Practical No. 1

Aim: Implementing advanced deep learning algorithms such as convolutional neural networks (CNNs) using TensorFlow.

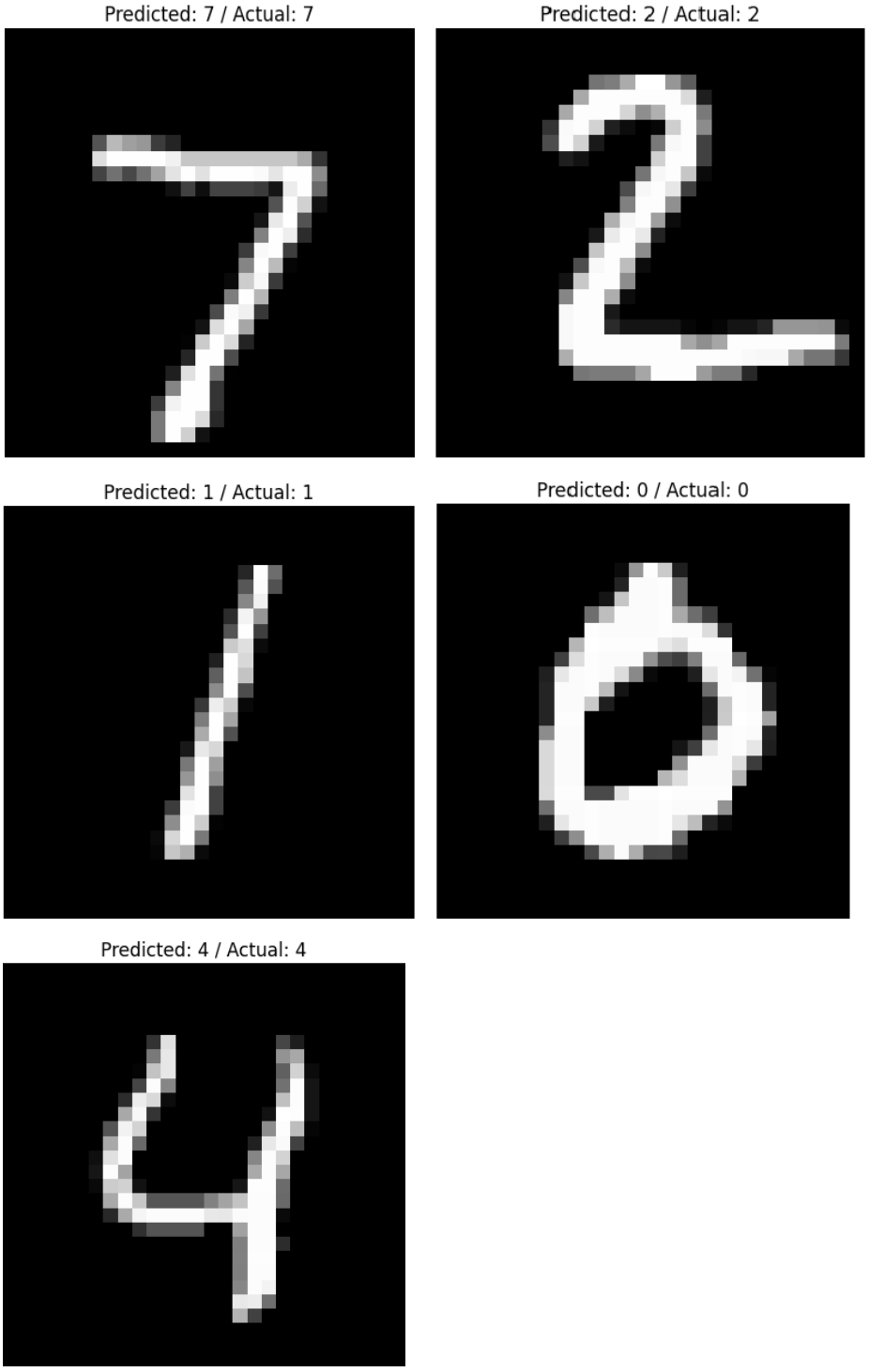
Writeup:

