<https://github.com/ved42/Deep_Learning_Practicals/tree/main>

a2

*# Implementing Feedforward neural networks with Keras and TensorFlow*

*# a. Import the necessary packages*

*# b. Load the training and testing data (MNIST/CIFAR10)*

*# c. Define the network architecture using Keras*

*# d. Train the model using SGD*

*# e. Evaluate the network*

*# f. Plot the training loss and accuracy*

**feedforward neural network**

A feedforward neural network, in simple words, is a type of artificial neural network that consists of layers of interconnected nodes, or neurons. It's called "feedforward" because information flows in one direction, from the input layer through intermediate hidden layers to the output layer, without forming loops or feedback connections.

Here's how it works:

Input Layer: The network receives input data, which could be anything from numbers, images, or text.

Hidden Layers: Between the input and output layers, there can be one or more hidden layers. Each layer contains multiple neurons, and these neurons are connected to each other and to the neurons in adjacent layers. Neurons in hidden layers perform mathematical operations on the input data and pass the result to the next layer.

Output Layer: The final layer produces the network's output, which could be a prediction, classification, or any other relevant information based on the input data and the network's learned patterns.

**Tensorflow**

TensorFlow is a tool that helps you teach computers to learn from data and make predictions, which can be used for things like recognizing images, understanding language, or making recommendations. It's like a toolbox for building smarter computer programs.

In [2]:

*# a. Import the necessary packages*

import pandas as pd*#used for convert raw data into tabular format*

import numpy as np*# used for mathematical operations*

import matplotlib.pyplot as plt *#used for visulazation*

import tensorflow as tf

import os

import seaborn as sns*#used advanced visulazation*

import keras

In [3]:

*# b. Load the training and testing data (MNIST/CIFAR10)*

*#converting the whole dataset into 4 arrays*

mnist=tf.keras.datasets.mnist

(X\_train\_full,y\_train\_full),(X\_test,y\_test)=mnist.load\_data()

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

11490434/11490434 [==============================] - 2s 0us/step

10000: This is the number of images in your dataset. In this case, you have 10,000 images.

28: This number represents the height of each image. Each image is 28 pixels tall.

28: This number represents the width of each image. Each image is 28 pixels wide.

In [7]:

X\_train\_full.shape,y\_train\_full.shape,X\_test.shape,y\_test.shape

Out[7]:

((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))

In [ ]:

print(f"data type of x\_train\_full:{X\_train\_full.dtype},\n shape of X\_train\_full:{X\_train\_full.shape}")

data type of x\_train\_full:uint8,

shape of X\_train\_full:(60000, 28, 28)

In [10]:

Out[10]:

array([[[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, 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, 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],

[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]]], dtype=uint8)

**Dividing the data by 255 is done to scale the pixel values of images between 0 and 1. This makes it easier for neural networks to learn from the data, as it helps with faster training and better performance. It's like adjusting the volume on a music player to a standard level before listening to different songs for better clarity and consistency.**

In [11]:

*# Create a validation dataset from the full training data*

*# Scale the data between 0 to 1 by dividing it by 255,as its an unsigned data between 0-255 range*

X\_valid,X\_train=X\_train\_full[:5000]/255.,X\_train\_full[5000:]/255.

y\_valid,y\_train=y\_train\_full[:5000],y\_train\_full[5000:]

*#Scale the test set as well*

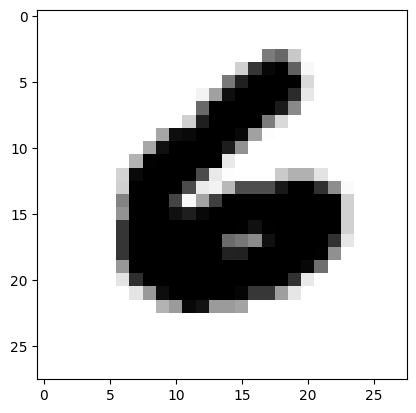
X\_test=X\_test/255.

In [15]:

*# Lets view some data*

plt.imshow(X\_train[18],cmap='binary')

plt.show()



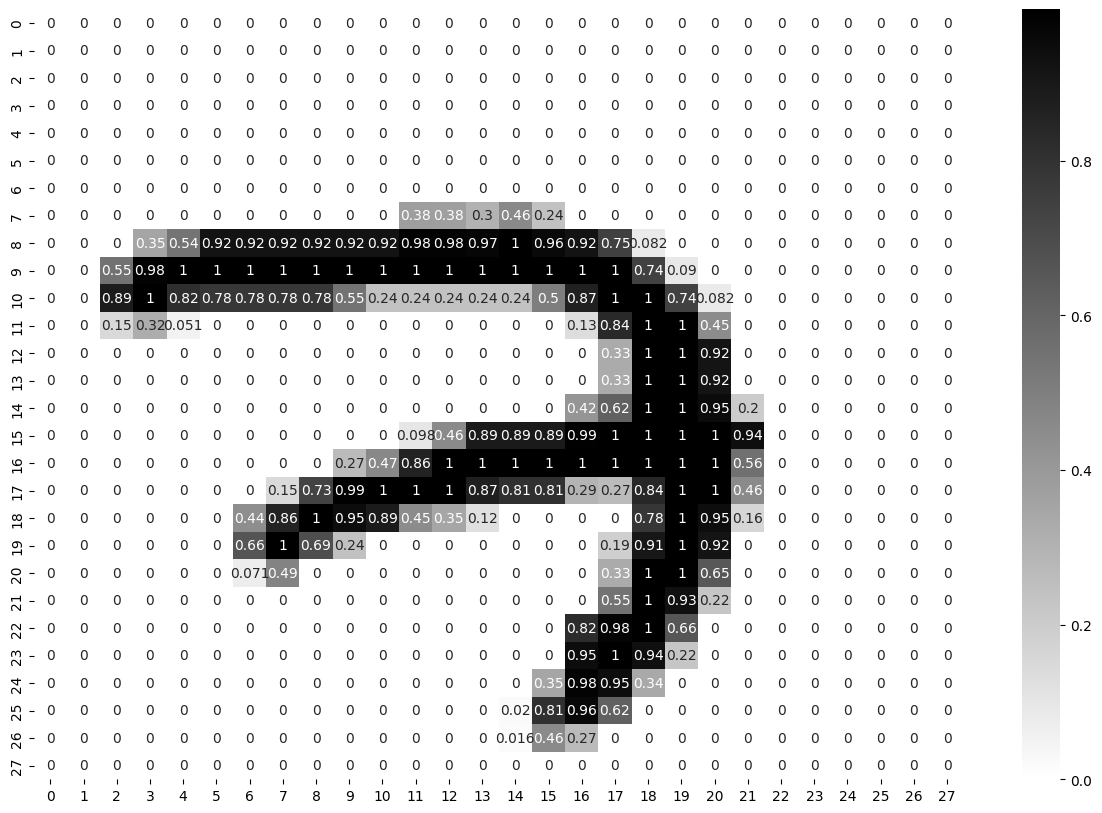
In [16]:

plt.figure(figsize=(15,10))

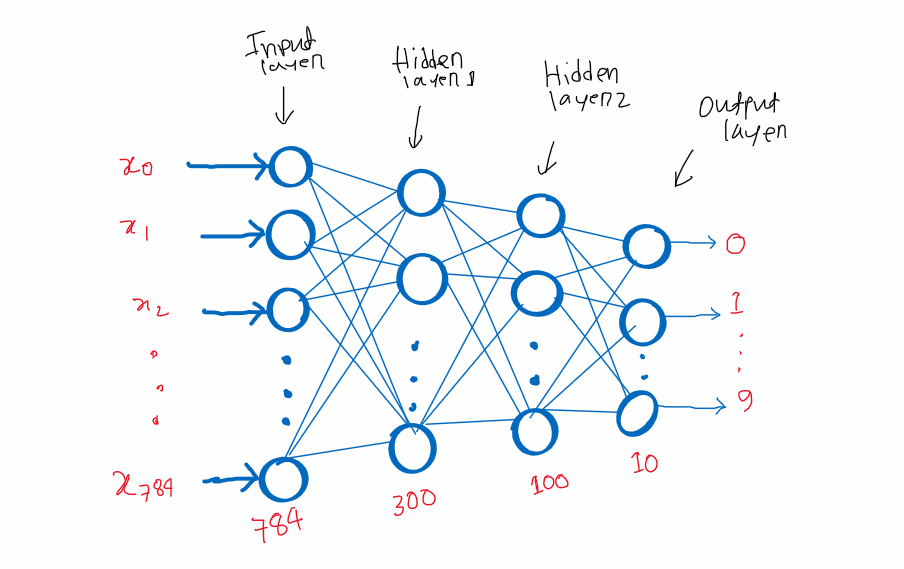
sns.heatmap(X\_train[0],annot=True,cmap='binary')

