.legend(loc='upper right')

plt.title('Training and Validation Loss')

plt.show()



data\_augmentation = keras.Sequential(

  [

    layers.RandomFlip("horizontal",

                      input\_shape=(img\_height,

                                  img\_width,

                                  3)),

    layers.RandomRotation(0.1),

    layers.RandomZoom(0.1),

  ]

)

plt.figure(figsize=(10, 10))

for images, \_ in train\_ds.take(1):

  for i in range(9):

    augmented\_images = data\_augmentation(images)

    ax = plt.subplot(3, 3, i + 1)

    plt.imshow(augmented\_images[0].numpy().astype("uint8"))

    plt.axis("off")



model = Sequential([

  data\_augmentation,

  layers.Rescaling(1./255),

  layers.Conv2D(16, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Conv2D(32, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Conv2D(64, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Dropout(0.2),

  layers.Flatten(),

  layers.Dense(128, activation='relu'),

  layers.Dense(num\_classes)

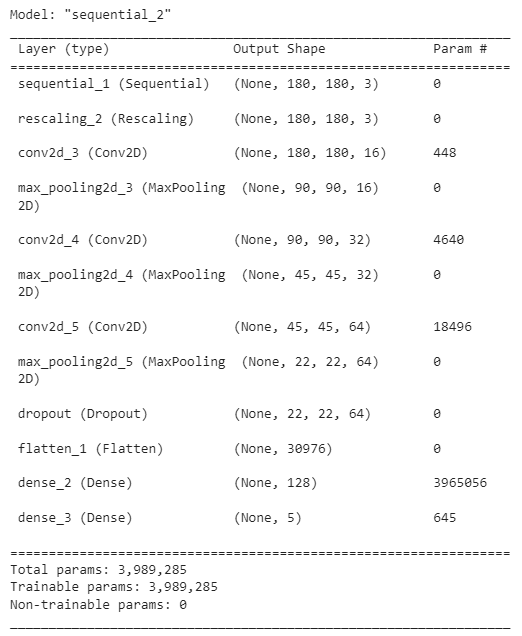
])

model.compile(optimizer='adam',

              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

              metrics=['accuracy'])

model.summary()



epochs = 15

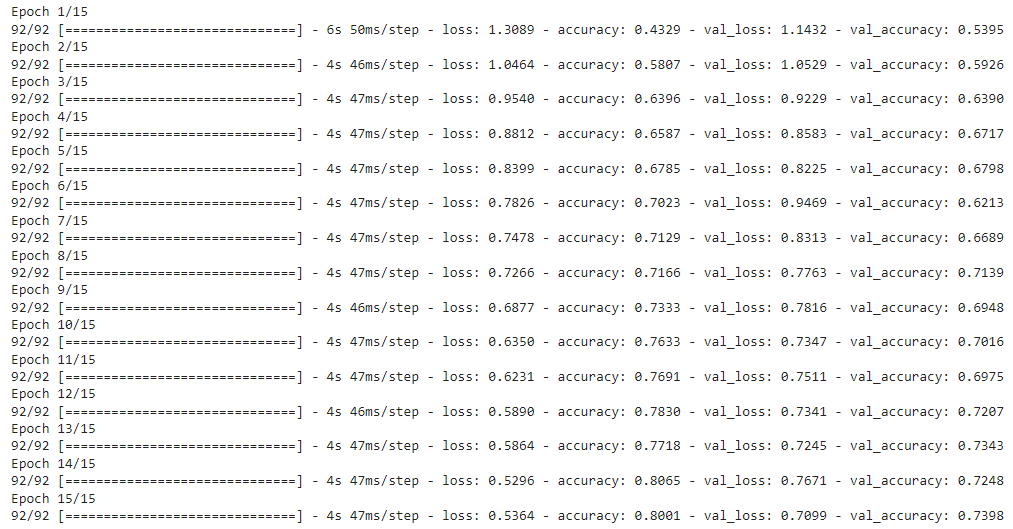
history = model.fit(

  train\_ds,

  validation\_data=val\_ds,

  epochs=epochs

)



acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs\_range = range(epochs)

plt.figure(figsize=(8, 8))

plt.subplot(1, 2, 1)

plt.plot(epochs\_range, acc, label='Training Accuracy')

plt.plot(epochs\_range, val\_acc, label='Validation Accuracy')

plt.legend(loc='lower right')

plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)