- **Import Libraries:** It starts by importing tensorflow and specific layers and models from tensorflow.keras. numpy and matplotlib.pyplot are also imported for data handling and visualization.
- **Load and Preprocess Data:** The MNIST dataset is loaded, which consists of grayscale images of digits (0-9). The pixel values are normalized by dividing by 255.0 to scale them between 0 and 1. A channel dimension is added to the images (x_train[..., tf.newaxis]) because CNNs typically expect input with a channel dimension (e.g., 1 for grayscale, 3 for RGB).
- **Build the CNN Model:** A models.Sequential model is created, which is a linear stack of layers.
    - ***layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)):*** This is the first convolutional layer. It uses 32 filters (which learn features), each of size 3x3. The activation='relu' applies the ReLU activation function, which introduces non-linearity. input_shape=(28, 28, 1) specifies the shape of the input images (28x28 pixels with 1 channel).
    - ***layers.MaxPooling2D((2, 2)):*** This is a max pooling layer with a pool size of 2x2. It reduces the spatial dimensions of the feature maps, helping to reduce computation and extract the most important features.
    - ***layers.Conv2D(64, (3, 3), activation='relu'):*** Another convolutional layer, this time with 64 filters.
    - ***layers.MaxPooling2D((2, 2)):*** Another max pooling layer.
    - ***layers.Flatten():*** This layer flattens the output of the convolutional and pooling layers into a 1D vector. This is necessary before passing the data to the dense layers.
    - ***layers.Dense(64, activation='relu'):*** A fully connected (dense) layer with 64 neurons and ReLU activation.
    - ***layers.Dense(10, activation='softmax'):*** The output layer. It's a dense layer with 10 neurons (one for each digit class). The softmax activation function outputs a probability distribution over the 10 classes, indicating the model's confidence in each digit.
- **Compile the Model:** The model is compiled with an optimizer ('adam'), a loss function ('sparse_categorical_crossentropy' which is suitable for multi-class classification with integer labels), and a metric to monitor during training ('accuracy').
- **Train the Model:** The model is trained using the model.fit() method on the training data (x_train, y_train) for a specified number of epochs. validation_split=0.1 sets aside 10% of the training data for validation during training.
- **Evaluate the Model:** The trained model is evaluated on the test data (x_test, y_test) using model.evaluate() to determine its performance on unseen data.
- **Make and Visualize Predictions:** The model makes predictions on the test data using model.predict(). A loop then visualizes the first few test images along with the model's predicted digit and the actual digit.

Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Normalize the images (scale from 0-255 → 0-1)
x_train = x_train / 255.0
x_test = x_test / 255.0
# Add a channel dimension: (28,28) → (28,28,1)
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 digits: 0–9
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, validation_split=0.1)
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
predictions = model.predict(x_test)
# Plot a few predictions
for i in range(5):
    plt.imshow(x_test[i].squeeze(), cmap='gray')
    plt.title(f"Predicted: {np.argmax(predictions[i])} / Actual: {y_test[i]}")
    plt.axis('off')
    plt.show()
```

Output:


Predicted: 7 / Actual: 7
Predicted: 2 / Actual: 2
Predicted: 1 / Actual: 1
Predicted: 0 / Actual: 0
Predicted: 4 / Actual: 4

Practical No. 2

Aim: Building a natural language processing (NLP) model for sentiment analysis or text classification.

Writeup:

- **Import Libraries:** It imports necessary libraries like tensorflow , Tokenizer and pad_sequences for text preprocessing, Sequential for building the model, and layers like Embedding, Flatten, and Dense. numpy is used for handling labels.
- **Prepare Data:**
  - ***texts:*** This is your input text data, which are short sentences expressing different sentiments.
  - ***labels:*** These are the corresponding sentiment labels for each sentence (1 for positive, 0 for negative). This is the target variable your model will learn to predict.
- **Text Preprocessing:**
  - ***Tokenizer:*** This is a crucial step in NLP. The Tokenizer converts text into numerical sequences. It builds a vocabulary from your text data, assigning a unique integer to each word. The oov_token="<OOV>" handles words that are not in the vocabulary during later processing.
  - ***fit_on_texts(texts):*** This method trains the tokenizer on your texts to build the vocabulary.
  - ***texts_to_sequences(texts):*** This converts your text sentences into sequences of integers based on the vocabulary created by the tokenizer.
  - ***pad_sequences:*** Neural networks typically require input sequences of the same length. pad_sequences adds padding (usually zeros) to the sequences to make them all the same length (maxlen=5 in this case). padding='post' means padding is added at the end of the sequence.
- **Build the NLP Model:** A Sequential model is created:
  - ***Embedding(input_dim=vocab_size, output_dim=8, input_length=5):*** This is the embedding layer, which is fundamental for many NLP tasks. It converts the integer sequences into dense vectors of fixed size (8 in this case). Each word in your vocabulary will be represented by a unique vector. This layer learns to represent words in a way that captures their semantic meaning. The input_length=5 specifies the length of the input sequences.
  - ***Flatten():*** This layer flattens the output of the embedding layer into a 1D vector, preparing it for the dense layers.
  - ***Dense(6, activation='relu'):*** A dense layer with 6 neurons and ReLU activation. These layers learn to combine the features from the embedding layer.
  - ***Dense(1, activation='sigmoid'):*** The output layer. It has 1 neuron with a sigmoid activation function. The sigmoid function outputs a value between 0 and 1, which can be interpreted as the probability of the input text having positive sentiment (closer to 1) or negative sentiment (closer to 0).
- **Compile the Model:** The model is compiled with the adam optimizer and binary_crossentropy loss function, which is suitable for binary classification tasks like sentiment analysis (positive or negative). metrics=['accuracy'] tracks the model's accuracy during training.

- **Train the Model:** The model is trained on the padded sequences and corresponding labels for a specified number of epochs.
- **Test and Predict:**
  - ***test_texts:*** New sentences are defined to test the trained model.
  - These test sentences are preprocessed using the same tokenizer and padding as the training data.
  - ***model.predict(test_pad):*** The trained model makes predictions on the preprocessed test data. The output is a probability score for each test sentence.

The code then iterates through the test sentences and their predictions, classifying the sentiment as "Positive" if the probability is above 0.51 and "Negative" otherwise. The sentiment and the prediction probability are printed.

Code:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense
import numpy as np
texts = [
    "I love this product",
    "This is the worst movie",
    "I am so happy",
    "I hate this book",
    "What a great day",
    "I am very disappointed"
]
labels = [1, 0, 1, 0, 1, 0]  # 1 = Positive, 0 = Negative
# Initialize tokenizer
tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(texts)
# Convert to sequences
sequences = tokenizer.texts_to_sequences(texts)
padded = pad_sequences(sequences, padding='post', maxlen=5)
# Vocabulary size
vocab_size = len(tokenizer.word_index) + 1
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=8, input_length=5),
    Flatten(),
    Dense(6, activation='relu'),
    Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
#model.summary()
# Convert labels to NumPy array
labels = np.array(labels)
```

```
# Train
model.fit(padded, labels, epochs=50, verbose=1)  # Set verbose=1 if you want output
# Test new sentences
#test_texts = ["so happy", "I hate this taste", "What a worst day", "Great movie"]
test_texts = [
    "I love this product",
    "worst movie",
    "I am so happy",
    "I hate this book",
    "great day",
    "I am very disappointed"
]
# Tokenize and pad
test_seq = tokenizer.texts_to_sequences(test_texts)
test_pad = pad_sequences(test_seq, maxlen=5, padding='post')
predictions = model.predict(test_pad)

for i, text in enumerate(test_texts):
    sentiment = "Positive" if predictions[i] > 0.51 else "Negative"
    print(f"{text} → {sentiment} ({predictions[i][0]:.2f})")

for i, text in enumerate(test_texts):
    if predictions[i] > 0.51:
      sentiment = "Positive"
    else:
      sentiment = "Negative"
    print(f"{text} → {sentiment} ({predictions[i][0]:.2f})")
```

Output:

```
I love this product → Positive (0.53)
worst movie → Negative (0.51)
I am so happy → Positive (0.56)
I hate this book → Negative (0.50)
great day → Positive (0.51)
I am very disappointed → Negative (0.45)
```

Practical No. 3

Aim: Creating a chatbot using advanced techniques like transformer models

Writeup:

