I
N
D
E
X

PRACTICAL No.: 01

Implementing advanced deep learning algorithms such as CNNs or RNNs

PRACTICAL No.: 01 Implementing advanced deep learning algorithms such as CNNs or RNNs

Code:

```
import tensorflow as tf import
numpy as np
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.text import Tokenizer from
tensorflow.keras.preprocessing.sequence import pad_sequences from sklearn.model_selection
import train_test_split from sklearn.metrics import accuracy_score texts = [                    # Sample
data (For a full implementation, replace this with a larger dataset)
   "I love this movie",
   "This movie is terrible",
   "Amazing plot and great acting",
   "I hate this movie",
   "The movie was okay, not great",
   "Fantastic experience, very enjoyable",
   "Not worth watching",
   "I would watch it again, highly recommend",
   "I dislike the plot but the acting was good",
   "A masterpiece in cinema"
]
labels = [1, 0, 1, 0, 0, 1, 0, 1, 0, 1]  # 1: Positive, 0: Negative
tokenizer = Tokenizer(num_words=10000)         # Tokenize the text data tokenizer.fit_on_texts(texts)
X = tokenizer.texts_to_sequences(texts)
X = pad_sequences(X, maxlen=10)                    # Pad sequences to ensure they are of the same length
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2, random_state=42)         # Train-test split
X_train = np.array(X_train)                  # Ensure data is in the right format
X_test = np.array(X_test) y_train
= np.array(y_train) y_test =
np.array(y_test)
model_rnn = models.Sequential([           # Define the LSTM-based RNN model
layers.Embedding(input_dim=10000, output_dim=64, input_length=10),     layers.LSTM(64,
return_sequences=False),
   layers.Dense(64, activation='relu'),     layers.Dense(1, activation='sigmoid')  #
Binary classification (positive/negative)
])
model_rnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model_rnn.fit(X_train, y_train, epochs=5, batch_size=2, validation_data=(X_test, y_test)) # Train RNN model
y_pred_rnn = (model_rnn.predict(X_test) > 0.5).astype(int)         # Evaluate the model print("RNN (LSTM)
Accuracy:", accuracy_score(y_test, y_pred_rnn))
```

Output:

```
Epoch 1/5   4/4 ——— 6s 288ms/step - accuracy: 0.3750 - loss: 0.6953 -val_accuracy: 0.0000e+00 - val_loss: 0.7018
Epoch 2/5    4/4 ——— 0s 54ms/step - accuracy: 0.6250 - loss: 0.6884 - val_accuracy: 0.0000e+00 - val_loss: 0.7084
Epoch 3/5    4/4 ——— 0s 55ms/step - accuracy: 0.6250 - loss: 0.6824 - val_accuracy: 0.0000e+00 - val_loss: 0.7137
Epoch 4/5    4/4 ——— 0s 66ms/step - accuracy: 0.6250 - loss: 0.6776 -val_accuracy: 0.0000e+00 - val_loss: 0.7171
Epoch 5/5    4/4 ——— 0s 39ms/step - accuracy: 0.7500 - loss: 0.6703 - val_accuracy: 0.0000e+00 - val_loss: 0.7238
1/1 ———————————————————————— 0s 418ms/step RNN
(LSTM) Accuracy: 0.0
```

PRACTICAL No.: 02

Building NLP model for sentiment analysis

PRACTICAL No.: 02

Building NLP model for sentiment analysis

Code:

```
import nltk import string import numpy as np import pandas as
pd from sklearn.model_selection import train_test_split from
sklearn.feature_extraction.text import TfidfVectorizer from
sklearn.linear_model import LogisticRegression from
sklearn.metrics import accuracy_score, classification_report from
nltk.corpus import movie_reviews from sklearn.pipeline import
make_pipeline from sklearn.preprocessing import LabelEncoder
nltk.download('movie_reviews')                                  # Download NLTK data (movie reviews dataset)
nltk.download('stopwords') documents = [(list(movie_reviews.words(fileid)), category)          for category in
movie_reviews.categories()
        for fileid in movie_reviews.fileids(category)]          # Loading the movie reviews dataset from nltk
texts = [' '.join(doc) for doc, _ in documents]                 # Preprocessing the data: Combine text and labels
labels = [category for _, category in documents]                # Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(texts, labels, test_size=0.2, random_state=42)
# Initialize the vectorizer (convert text to features) and classifier vectorizer = TfidfVectorizer(stop_words='english',
max_features=1000) classifier = LogisticRegression(max_iter=1000) model = make_pipeline(vectorizer, classifier)
# Create a pipeline that first vectorizes text & then trains classifier
model.fit(X_train, y_train)                                     # Train the model
y_pred = model.predict(X_test)                                  # Predict the labels for the test set
accuracy = accuracy_score(y_test, y_pred)                       # Evaluating the mode report =
classification_report(y_test, y_pred)
print(f"Accuracy: {accuracy}") print("Classification Report:") print(report) example_text = ["I love this movie,
it's amazing!", "I hate this movie, it's terrible."]   # Example prediction predictions =
model.predict(example_text) print("\nPredictions for Example Texts:") for text, pred in zip(example_text,
predictions):
   print(f"Text: {text} -> Predicted Sentiment: {pred}")
```

