**UNIVERSITY OF MUMBAI DEPARTMENT OF COMPUTER SCIENCE**

M.Sc. Computer Science – Semester IV **ADVANCE DEEP LEARNING** JOURNAL

2022-2023

SEAT NUMBER

UNIVERSITY OF MUMBAI

**DEPARTMENT OF COMPUTER SCIENCE**

**CERTIFICATE**

This is to certify that the work entered in this journal was done in the University Department of Computer Science laboratory by Mr./Ms. Seat No. for the course of M.Sc. Computer Science - Semester IV during the academic year 2022 - 2023 in a satisfactory manner.

**Subject In-charge Head of Department**

**External Examiner**

# Index

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Content** | **Page No** | **Date** | **Sign** |
| 1 | Implement Feed-forward Neural Network and train the networkwith different optimizers and compare the results. | 1 |  |  |
| 2 | Write a Program to implement regularization to prevent the model from overfitting | 4 |  |  |
| 3 | Implement deep learning for recognizing classes for datasets like CIFAR-10 images for previously unseen images and assign themto one of the 10 classes. | 7 |  |  |
| 4 | Implement deep learning for the Prediction of the autoencoderfrom the test data  (e.g., MNIST data set) | 9 |  |  |
| 5 | Implement Convolutional Neural Network for Digit Recognitionon the MNIST Dataset | 12 |  |  |
| 6 | Write a program to implement Transfer Learning on the suitabledataset (e.g., classify the cats versus dog’s dataset from Kaggle). | 15 |  |  |
| 7 | Write a program for the Implementation of a Generative  Adversarial Network for generating synthetic shapes (like digits) | 21 |  |  |
| 8 | Write a program to implement a simple form of a recurrent neuralnetwork.   1. E.g. (4-to-1 RNN) to show that the quantity of rain on acertain day also depends on the values of the previous day 2. LSTM for sentiment analysis on datasets like UMICH SI650for similar. | 30  33 |  |  |
| 9 | Write a program for object detection from the image/video. | 36 |  |  |
| 10 | Write a program for object detection using pre-trained models touse object detection. | 37 |  |  |

**Practical 1**

**Aim**: Implement Feed-forward Neural Network and train the network with different optimizers and compare the results.

**Theory:** A Feed Forward Neural Network is an artificial neural network in which the connections between nodes does not form a cycle. The opposite of a feed forward neural network is a recurrent neural network, in which certain pathways are cycled. The feed forward model is the simplest form of neural network as information is only processed in one direction. While the data may pass through multiple hidden nodes, it always moves in one direction and never backwards.

A Feed Forward Neural Network is commonly seen in its simplest form as a single layer perceptron. In this model, a series of inputs enter the layer and are multiplied by the weights. Each value is then added together to get a sum of the weighted input values. If the sum of the values is above a specific threshold, usually set at zero, the value produced is often 1, whereas if the sum falls below the threshold, the output value is -1. The single layer perceptron is an important model of feed forward neural networks and is often used in classification tasks. Furthermore, single layer perceptron’s can incorporate aspects of machine learning.

**Code:**

import tensorflow as tfimport numpy as np

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_splitfrom sklearn.preprocessing import LabelBinarizer

# Load Iris dataset

iris = load\_iris() # Loading Iris dataset into a variable.X = iris.data # Features of the dataset.

y = iris.target # Class labels of the dataset.# One-hot encode labels

lb = LabelBinarizer() # Creating an instance of LabelBinarizer class for one-hotencoding. y = lb.fit\_transform(y) # One-hot encoding the class labels.# Split data into train

and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,test\_size=0.2,random\_state=42)

# Splitting the dataset into training and testing sets with test size of 20%.# Define model architecture model = tf.keras.Sequential([

# First hidden layer with 16 neurons and input shape of 4 features. ReLUactivation function is used. tf.keras.layers.Dense(16, input\_shape=(4,), activation='relu'),

# Second hidden layer with 8 neurons. ReLU activation function is used.tf.keras.layers.Dense(8, activation='relu'),

# Output layer with 3 neurons for 3 classes. Softmax activation functionis used for multiclass␣classification task.

tf.keras.layers.Dense(3, activation='softmax')

])

# Compile model with different optimizersoptimizers = ['sgd', 'adam', 'rmsprop']

# List of optimizers to be used for training the model.

for optimizer in optimizers: # Looping over each optimizer.

# Compiling the model with 'categorical\_crossentropy' as the loss function,the current optimizer and accuracy as the metric to be calculated.model.compile(loss='categorical\_crossentropy', optimizer=optimizer,

metrics=['accuracy'])# Train model

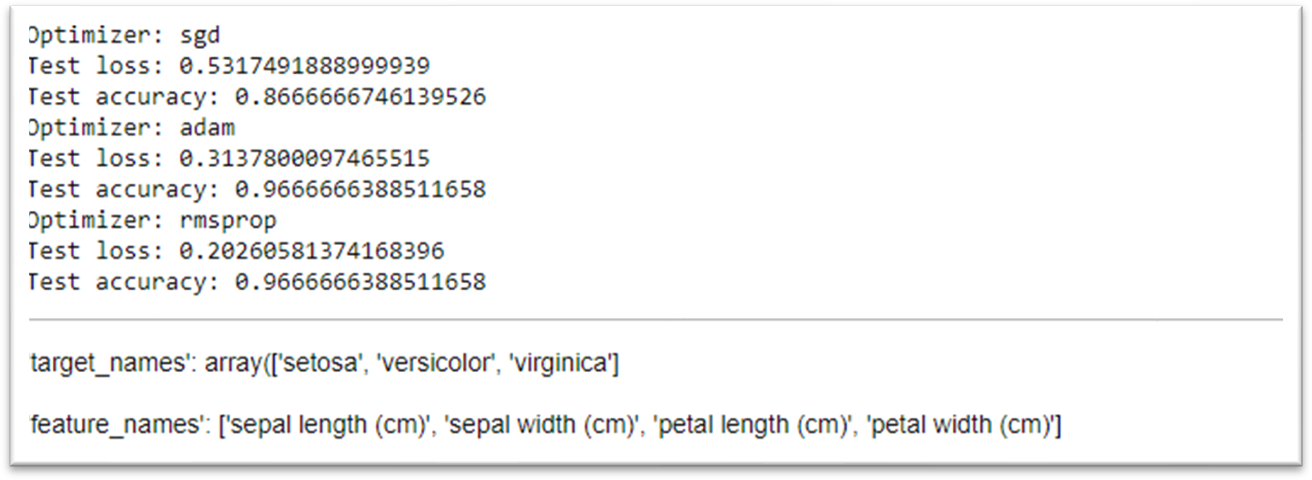
history = model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test),epochs=50, verbose=0) # Training the model for 50 epochs with verbose=0 to suppress the output.

# Evaluate model

loss, accuracy = model.evaluate(X\_test, y\_test, verbose=0) # Evaluating themodel on the test set and calculating the loss and accuracy.

print('Optimizer:', optimizer) # Printing the optimizer name. print('Test loss:', loss) # Printing the loss value on the test set.

print('Test accuracy:', accuracy) # Printing the accuracy value on the testset.



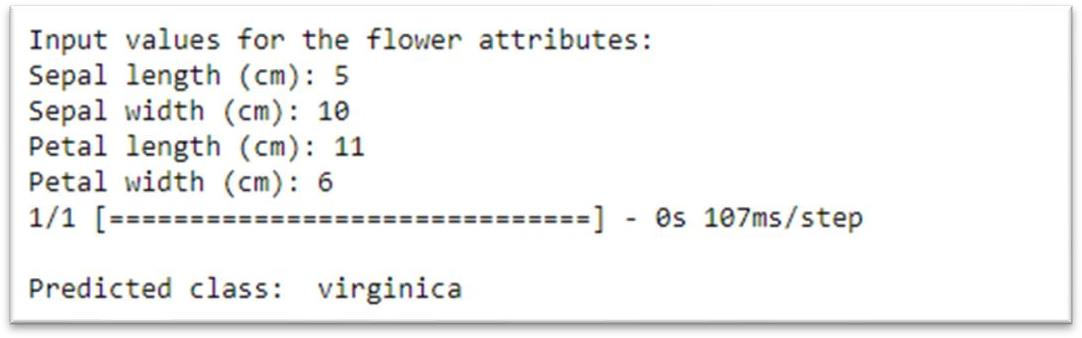
# Allow user to input values for the flower attributesprint('\nInput values for the flower attributes:') sepal\_length = float(input('Sepal length (cm): ')) sepal\_width = float(input('Sepal width (cm): ')) petal\_length = float(input('Petal length (cm): ')) petal\_width = float(input('Petal width (cm): '))

# Predict class of flower based on input values

input\_values = np.array([[sepal\_length, sepal\_width, petal\_length,petal\_width]])prediction = model.predict(input\_values)

predicted\_class = np.argmax(prediction)class\_names = iris.target\_names

print('\nPredicted class: ', class\_names[predicted\_class])



#memory optimizers = {

'sgd': tf.keras.optimizers.SGD(), 'adam': tf.keras.optimizers.Adam(), 'rmsprop': tf.keras.optimizers.RMSprop()

}

# Compile model with different optimizers

for optimizer\_name, optimizer in optimizers.items(): model.compile(loss='categorical\_crossentropy', optimizer=optimizer,metrics=['accuracy'])