Out[16]:

<Axes: >



**Architecture Used:**



**Softmax**

In simple words, the softmax activation function is used in machine learning and deep learning to turn a set of numbers into probabilities. It takes a bunch of numbers as input and transforms them in such a way that the largest number becomes close to 1, and the other numbers get smaller. This allows you to represent how likely each option is from a list of choices.

**Relu**

ReLU, which stands for Rectified Linear Unit, is a simple and popular activation function in neural networks. In short, it works like this:

If the input is positive, it keeps the input as it is.

If the input is negative, it turns it into zero.

So, ReLU is like a switch that turns off for negative numbers and lets positive numbers through. It's used to introduce non-linearity in the neural network and helps the network learn complex patterns and relationships in data.

**Dense**

In simple words, the Dense function in deep learning is like a versatile building block that allows a neural network to learn and represent patterns and relationships in data. It connects all the input elements to the output, applies non-linear transformations, and is commonly used for tasks like classification, regression, and feature learning.

**sparse\_categorical\_crossentropy**

In simple words, sparse\_categorical\_crossentropy is a mathematical formula used to measure how well a machine learning model (like a neural network) is performing in tasks where you want to classify things into different categories (classification tasks).

The sparse\_categorical\_crossentropy is a way to calculate how close these predictions are to the actual labels (the correct answers). If the predictions are very close to the actual labels, the value of sparse\_categorical\_crossentropy is low. If the predictions are way off, the value is high. The goal is to train the model to minimize this value, making its predictions as accurate as possible.

In [17]:

*# c. Define the network architecture using Keras*

*# Creating Layers of ANN*

LAYERS=[tf.keras.layers.Flatten(input\_shape=[28,28],name='inputLayer'),

tf.keras.layers.Dense(300,activation="relu",name="HiddenLayer1"),

tf.keras.layers.Dense(100,activation="relu",name="HiddenLayer2"),

tf.keras.layers.Dense(10,activation="softmax",name="outputlayer")]

model\_clf=tf.keras.models.Sequential(LAYERS)

In [18]:

model\_clf.layers

Out[18]:

[<keras.src.layers.reshaping.flatten.Flatten at 0x7eea2464eb00>,

<keras.src.layers.core.dense.Dense at 0x7eea2464f8e0>,

<keras.src.layers.core.dense.Dense at 0x7eea2464f940>,

<keras.src.layers.core.dense.Dense at 0x7eea2464fd30>]

In [19]:

model\_clf.summary()

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

inputLayer (Flatten) (None, 784) 0

HiddenLayer1 (Dense) (None, 300) 235500

HiddenLayer2 (Dense) (None, 100) 30100

outputlayer (Dense) (None, 10) 1010

=================================================================

Total params: 266610 (1.02 MB)

Trainable params: 266610 (1.02 MB)

Non-trainable params: 0 (0.00 Byte)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [20]:

*# first Layer \* second Layer + bias*

784\*300 + 300, 300\*100+100, 100\*10+10

Out[20]:

(235500, 30100, 1010)

**SGD- It will move to each record(row) of the dataset calculate the loss function ,gradient and then it update the parameters of each row**

**Loss function-Loss function is similar to cost dunction only the diffrence is it is calculated for single record..loss function is used to minimized the error and also to update the weights so the weights can reach to the global minima**

In [21]:

*# d. Train the model using SGD*

*# SGD- It will move to each record(row) of the dataset calculate the loss function ,gradient and then it update the parameters of each row*

*# Loss function-Loss function is similar to cost dunction only the diffrence is it is calculated for single record..loss function is used to minimized the error and also to update the weights so the weights can reach to the global minima*

LOSS\_FUNCTION="sparse\_categorical\_crossentropy"*#use=>tf.losses.sparse\_categorical\_crossentropy*

OPTIMIZER="SGD"*#or use with custom learning rate=>tf.keras.optimizers.SGD(0.02)*

METRICS=['accuracy']

model\_clf.compile(loss=LOSS\_FUNCTION,

optimizer=OPTIMIZER,

metrics=METRICS)

**If you set the number of epochs to 30 during the training of an Artificial Neural Network (ANN), it means that the network will go through your entire training dataset 30 times to learn from it and improve its predictions.**

In [22]:

*# training*

*# If you set the number of epochs to 30 during the training of an Artificial Neural Network (ANN), it means that the network will go through your entire training dataset 30 times to learn from it and improve its predictions.*

EPOCHS=30

VALIDATION\_SET=(X\_valid,y\_valid)

history=model\_clf.fit(X\_train,y\_train,epochs=EPOCHS,validation\_data=VALIDATION\_SET,batch\_size=32)

Epoch 1/30

1719/1719 [==============================] - 11s 3ms/step - loss: 0.5992 - accuracy: 0.8480 - val\_loss: 0.3120 - val\_accuracy: 0.9148

Epoch 2/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.2892 - accuracy: 0.9179 - val\_loss: 0.2488 - val\_accuracy: 0.9318

Epoch 3/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.2383 - accuracy: 0.9321 - val\_loss: 0.2041 - val\_accuracy: 0.9438

Epoch 4/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.2034 - accuracy: 0.9422 - val\_loss: 0.1794 - val\_accuracy: 0.9502

Epoch 5/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.1772 - accuracy: 0.9494 - val\_loss: 0.1626 - val\_accuracy: 0.9544

Epoch 6/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.1564 - accuracy: 0.9550 - val\_loss: 0.1458 - val\_accuracy: 0.9614

Epoch 7/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.1394 - accuracy: 0.9607 - val\_loss: 0.1336 - val\_accuracy: 0.9640

Epoch 8/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.1263 - accuracy: 0.9642 - val\_loss: 0.1240 - val\_accuracy: 0.9656

Epoch 9/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.1141 - accuracy: 0.9675 - val\_loss: 0.1135 - val\_accuracy: 0.9696

Epoch 10/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.1044 - accuracy: 0.9709 - val\_loss: 0.1112 - val\_accuracy: 0.9690

Epoch 11/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0963 - accuracy: 0.9730 - val\_loss: 0.1011 - val\_accuracy: 0.9704

Epoch 12/30

1719/1719 [==============================] - 7s 4ms/step - loss: 0.0883 - accuracy: 0.9751 - val\_loss: 0.0975 - val\_accuracy: 0.9730

Epoch 13/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0816 - accuracy: 0.9770 - val\_loss: 0.0915 - val\_accuracy: 0.9746

Epoch 14/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.0757 - accuracy: 0.9787 - val\_loss: 0.0902 - val\_accuracy: 0.9740

Epoch 15/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0701 - accuracy: 0.9803 - val\_loss: 0.0854 - val\_accuracy: 0.9758

Epoch 16/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.0656 - accuracy: 0.9817 - val\_loss: 0.0837 - val\_accuracy: 0.9758

Epoch 17/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0613 - accuracy: 0.9829 - val\_loss: 0.0793 - val\_accuracy: 0.9766

Epoch 18/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.0573 - accuracy: 0.9841 - val\_loss: 0.0773 - val\_accuracy: 0.9772