Output:

Accuracy: 0.8125 Classification
Report:

	precision	recall	f1-score	support
neg	0.82	0.80	0.81	199
pos	0.81	0.82	0.81	201
accuracy			0.81	400
macro avg	0.81	0.81	0.81	400
weighted avg	0.81	0.81	0.81	400

Predictions for Example Texts:
Text: I love this movie, it's amazing! -> Predicted Sentiment: pos
Text: I hate this movie, it's terrible. -> Predicted Sentiment: neg

PRACTICAL No.: 03

Creating a Chatbot using advanced technique

PRACTICAL No.: 03

Aim: Creating a Chatbot using advanced technique

Code:

```
import torch from transformers import GPT2LMHeadModel,
GPT2Tokenizer

# Load the pre-trained GPT-2 model and tokenizer
model_name = "gpt2" tokenizer =
GPT2Tokenizer.from_pretrained(model_name) model =
GPT2LMHeadModel.from_pretrained(model_name)

# Set the model to evaluation mode model.eval()

def generate_response(prompt, max_length=50):
    input_ids = tokenizer.encode(prompt, return_tensors="pt")

    # Generate response     with torch.no_grad():         output = model.generate(input_ids,
max_length=max_length, num_return_sequences=1, pad_token_id=50256)

    response = tokenizer.decode(output[0], skip_special_tokens=True)
return response

print("Chatbot: Hi there! How can I help you?") while
True:
    user_input = input("You: ")
if user_input.lower() == "exit":
print("Chatbot: Goodbye!")
break

    response = generate_response(user_input)
    print("Chatbot:", response)
```

Output:

```
Chatbot: Hi there! How can I help you?
You: What is AI?
Chatbot: What is AI?

AI is a new field of research that has been gaining traction in recent years. It is a field that has been gaining traction
in recent years. It is a field that has been gaining traction in recent years. It is You: What is ML?
Chatbot: What is ML?

ML is a programming language that is designed to be used in a variety of applications. ML is a programming
language that is designed to be used in a variety of applications. You: exit
Chatbot: Goodbye!
```