# Train model

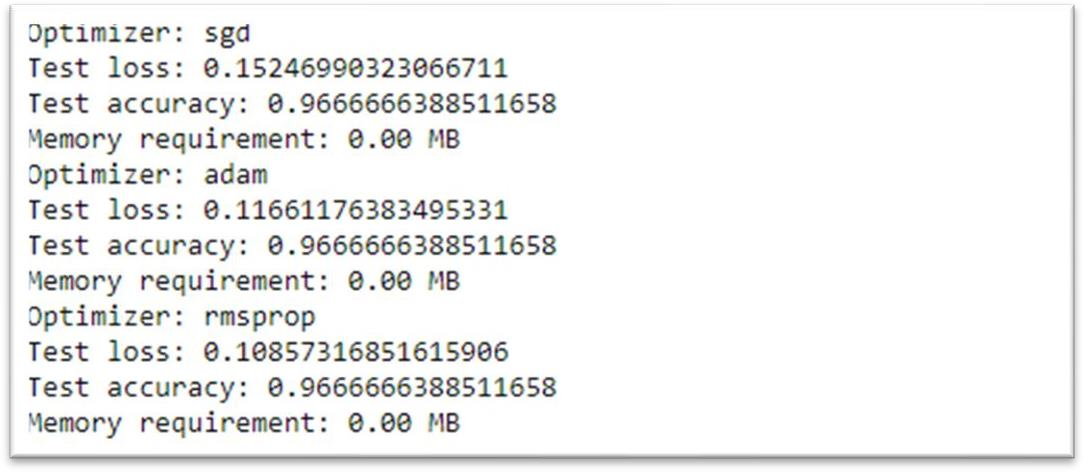
history = model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test),epochs=50, verbose=0) # Evaluate model

loss, accuracy = model.evaluate(X\_test, y\_test, verbose=0)print('Optimizer:', optimizer\_name) print('Test loss:', loss) print('Test accuracy:',

accuracy)# Estimate memory requirement

size\_in\_bytes = model.count\_params() \* 4 # each parameter is a 32-bit floatsize\_in\_mb = size\_in\_bytes / (1024 \* 1024)

print(f'Memory requirement: {size\_in\_mb:.2f} MB')



# Practical 2

**Aim**: Program to implement regularization to prevent the model from overfitting.

**Theory:** Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model’s performance on the unseen data as well. L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

Cost function = Loss (say, binary cross entropy) + Regularization term

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent. However, this regularization term differs in L1 and L2.

In L2, we have:

Here, lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero).

In L1, we have:

In this, we penalize the absolute value of the weights. Unlike L2, the weights may be reduced to zero here. Hence, it is very useful when we are trying to compress our model. Otherwise, we usually prefer L2 over it.

**Code:**

# Import TensorFlow libraryimport tensorflow as tf

# Load the data # Load MNIST dataset'''

loads the MNIST dataset using the load\_data() function provided by Keras, a high-level API of TensorFlow.

The MNIST dataset contains 70,000 images of handwritten digits that are split into60,000 training images and 10,000 testing images.

'''

(train\_data, train\_labels), (test\_data, test\_labels) = tf.keras.datasets.mnist.

load\_data()

# Preprocess the data'''

Preprocess the data. The images are first reshaped from a 3D array (28x28 pixels)to a 2D array (784 pixels). Then, the pixel values are normalized to be between 0 and 1 by dividing by 255. The labels are converted to one-hot encoding format using the to\_categorical()function provided by Keras. This is done to make it easier for the model to classifythe images into 10 different classes (one for each digit).

'''

# Reshape and normalize training data

train\_data = train\_data.reshape((60000, 784)) / 255.0# Reshape and normalize testing data

test\_data = test\_data.reshape((10000, 784)) / 255.0# Convert training labels to one-hot encoding

train\_labels = tf.keras.utils.to\_categorical(train\_labels)# Convert testing labels to one-hot encoding

test\_labels = tf.keras.utils.to\_categorical(test\_labels)

# Define the model architecture'''

This code defines the architecture of the neural network model. The Sequential () function is used to create a sequential model, meaning that the layers are added insequence. Three fully connected layers are defined using the Dense () function.

The first layer has 128 units, ReLU activation, and L2 regularization with a regularization parameter of 0.01. The second layer has 64 units, ReLU activation, and L2 regularization with a regularization parameter of 0.01. The third and final layer has 10 units, softmax activation, and is used for the classification task.

'''

model = tf.keras.models.Sequential([ # Define sequential model

#Add a fully connected layer with 128 units, ReLU activation, and L2regularization tf.keras.layers.Dense(128, activation='relu', input\_shape=(784,), kernel\_regularizer=tf.keras.regularizers.l2(0.01)),

# Add another fully connected layer with 64 units,ReLU activation, and L2regularization tf.keras.layers.Dense(64, activation='relu', kernel\_regularizer=tf.keras.regularizers.l2(0.01)), # Add a final output layer with 10 units (one for each class), softmaxactivation tf.keras.layers.Dense(10, activation='softmax')

])

# Compile the model'''

This code compiles the model. The compile () function configures the model for training by specifying the optimizer, loss function, and metrics to monitor duringtraining. In this case, the Adam optimizer is used with a learning rate of 0.001,categorical cross-entropy is used as the loss function, and accuracy is monitoredduring training.

'''

model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=0.001),# Use Adam optimizer with learning rate 0.001

loss='categorical\_crossentropy',

# Use categorical cross-entropy loss function metrics=['accuracy']) # Monitor accuracy during training

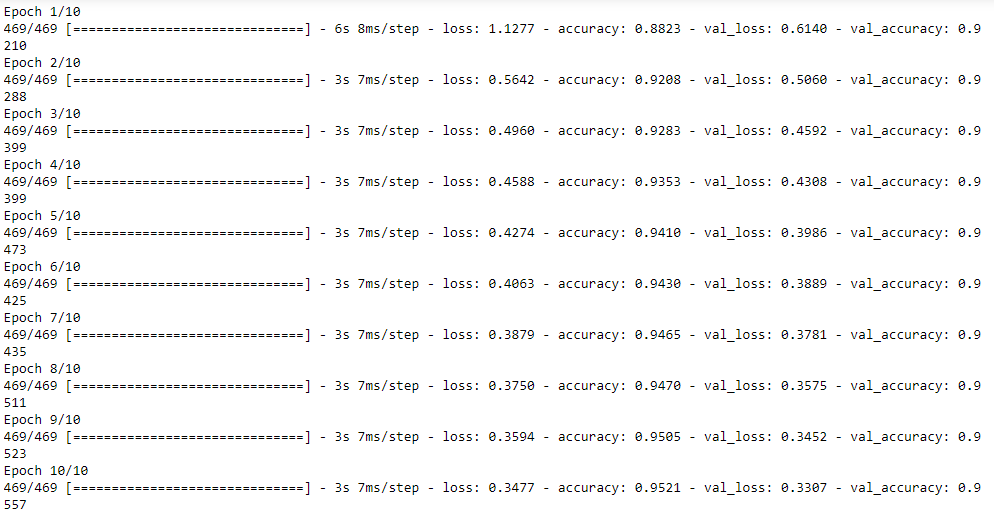
# Train the model'''

This code trains the model using the fit () function. The training data and labelsare passed in as arguments, along with the number of epochs to train for, the batchsize to use, and the validation data to use for monitoring model performance during training. The fit () function returns a history object that contains information about the training process, such as the loss and accuracy at each epoch. The purposeof this program is to demonstrate how to implement a neural network model for image classification using TensorFlow/Keras. The model uses regularization techniques toprevent overfitting and achieves high accuracy on the MNIST dataset.

'''

history = model.fit (train\_data, train\_labels, epochs=10, batch\_size=128, # Train the model for 10 epochs, using batches of size 128, andvalidate on the testing data at the end of each epoch

validation\_data= (test\_data, test\_labels))



# Practical 3

**Aim**: Implement deep learning for recognizing classes for datasets like CIFAR-10 images for previously unseen images and assign them to one of the 10 classes.

**Theory:** The CIFAR-10 dataset (Canadian Institute for Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

Computer algorithms for recognizing objects in photos often learn by example. CIFAR-10 is a set of images that can be used to teach a computer how to recognize objects. Since the images in CIFAR-10 are low-resolution (32x32), this dataset can allow researchers to quickly try different algorithms to see what works.

CIFAR-10 is a labeled subset of the 80 million Tiny Images dataset from 2008, published in 2009. When the dataset was created, students were paid to label all of the images. Various kinds of convolutional neural networks tend to be the best at recognizing the images in CIFAR-10.

**Code:**

import tensorflow as tf from tensorflow import keras

from tensorflow.keras import layers

# Load the data

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

# Preprocess the data

x\_train = x\_train.astype("float32") / 255.0x\_test = x\_test.astype("float32") / 255.0

# Convert labels to one-hot encoding format y\_train = keras.utils.to\_categorical(y\_train, 10)y\_test = keras.utils.to\_categorical(y\_test, 10)

# Define the model architecturemodel = keras.Sequential([

keras.Input(shape=(32, 32, 3)),

layers.Conv2D(32, kernel\_size=(3, 3), activation="relu"),layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation="relu"),layers.MaxPooling2D(pool\_size=(2, 2)), layers.Flatten(),

layers.Dropout(0.5),

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

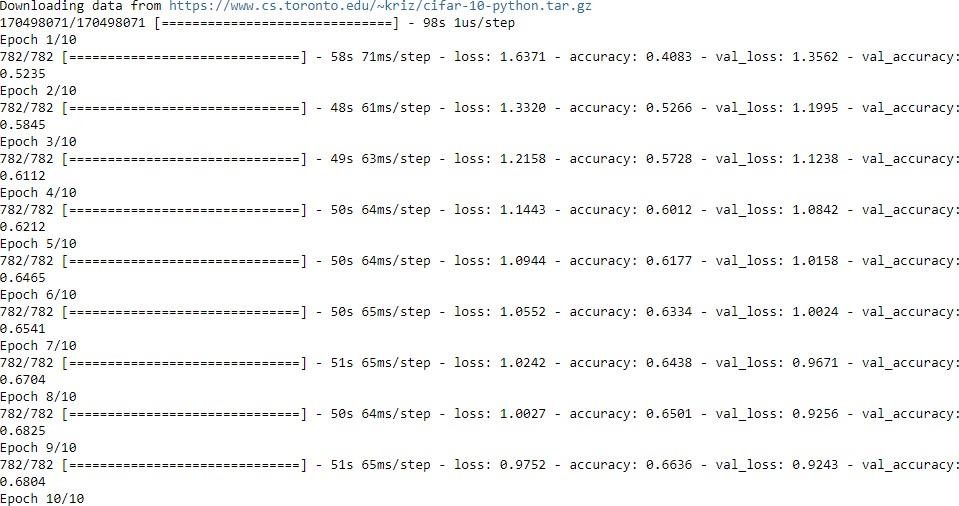
])