- **Import Libraries:** It imports tensorflow and several components from the transformers library, including BertTokenizer, TFBertForQuestionAnswering, and pipeline. These are essential for loading and using pre-trained transformer models.
- **Define Context and Questions:**
  - ***context:*** This is the block of text from which the model will find answers to the questions. It contains information about various Indian national symbols.
  - ***question:*** This is a list of questions about the information provided in the context.
- **Load Model and Tokenizer:**
  - ***AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2"):*** This line loads the tokenizer associated with the "deepset/bert-base-cased-squad2" pre-trained model. The tokenizer is responsible for converting text (both the context and the questions) into numerical IDs that the BERT model can understand.
  - ***TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-squad2",from_pt=True):*** This line loads the pre-trained BERT model specifically fine-tuned for the Question Answering task on the SQuAD2.0 dataset. from_pt=True indicates that the model weights are being loaded from a PyTorch checkpoint.
- **Build the Question Answering Pipeline:**
  - ***pipeline('question-answering',model = model,tokenizer = tokenizer):*** The pipeline utility from transformers is used to create a streamlined workflow for the question answering task. It combines the loaded model and tokenizer into a single object that can easily take a question and context as input and return the predicted answer.
- **Define Chatbot Function:**
  - ***def chatbot(question,context)::*** This function encapsulates the question answering process.
  - Inside the function, ***nlp({"question": question, "context": context})*** uses the question answering pipeline to find the answer to the given question within the provided context.
  - ***return output['answer']:*** The function extracts and returns the predicted answer from the pipeline's output.
- **Ask Questions and Print Answers:** The code then iterates through a few of the questions defined earlier, calls the chatbot function with each question and the context, and prints the question along with the answer returned by the chatbot.

Code:

```
#optional → !pip install transformers torch tensorflow
import tensorflow as tf
import transformers
from transformers import BertTokenizer
from transformers import TFBertForQuestionAnswering
```

```
from transformers import AutoTokenizer
from transformers import pipeline
print("Transformers version:", transformers.__version__)
print("TensorFlow version:", tf.__version__)
print("Model path or name:", "deepset/bert-base-cased-squad2")
# This is the given context
context = """
The Bengal tiger was chosen as the national animal in a meeting of the Indian wildlife board in
1972 and was adopted officially in April 1973. It was chosen over the Asiatic lion due to the
wider presence of the tiger across India.
Indian peacock was designated as the national bird of India in February 1963.
Indian elephant is the largest terrestrial mammal in India and a cultural symbol throughout its
range, appearing in various religious traditions and mythologies.
Indian banyan is a large tree native to the Indian subcontinent and produces aerial roots from the
branches which grow downwards, eventually becoming trunks
Mango is a large fruit tree with many varieties, believed to have originated in northeast India.
Lotus is an aquatic plant adapted to grow in the flood plains. Lotus seeds can remain dormant
and viable for many years, therefore the plant is regarded as a symbol of longevity.
"""
# These are some questions which we are going to ask to BERT model
question = [
    "What is the National animal of India?",
    "What is the National bird of India?",
    "What is the largest terrestrial mammal in India?",
    "Which tree produces aerial roots?",
    "Which fruit originated in northeast India?",
    "What is the symbol of longevity?",
]
# Importing the tokenizer of the BERT model
tokenizer = AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2")
# Importing the BERT model
model = TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-
squad2",from_pt=True)
# Tokenizing the first question and print the encoded output
tokenizer.encode(question[0],truncation = True, padding = True)
# Building the BERT "Question - Answering" pipeline
nlp = pipeline('question-answering',model = model,tokenizer = tokenizer)
# Making a chatbot function for question answering
def chatbot(question,context):
  output = nlp({
      "question": question,
      "context": context
  })
  return output['answer']
# Asking the first question to the chatbot
print("Question: ",question[1])
```

```
print("Answer: ",chatbot(question[1],context))
print("Question: ",question[2])
print("Answer: ",chatbot(question[2],context))
print("Question: ",question[3])
print("Answer: ",chatbot(question[3],context))
print("Question: ",question[4])
print("Answer: ",chatbot(question[4],context))
print("Question: ",question[5])
print("Answer: ",chatbot(question[5],context))
```

Output:

```
Transformers version: 4.55.4
TensorFlow version: 2.19.0
Model path or name: deepset/bert-base-cased-squad2

[101, 1327, 1110, 1103, 1305, 3724, 1104, 1726, 136, 102]
Question:  What is the National bird of India?
Answer:  Indian peacock
Question:  What is the largest terrestrial mammal in India?
Answer:  Indian elephant
Question:  Which tree produces aerial roots?
Answer:  Indian banyan
Question:  Which fruit originated in northeast India?
Answer:  Mango
Question:  What is the symbol of longevity?
Answer:  Lotus
```

Practical No. 4

Aim: Developing a recommendation system using collaborative filtering or deep learning approaches.