PRACTICAL No.: 04 Developing a recommendation system

PRACTICAL No.: 04 Aim: Developing a recommendation system

Code:

```
import numpy as np import
pandas as pd import
tensorflow as tf
from tensorflow.keras.models import Model from tensorflow.keras.layers import
Embedding, Flatten, Dense, Input, Concatenate from tensorflow.keras.optimizers
import Adam
data = { 'user_id': [1, 1, 1, 2, 2, 3, 3, 3, 4, 4], 'item_id': ['A', 'B', 'C', 'A', 'C', 'A', 'B', 'C', 'B', 'C'], 'rating': [5, 4, 3, 4, 5,
3, 2, 5, 5, 3] }
df = pd.DataFrame(data)  # Convert the data into a pandas DataFrame user_map = {user: idx for idx,
user in enumerate(df['user_id'].unique())} # Map item_id and user_id to integers item_map = {item: idx for
idx, item in enumerate(df['item_id'].unique())} df['user_id'] = df['user_id'].map(user_map) df['item_id'] =
df['item_id'].map(item_map)
X = df[['user_id', 'item_id']].values  # Split data into features and labels y = df['rating'].values user_input =
Input(shape=(1,)) # Define the model architecture for Neural Collaborative Filtering (NCF) item_input =
Input(shape=(1,)) user_embedding = Embedding(input_dim=len(user_map), output_dim=10)(user_input) #
Embedding layers for  item_embedding = Embedding(input_dim=len(item_map), output_dim=10)(item_input) #
users & items user_embedding = Flatten()(user_embedding)  # Flatten the embeddings item_embedding =
Flatten()(item_embedding) merged = Concatenate()([user_embedding, item_embedding])  # Concatenate
the embeddings dense = Dense(128, activation='relu')(merged)  # Fully connected layers dense = Dense(64,
activation='relu')(dense) output = Dense(1)(dense) model = Model(inputs=[user_input, item_input],
outputs=output)  # Compile the model model.compile(optimizer=Adam(), loss='mean_squared_error')
model.fit([X[:, 0], X[:, 1]], y, epochs=10, batch_size=2, verbose=1)  # Train the model def
recommend(user_id, top_n=3):  # Function to make recommendations for a specific user  all_items =
list(item_map.values())  # Get all items  user_input_data = np.array([user_map[user_id]] * len(all_items))  #
Predict ratings for each item for the user  item_input_data = np.array(all_items)
  predictions = model.predict([user_input_data, item_input_data])
  item_ids_sorted = np.argsort(predictions.flatten())[::-1]  # Sort the predictions by predicted rating
recommended_item_ids = item_ids_sorted[:top_n]  # Get the top_n recommended item IDs
  recommended_items = [list(item_map.keys())[list(item_map.values()).index(idx)] for idx in
recommended_item_ids]  # Convert back to original item IDs  return
recommended_items user_id = 1  # Example: Get top 3 recommendations for user 1
recommended_items = recommend(user_id) print(f"Top 3 recommendations for user
{user_id}: {recommended_items}")
```

Output:

```
Epoch 1/10  5/5 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - loss: 15.9476
Epoch 2/10  5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 15.3459
     :      :                          :    :    :    :
Epoch 9/10  5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - loss: 4.6208
Epoch 10/10  5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - loss: 2.6689
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 205ms/step
Top 3 recommendations for user 1: ['B', 'C', 'A']
```

PRACTICAL No.: 05 Implementing a computer vision project

PRACTICAL No.: 05

Aim: Implementing computer vision project (Object Detection/Image segmentation)