# Compile the model model.compile(loss="categorical\_crossentropy",optimizer="adam",metrics=["accuracy"

])

# Train the model model.fit(x\_train,y\_train,batch\_size=64,epochs=10,validation\_data=(x\_test,y\_test))

# Save the trained model to a file model.save("cifar10\_model.h5")



import numpy as np from PIL import Image

# Load the saved model

model = keras.models.load\_model("cifar10\_model.h5")

# Load and preprocess the test imageimg = Image.open("two.png")

img = img.resize((32, 32)) img\_array = np.array(img)

img\_array = img\_array.astype("float32") / 255.0img\_array = np.expand\_dims(img\_array, axis=0)

# Make predictions on the test image predictions = model.predict(img\_array)# Get the predicted class label class\_label = np.argmax(predictions)

# Print the predicted class label print("Predicted class label:", class\_label)

# Practical 4

**Aim**: Implement deep learning for the Prediction of the autoencoder from the test data (e.g., MNIST (data set)

**Theory:** An autoencoder is a special type of neural network that is trained to copy its input to its output. For example, given an image of a handwritten digit, an autoencoder first encodes the image into a lower dimensional latent representation, then decodes the latent representation back to an image. An autoencoder learns to compress the data while minimizing the reconstruction error.

The encoder part of the network is used for encoding and sometimes even for data compression purposes although it is not very effective as compared to other general compression techniques like JPEG. Encoding is achieved by the encoder part of the network which has a decreasing number of hidden units in each layer. Thus, this part is forced to pick up only the most significant and representative features of the data. The second half of the network performs the Decoding function. This part has an increasing number of hidden units in each layer and thus tries to reconstruct the original input from the encoded data. Thus Auto-encoders are an unsupervised learning technique.

**Code:**

This program first loads the MNIST dataset and pre-processes it. It then defines the encoder and decoder architectures and combines them into an autoencoder model. The autoencoder model is compiled and trained on the training data. The program then uses the trained autoencoder to predict the reconstructed images for the test data. The reconstructed images are plotted alongside the original test images for comparison. Note that in this program, we’re not using the labels of the MNIST dataset since we’re only interested in reconstructing the input images. Also, the loss function used in the autoencoder is binary crossentropy, since we’re treating each pixel value as a binary classification problem (i.e., is the pixel on or off?). Finally, the images are plotted using the matplotlib library.

import tensorflow as tf from tensorflow import kerasimport numpy as np

import matplotlib.pyplot as plt

# Load the MNIST dataset

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

# Normalize the pixel values to be between 0 and 1x\_train = x\_train.astype("float32") / 255.0 x\_test = x\_test.astype("float32") / 255.0

# Define the encoder architecture encoder = keras.models.Sequential([

keras.layers.Flatten(input\_shape=[28, 28]), keras.layers.Dense(128, activation="relu"), keras.layers.Dense(64, activation="relu"), keras.layers.Dense(32, activation="relu"),

])

# Define the decoder architecture decoder = keras.models.Sequential([

keras.layers.Dense(64, activation="relu", input\_shape=[32]),keras.layers.Dense(128, activation="relu"), keras.layers.Dense(28 \* 28, activation="sigmoid"), keras.layers.Reshape([28, 28]),

])

# Combine the encoder and decoder into an autoencoder modelautoencoder = keras.models.Sequential([encoder, decoder])

# Compile the autoencoder model autoencoder.compile(loss="binary\_crossentropy", optimizer=keras.optimizers.Adam(learning\_rate=0.001))

# Train the autoencoder model

history = autoencoder.fit(x\_train, x\_train, epochs=10, batch\_size=128,validation\_data=(x\_test, x\_test))

# Use the trained autoencoder to predict the reconstructed images for the test datadecoded\_imgs = autoencoder.predict(x\_test)

#Plot some of the original test images and their reconstructed counterpartsn = 10 # number of images to display

plt.figure(figsize=(20, 4))for i in range(n):

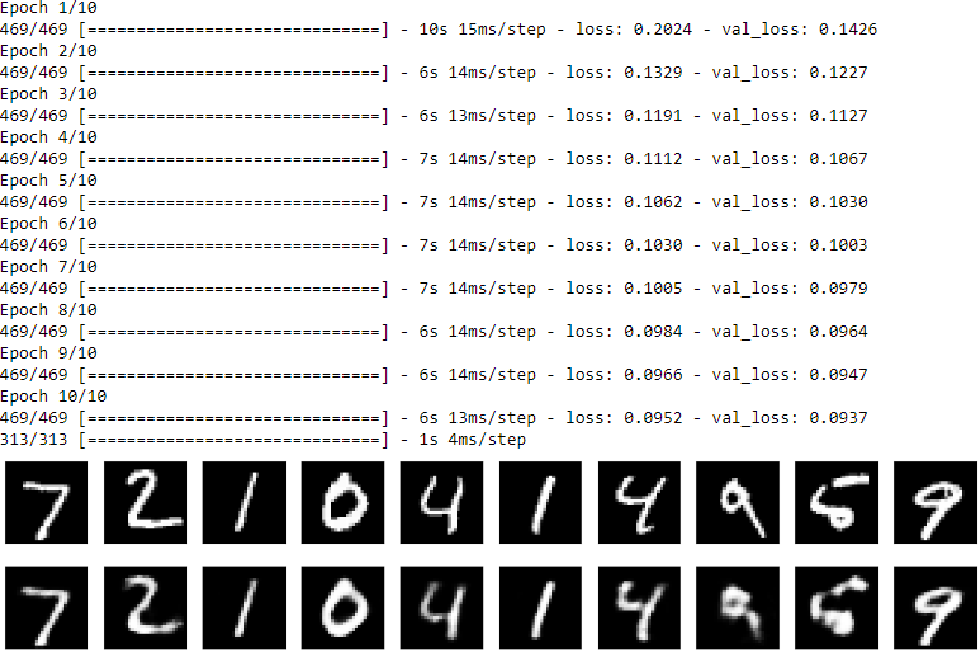
# Display original images ax = plt.subplot(2, n, i + 1)

plt.imshow(x\_test[i]) plt.gray() ax.get\_xaxis().set\_visible(False)ax.get\_yaxis().set\_visible(False)

# Display reconstructed images ax = plt.subplot(2, n, i + n + 1)

plt.imshow(decoded\_imgs[i]) plt.gray() ax.get\_xaxis().set\_visible(False) ax.get\_yaxis().set\_visible(False)

plt.show()



# Practical 5

**Aim**: Implement Convolutional Neural Network for Digit Recognition on the MNIST Dataset.

**Theory:** A Convolutional Neural Network (CNN) is a type of deep learning algorithm that is particularly well-suited for image recognition and processing tasks. It is made up of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

The convolutional layers are the key component of a CNN, where filters are applied to the input image to extract features such as edges, textures, and shapes. The output of the convolutional layers is then passed through pooling layers, which are used to down-sample the feature maps, reducing the spatial dimensions while retaining the most important information. The output of the pooling layers is then passed through one or more fully connected layers, which are used to make a prediction or classify the image.

Convolutional Neural Network Design:

* The construction of a convolutional neural network is a multi-layered feed-forward neural network, made by assembling many unseen layers on top of each other in a particular order.
* It is the sequential design that give permission to CNN to learn hierarchical attributes.
* In CNN, some of them followed by grouping layers and hidden layers are typically convolutional layers followed by activation layers.
* The pre-processing needed in a ConvNet is kindred to that of the related pattern of neurons in the human brain and was motivated by the organization of the Visual Cortex.

**Code:**

import tensorflow as tf from tensorflow import kerasimport numpy as np

import matplotlib.pyplot as plt

# Load the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

# Preprocess the data

x\_train = x\_train.astype("float32") / 255.0x\_test = x\_test.astype("float32") / 255.0 x\_train = np.expand\_dims(x\_train, -1) x\_test = np.expand\_dims(x\_test, -1)

# Define the CNN architecture model = keras.models.Sequential([

keras.layers.Conv2D(32, (3, 3), activation="relu", input\_shape=(28, 28,

1)),

keras.layers.MaxPooling2D((2, 2)),

keras.layers.Conv2D(64, (3, 3), activation="relu"),

keras.layers.MaxPooling2D((2, 2)),

keras.layers.Flatten(), keras.layers.Dense(64, activation="relu"), keras.layers.Dense(10, activation="softmax")

])

# Compile the model

model.compile(optimizer="adam", loss="sparse\_categorical\_crossentropy",metrics=["accuracy"])

# Train the model

history = model.fit(x\_train, y\_train, epochs=10, batch\_size=128,validation\_data=(x\_test, y\_test))

# Evaluate the model on the test data

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)print("Test accuracy:", test\_acc)