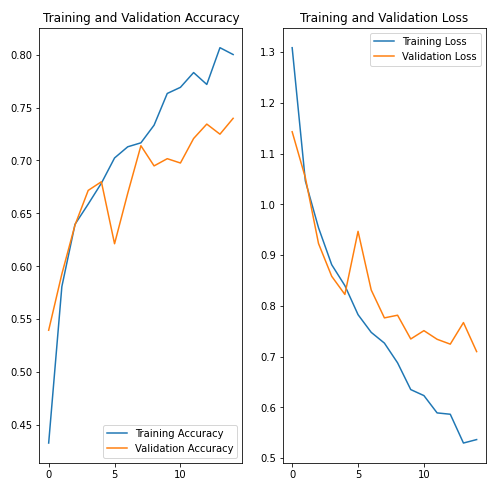
plt.plot(epochs\_range, loss, label='Training Loss')

plt.plot(epochs\_range, val\_loss, label='Validation Loss')

plt.legend(loc='upper right')

plt.title('Training and Validation Loss')

plt.show()



sunflower\_url = "https://storage.googleapis.com/download.tensorflow.org/example\_images/592px-Red\_sunflower.jpg"

sunflower\_path = tf.keras.utils.get\_file('Red\_sunflower', origin=sunflower\_url)

img = tf.keras.utils.load\_img(

    sunflower\_path, target\_size=(img\_height, img\_width)

)

img\_array = tf.keras.utils.img\_to\_array(img)

img\_array = tf.expand\_dims(img\_array, 0) # Create a batch

predictions = model.predict(img\_array)

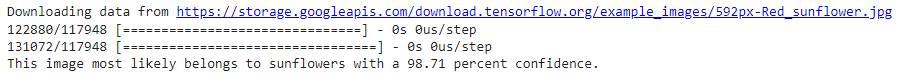
score = tf.nn.softmax(predictions[0])

print(

    "This image most likely belongs to {} with a {:.2f} percent confidence."

    .format(class\_names[np.argmax(score)], 100 \* np.max(score))

)



**PRACTICAL 5C**

**DATA AUGMENTATION**

Data augmentation is a strategy that enables practitioners to significantly increase the diversity of data available for training models, without actually collecting new data. Data augmentation techniques such as cropping, padding, and horizontal flipping are commonly used to train large neural networks.

**CODE:**

import matplotlib.pyplot as plt

import numpy as np

import tensorflow as tf

import tensorflow\_datasets as tfds

from tensorflow.keras import layers

(train\_ds, val\_ds, test\_ds), metadata = tfds.load(

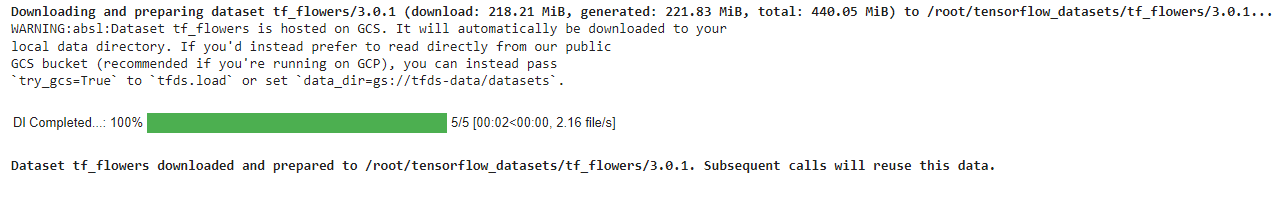
    'tf\_flowers',

    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],

    with\_info=True,

    as\_supervised=True,

)



num\_classes = metadata.features['label'].num\_classes

print(num\_classes)