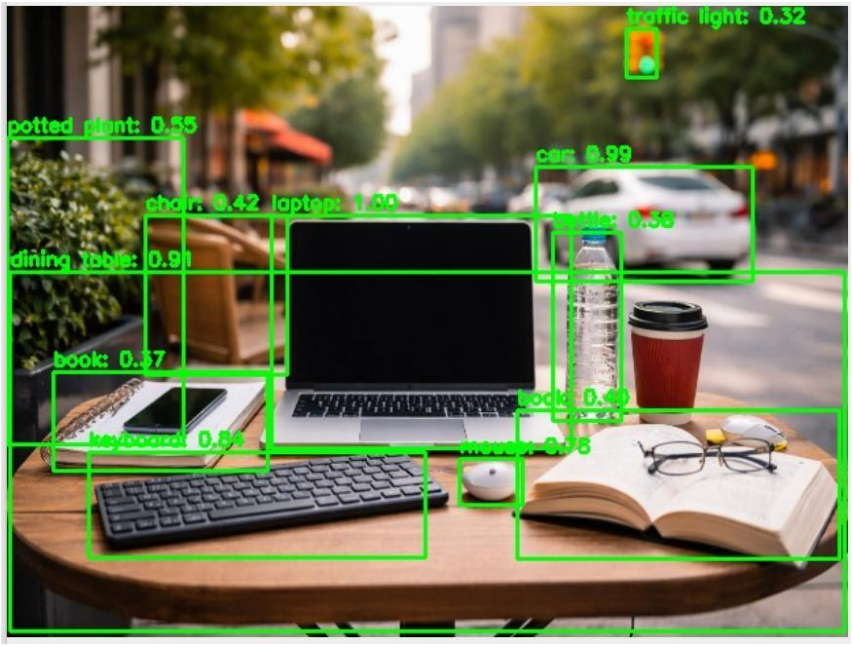
Code:

```
import numpy as np
import os import
tensorflow as tf
import cv2

# Load the pre-trained model
MODEL_NAME = 'ssd_mobilenet_v2_coco_2018_03_29'
PATH_TO_CKPT = os.path.join(MODEL_NAME, 'frozen_inference_graph.pb')
NUM_CLASSES = 90

detection_graph = tf.Graph() with
detection_graph.as_default():
   od_graph_def = tf.compat.v1.GraphDef()     with
tf.io.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
serialized_graph = fid.read()
od_graph_def.ParseFromString(serialized_graph)
tf.import_graph_def(od_graph_def, name='')

# Load label map
#PATH_TO_LABELS = os.path.join('data', 'mscoco_label_map.pbtxt')
PATH_TO_LABELS = "mscoco_label_map.pbtxt"
category_index = {} with
open(PATH_TO_LABELS, 'r') as f:
   lines = f.readlines()
for line in lines:
if 'id:' in line:
        id_index = int(line.strip().split(':')[1])
if 'display_name:' in line:
        name = line.strip().split(':')[1].strip().strip('"')
category_index[id_index] = {'name': name}

# Function to perform object detection def
detect_objects(image):
   image = cv2.resize(image, (640, 480))     with
detection_graph.as_default():        with
tf.compat.v1.Session(graph=detection_graph) as sess:
        # Expand dimensions since the model expects images to have shape: [1, None, None, 3]
image_expanded = np.expand_dims(image, axis=0)           image_tensor =
detection_graph.get_tensor_by_name('image_tensor:0')
        # Each box represents a part of the image where a particular object was detected.
boxes = detection_graph.get_tensor_by_name('detection_boxes:0')           # Each score
represents the level of confidence for each of the objects.           # The score is shown
on the result image, together with the class label.           scores =
detection_graph.get_tensor_by_name('detection_scores:0')           classes =
detection_graph.get_tensor_by_name('detection_classes:0')           num_detections =
detection_graph.get_tensor_by_name('num_detections:0')           # Actual detection.
        (boxes, scores, classes, num_detections) = sess.run([boxes, scores, classes,
num_detections],feed_dict={image_tensor: image_expanded})
```

```
        print("Max score:", np.max(scores))
        # Visualization of the results of a detection.
        for i in range(len(scores[0])):                if scores[0][i] > 0.3:  # Adjust
confidence threshold as needed                    class_id = int(classes[0][i])
class_name = category_index.get(class_id, {'name': 'object'})['name']
score = float(scores[0][i])
                ymin, xmin, ymax, xmax = boxes[0][i]
                (left, right, top, bottom) = (xmin * image.shape[1], xmax * image.shape[1], ymin * image.shape[0],
ymax * image.shape[0])
                cv2.rectangle(image, (int(left), int(top)), (int(right), int(bottom)), (0, 255, 0), 2)
                cv2.putText(image, '{}: {:.2f}'.format(class_name, score), (int(left), int(top - 5)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)     return image

# Object detection on an image
input_image = cv2.imread("images.jpg")

if input_image is None:
   print("Error: Image not found") else:
   output_image = detect_objects(input_image)
cv2.imwrite("output_detected.jpg", output_image)     print("Detection
completed. Output saved as output_detected.jpg")
```

Output:

Max score: 0.9871206
Detection completed. Output saved as output_detected.jpg



PRACTICAL No.: 06

Training a generative adversarial network (GAN)

PRACTICAL No.: 06 Aim: Training a generative adversarial network (GAN)