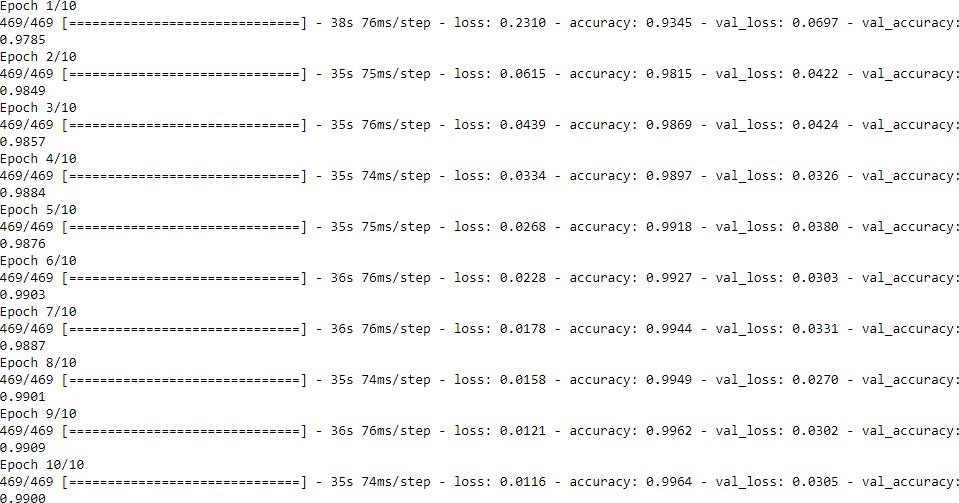
# Show predictions for a sample input imagesample\_img = x\_test[0]

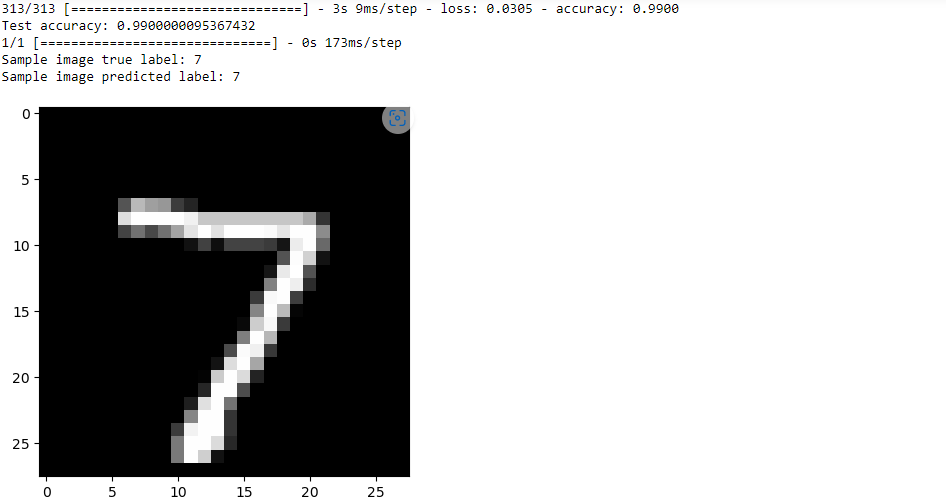
sample\_label = y\_test[0]

sample\_img = np.expand\_dims(sample\_img, 0)pred = model.predict(sample\_img) pred\_label = np.argmax(pred)

print("Sample image true label:", sample\_label) print("Sample image predicted label:", pred\_label)

# Display the sample image plt.imshow(sample\_img.squeeze(), cmap='gray')plt.show()





# Practical 6

**Aim**: Implement Transfer Learning on the suitable public dataset (e.g., classify the cats versus dog’s dataset from Kaggle or UCI or inbuilt dataset).

**Theory:** Transfer learning is a machine learning (ML) method that reuses a trained model designed for a particular task to accomplish a different yet related task. The knowledge acquired from task one is thereby transferred to the second model that focuses on the new task.

The term ‘transfer learning’ is related to human psychology. For example, consider an individual who is an expert guitarist. It is quite easy for him to learn to play other stringed instruments, such as a sitar or mandolin, compared to someone with no experience playing any musical instrument.

Transfer learning speeds up the overall process of training a new model and consequently improves its performance. It is primarily used when a model requires large amount of resources and time for training. Due to these reasons, transfer learning is employed in several deep learning projects, such as neural networks that accomplish NLP or CV tasks, such as sentiment analysis.

**Code:**

import tensorflow as tfimport numpy as np

import matplotlib.pyplot as pltimport os import zipfile

from tensorflow.keras.preprocessing.image import ImageDataGeneratorfrom tensorflow.keras.applications import VGG16

# Download and extract dataset

url = "https://storage.googleapis.com/mledu-datasets/cats\_and\_dogs\_filtered.zip"filename = os.path.join(os.getcwd(), "cats\_and\_dogs\_filtered.zip") tf.keras.utils.get\_file(filename, url)

with zipfile.ZipFile("cats\_and\_dogs\_filtered.zip", "r") as zip\_ref:zip\_ref.extractall() # Define data generators

train\_dir = os.path.join(os.getcwd(), "cats\_and\_dogs\_filtered", "train") validation\_dir = os.path.join(os.getcwd(), "cats\_and\_dogs\_filtered","validation")

train\_datagen = ImageDataGenerator(rescale=1./255,

rotation\_range=20, width\_shift\_range=0.2, height\_shift\_range=0.2, shear\_range=0.2, zoom\_range=0.2, horizontal\_flip=True)

validation\_datagen = ImageDataGenerator(rescale=1./255)

train\_generator = train\_datagen.flow\_from\_directory(train\_dir,

target\_size=(150, 150), batch\_size=20, class\_mode="binary")

validation\_generator = validation\_datagen.flow\_from\_directory(validation\_dir, target\_size=(150,150),batch\_size=20,class\_mode="binary")

# Load pre-trained VGG16 model conv\_base = VGG16(weights="imagenet",

include\_top=False, input\_shape=(150, 150, 3))

# Freeze convolutional base layersconv\_base.trainable = False

# Build model on top of the convolutional basemodel = tf.keras.models.Sequential() model.add(conv\_base) model.add(tf.keras.layers.Flatten()) model.add(tf.keras.layers.Dense(256, activation="relu"))

model.add(tf.keras.layers.Dropout(0.5)) model.add(tf.keras.layers.Dense(1, activation="sigmoid"))

# Compile model model.compile(loss="binary\_crossentropy",

optimizer=tf.keras.optimizers.RMSprop(learning\_rate=2e-5),metrics=["accuracy"])

# Train model

history = model.fit(train\_generator,

steps\_per\_epoch=100,epochs=30, validation\_data=validation\_generator, validation\_steps=50)

# Show sample input and its predicted classx, y\_true = next(validation\_generator) y\_pred = model.predict(x) class\_names = ['cat', 'dog']for i in

range(len(x)):

plt.imshow(x[i])

plt.title(f'Predicted class: {class\_names[int(round(y\_pred[i][0]))]}, Trueclass:

{class\_names[int(y\_true[i])]}') plt.show()

# Plot accuracy and loss over timeacc = history.history["accuracy"]

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

loss = history.history["loss"] val\_loss = history.history["val\_loss"]

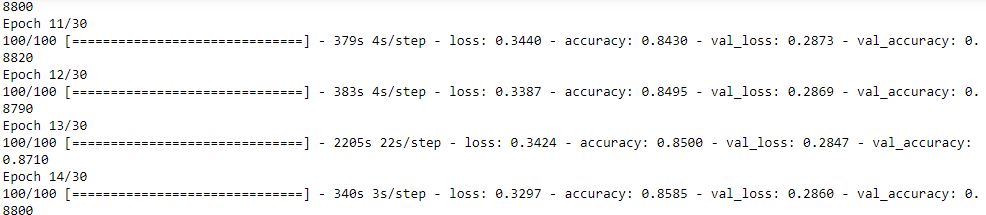
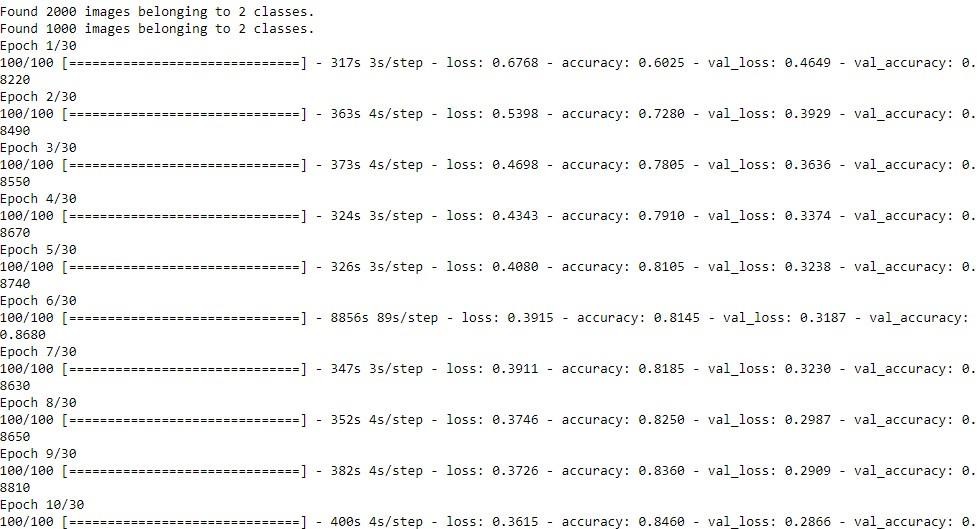
epochs = range(1, len(acc) + 1)