Epoch 19/30

1719/1719 [==============================] - 5s 3ms/step - loss: 0.0534 - accuracy: 0.9852 - val\_loss: 0.0778 - val\_accuracy: 0.9772

Epoch 20/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0502 - accuracy: 0.9864 - val\_loss: 0.0737 - val\_accuracy: 0.9768

Epoch 21/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0472 - accuracy: 0.9873 - val\_loss: 0.0733 - val\_accuracy: 0.9780

Epoch 22/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0443 - accuracy: 0.9885 - val\_loss: 0.0741 - val\_accuracy: 0.9786

Epoch 23/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0414 - accuracy: 0.9893 - val\_loss: 0.0722 - val\_accuracy: 0.9780

Epoch 24/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0389 - accuracy: 0.9903 - val\_loss: 0.0716 - val\_accuracy: 0.9794

Epoch 25/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.0370 - accuracy: 0.9905 - val\_loss: 0.0694 - val\_accuracy: 0.9780

Epoch 26/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0347 - accuracy: 0.9914 - val\_loss: 0.0697 - val\_accuracy: 0.9792

Epoch 27/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.0326 - accuracy: 0.9919 - val\_loss: 0.0703 - val\_accuracy: 0.9788

Epoch 28/30

1719/1719 [==============================] - 6s 3ms/step - loss: 0.0310 - accuracy: 0.9926 - val\_loss: 0.0692 - val\_accuracy: 0.9798

Epoch 29/30

1719/1719 [==============================] - 6s 4ms/step - loss: 0.0290 - accuracy: 0.9932 - val\_loss: 0.0665 - val\_accuracy: 0.9804

Epoch 30/30

1719/1719 [==============================] - 5s 3ms/step - loss: 0.0273 - accuracy: 0.9940 - val\_loss: 0.0664 - val\_accuracy: 0.9806

In [23]:

history.params

Out[23]:

{'verbose': 1, 'epochs': 30, 'steps': 1719}

**The evaluation of the model on the dataset can be done using the evaluate() function. It takes two arguments i.e, input and output. It will generate a prediction for input**

In [24]:

*# e. Evaluate the network*

model\_clf.evaluate(X\_test,y\_test)

313/313 [==============================] - 1s 2ms/step - loss: 0.0709 - accuracy: 0.9778

Out[24]:

[0.07088310271501541, 0.9778000116348267]

In [25]:

y\_prob=model\_clf.predict(X\_test)

y\_prob

313/313 [==============================] - 1s 2ms/step

Out[25]:

array([[5.25801534e-06, 1.59523083e-07, 5.92304059e-05, ...,

9.99719679e-01, 5.14412818e-07, 2.07506255e-05],

[4.57398119e-06, 3.42297935e-05, 9.99755323e-01, ...,

2.91353747e-11, 3.16736623e-05, 1.68335235e-12],

[8.94955519e-06, 9.96707022e-01, 4.24500409e-04, ...,

1.07333134e-03, 1.15623500e-03, 2.03242580e-05],

...,

[5.28636232e-11, 1.00047345e-10, 2.45209647e-12, ...,

3.38100631e-06, 3.85558394e-07, 4.21060076e-06],

[9.33798816e-08, 5.23022976e-08, 3.02149555e-10, ...,

1.64201397e-09, 1.51739432e-05, 2.72978751e-10],

[8.25444815e-08, 2.14207110e-10, 6.74709213e-07, ...,

5.83831638e-12, 7.07162640e-09, 2.39068348e-10]], dtype=float32)

In [27]:

x\_new=X\_test[:3]

In [28]:

y\_prob=model\_clf.predict(x\_new)

y\_prob.round(3)

1/1 [==============================] - 0s 27ms/step

Out[28]:

array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. ,

0. ],

[0. , 0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. ,

0. ],

[0. , 0.997, 0. , 0. , 0. , 0. , 0. , 0.001, 0.001,

0. ]], dtype=float32)

In [29]:

y\_pred=np.argmax(y\_prob,axis=-1)

y\_pred

Out[29]:

array([7, 2, 1])

In [30]:

actual=y\_test[:3]

actual

Out[30]:

array([7, 2, 1], dtype=uint8)

In [31]:

*# plot*

for data, pred, actual\_data in zip(x\_new, y\_pred, actual):

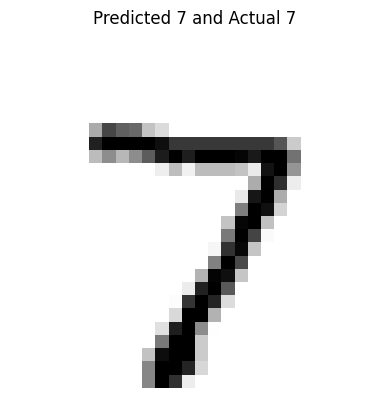
plt.imshow(data, cmap="binary")

plt.title(f"Predicted {pred} and Actual {actual\_data}")

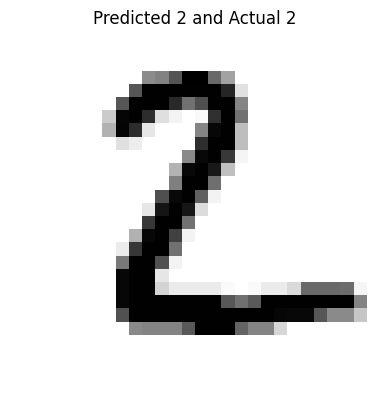
plt.axis("off")

plt.show()

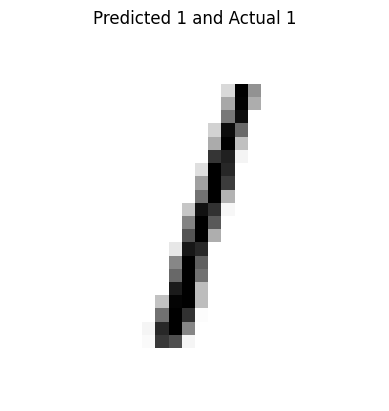
print("######################")



######################



######################



######################

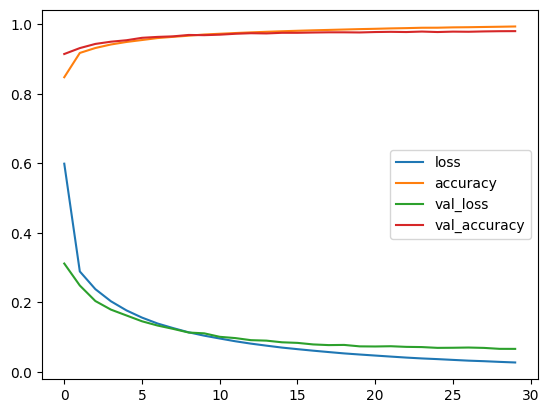
In [26]:

*# f. Plot the training loss and accuracy*

pd.DataFrame(history.history).plot()

Out[26]:

<Axes: >



A3

*# Build the Image classification model by dividing the model into following 4 stages:*

*# a. Loading and preprocessing the image data*

*# b. Defining the model’s architecture*

*# c. Training the model*

*# d. Estimating the model’s performance*

In [ ]:

import tensorflow as tf

from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt

import numpy as np

In [ ]:

(X\_train, y\_train), (X\_test,y\_test) = datasets.cifar10.load\_data()

X\_train.shape

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

170498071/170498071 [==============================] - 2s 0us/step

Out[ ]:

(50000, 32, 32, 3)

In [ ]:

y\_train.shape

Out[ ]:

(50000, 1)

In [ ]:

y\_train[:5]

Out[ ]:

array([[6],

[9],

[9],

[4],

[1]], dtype=uint8)

In [ ]:

*# y\_train is a 2D array, for our classification having 1D array is good enough. so we will convert this to now 1D array*

In [ ]:

y\_train = y\_train.reshape(-1,)

y\_train[:5]

Out[ ]:

array([6, 9, 9, 4, 1], dtype=uint8)

In [ ]:

y\_test = y\_test.reshape(-1,)

In [ ]:

classes = ["airplane","automobile","bird","cat","deer","dog","frog","horse","ship","truck"]

In [ ]:

def plot\_sample(X, y, index):

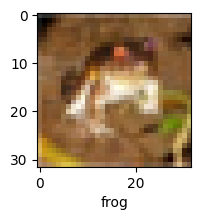
plt.figure(figsize = (15,2))

plt.imshow(X[index])

plt.xlabel(classes[y[index]])

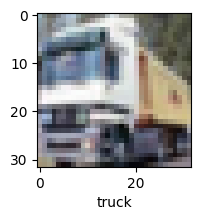
In [ ]:

plot\_sample(X\_train, y\_train, 0)



In [ ]:

plot\_sample(X\_train, y\_train, 1)



Normalize the images to a number from 0 to 1. Image has 3 channels (R,G,B) and each value in the channel can range from 0 to 255. Hence to normalize in 0-->1 range, we need to divide it by 255

Normalizing the training data

In [ ]:

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

**sparse\_categorical\_crossentropy**

In simple words, sparse\_categorical\_crossentropy is a mathematical formula used to measure how well a machine learning model (like a neural network) is performing in tasks where you want to classify things into different categories (classification tasks).