Code:

```
import tensorflow as tf from
tensorflow.keras import layers
import numpy as np import matplotlib.pyplot as plt import os
np.random.seed(42)                  # Set random seed for reproducibility
tf.random.set_seed(42)
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()    # Load and preprocess MNIST dataset
x_train = x_train.astype('float32')          # Normalize the images to the range [-1, 1] for better GAN performance
x_train = (x_train - 127.5) / 127.5  # Normalize to [-1, 1] x_train = np.expand_dims(x_train, axis=-1)  # Add
channel dimension (1, height, width, 1) latent_dim = 100            # Set parameters           # Dimension of the
latent space (noise) batch_size = 64 epochs = 10000 save_interval = 1000
def build_generator():    # Create the Generator model
   model = tf.keras.Sequential() # Fully connected layer with ReLU activation     model.add(layers.Dense(128 *
7 * 7, input_dim=latent_dim))  # Output enough features for reshape     model.add(layers.LeakyReLU(0.2))
model.add(layers.BatchNormalization(momentum=0.8))    # Reshape the output of the Dense layer into a 3D
model.add(layers.Reshape((7, 7, 128)))  # tensor (7x7x128)          # 7x7 image with 128 filters
model.add(layers.Conv2DTranspose(128, kernel_size=3, strides=2, padding='same'))    # Upsample to 14x14
model.add(layers.LeakyReLU(0.2))
   model.add(layers.BatchNormalization(momentum=0.8))     model.add(layers.Conv2DTranspose(64,
kernel_size=3, strides=2, padding='same'))    # Upsample to 28x28     model.add(layers.LeakyReLU(0.2))
model.add(layers.BatchNormalization(momentum=0.8))                # Final layer: 28x28 image with tanh activation
model.add(layers.Conv2DTranspose(1, kernel_size=3, strides=1, padding='same', activation='tanh'))     return
model
def build_discriminator():          # Create the Discriminator model
   model = tf.keras.Sequential()      # Convolutional layers with LeakyReLU activations
model.add(layers.Conv2D(64, kernel_size=3, strides=2, padding='same', input_shape=(28, 28, 1)))
model.add(layers.LeakyReLU(0.2))     model.add(layers.Dropout(0.25))
model.add(layers.Conv2D(128, kernel_size=3, strides=2, padding='same'))
model.add(layers.LeakyReLU(0.2))     model.add(layers.Dropout(0.25))
model.add(layers.Flatten())     model.add(layers.Dense(1, activation='sigmoid'))  # Output a single
value (real/fake)     return model
def build_gan(generator, discriminator): # Build the GAN model by combining the Generator & the Discriminator
discriminator.trainable = False        # Freeze the discriminator during GAN training (we only train the generator)
gan_input = layers.Input(shape=(latent_dim,))     x = generator(gan_input)     gan_output = discriminator(x)     gan
= tf.keras.Model(gan_input, gan_output)     return gan
discriminator = build_discriminator()      # Compile the Discriminator and GAN
discriminator.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
loss='binary_crossentropy', metrics=['accuracy'])

generator = build_generator() gan = build_gan(generator, discriminator)
gan.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5), loss='binary_crossentropy')
def train_gan(epochs, batch_size, save_interval):              # Function to train the GAN
   half_batch = batch_size // 2  # Half the batch size for real and fake     valid = np.ones((batch_size,
1))  # Change to match full batch size          # Adversarial ground truths     fake = np.zeros((batch_size,
1))  # Change to match full batch size     for epoch in range(epochs):          idx = np.random.randint(0,
x_train.shape[0], half_batch)            # Train Discriminator          real_images = x_train[idx]          noise
= np.random.normal(0, 1, (half_batch, latent_dim))                # Generate fake images
generated_images = generator.predict(noise)                          # Train on real images
      d_loss_real = discriminator.train_on_batch(real_images, valid[:half_batch])  # Use only half batch for real
d_loss_fake = discriminator.train_on_batch(generated_images, fake[:half_batch])# Use only 1/2 batch for fake
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)          noise = np.random.normal(0, 1, (batch_size, latent_dim))
      # Train Generator via GAN          g_loss = gan.train_on_batch(noise, valid)  # Use full batch for generator
```

```
if epoch % save_interval == 0:                    # Print the progress            print(f"{epoch}/{epochs} [D loss:
{d_loss[0]} | D accuracy: {100 * d_loss[1]}] [G loss: {g_loss}]")            save_generated_images(epoch) def
save_generated_images(epoch, examples=10, dim=(1, 10), figsize=(10, 1)): # Function save generated images
noise = np.random.normal(0, 1, (examples, latent_dim))     generated_images = generator.predict(noise)
   generated_images = (generated_images + 1) / 2.0  # Rescale images to [0, 1]
plt.figure(figsize=figsize)               # Plot and save images     for i in
range(examples):
       plt.subplot(dim[0], dim[1], i + 1)
plt.imshow(generated_images[i, :, :, 0], cmap='gray')
plt.axis('off')
   plt.tight_layout()
plt.savefig(f"gan_generated_image_{epoch}.png")     plt.close()
train_gan(epochs, batch_size, save_interval)        # Train the GAN
```

Output:

```
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 109ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 114ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 102ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 103ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 111ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 116ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 117ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 113ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step
```