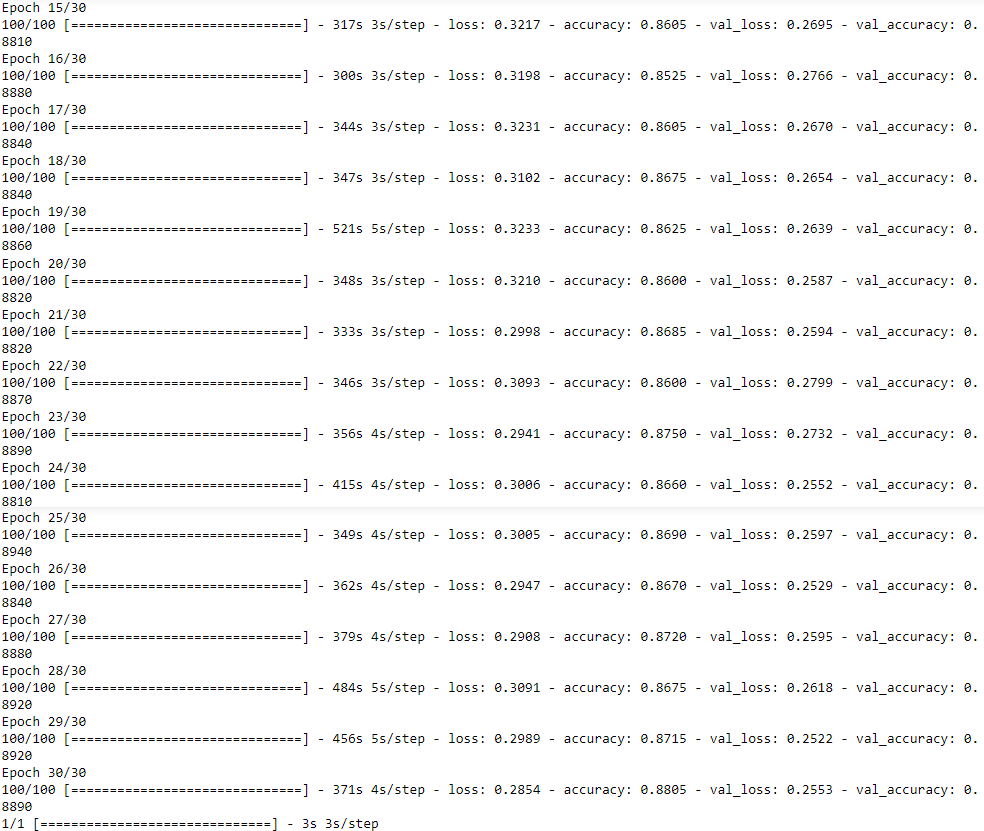
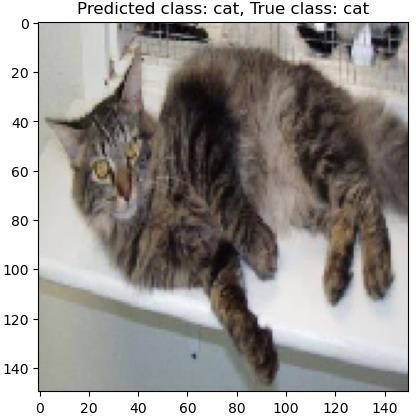
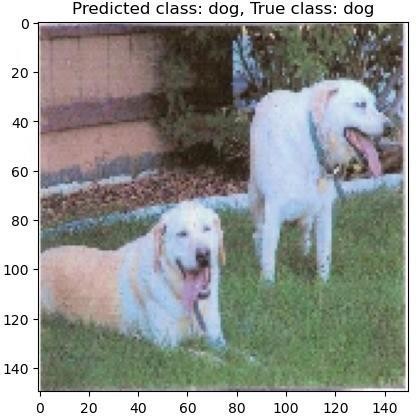
plt.plot(epochs, acc, "bo", label="Training acc") plt.plot(epochs, val\_acc, "b", label="Validation acc")plt.title("Training and validation accuracy") plt.legend()

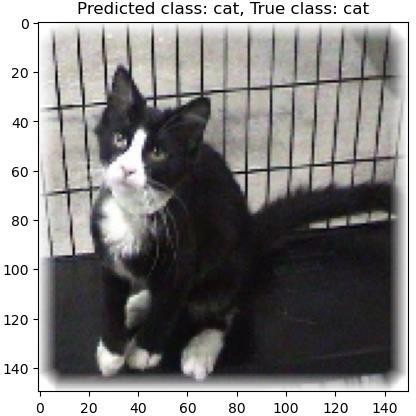
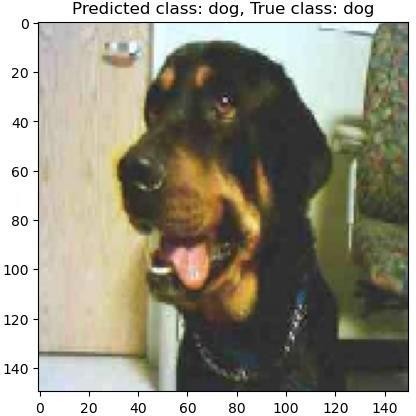
plt.figure()

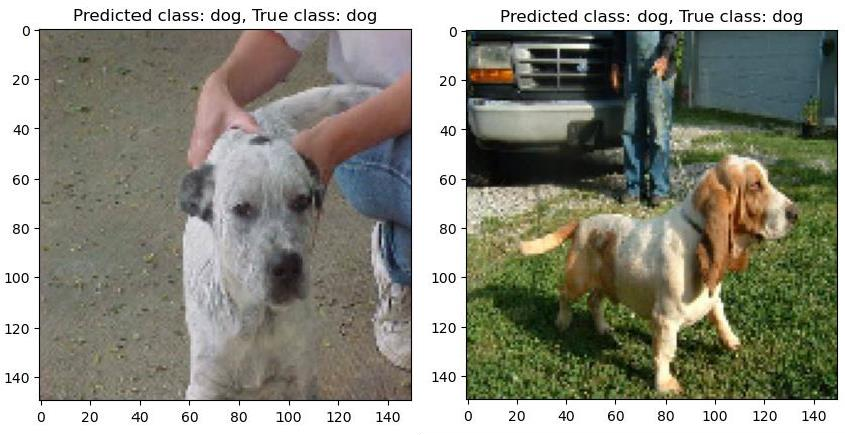
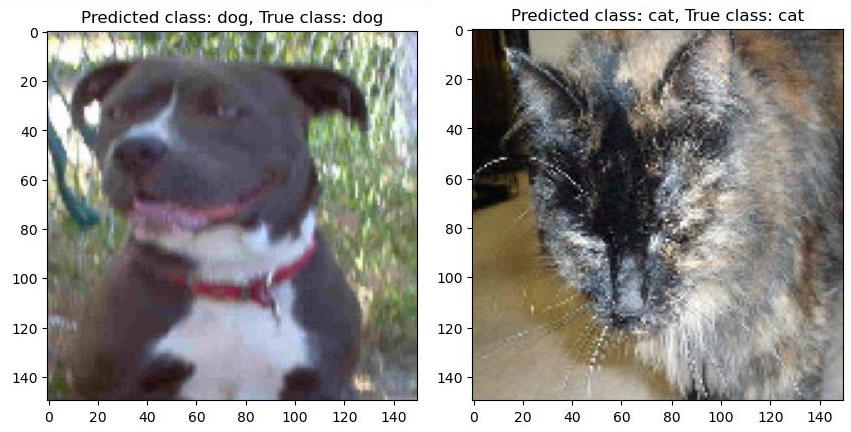
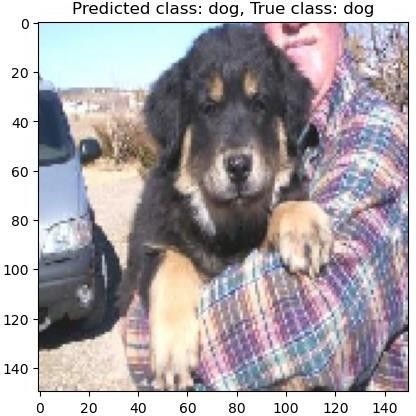
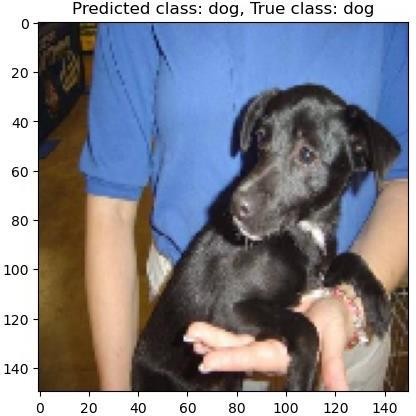
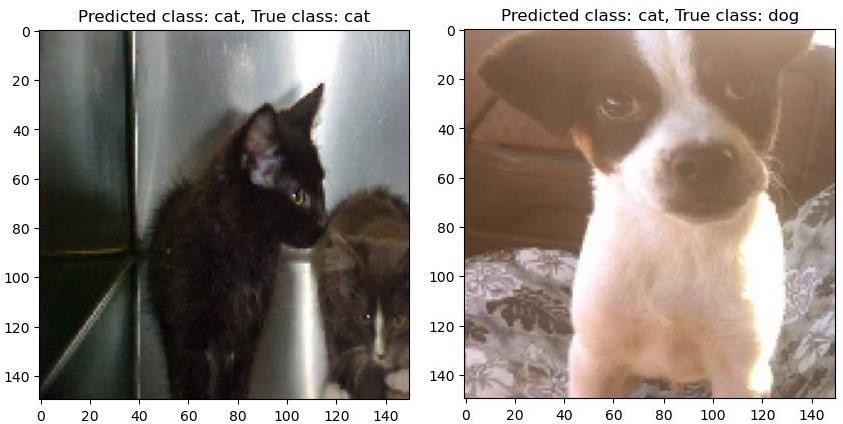
plt.plot(epochs, loss, "bo", label="Training loss") plt.plot(epochs, val\_loss, "b", label="Validation loss")plt.title("Training and validation loss")

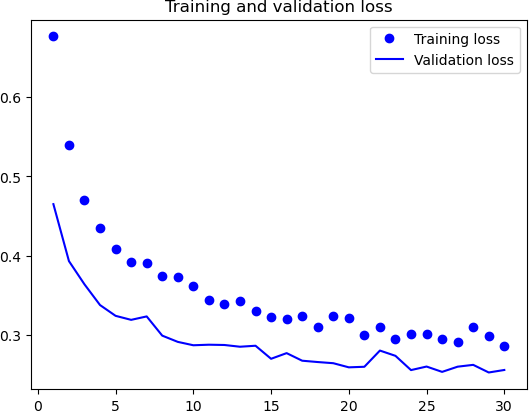
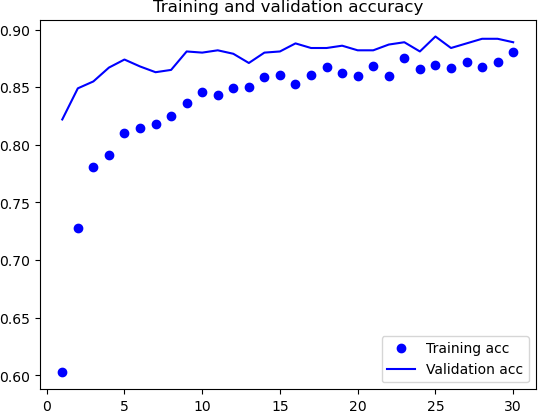
plt.legend() plt.show()









# Practical 7

**Aim**: Write a program for the Implementation of a Generative Adversarial Network for generating synthetic shapes (like digits).