The sparse\_categorical\_crossentropy is a way to calculate how close these predictions are to the actual labels (the correct answers). If the predictions are very close to the actual labels, the value of sparse\_categorical\_crossentropy is low. If the predictions are way off, the value is high. The goal is to train the model to minimize this value, making its predictions as accurate as possible.

**Softmax**

In simple words, the softmax activation function is used in machine learning and deep learning to turn a set of numbers into probabilities. It takes a bunch of numbers as input and transforms them in such a way that the largest number becomes close to 1, and the other numbers get smaller. This allows you to represent how likely each option is from a list of choices.

**Relu**

ReLU, which stands for Rectified Linear Unit, is a simple and popular activation function in neural networks. In short, it works like this:

If the input is positive, it keeps the input as it is.

If the input is negative, it turns it into zero.

So, ReLU is like a switch that turns off for negative numbers and lets positive numbers through. It's used to introduce non-linearity in the neural network and helps the network learn complex patterns and relationships in data.

**Dense**

In simple words, the Dense function in deep learning is like a versatile building block that allows a neural network to learn and represent patterns and relationships in data. It connects all the input elements to the output, applies non-linear transformations, and is commonly used for tasks like classification, regression, and feature learning.

In [ ]:

ann = models.Sequential([

layers.Flatten(input\_shape=(32,32,3)),

layers.Dense(3000, activation='relu'),

layers.Dense(1000, activation='relu'),

layers.Dense(10, activation='softmax')

])

ann.compile(optimizer='SGD',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

ann.fit(X\_train, y\_train, epochs=5)

Epoch 1/5

1563/1563 [==============================] - 160s 102ms/step - loss: 1.8149 - accuracy: 0.3523

Epoch 2/5

1563/1563 [==============================] - 165s 106ms/step - loss: 1.6259 - accuracy: 0.4250

Epoch 3/5

1563/1563 [==============================] - 141s 90ms/step - loss: 1.5430 - accuracy: 0.4556

Epoch 4/5

1563/1563 [==============================] - 140s 90ms/step - loss: 1.4838 - accuracy: 0.4778

Epoch 5/5

1563/1563 [==============================] - 148s 95ms/step - loss: 1.4353 - accuracy: 0.4943

Out[ ]:

<keras.src.callbacks.History at 0x795fbcd16050>

In [ ]:

from sklearn.metrics import confusion\_matrix , classification\_report

import numpy as np

y\_pred = ann.predict(X\_test)

y\_pred\_classes = [np.argmax(element) for element in y\_pred]

print("Classification Report: \n", classification\_report(y\_test, y\_pred\_classes))

313/313 [==============================] - 16s 50ms/step

Classification Report:

precision recall f1-score support

0 0.62 0.49 0.55 1000

1 0.57 0.66 0.61 1000

2 0.46 0.23 0.31 1000

3 0.36 0.34 0.35 1000

4 0.67 0.12 0.21 1000

5 0.42 0.34 0.37 1000

6 0.45 0.65 0.53 1000

7 0.33 0.80 0.47 1000

8 0.74 0.47 0.57 1000

9 0.54 0.59 0.56 1000

accuracy 0.47 10000

macro avg 0.51 0.47 0.45 10000

weighted avg 0.51 0.47 0.45 10000

In [ ]:

**Conv2D**

Certainly! In simple words, the Conv2D function in deep learning is like a detective that scans an image to find important patterns, shapes, and features. It's used to understand what's in the image, helping computers recognize objects, like cats or dogs, by examining tiny parts of the picture at a time. This is a crucial step in tasks like image recognition and classification.

**MaxPooling2D**

MaxPooling2D, in simple terms, is like taking a quick summary of an image. It divides the image into small blocks and, for each block, keeps only the largest value. This helps reduce the amount of data to process while retaining the essential features. It's like zooming out to see the bigger picture in a more manageable way, often used in image processing to make computations faster and focus on important details.

**Adam Optimizer**

In simple words, the Adam optimizer is like a guide that helps a machine learning model learn and improve. It adjusts the model's learning rate as it trains, making it go faster in the beginning (to learn quickly) and slower later (to fine-tune). This helps the model find the best solution efficiently, making it a popular choice for training neural networks.

**CNN**

In simple words, a CNN (Convolutional Neural Network) is a type of computer program that's really good at understanding pictures. It breaks down the image into smaller pieces, looks for patterns, and uses those patterns to figure out what's in the picture. It's like how our brain recognizes shapes and objects by looking at different parts of an image. CNNs are used in things like facial recognition, self-driving cars, and even telling you what's in a photo.

In [ ]:

cnn = models.Sequential([

layers.Conv2D(filters=32, kernel\_size=(3, 3), activation='relu', input\_shape=(32, 32, 3)),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(filters=64, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Flatten(),

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

layers.Dense(10, activation='softmax')

])

In [ ]:

cnn.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

In [ ]:

cnn.fit(X\_train, y\_train, epochs=10)

Epoch 1/10

1563/1563 [==============================] - 65s 41ms/step - loss: 1.4585 - accuracy: 0.4764

Epoch 2/10

1563/1563 [==============================] - 65s 42ms/step - loss: 1.1014 - accuracy: 0.6136

Epoch 3/10

1563/1563 [==============================] - 64s 41ms/step - loss: 0.9772 - accuracy: 0.6594

Epoch 4/10

1563/1563 [==============================] - 64s 41ms/step - loss: 0.8918 - accuracy: 0.6910

Epoch 5/10

1563/1563 [==============================] - 63s 40ms/step - loss: 0.8204 - accuracy: 0.7133

Epoch 6/10

1563/1563 [==============================] - 66s 42ms/step - loss: 0.7608 - accuracy: 0.7340

Epoch 7/10

1563/1563 [==============================] - 64s 41ms/step - loss: 0.7094 - accuracy: 0.7541

Epoch 8/10

1563/1563 [==============================] - 64s 41ms/step - loss: 0.6659 - accuracy: 0.7662

Epoch 9/10

1563/1563 [==============================] - 63s 41ms/step - loss: 0.6189 - accuracy: 0.7839

Epoch 10/10

1563/1563 [==============================] - 63s 41ms/step - loss: 0.5820 - accuracy: 0.7950

Out[ ]:

<keras.src.callbacks.History at 0x795fc3e34a60>

With CNN, at the end 5 epochs, accuracy was at around 70% which is a significant improvement over ANN. CNN's are best for image classification and gives superb accuracy. Also computation is much less compared to simple ANN as maxpooling reduces the image dimensions while still preserving the features

In [ ]:

cnn.evaluate(X\_test,y\_test)

313/313 [==============================] - 5s 16ms/step - loss: 0.9598 - accuracy: 0.6867

Out[ ]:

[0.9598063230514526, 0.6866999864578247]

In [ ]:

y\_pred = cnn.predict(X\_test)

y\_pred[:5]