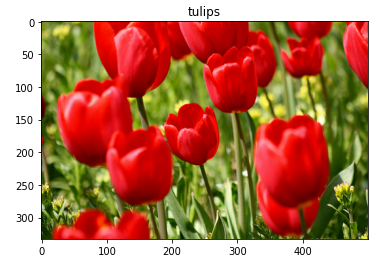


get\_label\_name = metadata.features['label'].int2str

image, label = next(iter(train\_ds))

\_ = plt.imshow(image)

\_ = plt.title(get\_label\_name(label))



IMG\_SIZE = 180

resize\_and\_rescale = tf.keras.Sequential([

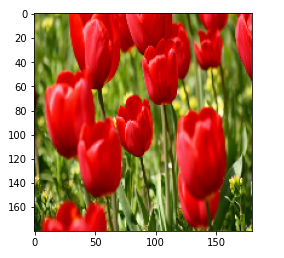
  layers.Resizing(IMG\_SIZE, IMG\_SIZE),

  layers.Rescaling(1./255)

])

result = resize\_and\_rescale(image)

\_ = plt.imshow(result)



print("Min and max pixel values:", result.numpy().min(), result.numpy().max())



data\_augmentation = tf.keras.Sequential([

  layers.RandomFlip("horizontal\_and\_vertical"),

  layers.RandomRotation(0.2),

])

# Add the image to a batch.

image = tf.expand\_dims(image, 0)

plt.figure(figsize=(10, 10))

for i in range(9):

  augmented\_image = data\_augmentation(image)

  ax = plt.subplot(3, 3, i + 1)

  plt.imshow(augmented\_image[0])

  plt.axis("off")



model = tf.keras.Sequential([

  # Add the preprocessing layers you created earlier.

  resize\_and\_rescale,

  data\_augmentation,

  layers.Conv2D(16, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  # Rest of your model.

])

aug\_ds = train\_ds.map(

  lambda x, y: (resize\_and\_rescale(x, training=True), y))

batch\_size = 32

AUTOTUNE = tf.data.AUTOTUNE

def prepare(ds, shuffle=False, augment=False):

  # Resize and rescale all datasets.

  ds = ds.map(lambda x, y: (resize\_and\_rescale(x), y),

              num\_parallel\_calls=AUTOTUNE)

  if shuffle:

    ds = ds.shuffle(1000)

  # Batch all datasets.

  ds = ds.batch(batch\_size)

  # Use data augmentation only on the training set.

  if augment:

    ds = ds.map(lambda x, y: (data\_augmentation(x, training=True), y),

                num\_parallel\_calls=AUTOTUNE)

  # Use buffered prefetching on all datasets.

  return ds.prefetch(buffer\_size=AUTOTUNE)

train\_ds = prepare(train\_ds, shuffle=True, augment=True)

val\_ds = prepare(val\_ds)

test\_ds = prepare(test\_ds)

model = tf.keras.Sequential([

  layers.Conv2D(16, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Conv2D(32, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Conv2D(64, 3, padding='same', activation='relu'),

  layers.MaxPooling2D(),

  layers.Flatten(),

  layers.Dense(128, activation='relu'),

  layers.Dense(num\_classes)

])

model.compile(optimizer='adam',

              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

              metrics=['accuracy'])

epochs=5

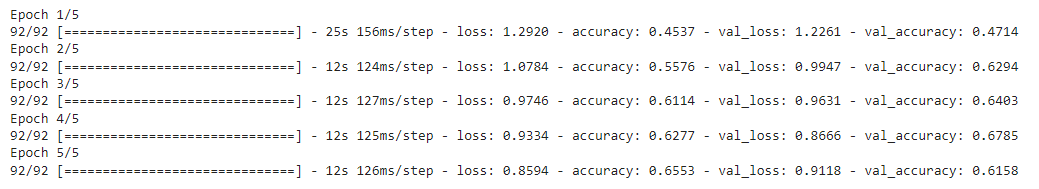
history = model.fit(

  train\_ds,

  validation\_data=val\_ds,

  epochs=epochs

)