Writeup:

- **Import Libraries:** It imports numpy for numerical operations and modules from tensorflow.keras for building the neural network model (Input, Embedding, Dot, Flatten, Model).
- **Sample Data:** ratings is a small NumPy array representing sample user-item ratings. Each row is a [user_id, item_id, rating] triplet.
- **Determine Dimensions:** It calculates the number of unique users (num_users) and items (num_items) from the ratings data to set the input dimensions for the embedding layers.
- **Define Embedding Size:** embedding_size is set to 2. This determines the size of the learned vector representation (embedding) for each user and item.
- **Define Model Inputs:** user_input and item_input are defined as Keras Input layers, specifying that the model will take two separate inputs, one for user IDs and one for item IDs.
- **Create Embedding Layers:**
  - ***user_embedding = Embedding(...):*** An Embedding layer is created for users. It takes the num_users as input dimension and outputs a vector of size embedding_size for each user ID.
  - ***item_embedding = Embedding(...):*** An Embedding layer is created similarly for items. These layers learn to represent users and items as dense vectors in a low-dimensional space.
- **Calculate Dot Product:**
  - ***dot_product = Dot(axes=2)([user_embedding, item_embedding]):*** A Dot layer is used to calculate the dot product between the user embedding and the item embedding. The dot product of these learned vectors is used to predict the rating. axes=2 specifies that the dot product is calculated across the last dimension of the input tensors (the embedding dimension).
- **Flatten Output:** ***dot_product = Flatten()(dot_product):*** The output of the dot product is flattened into a 1D tensor.
- **Define the Model:** ***model = Model(inputs=[user_input, item_input], outputs=dot_product):*** A Keras Model is created, specifying the input layers (user and item inputs) and the output layer (the flattened dot product). This model takes a user ID and an item ID and outputs a predicted rating.
- **Compile the Model:** ***model.compile(...):*** The model is compiled with the adam optimizer and mse (Mean Squared Error) loss function. The goal is to minimize the difference between the predicted rating and the actual rating.
- **Prepare Training Data:** The ratings data is separated into user IDs (X_users), item IDs (X_items), and actual ratings (y_ratings). The data types are cast to integers and floats as needed.
- **Train the Model:** ***model.fit([X_users, X_items], y_ratings, epochs=100, verbose=1):*** The model is trained on the prepared data. The model learns the user and item embeddings such

that their dot product approximates the known ratings. It trains for 100 epochs, and verbose=1 shows the training progress.

- **Make Predictions:** The code then defines arrays of user_ids and item_ids for which to predict ratings. ***model.predict()*** is used to get the predicted ratings for these user-item pairs.
- **Print Predictions:** The code iterates through the predicted ratings and prints the predicted rating for each user-item pair.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Embedding, Dot, Flatten
from tensorflow.keras.models import Model
# Sample user-item-rating data
ratings = np.array([
    [0, 0, 5.0],  # user 0 rated item 0 with 5
    [0, 1, 3.0],  # user 0 rated item 1 with 3
    [1, 0, 4.0],  # user 1 rated item 0 with 4
    [1, 2, 2.0],  # user 1 rated item 2 with 2
    [2, 1, 4.0],  # user 2 rated item 1 with 5
    [2, 2, 5.0],  # user 2 rated item 2 with 5
])
# Number of users and items
num_users = int(np.max(ratings[:, 0])) + 1  # 3 users
num_items = int(np.max(ratings[:, 1])) + 1  # 3 items
embedding_size = 2
print(np.max(ratings[:, 0]),'\n',int(np.max(ratings[:, 0])))
# Inputs for user and item
user_input = Input(shape=(1,))
item_input = Input(shape=(1,))
embedding_size = 2
user_embedding = Embedding(input_dim=num_users,output_dim=embedding_size)(user_input)
item_embedding =Embedding(input_dim=num_items,output_dim=embedding_size)(item_input)
# Dot product of user and item embeddings
dot_product = Dot(axes=2)([user_embedding, item_embedding])
# Flatten the result
dot_product = Flatten()(dot_product)
# Define the model
model = Model(inputs=[user_input, item_input], outputs=dot_product)
model.compile(optimizer='adam', loss='mse')
# Training data
X_users = ratings[:, 0].astype('int32')
X_items = ratings[:, 1].astype('int32')
y_ratings = ratings[:, 2].astype('float32')
X_users
X_items
y_ratings
```