313/313 [==============================] - 4s 13ms/step

Out[ ]:

array([[3.7033547e-05, 3.7818354e-05, 2.7866499e-04, 7.1656936e-01,

6.5130298e-05, 2.7911690e-01, 3.6428377e-03, 4.7474983e-05,

1.8147808e-04, 2.3268634e-05],

[6.4340471e-03, 2.6215971e-04, 6.5784894e-07, 7.0468438e-08,

5.1743282e-08, 5.7075893e-12, 2.0128017e-09, 4.2582340e-12,

9.9329782e-01, 5.2164828e-06],

[1.0339052e-02, 4.4135097e-02, 3.5547378e-04, 8.7805500e-04,

9.7768709e-05, 4.8042143e-06, 4.9840124e-05, 1.0674240e-05,

9.3856090e-01, 5.5683455e-03],

[9.5089322e-01, 3.4475102e-04, 8.0184033e-03, 9.8825304e-04,

7.9321619e-03, 8.8448523e-06, 3.2401126e-04, 4.6731827e-07,

3.1232288e-02, 2.5764402e-04],

[5.8452320e-06, 1.7955457e-05, 1.8396771e-02, 1.0334741e-02,

3.8233700e-01, 1.2916062e-04, 5.8875591e-01, 1.1621773e-05,

1.0291834e-05, 5.8233684e-07]], dtype=float32)

In [ ]:

y\_classes = [np.argmax(element) for element in y\_pred]

y\_classes[:5]

Out[ ]:

[3, 8, 8, 0, 6]

In [ ]:

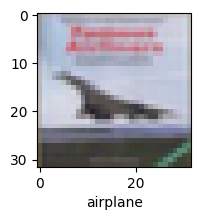
y\_test[:5]

Out[ ]:

array([3, 8, 8, 0, 6], dtype=uint8)

In [ ]:

plot\_sample(X\_test, y\_test,3)



In [ ]:

classes[y\_classes[3]]

Out[ ]:

'airplane'

A4

import keras

from keras import layers

1. Use Autoencoder to implement anomaly detection. Build the model by using:

a. Import required libraries

b. Upload / access the dataset

c. Encoder converts it into latent representation

d. Decoder networks convert it back to the original input

e. Compile the models with Optimizer, Loss, and Evaluation Metrics

**Autoencoder**

An autoencoder is a type of artificial neural network used in unsupervised learning and dimensionality reduction. It's designed to encode and then decode data, often used for tasks like data compression, denoising, and feature learning. Here's how it works:

* Encoder: The first part of the autoencoder, called the encoder, takes input data and compresses it into a lower-dimensional representation. This lower-dimensional representation is sometimes referred to as the "encoding" or "latent space." The encoder consists of one or more layers of neurons that learn to capture the most important features or patterns in the input data.
* Decoder: The second part of the autoencoder, called the decoder, takes the encoded representation and attempts to reconstruct the original data from it. Like the encoder, the decoder consists of one or more layers that learn to convert the encoded data back into the original format.

The goal of an autoencoder is to learn a compact representation of the input data in the encoding step and then learn to reconstruct the original data from this compact representation in the decoding step. During training, the autoencoder's parameters (weights) are adjusted to minimize the difference between the input data and the reconstructed data.

In [2]:

*# This is the size of our encoded representations*

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

In [3]:

*# This is our input image*

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

So, the purpose of this line is to create a dense layer with encoding\_dim neurons, using the ReLU activation function, and connect it to the input\_img, forming a part of a neural network model.

In [4]:

*# "encoded" is the encoded representation of the input*

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

So, the purpose of this line is to create a dense layer with 784 neurons, using the sigmoid activation function, and connect it to the encoded layer. This is commonly used in autoencoders to decode the reduced-dimensional representation (encoding) back into the original data space, such as image reconstruction.

In [5]:

*# "decoded" is the lossy reconstruction of the input*

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

In summary, autoencoder = keras.Model(input\_img, decoded) is used to create a Keras model that defines the architecture of your autoencoder, allowing you to train, evaluate, and use it for various tasks related to encoding and decoding data.

In [6]:

*# This model maps an input to its reconstruction*

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

autoencoder.layers

Out[6]:

[<keras.src.engine.input\_layer.InputLayer at 0x78de0e5ffb80>,

<keras.src.layers.core.dense.Dense at 0x78de0e62a0b0>,

<keras.src.layers.core.dense.Dense at 0x78de0e62bc70>]

In simpler words, this model is used to extract a compressed, simplified representation of your input data. It's like creating a "data compressor" that can take in your data and reduce it to a more compact form, which can be useful for tasks like data compression, feature extraction, or dimensionality reduction.

In [7]:

*# This model maps an input to its encoded representation*

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

The line encoded\_input = keras.Input(shape=(encoding\_dim,)) creates an input layer for your neural network model with a specific shape.

In [8]:

*# This is our encoded (32-dimensional) input*

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

In [9]:

*# Retrieve the last layer of the autoencoder model (output layer)*

decoder\_layer = autoencoder.layers[-1]

In [10]:

*# 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:

The binary\_crossentropy function is typically used in the context of binary classification tasks in machine learning and deep learning. In simple words, it's a way to measure how well a model is performing when it needs to decide between two choices, often denoted as "0" and "1," "yes" and "no," or "positive" and "negative."

In [11]:

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).

In [12]:

from keras.datasets import mnist

import numpy as np

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

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

11490434/11490434 [==============================] - 0s 0us/step

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

In [13]:

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)

(60000, 784)

(10000, 784)

Now let's train our autoencoder for 50 epochs:

In [14]:

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

epochs=40,

batch\_size=256,

shuffle=True,

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

Epoch 1/40

235/235 [==============================] - 4s 15ms/step - loss: 0.2762 - val\_loss: 0.1887

Epoch 2/40

235/235 [==============================] - 3s 12ms/step - loss: 0.1708 - val\_loss: 0.1537

Epoch 3/40

235/235 [==============================] - 3s 11ms/step - loss: 0.1444 - val\_loss: 0.1335

Epoch 4/40

235/235 [==============================] - 3s 11ms/step - loss: 0.1285 - val\_loss: 0.1216

Epoch 5/40

235/235 [==============================] - 3s 11ms/step - loss: 0.1188 - val\_loss: 0.1138

Epoch 6/40

235/235 [==============================] - 3s 15ms/step - loss: 0.1120 - val\_loss: 0.1083

Epoch 7/40

235/235 [==============================] - 3s 11ms/step - loss: 0.1069 - val\_loss: 0.1036

Epoch 8/40

235/235 [==============================] - 3s 11ms/step - loss: 0.1030 - val\_loss: 0.1002

Epoch 9/40

235/235 [==============================] - 3s 11ms/step - loss: 0.1000 - val\_loss: 0.0976

Epoch 10/40

235/235 [==============================] - 3s 14ms/step - loss: 0.0978 - val\_loss: 0.0957

Epoch 11/40

235/235 [==============================] - 3s 12ms/step - loss: 0.0963 - val\_loss: 0.0947

Epoch 12/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0954 - val\_loss: 0.0938

Epoch 13/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0947 - val\_loss: 0.0934

Epoch 14/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0943 - val\_loss: 0.0930

Epoch 15/40

235/235 [==============================] - 4s 15ms/step - loss: 0.0940 - val\_loss: 0.0927

Epoch 16/40

235/235 [==============================] - 2s 11ms/step - loss: 0.0938 - val\_loss: 0.0925

Epoch 17/40

235/235 [==============================] - 2s 10ms/step - loss: 0.0937 - val\_loss: 0.0924

Epoch 18/40

235/235 [==============================] - 2s 10ms/step - loss: 0.0935 - val\_loss: 0.0923

Epoch 19/40

235/235 [==============================] - 3s 12ms/step - loss: 0.0934 - val\_loss: 0.0922

Epoch 20/40

235/235 [==============================] - 3s 13ms/step - loss: 0.0933 - val\_loss: 0.0921