loss, acc = model.evaluate(test\_ds)

print("Accuracy", acc)



def random\_invert\_img(x, p=0.5):

  if  tf.random.uniform([]) < p:

    x = (255-x)

  else:

    x

  return x

def random\_invert(factor=0.5):

  return layers.Lambda(lambda x: random\_invert\_img(x, factor))

random\_invert = random\_invert()

plt.figure(figsize=(10, 10))

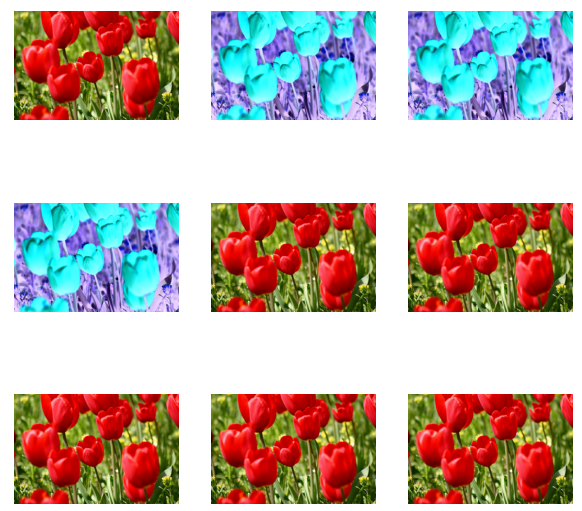
for i in range(9):

  augmented\_image = random\_invert(image)

  ax = plt.subplot(3, 3, i + 1)

  plt.imshow(augmented\_image[0].numpy().astype("uint8"))

  plt.axis("off")



class RandomInvert(layers.Layer):

  def \_\_init\_\_(self, factor=0.5, \*\*kwargs):

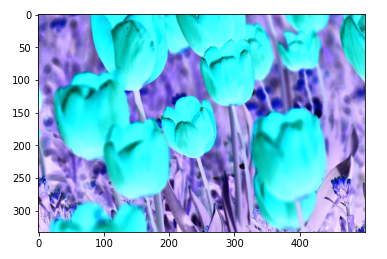
    super().\_\_init\_\_(\*\*kwargs)

    self.factor = factor

  def call(self, x):

    return random\_invert\_img(x)

\_ = plt.imshow(RandomInvert()(image)[0])



(train\_ds, val\_ds, test\_ds), metadata = tfds.load(

    'tf\_flowers',

    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],

    with\_info=True,

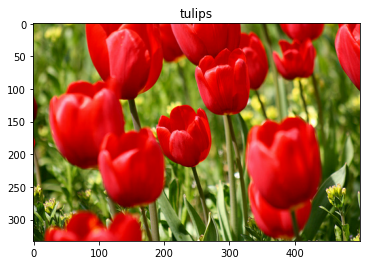
    as\_supervised=True,

)

image, label = next(iter(train\_ds))

\_ = plt.imshow(image)

\_ = plt.title(get\_label\_name(label))



def visualize(original, augmented):

  fig = plt.figure()

  plt.subplot(1,2,1)

  plt.title('Original image')

  plt.imshow(original)

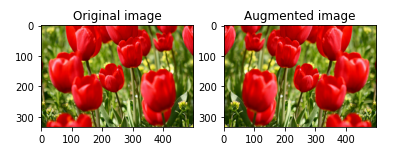
  plt.subplot(1,2,2)

  plt.title('Augmented image')

  plt.imshow(augmented)

flipped = tf.image.flip\_left\_right(image)