**Theory:** A generative adversarial network (GAN) is a class of machine learning frameworks and a prominent framework for approaching generative AI. In a GAN, two neural networks contest with each other in the form of a zero-sum game, where one agent's gain is another agent's loss.

Given a training set, this technique learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics. Though originally proposed as a form of generative model for unsupervised learning, GANs have also proved useful for semi-supervised learning, fully supervised learning, and reinforcement learning.

The core idea of a GAN is based on the "indirect" training through the discriminator, another neural network that can tell how "realistic" the input seems, which itself is also being updated dynamically. This means that the generator is not trained to minimize the distance to a specific image, but rather to fool the discriminator. This enables the model to learn in an unsupervised manner.

**Code:**

import tensorflow as tfimport numpy as np

import matplotlib.pyplot as plt

# Load the MNIST dataset

(train\_images, \_), (\_, \_) = tf.keras.datasets.mnist.load\_data() train\_images = train\_images.reshape(train\_images.shape[0], 28, 28,1).astype('float32') train\_images = (train\_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

# Define the generator model generator = tf.keras.Sequential([

tf.keras.layers.Dense(7\*7\*256, use\_bias=False, input\_shape=(100,)), tf.keras.layers.BatchNormalization(),

tf.keras.layers.LeakyReLU(), tf.keras.layers.Reshape((7, 7,

256)),

tf.keras.layers.Conv2DTranspose(128, (5, 5), strides=(1,1),padding='same', use\_bias=False), tf.keras.layers.BatchNormalization(), tf.keras.layers.LeakyReLU(), tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),

padding='same',use\_bias=False),

tf.keras.layers.BatchNormalization(), tf.keras.layers.LeakyReLU(), tf.keras.layers.Conv2DTranspose(32, (5, 5), strides=(2, 2),

padding='same',use\_bias=False),

tf.keras.layers.BatchNormalization(), tf.keras.layers.LeakyReLU(), tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),

padding='same',use\_bias=False, activation='tanh')

])

# Define the discriminator model discriminator = tf.keras.Sequential([

tf.keras.layers.Conv2D(32, (5, 5), strides=(2, 2),

padding='same',input\_shape=[28, 28, 1]),

tf.keras.layers.LeakyReLU(), tf.keras.layers.Dropout(0.3),

tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same'), tf.keras.layers.LeakyReLU(),

tf.keras.layers.Dropout(0.3), tf.keras.layers.Conv2D(128, (5, 5),

strides=(2, 2),

padding='same'),

tf.keras.layers.LeakyReLU(),tf.keras.layers.Dropout(0.3), tf.keras.layers.Flatten(), tf.keras.layers.Dense(1)

])

# Define the loss functions and optimizers

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)

def discriminator\_loss(real\_output, fake\_output):

real\_loss = cross\_entropy(tf.ones\_like(real\_output), real\_output)fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)total\_loss = real\_loss + fake\_loss return total\_loss

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

generator\_optimizer = tf.keras.optimizers.Adam(1e-4) discriminator\_optimizer = tf.keras.optimizers.Adam(1e-4)

# Define the training loopEPOCHS

= 50

noise\_dim = 100

num\_examples\_to\_generate = 16

seed = tf.random.normal([num\_examples\_to\_generate, noise\_dim])

@tf.function

def train\_step(images):

noise = tf.random.normal([BATCH\_SIZE, noise\_dim])

with tf.GradientTape() as gen\_tape, tf.GradientTape() as disc\_tape:generated\_images = generator(noise, training=True)

real\_output = discriminator(images, training=True) fake\_output = discriminator(generated\_images, training=True)gen\_loss = generator\_loss(fake\_output)

disc\_loss = discriminator\_loss(real\_output, fake\_output)

gradients\_of\_generator = gen\_tape.gradient(gen\_loss,generator.trainable\_variables) gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss,discriminator.trainable\_variables) generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator,generator.trainable\_variables))

# Apply gradients to the discriminator variables discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator,discriminator.trainable\_variables))

# Train the generator

with tf.GradientTape() as gen\_tape:

# Generate fake images using the generator generated\_images = generator(noise, training=True)

# Get discriminator's prediction of the generated images gen\_preds = discriminator(generated\_images, training=False)# Calculate generator's loss gen\_loss = generator\_loss(gen\_preds)

# Get gradients of the generator loss with respect to the generator variablesgradients\_of\_generator = gen\_tape.gradient(gen\_loss,

generator.trainable\_variables)

# Apply gradients to the generator variables generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.trainable\_variables))

# Print the losses

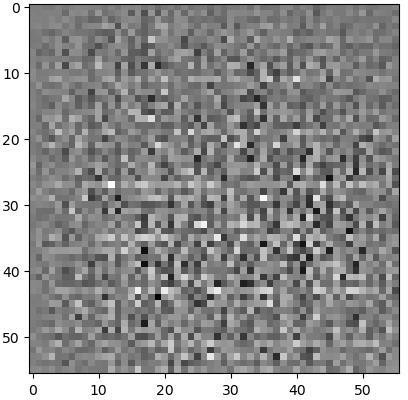
print("Discriminator loss:", disc\_loss.numpy(), "Generator loss:",gen\_loss.numpy())

# Save checkpoint ckpt\_manager.save()

# Generate and save 10 random images from the generator after trainingNOISE\_DIM = 100 for i in range(10):

noise = tf.random.normal([1, NOISE\_DIM]) generated\_images =

generator(noise, training=False)img = tf.squeeze(generated\_images[0]) plt.imshow(img, cmap='gray') plt.savefig(f'generated\_image\_{i}.png')



import tensorflow as tfimport numpy as np

import matplotlib.pyplot as plt

# Check if TensorFlow is able to detect a GPUprint(tf.config.list\_physical\_devices('GPU')) # Set the GPU device to use

device\_name = '/device:GPU:0'

mnist = tf.keras.datasets.mnist

(train\_images, train\_labels), (\_, \_) = mnist.load\_data()

# Normalize the images to [-1, 1]

train\_images = (train\_images.astype('float32') - 127.5) / 127.5

# Reshape the images to (28, 28, 1) and add a channel dimensiontrain\_images = np.expand\_dims(train\_images, axis=-1)

# Batch and shuffle the data BUFFER\_SIZE = 60000

BATCH\_SIZE = 256

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE) def make\_generator\_model(): model =

tf.keras.Sequential()

model.add(tf.keras.layers.Dense(7\*7\*256, use\_bias=False,input\_shape=(100,))) model.add(tf.keras.layers.BatchNormalization()) model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Reshape((7, 7, 256)))

assert model.output\_shape == (None, 7, 7, 256)

model.add(tf.keras.layers.Conv2DTranspose(128, (5, 5), strides=(1,1),padding='same', use\_bias=False))

assert model.output\_shape == (None, 7, 7, 128) model.add(tf.keras.layers.BatchNormalization())model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),padding='same', use\_bias=False))

assert model.output\_shape == (None, 14, 14, 64) model.add(tf.keras.layers.BatchNormalization()) model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2,2),padding='same', use\_bias=False, activation='tanh'))

assert model.output\_shape == (None, 28, 28, 1)return model

def make\_discriminator\_model(): model = tf.keras.Sequential()

model.add(tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2),padding='same',

input\_shape=[28, 28, 1])) model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Dropout(0.3)) model.add(tf.keras.layers.Conv2D(128, (5, 5), strides=(2, 2),padding='same')) model.add(tf.keras.layers.LeakyReLU())

model.add(tf.keras.layers.Dropout(0.3)) model.add(tf.keras.layers.Flatten()) model.add(tf.keras.layers.Dense(1))

return model

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)def discriminator\_loss(real\_output, fake\_output):

real\_loss = cross\_entropy(tf.ones\_like(real\_output), real\_output)fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)total\_loss = real\_loss + fake\_loss return total\_loss

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

# Define the models

generator = make\_generator\_model() discriminator = make\_discriminator\_model()

# Define the optimizers

generator\_optimizer = tf.keras.optimizers.Adam(1e-4) discriminator\_optimizer = tf.keras.optimizers.Adam(1e-4)

# Define the training loopEPOCHS

= 100

noise\_dim = 100

num\_examples\_to\_generate = 16

@tf.function

def train\_step(images): #Generate noise

noise = tf.random.normal([BATCH\_SIZE, noise\_dim])

with tf.GradientTape() as gen\_tape, tf.GradientTape() as disc\_tape:#Generate fake images generated\_images = generator(noise, training=True)# Evaluate

discriminator on real and fake images real\_output = discriminator(images, training=True)

fake\_output = discriminator(generated\_images, training=True)