PRACTICAL No.: 07 Applying reinforcement learning algorithms

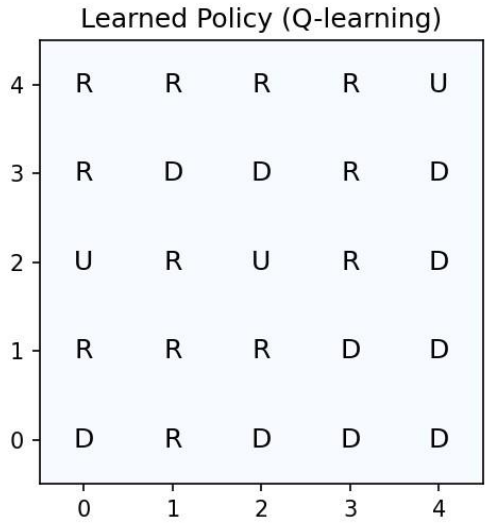
PRACTICAL No.: 07

Aim: Applying reinforcement learning algorithms

Code:

```
import numpy as np import
random import
matplotlib.pyplot as plt

# Define the environment size (grid dimensions)
grid_size = (5, 5)  # 5x5 grid
goal_state = (4, 4)  # goal is at the bottom-right corner start_state
= (0, 0)  # start at the top-left corner

actions = ['up', 'down', 'left', 'right']          # Define the actions action_map
= {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

# Q-learning parameters learning_rate
= 0.1    # Alpha discount_factor = 0.9
# Gamma
epsilon = 0.1         # Epsilon for epsilon-greedy policy num_episodes
= 1000   # Number of episodes for training max_steps_per_episode
= 100  # Maximum steps per episode

# Initialize Q-table (state-action value table)
q_table = np.zeros((grid_size[0], grid_size[1], len(actions)))

def reward_function(state):         # Reward function: -1 for each step, +100 for reaching the goal
if state == goal_state:
        return 100  # reward for reaching the goal
return -1  # penalty for each step

def choose_action(state):             # Choose action based on epsilon-greedy policy
if random.uniform(0, 1) < epsilon:
        return random.choice(actions)         # Exploration: random action
else:
        state_x, state_y = state                    # Exploitation: choose the best action based on Q-table
q_values = q_table[state_x, state_y, :]         max_q_value = np.max(q_values)         max_actions
= [actions[i] for i in range(len(actions)) if q_values[i] == max_q_value]         return
random.choice(max_actions)

def take_action(state, action):     # Take action and get the next state
state_x, state_y = state     move = action_map[action]
    new_x = max(0, min(grid_size[0] - 1, state_x + move[0]))  # Ensure we don't go out of bounds
new_y = max(0, min(grid_size[1] - 1, state_y + move[1]))  # Ensure we don't go out of bounds     return
(new_x, new_y)

def train_q_learning():   # Q-learning algorithm     for
episode in range(num_episodes):         state =
start_state         total_reward = 0         for step in
range(max_steps_per_episode):             action =
choose_action(state)             next_state =
take_action(state, action)
```

reward = reward_function(next_state) state_x, state_y = state



```
# Update Q-value using the Q-learning formula            next_state_x,
next_state_y = next_state            action_index = actions.index(action)
        # Bellman equation: Q(s, a) = Q(s, a) + alpha * [reward + gamma * max(Q(s', a')) - Q(s, a)]
max_future_q = np.max(q_table[next_state_x, next_state_y, :])            q_table[state_x, state_y, action_index]
+= learning_rate * (reward + discount_factor * max_future_q - q_table[state_x, state_y, action_index])
        state = next_state
        total_reward += reward

        if state == goal_state:      # If the agent reaches the goal, break the episode
break

      if (episode + 1) % 100 == 0:
        print(f"Episode {episode + 1} completed")

def visualize_policy():   # Visualization of learned policy (after training)
policy_grid = np.full(grid_size, '', dtype=object)     for x in range(grid_size[0]):
for y in range(grid_size[1]):
        best_action_index = np.argmax(q_table[x, y, :])            best_action =
actions[best_action_index]            policy_grid[x, y] = best_action[0].upper()  # Show the
first letter of the best action

   plt.figure(figsize=(6, 6))         # Plotting the policy grid
plt.imshow(np.zeros(grid_size), cmap="Blues", interpolation='none')
for x in range(grid_size[0]):         for y in range(grid_size[1]):
        plt.text(y, x, policy_grid[x, y], ha='center', va='center', color='black', fontsize=12)

   plt.title("Learned Policy (Q-learning)")     plt.xticks(range(grid_size[1]))     plt.yticks(range(grid_size[0]))
plt.gca().invert_yaxis()     plt.show()

train_q_learning() # Train the agent using Q-learning visualize_policy() # Visualize the learned policy
```

Output:

```
Episode 100 completed
Episode 200 completed
Episode 300 completed
Episode 400 completed
Episode 500 completed
Episode 600 completed
Episode 700 completed
Episode 800 completed
Episode 900 completed
Episode 1000 completed
```