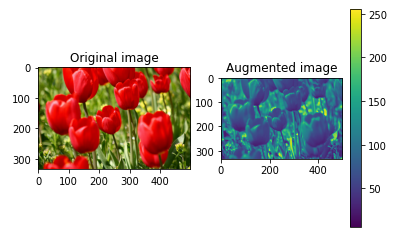
visualize(image, flipped)



grayscaled = tf.image.rgb\_to\_grayscale(image)

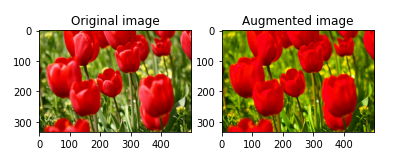
visualize(image, tf.squeeze(grayscaled))

\_ = plt.colorbar()



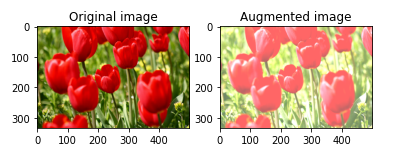
saturated = tf.image.adjust\_saturation(image, 3)

visualize(image, saturated)



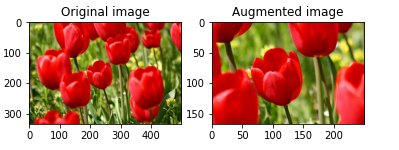
bright = tf.image.adjust\_brightness(image, 0.4)

visualize(image, bright)



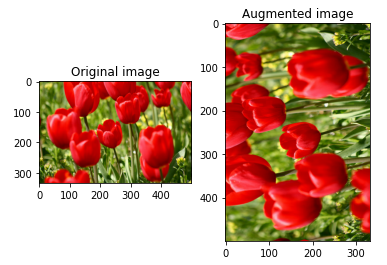
cropped = tf.image.central\_crop(image, central\_fraction=0.5)

visualize(image,cropped)



rotated = tf.image.rot90(image)

visualize(image, rotated)



**PRACTICAL 6**

**BUILDING RNN USING SINGLE NEURON**

Recurrent neural networks (RNN) are a class of neural networks that is powerful for modeling sequence data such as time series or natural language.

Schematically, a RNN layer uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about the timesteps it has seen so far.

The Keras RNN API is designed with a focus on:

* **Ease of use**: the built-in keras.layers.RNN, keras. layers.LSTM, keras.layers.GRU layers enable you to quickly build recurrent models without having to make difficult configuration choices.

**CODE**:

import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential()

# Add an Embedding layer expecting input vocab of size 1000, and

# output embedding dimension of size 64.

model.add(layers.Embedding(input\_dim=1000, output\_dim=64))

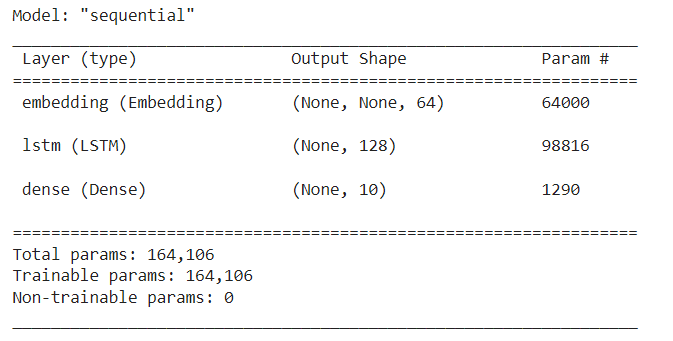
# Add a LSTM layer with 128 internal units.

model.add(layers.LSTM(128))

# Add a Dense layer with 10 units.

model.add(layers.Dense(10))

model.summary()



model = keras.Sequential()

model.add(layers.Embedding(input\_dim=1000, output\_dim=64))