# Calculate the losses

gen\_loss = generator\_loss(fake\_output)

disc\_loss = discriminator\_loss(real\_output, fake\_output)

gradients\_of\_generator = gen\_tape.gradient(gen\_loss,generator.trainable\_variables) gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss,discriminator.trainable\_variables) generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator,generator.trainable\_variables))

# Apply gradients to the discriminator variables

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator,discriminator.trainable\_variables)) def generate\_and\_save\_images(model, epoch, test\_input):# Generate images

from the model

predictions = model(test\_input, training=False)# Rescale to [0, 1] predictions = (predictions + 1) / 2.0

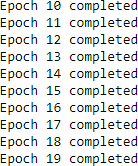
# Plot the images

fig = plt.figure(figsize=(4, 4))

for i in range(predictions.shape[0]):plt.subplot(4, 4, i+1) plt.imshow(predictions[i, :, :, 0], cmap='gray')plt.axis('off')

# Save the figure plt.savefig('image\_at\_epoch\_{:04d}.png'.format(epoch)) plt.show()

# Generate a fixed set of noise for evaluating the model during trainingfixed\_noise = tf.random.normal([num\_examples\_to\_generate, noise\_dim])

# Train the model

for epoch in range(EPOCHS):

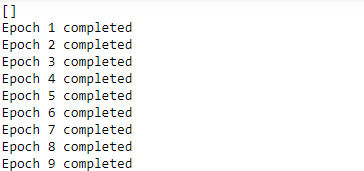
for image\_batch in train\_dataset: train\_step(image\_batch)

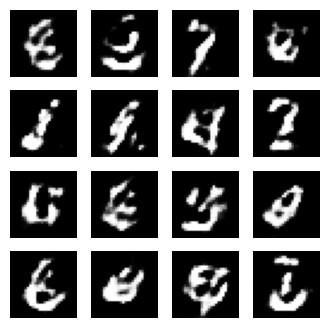
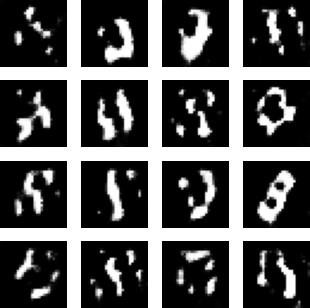
# Generate and save images every 10 epochsif (epoch + 1)

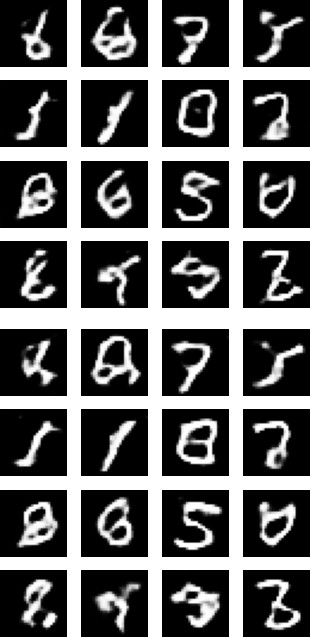
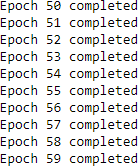
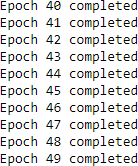
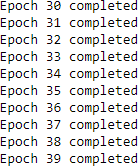
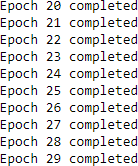
% 10 == 0:

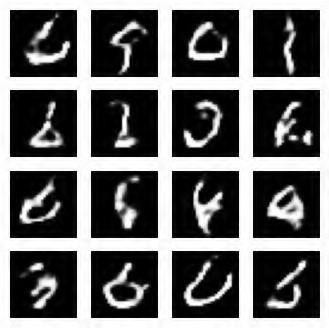
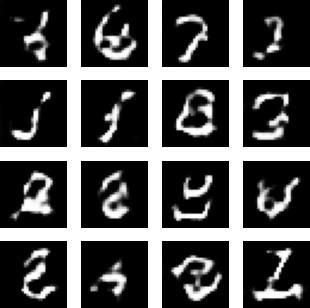
generate\_and\_save\_images(generator, epoch + 1, fixed\_noise)

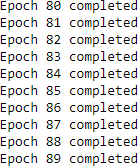
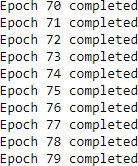
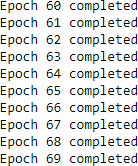
# Print progress every epoch

print('Epoch {} completed'.format(epoch + 1))

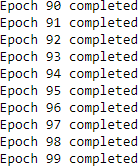


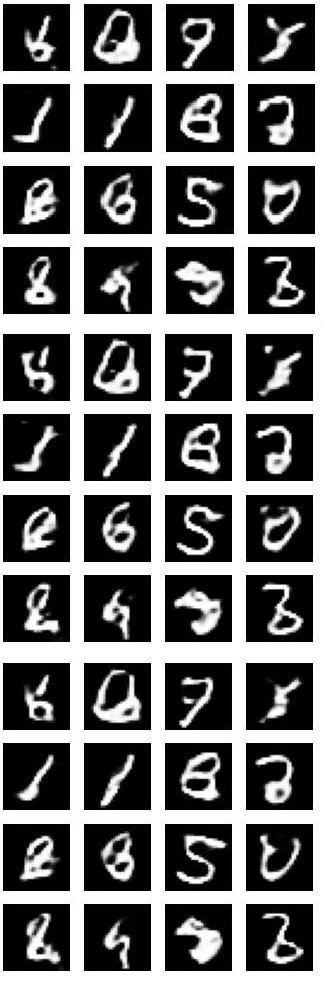








import os os.environ["CUDA\_VISIBLE\_DEVICES"]="0"



# Practical 8(A)

**Aim**: Write a program to implement a simple form of a recurrent neural network e.g., (4-to-1 RNN) to show that the quantity of rain on a certain day also depends on the values of the previous day.

**Theory:** Recurrent Neural Network (RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

**Code:**

import tensorflow as tfimport numpy as np

import matplotlib.pyplot as plt

# Define sequence of 50 days of rain data

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| rain\_data = np.array([2.3, | 1.5, | 3.1, | 2.0, | 2.5, | 1.7, | 2.9, | 3.5, | 3.0, | 2.1, |
| 2.5, | 2.2, | 2.8, | 3.2, | 1.8, | 2.7, | 1.9, | 3.1, | 3.3, | 2.0, |
| 2.5, | 2.2, | 2.4, | 3.0, | 2.1, | 2.5, | 3.2, | 3.1, | 1.9, | 2.7, |
| 2.2, | 2.8, | 3.1, | 2.0, | 2.5, | 1.7, | 2.9, | 3.5, | 3.0, | 2.1, |
| 2.5, | 2.2, | 2.8, | 3.2, | 1.8, | 2.7, | 1.9, | 3.1, | 3.3, | 2.0]) |

# Create input and output sequences for trainingdef create\_sequences(values, time\_steps):

x = []

y = []

for i in range(len(values)-time\_steps): x.append(values[i:i+time\_steps]) y.append(values[i+time\_steps])

return np.array(x), np.array(y)

time\_steps = 4

x\_train, y\_train = create\_sequences(rain\_data, time\_steps)

# Define RNN model

model = tf.keras.models.Sequential([

tf.keras.layers.SimpleRNN(8, input\_shape=(time\_steps, 1)),tf.keras.layers.Dense(1)

])

# Compile model

model.compile(optimizer="adam", loss="mse")

# Train model

history = model.fit(x\_train.reshape(-1, time\_steps, 1), y\_train, epochs=100)

# Plot loss over time

loss = history.history["loss"] epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, "bo", label="Training loss")plt.title("Training loss") plt.legend()

plt.show()

# Test model on new sequence

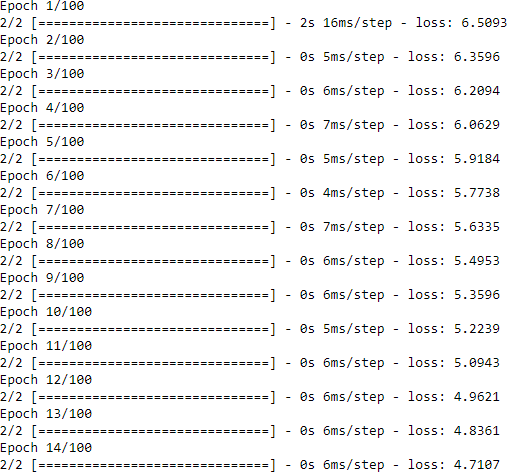
test\_sequence = np.array([2.5, 2.2, 2.8, 3.2])x\_test = np.array([test\_sequence])

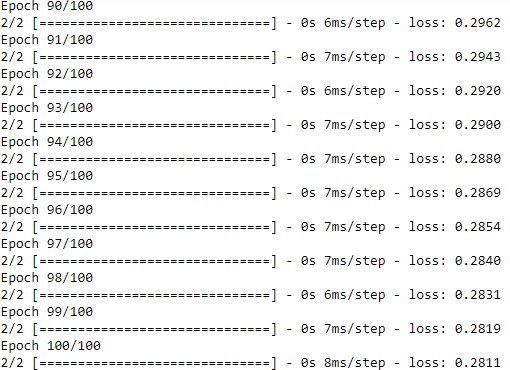
y\_test = model.predict(x\_test.reshape(-1, time\_steps, 1))

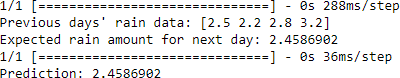
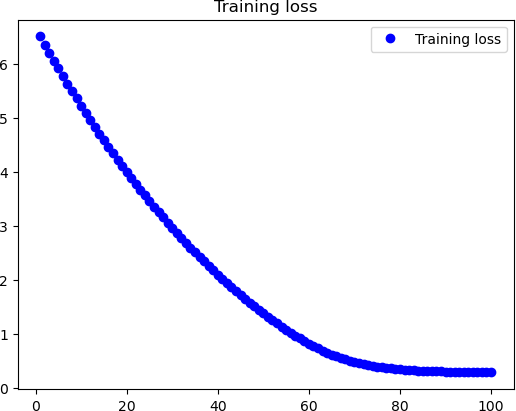
# Print input, output, and prediction print("Previous days' rain data:", test\_sequence)

print("Expected rain amount for next day:", y\_test[0][0])

prediction = model.predict(np.array([test\_sequence]).reshape(1, time\_steps, 1))print("Prediction:", prediction[0][0])







The output of this program will show the loss of the training data over time, as well as the expected rain amount for the next day given the previous 4 days’ rain data, and the model’s prediction of the next day’s rain amount. Note that the expected rain amount is simply the true value for the next day in

# Practical 8(B)

**Aim**: Write a program to implement a simple form of a recurrent neural network like LSTM for sentiment analysis on datasets like UMICH SI650 for similar.