PRACTICAL No.: 08 Use Python libraries (GPT-2 or textgenrnn) to train generative models

PRACTICAL No.: 08 Aim: Use Python libraries (GPT-2 or textgenrnn) to train generative models

Code:

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load pre-trained model and tokenizer model_name = 'gpt2'
model = GPT2LMHeadModel.from_pretrained(model_name)
tokenizer = GPT2Tokenizer.from_pretrained(model_name)

# Encode the input text input_text = "Once upon a time"
input_ids = tokenizer.encode(input_text, return_tensors='pt')

# Generate text
output = model.generate(input_ids, max_length=100, num_return_sequences=1)

# Decode and print the generated text
generated_text = tokenizer.decode(output[0], skip_special_tokens=True) print(generated_text)
```

Output:

```
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior.
Please pass your input's `attention_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
The attention mask is not set and cannot be inferred from input because pad token is same as eos token. As a
consequence, you may observe unexpected behavior. Please pass your input's `attention_mask` to obtain reliable
results.
Once upon a time, the world was a place of great beauty and great danger. The world was a place of great danger,
and the world was a place of great danger. The world was a place of great danger, and the world was a place of
great danger. The world was a place of great danger, and the world was a place of great danger. The world was a
place of great danger, and the world was a place of great danger. The world was a place of great
```

PRACTICAL NO.: 09 Experiment with neural networks like GANs

PRACTICAL NO.: 09

Aim: Experiment with neural networks like GANs

Code:

```
import torch import torch.nn as nn import
torch.optim as optim import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt import os
import torchvision.utils as vutils
from torch.utils.data import DataLoader

# Generator Model class
Generator(nn.Module):
def __init__(self):
        super(Generator, self).__init__()
self.fc1 = nn.Linear(100, 256)         self.fc2
= nn.Linear(256, 512)         self.fc3 =
nn.Linear(512, 1024)         self.fc4 =
nn.Linear(1024, 784)         self.relu =
nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, z):
        x = self.relu(self.fc1(z))         x = self.relu(self.fc2(x))
x = self.relu(self.fc3(x))         x = self.tanh(self.fc4(x))  #
Output is in the range [-1, 1]         return x.view(-1, 1, 28,
28)  # Reshape to 28x28 image

# Discriminator Model class
Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
self.fc1 = nn.Linear(784, 1024)
self.fc2 = nn.Linear(1024, 512)
self.fc3 = nn.Linear(512, 256)         self.fc4
= nn.Linear(256, 1)         self.leaky_relu =
nn.LeakyReLU(0.2)         self.sigmoid =
nn.Sigmoid()

    def forward(self, x):
        x = x.view(-1, 784)  # Flatten the image to a vector
x = self.leaky_relu(self.fc1(x))         x =
self.leaky_relu(self.fc2(x))         x =
self.leaky_relu(self.fc3(x))         x = self.sigmoid(self.fc4(x))
# Output is between 0 and 1         return x

# Load MNIST dataset def
load_data():
    transform = transforms.Compose([
transforms.ToTensor(),         transforms.Normalize((0.5,),
(0.5,))     ])
    train_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)      return train_loader
```

```
# Train the GAN def train_gan(generator, discriminator, train_loader, num_epochs=50,
lr=0.0002, beta1=0.5, output_dir="generated_images"):
   # Create output directory if it doesn't exist
if not os.path.exists(output_dir):
       os.makedirs(output_dir)

   # Define loss and optimizers     criterion = nn.BCELoss()     optimizer_g =
optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))     optimizer_d =
optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.999))

   # Labels
real_label = 1
   fake_label = 0

   for epoch in range(num_epochs):         for i,
(real_images, _) in enumerate(train_loader):
          # Generate random noise
          z = torch.randn(real_images.size(0), 100)  # Latent vector size = 100
real_labels = torch.ones(real_images.size(0), 1)            fake_labels =
torch.zeros(real_images.size(0), 1)

          # Train Discriminator with real images
optimizer_d.zero_grad()            output_real =
discriminator(real_images)            loss_real =
criterion(output_real, real_labels)
loss_real.backward()

          # Train Discriminator with fake images generated by the Generator            fake_images =
generator(z)            output_fake = discriminator(fake_images.detach())  # Detach to avoid backprop through
the generator            loss_fake = criterion(output_fake, fake_labels)            loss_fake.backward()
          optimizer_d.step()