# The output of GRU will be a 3D tensor of shape (batch\_size, timesteps, 256)

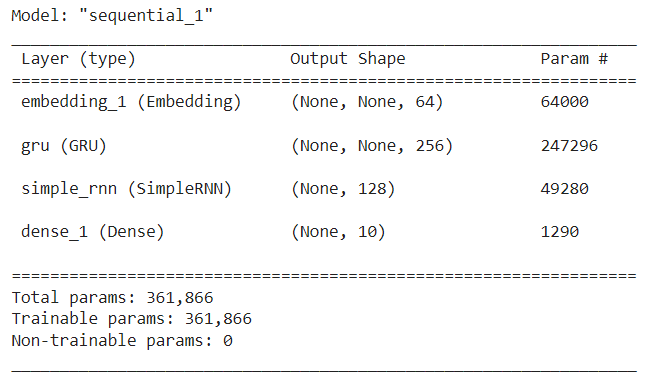
model.add(layers.GRU(256, return\_sequences=True))

# The output of SimpleRNN will be a 2D tensor of shape (batch\_size, 128)

model.add(layers.SimpleRNN(128))

model.add(layers.Dense(10))

model.summary()



encoder\_vocab = 1000

decoder\_vocab = 2000

encoder\_input = layers.Input(shape=(None,))

encoder\_embedded = layers.Embedding(input\_dim=encoder\_vocab, output\_dim=64)(

    encoder\_input

)

# Return states in addition to output

output, state\_h, state\_c = layers.LSTM(64, return\_state=True, name="encoder")(

    encoder\_embedded

)

encoder\_state = [state\_h, state\_c]

decoder\_input = layers.Input(shape=(None,))

decoder\_embedded = layers.Embedding(input\_dim=decoder\_vocab, output\_dim=64)(

    decoder\_input

)

# Pass the 2 states to a new LSTM layer, as initial state

decoder\_output = layers.LSTM(64, name="decoder")(

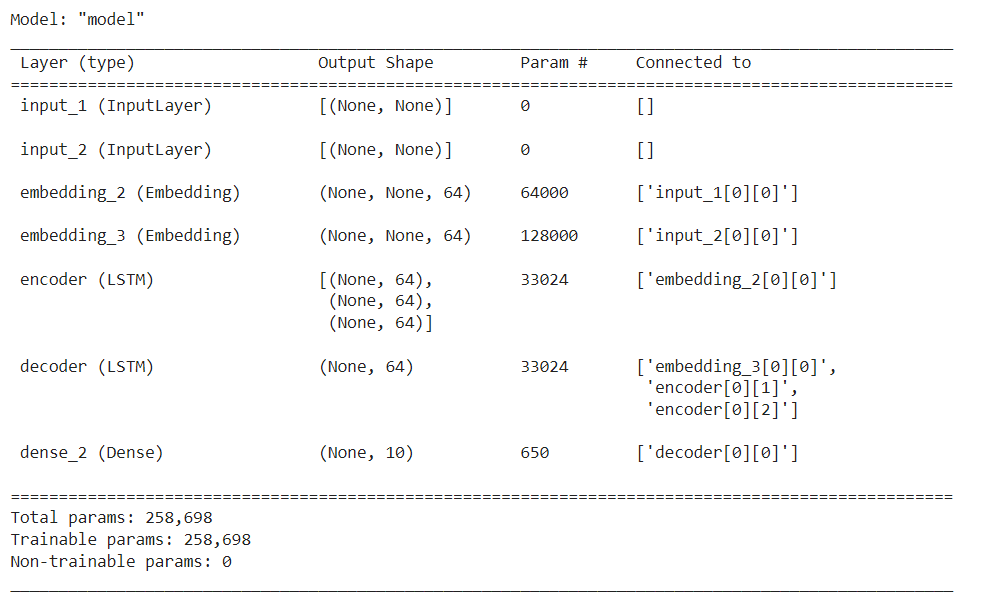
    decoder\_embedded, initial\_state=encoder\_state

)

output = layers.Dense(10)(decoder\_output)

model = keras.Model([encoder\_input, decoder\_input], output)

model.summary()



**PRACTICAL 7**

**USING COVNET TO BUILD DEEP LEARNING MODEL**

This demo trains a Convolutional Neural Network on the [MNIST digits dataset](http://yann.lecun.com/exdb/mnist/) in your browser, with nothing but Javascript. The dataset is fairly easy and one should expect to get somewhere around 99% accuracy within few minutes. I used [this python script](https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist_parse.zip) to parse the [original files](http://deeplearning.net/tutorial/gettingstarted.html) into batches of images that can be easily loaded into page DOM with img tags.