Epoch 21/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0933 - val\_loss: 0.0921

Epoch 22/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0932 - val\_loss: 0.0920

Epoch 23/40

235/235 [==============================] - 2s 11ms/step - loss: 0.0932 - val\_loss: 0.0920

Epoch 24/40

235/235 [==============================] - 3s 13ms/step - loss: 0.0931 - val\_loss: 0.0919

Epoch 25/40

235/235 [==============================] - 3s 12ms/step - loss: 0.0930 - val\_loss: 0.0919

Epoch 26/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0930 - val\_loss: 0.0919

Epoch 27/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0930 - val\_loss: 0.0918

Epoch 28/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0929 - val\_loss: 0.0918

Epoch 29/40

235/235 [==============================] - 3s 15ms/step - loss: 0.0929 - val\_loss: 0.0917

Epoch 30/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0929 - val\_loss: 0.0917

Epoch 31/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0929 - val\_loss: 0.0918

Epoch 32/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0928 - val\_loss: 0.0917

Epoch 33/40

235/235 [==============================] - 3s 13ms/step - loss: 0.0928 - val\_loss: 0.0917

Epoch 34/40

235/235 [==============================] - 3s 12ms/step - loss: 0.0928 - val\_loss: 0.0917

Epoch 35/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0928 - val\_loss: 0.0917

Epoch 36/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0928 - val\_loss: 0.0917

Epoch 37/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val\_loss: 0.0916

Epoch 38/40

235/235 [==============================] - 3s 15ms/step - loss: 0.0927 - val\_loss: 0.0916

Epoch 39/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val\_loss: 0.0916

Epoch 40/40

235/235 [==============================] - 3s 11ms/step - loss: 0.0927 - val\_loss: 0.0916

Out[14]:

<keras.src.callbacks.History at 0x78de0c1d9510>

After 40 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.

In short, an 'encoder' is a component in machine learning used to convert input data (e.g., text, images, or numerical values) into a different representation, typically a numerical one. It's often used in various tasks like data compression, feature extraction, and creating embeddings for downstream applications.

In short, a 'decoder' is a component in machine learning used to convert a hidden representation or encoded data back into a more understandable or original format. It's commonly used in tasks like language translation, image generation, and sequence-to-sequence tasks.

In [15]:

*# 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)

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

In [16]:

*# 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)

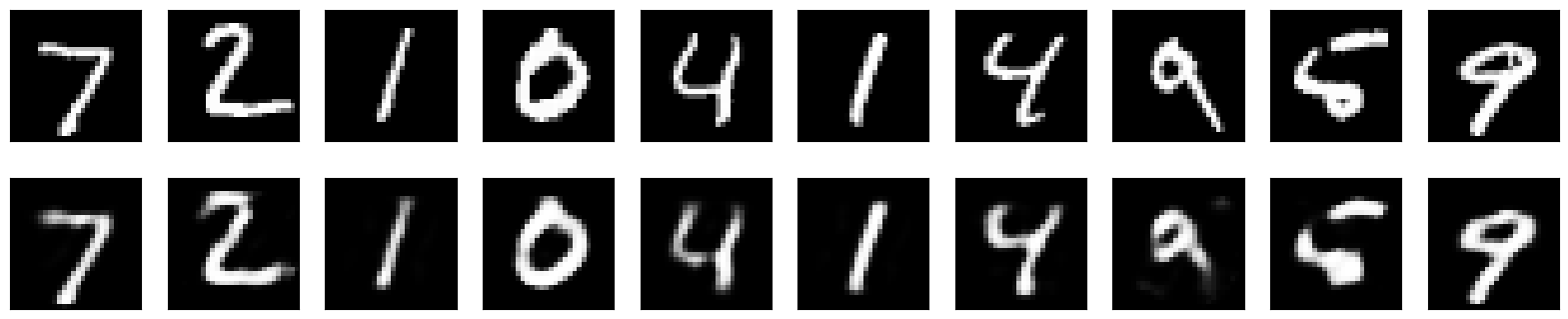
plt.imshow(decoded\_imgs[i].reshape(28, 28))

plt.gray()

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

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

plt.show()



Here's what we get. The top row is the original digits, and the bottom row is the reconstructed digits. We are losing quite a bit of detail with this basic approach.

In [ ]:

A5

Certainly! Let me explain the Continuous Bag of Words (CBOW) model in simple words.

Imagine you have a big collection of sentences, like books or articles. The CBOW model is like a smart system that learns the meaning of words by looking at the words around them. It's trying to understand what a word means by examining the words that often appear close to it.

Here's how CBOW works:

1. It looks at a target word. Let's say the target word is "apple."
2. It checks the words that are usually found near "apple" in sentences, like "delicious," "fruit," and "red."
3. It uses these surrounding words to guess what "apple" means. For example, if it sees "delicious" and "fruit" a lot, it might think "apple" is related to being a tasty fruit.
4. CBOW repeats this process for many words in the text to learn what they mean based on their context.

So, in simple terms, CBOW is a method that learns the meaning of words by looking at the words around them. This helps it create word representations that understand the words' relationships in the language, making it useful for various language-related tasks.

Prepare the Data: Load your text corpus and preprocess it. Tokenize the sentences, remove punctuation, convert text to lowercase, and create a vocabulary with unique words. Assign an index to each word in the vernacular.

In short, Gensim is a Python library used for natural language processing (NLP) tasks. Its main functions include topic modeling, word vector embeddings, document similarity analysis, text preprocessing, document representation, and efficient handling of large textual datasets. It's a valuable tool for tasks like discovering topics in documents, measuring document similarity, and creating word embeddings for NLP applications.

In [7]:

import gensim

import pandas as pd

Reading and Exploring the Dataset The dataset we are using here is a subset of Amazon reviews from the Cell Phones & Accessories category. The data is stored as a JSON file and can be read using pandas.

Link to the Dataset: <http://snap.stanford.edu/data/amazon/productGraph/categoryFiles/reviews_Cell_Phones_and_Accessories_5.json.gz>

In [20]:

df=pd.read\_json("http://snap.stanford.edu/data/amazon/productGraph/categoryFiles/reviews\_Cell\_Phones\_and\_Accessories\_5.json.gz",lines=True)

Simple Preprocessing & Tokenization The first thing to do for any data science task is to clean the data. For NLP, we apply various processing like converting all the words to lower case, trimming spaces, removing punctuations. This is something we will do over here too.

Additionally, we can also remove stop words like 'and', 'or', 'is', 'the', 'a', 'an' and convert words to their root forms like 'running' to 'run'.

In [22]:

df.head()

Out[22]:

|  | **reviewerID** | **asin** | **reviewerName** | **helpful** | **reviewText** | **overall** | **summary** | **unixReviewTime** | **reviewTime** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | A30TL5EWN6DFXT | 120401325X | christina | [0, 0] | They look good and stick good! I just don't li... | 4 | Looks Good | 1400630400 | 05 21, 2014 |
| **1** | ASY55RVNIL0UD | 120401325X | emily l. | [0, 0] | These stickers work like the review says they ... | 5 | Really great product. | 1389657600 | 01 14, 2014 |
| **2** | A2TMXE2AFO7ONB | 120401325X | Erica | [0, 0] | These are awesome and make my phone look so st... | 5 | LOVE LOVE LOVE | 1403740800 | 06 26, 2014 |
| **3** | AWJ0WZQYMYFQ4 | 120401325X | JM | [4, 4] | Item arrived in great time and was in perfect ... | 4 | Cute! | 1382313600 | 10 21, 2013 |
| **4** | ATX7CZYFXI1KW | 120401325X | patrice m rogoza | [2, 3] | awesome! stays on, and looks great. can be use... | 5 | leopard home button sticker for iphone 4s | 1359849600 | 02 3, 2013 |

The simple\_preprocess function in Gensim is used to preprocess and tokenize a text document. It takes a text string as input and performs the following tasks:

Tokenization: It breaks the input text into individual words or tokens, splitting it based on spaces and punctuation.