**Theory:** LSTM stands for long short-term memory networks, used in the field of Deep Learning. It is a variety of recurrent neural networks (RNNs) that are capable of learning long-term dependencies, especially in sequence prediction problems. LSTM has feedback connections, i.e., it is capable of processing the entire sequence of data, apart from single data points such as images.

Sentiment Analysis is an NLP application that identifies a text corpus’s emotional or sentimental tone or opinion. Usually, emotions or attitudes towards a topic can be positive, negative, or neutral. Sentiment analysis is a potent tool with varied applications across industries. It is helpful for social media and brand monitoring, customer support and feedback analysis, market research, etc.

**Code:**

import pandas as pd import numpy as np import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequencesfrom sklearn.model\_selection import train\_test\_split

import matplotlib.pyplot as plt

#Load data

data = pd.read\_csv("training.txt", delimiter="\t", names=["label", "text"])

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data["text"],data["label"],test\_size=0.2, random\_state=42)

# Tokenize words

tokenizer = Tokenizer(num\_words=5000, oov\_token="<OOV>")tokenizer.fit\_on\_texts(X\_train)

# Convert words to sequences

X\_train\_seq = tokenizer.texts\_to\_sequences(X\_train)X\_test\_seq = tokenizer.texts\_to\_sequences(X\_test)

# Pad sequences to have same length max\_length = 100

X\_train\_pad = pad\_sequences(X\_train\_seq, maxlen=max\_length,padding="post",truncating="post") X\_test\_pad = pad\_sequences(X\_test\_seq, maxlen=max\_length,padding="post",truncating="post")

# Build LSTM model

model = tf.keras.models.Sequential([ tf.keras.layers.Embedding(input\_dim=5000, output\_dim=32, input\_length=max\_length),

tf.keras.layers.LSTM(units=64, dropout=0.2, recurrent\_dropout=0.2),tf.keras.layers.Dense(1, activation="sigmoid")

])

# Compile model

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

# Train model

history = model.fit(X\_train\_pad, y\_train, epochs=10, batch\_size=32,validation\_split=0.1)

# Evaluate model on test data

loss, accuracy = model.evaluate(X\_test\_pad, y\_test)print("Test loss:", loss) print("Test accuracy:", accuracy)

# Plot training and validation accuracy over time plt.plot(history.history["accuracy"], label="Training accuracy") plt.plot(history.history["val\_accuracy"], label="Validation accuracy")plt.xlabel("Epoch")

plt.ylabel("Accuracy") plt.legend() plt.show()

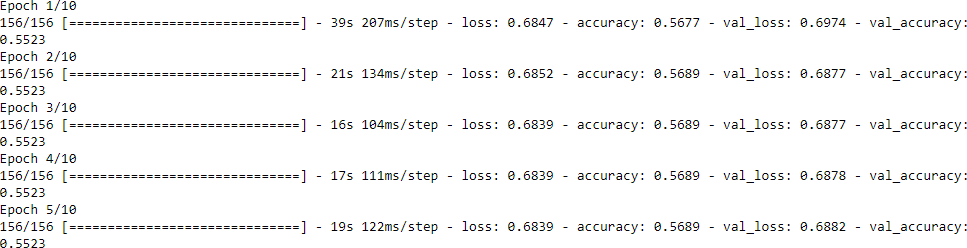
# Make predictions on test data predictions = model.predict(X\_test\_pad)

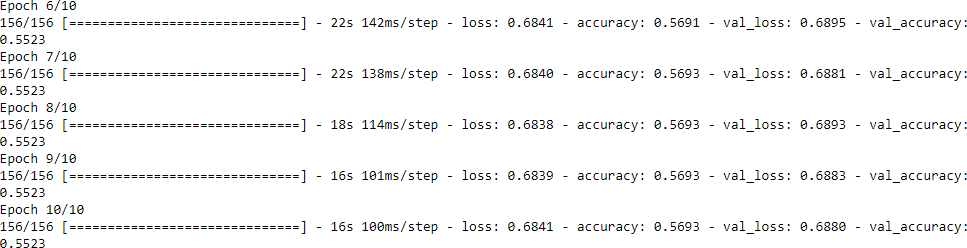
# Print input, output, and prediction for random exampleindex = np.random.randint(0, len(X\_test\_pad))

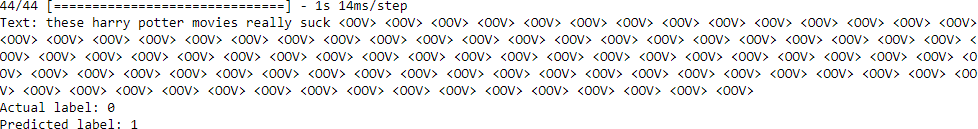
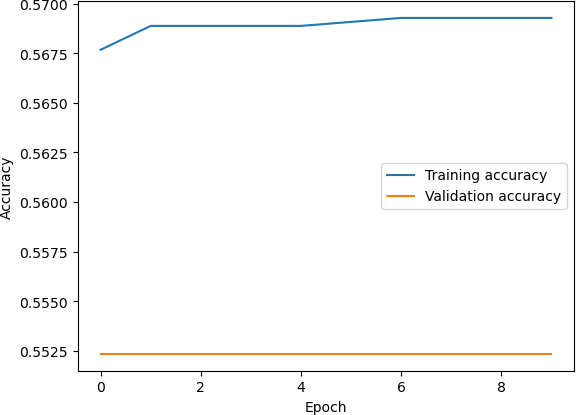
text = tokenizer.sequences\_to\_texts([X\_test\_pad[index]])[0]label = y\_test.values[index]

prediction = predictions[index][0] print("Text:", text) print("Actual label:", label)

print("Predicted label:", round(prediction))







The LSTM model predicted a label of 1 for the given text “i love the harry potter series if you can count that as a book also catcher in the tye jane eyre the virgin suicides yeah”, which means that the model classified this text as having a positive sentiment.

This code loads the UMICH SI650 dataset, splits it into training and testing sets, tokenizes the words, converts them to sequences, and pads the sequences to have the same length. It then builds an LSTM model with an embedding layer, an LSTM layer, and a dense output layer. The model is compiled with binary cross-entropy loss and accuracy as a metric. The model is trained for 10 epochs, and the training and validation accuracy are plotted over time. Finally, the model is evaluated on the test data, and a random example is chosen to print the input, output, and prediction.

# Practical 9

**Aim**: Write a program for object detection from the image.

**Code:**

import numpy as np import tensorflow as tf

from tensorflow.keras.applications.vgg16 import VGG16, preprocess\_input,decode\_predictions from tensorflow.keras.preprocessing.image import load\_img, img\_to\_array

# Load the VGG16 model with pre-trained weightsmodel = VGG16()

# Load the image to detect objects in

img = load\_img('objectdetectimage.jpg', target\_size=(224, 224))

# Convert the image to a numpy arrayimg\_arr = img\_to\_array(img)

img\_arr = np.expand\_dims(img\_arr, axis=0)img\_arr = preprocess\_input(img\_arr)

# Predict the objects in the imagepreds = model.predict(img\_arr)

decoded\_preds = decode\_predictions(preds, top=5)[0]

# Print the predicted objects and their probabilitiesfor pred in decoded\_preds:

print(f"{pred[1]}: {pred[2]\*100:.2f}%")



# Practical 10

**Aim**: Write a program for object detection using pre-trained models to use object detection.

**Theory:** VGG stands for Visual Geometry Group; it is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers. The “deep” refers to the number of layers with VGG-16 or VGG- 19 consisting of 16 and 19 convolutional layers.

**Code:**

import numpy as np import tensorflow as tf

from tensorflow.keras.applications.vgg16 import VGG16, preprocess\_input,decode\_predictions from tensorflow.keras.preprocessing.image import load\_img, img\_to\_array

# Load the VGG16 model with pre-trained weights model = VGG16()

# Load the image to detect objects in

image = load\_img('objectdetectimage.jpg', target\_size=(224, 224))

# Convert the image to a numpy array image = img\_to\_array(image)

# Reshape the image data for VGG

image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))

# Preprocess the image

image = preprocess\_input(image)

# Make predictions on the image using the VGG model predictions = model.predict(image)

# Decode the predictions

decoded\_predictions = decode\_predictions(predictions, top=2)

# Print the predictions with their probabilities

for i, prediction in enumerate(decoded\_predictions[0]):

print("Object ", i+1, ": ", prediction[1], ", Probability: ", prediction[2])

A picture containing text, font, screenshot, line  Description automatically generated