This network takes a 28x28 MNIST image and crops a random 24x24 window before training on it (this technique is called data augmentation and improves generalization). Similarly, to do prediction, 4 random crops are sampled and the probabilities across all crops are averaged to produce final predictions. The network runs at about 5ms for both forward and backward pass on my reasonably decent Ubuntu+Chrome machine.

# Instantiate a Network and Trainer

**CODE:**

layer\_defs = [];

layer\_defs.push({type:'input', out\_sx:24, out\_sy:24, out\_depth:1});

layer\_defs.push({type:'conv', sx:5, filters:8, stride:1, pad:2, activation:'relu'});

layer\_defs.push({type:'pool', sx:2, stride:2});

layer\_defs.push({type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'});

layer\_defs.push({type:'pool', sx:3, stride:3});

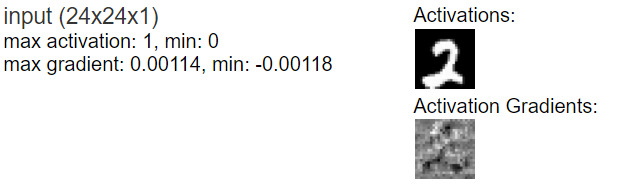
layer\_defs.push({type:'softmax', num\_classes:10});

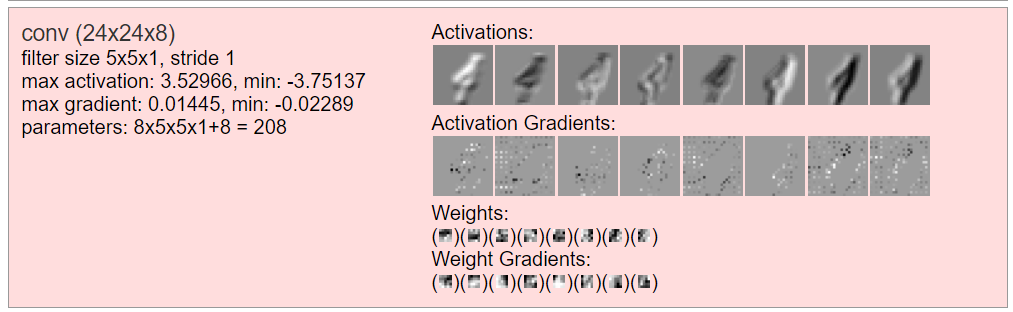
net = new convnetjs.Net();

net.makeLayers(layer\_defs);

trainer = new convnetjs.SGDTrainer(net, {method:'adadelta', batch\_size:20, l2\_decay:0.001});

# Network Visualization





**PRACTICAL 8**

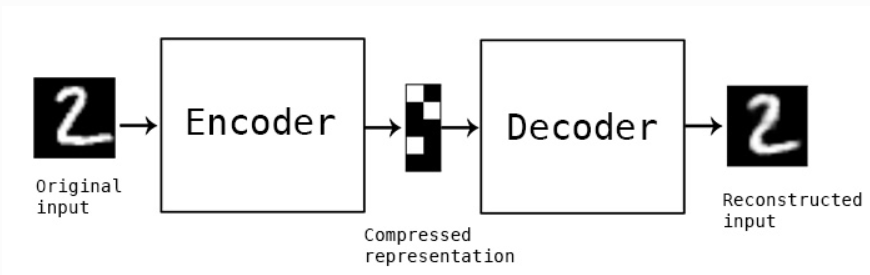
**IMPLEMENTING A SINGLE AUTOENCODER BASED ON FULLY CONNECTED LAYER**

"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) learned automatically from examples rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

1) Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds. An autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Autoencoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

3) Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.