Lowercasing: It converts all tokens to lowercase to ensure consistent handling of text.

Removing Short Tokens: By default, simple\_preprocess removes tokens that are shorter than 2 characters in length, as they are often considered less meaningful.

In [21]:

review\_text = df.reviewText.apply(gensim.utils.simple\_preprocess)

In [23]:

review\_text

Out[23]:

0 [they, look, good, and, stick, good, just, don...

1 [these, stickers, work, like, the, review, say...

2 [these, are, awesome, and, make, my, phone, lo...

3 [item, arrived, in, great, time, and, was, in,...

4 [awesome, stays, on, and, looks, great, can, b...

...

194434 [works, great, just, like, my, original, one, ...

194435 [great, product, great, packaging, high, quali...

194436 [this, is, great, cable, just, as, good, as, t...

194437 [really, like, it, becasue, it, works, well, w...

194438 [product, as, described, have, wasted, lot, of...

Name: reviewText, Length: 194439, dtype: object

In [24]:

review\_text.loc[0]

Out[24]:

['they',

'look',

'good',

'and',

'stick',

'good',

'just',

'don',

'like',

'the',

'rounded',

'shape',

'because',

'was',

'always',

'bumping',

'it',

'and',

'siri',

'kept',

'popping',

'up',

'and',

'it',

'was',

'irritating',

'just',

'won',

'buy',

'product',

'like',

'this',

'again']

In [25]:

df.reviewText.loc[0]

Out[25]:

"They look good and stick good! I just don't like the rounded shape because I was always bumping it and Siri kept popping up and it was irritating. I just won't buy a product like this again"

Training the Word2Vec Model Train the model for reviews. Use a window of size 10 i.e. 10 words before the present word and 10 words ahead. A sentence with at least 2 words should only be considered, configure this using min\_count parameter.

Workers define how many CPU threads to be used.

Initialize the model

In short, the Word2Vec function in Gensim is used to create word embeddings, which are numerical representations of words that capture their semantic meaning. These embeddings are helpful for tasks like word similarity, analogies, text classification, and recommendation systems in natural language processing.

In [26]:

model = gensim.models.Word2Vec(

window=10,

min\_count=2,

workers=4,

)

Build Vocabulary

In short, the build\_vocab function in Gensim is used to construct the vocabulary for training a Word2Vec model. It processes the text data to identify unique words and assigns a unique numerical ID to each word, which is crucial for subsequent training of the Word2Vec model.

In [27]:

model.build\_vocab(review\_text, progress\_per=1000)

Train the Word2Vec Model

In [28]:

model.train(review\_text, total\_examples=model.corpus\_count, epochs=model.epochs)

Out[28]:

(61505326, 83868975)

Save the Model

Save the model so that it can be reused in other applications

In [29]:

model.save("./word2vec-amazon-cell-accessories-reviews-short.model")

Finding Similar Words and Similarity between words

In [ ]:

<https://radimrehurek.com/gensim/models/word2vec.html>

the wv.most\_similar function is used to find words that are most similar to a given word in a Word2Vec model's word embedding space. It helps you discover words that have similar meanings or contexts to the word you provide as input.

In [30]:

model.wv.most\_similar("bad")

Out[30]:

[('terrible', 0.6989720463752747),

('shabby', 0.6393527388572693),

('good', 0.5959081053733826),

('horrible', 0.5899097323417664),

('crappy', 0.5518659949302673),

('pathetic', 0.5452446937561035),

('funny', 0.5432130098342896),

('disappointing', 0.533385694026947),

('okay', 0.529837429523468),

('awful', 0.521769106388092)]

In [31]:

model.wv.similarity(w1="cheap", w2="inexpensive")

Out[31]:

0.5554835

In [32]:

model.wv.similarity(w1="great", w2="good")

Out[32]:

0.78176177

Further Reading You can read about gensim more at <https://radimrehurek.com/gensim/models/word2vec.html>

Explore other Datasets related to Amazon Reviews: <http://jmcauley.ucsd.edu/data/amazon/>

A6

Object detection using Transfer Learning of CNN architectures a. Load in a pre-trained CNN model trained on a large dataset b. Freeze parameters (weights) in model’s lower convolutional layers c. Add custom classifier with several layers of trainable parameters to model d. Train classifier layers on training data available for task e. Fine-tune hyper parameters and unfreeze more layers as needed

Transfer Learning: Transfer learning in object detection is a technique where a pre-trained deep learning model, typically a convolutional neural network (CNN), is used as a starting point for training a new object detection model. Instead of training a deep neural network from scratch, transfer learning leverages the knowledge and features learned by the pre-trained model on a large dataset.

Here's how transfer learning works in object detection:

Pre-trained Model: A pre-trained CNN model, such as VGG, ResNet, or MobileNet, that has been trained on a large dataset, often for image classification, is selected as the starting point.

Feature Extraction: The pre-trained model's layers, especially the earlier convolutional layers, are retained. These layers have learned to extract meaningful features from images. They act as feature extractors for the object detection task.

Adaptation: A new set of layers, including object detection-specific layers like anchor boxes, region proposal networks (RPNs), and classification/regression heads, are added on top of the retained layers. These layers are specific to the object detection task and are trained from scratch.

Fine-Tuning: The entire model is trained on a smaller dataset for the specific object detection task. This fine-tuning allows the model to adapt to the new object classes and detection challenges.

In [1]:

import numpy as np

import cv2

import PIL.Image as Image

import os

import matplotlib.pylab as plt

import tensorflow as tf

import tensorflow\_hub as hub

from tensorflow import keras

from tensorflow.keras import layers

from tensorflow.keras.models import Sequential

Make predictions using ready made model (without any training)

In [2]:

IMAGE\_SHAPE = (224, 224)

classifier = tf.keras.Sequential([

hub.KerasLayer("https://tfhub.dev/google/tf2-preview/mobilenet\_v2/classification/4", input\_shape=IMAGE\_SHAPE+(3,))

])

In [3]:

gold\_fish = Image.open("/content/goldfish.jpeg").resize(IMAGE\_SHAPE)

gold\_fish

Out[3]:



In [4]:

gold\_fish = np.array(gold\_fish)/255.0

gold\_fish.shape

Out[4]:

(224, 224, 3)

In [5]:

gold\_fish[np.newaxis, ...]

Out[5]:

array([[[[0.10588235, 0.1254902 , 0.10196078],

[0.10588235, 0.1254902 , 0.10196078],

[0.10588235, 0.1254902 , 0.10196078],

...,

[0.20392157, 0.25882353, 0.2 ],

[0.20392157, 0.25882353, 0.2 ],

[0.18823529, 0.24313725, 0.18431373]],

[[0.10588235, 0.1254902 , 0.10196078],

[0.10588235, 0.1254902 , 0.10196078],

[0.10588235, 0.1254902 , 0.10196078],

...,

[0.21176471, 0.26666667, 0.20784314],

[0.20784314, 0.2627451 , 0.20392157],

[0.19215686, 0.24705882, 0.18823529]],

[[0.10588235, 0.1254902 , 0.10196078],

[0.10588235, 0.1254902 , 0.10196078],

[0.10588235, 0.1254902 , 0.10196078],

...,

[0.21568627, 0.27058824, 0.21176471],

[0.21568627, 0.27058824, 0.21176471],

[0.19215686, 0.24705882, 0.18823529]],

...,

[[0.05882353, 0.06666667, 0.01568627],

[0.05490196, 0.0627451 , 0.01176471],

[0.04705882, 0.05490196, 0.00392157],

...,

[0.03921569, 0.05098039, 0.00784314],

[0.03921569, 0.05098039, 0.00784314],

[0.03529412, 0.04705882, 0.00392157]],

[[0.05490196, 0.0627451 , 0.01176471],

[0.05098039, 0.05882353, 0.00784314],

[0.04313725, 0.05098039, 0. ],

...,

[0.03529412, 0.04705882, 0.00392157],

[0.03529412, 0.04705882, 0.00392157],

[0.03137255, 0.04313725, 0. ]],

[[0.04705882, 0.05490196, 0.00392157],

[0.04313725, 0.05098039, 0. ],

[0.03921569, 0.04705882, 0. ],

...,

[0.03529412, 0.04705882, 0.00392157],

[0.03137255, 0.04313725, 0. ],

[0.03137255, 0.04313725, 0. ]]]])