          # Train Generator to fool the Discriminator
optimizer_g.zero_grad()
          output_fake_g = discriminator(fake_images)            loss_g = criterion(output_fake_g,
real_labels)  # Want the discriminator to classify fake as real            loss_g.backward()
optimizer_g.step()

          # Print the losses
if (i + 1) % 100 == 0:
              print(f"Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], "
f"D Loss: {loss_real.item() + loss_fake.item():.4f}, G Loss: {loss_g.item():.4f}")

       # Save generated images at the end of each epoch
if (epoch + 1) % 10 == 0:            with
torch.no_grad():
              fake_images = generator(torch.randn(64, 100))  # Generate 64 fake images
fake_images = fake_images.cpu().detach()               # Save the generated images as
a grid
              vutils.save_image(fake_images, os.path.join(output_dir, f'epoch_{epoch+1}.png'), nrow=8,
normalize=True)               print(f"Generated images saved to {output_dir}/epoch_{epoch+1}.png")
```

```
# Main function to run everything if
__name__ == "__main__":
    # Load dataset
    train_loader = load_data()

    # Create models
generator = Generator()
    discriminator = Discriminator()

    # Start training the GAN
    train_gan(generator, discriminator, train_loader)

    # Save models
    torch.save(generator.state_dict(), 'generator.pth')
torch.save(discriminator.state_dict(), 'discriminator.pth')     print("Models
saved as 'generator.pth' and 'discriminator.pth'")
```

Output:

```
100%|| 9.91M/9.91M [00:15<00:00, 625kB/s]
100%|| 28.9k/28.9k [00:00<00:00, 71.3kB/s]
100%|| 1.65M/1.65M [00:02<00:00, 584kB/s]
100%|| 4.54k/4.54k [00:00<?, ?B/s]
Epoch [1/50], Step [100/938], D Loss: 1.6525, G Loss: 0.6811
Epoch [1/50], Step [200/938], D Loss: 0.2828, G Loss: 1.8373
Epoch [1/50], Step [300/938], D Loss: 0.8320, G Loss: 2.5801
Epoch [1/50], Step [400/938], D Loss: 0.1982, G Loss: 5.5415
Epoch [1/50], Step [500/938], D Loss: 0.4918, G Loss: 1.4684
Epoch [1/50], Step [600/938], D Loss: 0.6889, G Loss: 3.8545
Epoch [1/50], Step [700/938], D Loss: 0.2333, G Loss: 4.3858
Epoch [1/50], Step [800/938], D Loss: 0.3795, G Loss: 3.7269
Epoch [1/50], Step [900/938], D Loss: 0.1477, G Loss: 3.0551
```

PRACTICAL No.: 10 Optimization techniques (Bayesian optimization) for Hyperparameter tuning

PRACTICAL No.: 10 Aim: Optimization techniques (Bayesian optimization) for Hyperparameter tuning

Code:

```
import numpy as np
from sklearn.datasets import load_breast_cancer from
sklearn.model_selection import train_test_split, cross_val_score from
sklearn.ensemble import RandomForestClassifier from skopt import
BayesSearchCV

data = load_breast_cancer()                    # Load dataset
X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(          # Train-test split
   X, y, test_size=0.2, random_state=42
)

model = RandomForestClassifier(random_state=42)       # Define model

param_space = {          # Define hyperparameter search space
   'n_estimators': (50, 300),
   'max_depth': (3, 20),
   'min_samples_split': (2, 20),
   'min_samples_leaf': (1, 10)
}

bayes_search = BayesSearchCV(          # Bayesian Optimization
estimator=model,    search_spaces=param_space,    n_iter=25,
# Number of optimization steps    cv=3,    scoring='accuracy',
n_jobs=-1,
   random_state=42
)

bayes_search.fit(X_train, y_train)          # Run optimization print("Best
Hyperparameters Found:") # Best parameters
print(bayes_search.best_params_) print("\nBest Cross-Validation
Accuracy:")      # Best accuracy print(bayes_search.best_score_)
test_accuracy = bayes_search.score(X_test, y_test)          # Test accuracy
print("\nTest Set Accuracy:") print(test_accuracy)
```

Output:

```
Best Hyperparameters Found:
OrderedDict({'max_depth': 12, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50})

Best Cross-Validation Accuracy:
0.956009643313582

Test Set Accuracy:
0.9649122807017544
```