**CODE:**

import keras

from keras import layers

# This is the size of our encoded representations

encoding\_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# This is our input image

input\_img = keras.Input(shape=(784,))

# "encoded" is the encoded representation of the input

encoded = layers.Dense(encoding\_dim, activation='relu')(input\_img)

# "decoded" is the lossy reconstruction of the input

decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction

autoencoder = keras.Model(input\_img, decoded)

#Let's also create a separate encoder model:

# This model maps an input to its encoded representation

encoder = keras.Model(input\_img, encoded)

#As well as the decoder model:

# This is our encoded (32-dimensional) input

encoded\_input = keras.Input(shape=(encoding\_dim,))

# Retrieve the last layer of the autoencoder model

decoder\_layer = autoencoder.layers[-1]

# Create the decoder model

decoder = keras.Model(encoded\_input, decoder\_layer(encoded\_input))

#Now let's train our autoencoder to reconstruct MNIST digits.

#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer:

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels #(since we're only interested in encoding/decoding the input images).

from keras.datasets import mnist

import numpy as np

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

#We will normalize all values between 0 and 1 and we will flatten the 28x28 images into #vectors of size 784.

x\_train = x\_train.astype('float32') / 255.

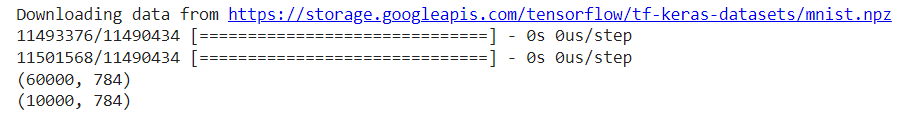
x\_test = x\_test.astype('float32') / 255.

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

print(x\_train.shape)

print(x\_test.shape)



#Now let's train our autoencoder for 50 epochs:

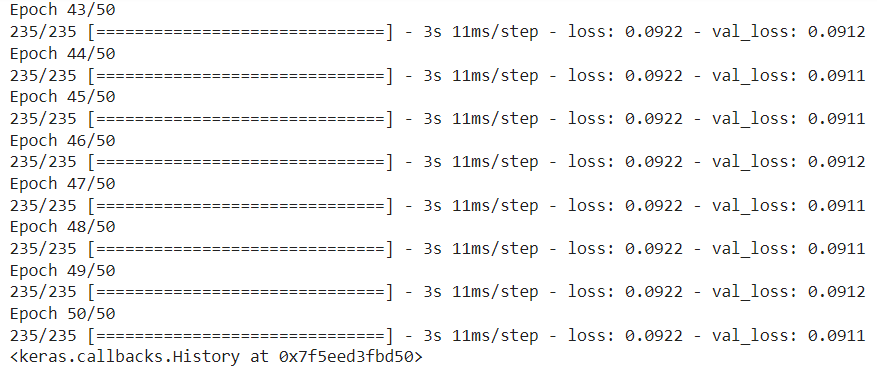
autoencoder.fit(x\_train, x\_train,

                epochs=50,

                batch\_size=256,

                shuffle=True,

                validation\_data=(x\_test, x\_test))



#After 50 epochs, the autoencoder seems to reach a stable train/validation loss value of #about 0.09. We can try to visualize the reconstructed inputs and the encoded #representations. We will use Matplotlib.

# Encode and decode some digits

# Note that we take them from the \*test\* set

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)

# Use Matplotlib (don't ask)

import matplotlib.pyplot as plt

n = 10  # How many digits we will display

plt.figure(figsize=(20, 4))

for i in range(n):

    # Display original

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x\_test[i].reshape(28, 28))

    plt.gray()

    ax.get\_xaxis().set\_visible(False)

    ax.get\_yaxis().set\_visible(False)

    # Display reconstruction

    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(decoded\_imgs[i].reshape(28, 28))

    plt.gray()

    ax.get\_xaxis().set\_visible(False)

    ax.get\_yaxis().set\_visible(False)

plt.show()