In [6]:

result = classifier.predict(gold\_fish[np.newaxis, ...])

result.shape

1/1 [==============================] - 3s 3s/step

Out[6]:

(1, 1001)

In [7]:

predicted\_label\_index = np.argmax(result)

predicted\_label\_index

Out[7]:

2

In [8]:

image\_labels = []

with open("ImageNetLabels.txt", "r") as f:

image\_labels = f.read().splitlines()

image\_labels[:5]

Out[8]:

['background', 'tench', 'goldfish', 'great white shark', 'tiger shark']

In [9]:

image\_labels[predicted\_label\_index]

Out[9]:

'goldfish'

Load flowers dataset

In [10]:

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, cache\_dir='.', untar=True)

*# cache\_dir indicates where to download data. I specified . which means current directory*

*# untar true will unzip it*

Downloading data from https://storage.googleapis.com/download.tensorflow.org/example\_images/flower\_photos.tgz

228813984/228813984 [==============================] - 5s 0us/step

In [11]:

import pathlib

data\_dir = pathlib.Path(data\_dir)

data\_dir

Out[11]:

PosixPath('datasets/flower\_photos')

In [12]:

list(data\_dir.glob('\*/\*.jpg'))[:5]

Out[12]:

[PosixPath('datasets/flower\_photos/tulips/8523133474\_d2c0845b54.jpg'),

PosixPath('datasets/flower\_photos/tulips/14097328354\_4f1469a170.jpg'),

PosixPath('datasets/flower\_photos/tulips/17078576150\_6f272ce73f\_n.jpg'),

PosixPath('datasets/flower\_photos/tulips/8673416556\_639f5c88f1\_n.jpg'),

PosixPath('datasets/flower\_photos/tulips/7166626128\_8e0983ac8e\_n.jpg')]

In [13]:

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

print(image\_count)

3670

In [14]:

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

roses[:5]

Out[14]:

[PosixPath('datasets/flower\_photos/roses/18741313803\_1bbf842fc6\_n.jpg'),

PosixPath('datasets/flower\_photos/roses/5332550500\_ab341aefd8.jpg'),

PosixPath('datasets/flower\_photos/roses/5897035797\_e67bf68124\_n.jpg'),

PosixPath('datasets/flower\_photos/roses/4979895172\_ca06eba616.jpg'),

PosixPath('datasets/flower\_photos/roses/4495885281\_fe2a3b671d.jpg')]

In [15]:

!pip install Pillow

Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (9.4.0)

In [16]:

from PIL import Image

In [17]:

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

Out[17]:



In [18]:

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

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

Out[18]:



Read flowers images from disk into numpy array using opencv

In [19]:

flowers\_images\_dict = {

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

'daisy': list(data\_dir.glob('daisy/\*')),

'dandelion': list(data\_dir.glob('dandelion/\*')),

'sunflowers': list(data\_dir.glob('sunflowers/\*')),

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

}

In [20]:

flowers\_labels\_dict = {

'roses': 0,

'daisy': 1,

'dandelion': 2,

'sunflowers': 3,

'tulips': 4,

}

In [21]:

flowers\_images\_dict['roses'][:5]

Out[21]:

[PosixPath('datasets/flower\_photos/roses/18741313803\_1bbf842fc6\_n.jpg'),

PosixPath('datasets/flower\_photos/roses/5332550500\_ab341aefd8.jpg'),

PosixPath('datasets/flower\_photos/roses/5897035797\_e67bf68124\_n.jpg'),

PosixPath('datasets/flower\_photos/roses/4979895172\_ca06eba616.jpg'),

PosixPath('datasets/flower\_photos/roses/4495885281\_fe2a3b671d.jpg')]

In [22]:

str(flowers\_images\_dict['roses'][0])

Out[22]:

'datasets/flower\_photos/roses/18741313803\_1bbf842fc6\_n.jpg'

In [23]:

img = cv2.imread(str(flowers\_images\_dict['roses'][0]))

In [24]:

img.shape

Out[24]:

(213, 320, 3)

In [25]:

cv2.resize(img,(224,224)).shape

Out[25]:

(224, 224, 3)

In [26]:

X, y = [], []

for flower\_name, images in flowers\_images\_dict.items():

for image in images:

img = cv2.imread(str(image))

resized\_img = cv2.resize(img,(224,224))

X.append(resized\_img)

y.append(flowers\_labels\_dict[flower\_name])

In [27]:

X = np.array(X)

y = np.array(y)

Train test split

In [28]:

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

Preprocessing the data

In [29]:

X\_train\_scaled = X\_train / 255

X\_test\_scaled = X\_test / 255

Make prediction using pre-trained model on new flowers dataset

In [30]:

X[1].shape

Out[30]:

(224, 224, 3)

In [31]:

plt.axis('off')

plt.imshow(X[0])

Out[31]:

<matplotlib.image.AxesImage at 0x7d4f6650f0a0>



In [32]:

plt.axis('off')

plt.imshow(X[1])

Out[32]:

<matplotlib.image.AxesImage at 0x7d4e45e6c9d0>



In [33]:

plt.axis('off')

plt.imshow(X[2])

Out[33]:

<matplotlib.image.AxesImage at 0x7d4e45ec6ad0>



In [34]:

x0\_resized = cv2.resize(X[0], IMAGE\_SHAPE)

x1\_resized = cv2.resize(X[1], IMAGE\_SHAPE)

x2\_resized = cv2.resize(X[2], IMAGE\_SHAPE)

In [35]:

predicted = classifier.predict(np.array([x0\_resized, x1\_resized, x2\_resized]))

predicted = np.argmax(predicted, axis=1)

predicted

1/1 [==============================] - 3s 3s/step

Out[35]:

array([795, 795, 795])

In [36]:

image\_labels[795]

Out[36]:

'shower curtain'

Now take pre-trained model and retrain it using flowers images

In [37]:

feature\_extractor\_model = "https://tfhub.dev/google/tf2-preview/mobilenet\_v2/feature\_vector/4"

pretrained\_model\_without\_top\_layer = hub.KerasLayer(

feature\_extractor\_model, input\_shape=(224, 224, 3), trainable=False)

In [38]:

num\_of\_flowers = 5

model = tf.keras.Sequential([

pretrained\_model\_without\_top\_layer,

tf.keras.layers.Dense(num\_of\_flowers)

])

model.summary()

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

keras\_layer\_1 (KerasLayer) (None, 1280) 2257984

dense (Dense) (None, 5) 6405

=================================================================

Total params: 2,264,389

Trainable params: 6,405

Non-trainable params: 2,257,984

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [39]:

model.compile(

optimizer="adam",

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

metrics=['acc'])

model.fit(X\_train\_scaled, y\_train, epochs=5)

Epoch 1/5

86/86 [==============================] - 131s 1s/step - loss: 0.8628 - acc: 0.6781

Epoch 2/5

86/86 [==============================] - 122s 1s/step - loss: 0.4156 - acc: 0.8663

Epoch 3/5

86/86 [==============================] - 125s 1s/step - loss: 0.3230 - acc: 0.8975

Epoch 4/5

86/86 [==============================] - 124s 1s/step - loss: 0.2722 - acc: 0.9190

Epoch 5/5

86/86 [==============================] - 122s 1s/step - loss: 0.2394 - acc: 0.9313

Out[39]:

<keras.callbacks.History at 0x7d4f896ed8d0>

In [40]:

model.evaluate(X\_test\_scaled,y\_test)

29/29 [==============================] - 43s 1s/step - loss: 0.3617 - acc: 0.8725

Out[40]:

[0.36165452003479004, 0.8725489974021912]