```
# Train the model
model.fit([X_users, X_items], y_ratings, epochs=100, verbose=1)
# Predict for multiple user-item pairs
user_ids = np.array([0, 1, 2])
item_ids = np.array([2, 0, 1])
predicted_batch = model.predict([user_ids, item_ids])
for u, i, p in zip(user_ids, item_ids, predicted_batch):
    print(f"Predicted rating for user {u} on item {i}: {p.item():.2f}")
ratings
# Predict for multiple user-item pairs
user_ids = np.array([0, 1, 2])
item_ids = np.array([0, 2, 1])
predicted_batch = model.predict([user_ids, item_ids])
for u, i, p in zip(user_ids, item_ids, predicted_batch):
    print(f" User {u} - item {i} - Predicted rating : {p.item():.2f}")
```

Output:

```
2.0
 2

array([0, 0, 1, 1, 2, 2], dtype=int32)
array([0, 1, 0, 2, 1, 2], dtype=int32)
array([5., 3., 4., 2., 4., 5.], dtype=float32)

Predicted rating for user 0 on item 2: 0.01
Predicted rating for user 1 on item 0: 0.04
Predicted rating for user 2 on item 1: 0.04

array([[0., 0., 5.],
       [0., 1., 3.],
       [1., 0., 4.],
       [1., 2., 2.],
       [2., 1., 4.],
       [2., 2., 5.]])

User 0 - item 0 - Predicted rating : 0.04
 User 1 - item 2 - Predicted rating : 0.01
 User 2 - item 1 - Predicted rating : 0.04
```

Practical No. 5

Aim: Implementing a computer vision project, such as object detection or image segmentation.

Writeup:

- **Import Libraries:** It imports cv2 (OpenCV) for image processing, numpy for numerical operations (especially for defining color ranges), and cv2_imshow from google.colab.patches to display images in Google Colab.
- **Load the Image:** ***image = cv2.imread('/content/image.png')*** loads an image file into a NumPy array, which is how OpenCV represents images.
- **Convert to HSV:** ***hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)*** converts the image from the BGR color space (OpenCV's default) to the HSV (Hue, Saturation, Value) color space. This conversion is crucial for color-based segmentation because colors are represented in a way that makes it easier to define color ranges.
    - **Hue (H):** Represents the color type (e.g., red, blue, green).
    - **Saturation (S):** Represents the purity or intensity of the color.
    - **Value (V):** Represents the brightness of the color.
- **Define Color Range** (Examples: blue, green, red):
    - ***lower_blue = np.array([100, 150, 0])*** and ***upper_blue = np.array([140, 255, 255]):*** These lines define the lower and upper bounds for the blue color range in HSV. Pixels with HSV values within this range will be considered blue.
    - Similar lower and upper bounds are defined for green and red. Choosing appropriate HSV ranges for the colors you want to segment is a key part of this technique.
- **Create Mask:** ***mask = cv2.inRange(hsv, lower_blue, upper_blue):*** This is the core segmentation step. cv2.inRange() takes the HSV image and the lower and upper color bounds. It creates a binary mask where pixels within the specified color range are set to white (255), and all other pixels are set to black (0). This mask essentially isolates the pixels of the target color.
- **Segment Image:** ***segmented = cv2.bitwise_and(image, image, mask=mask):*** This line applies the generated mask to the original image. cv2.bitwise_and() performs a bitwise AND operation between the original image and itself, but only where the mask is non-zero (white). This effectively keeps the original pixel values only for the areas where the mask is white (the segmented color) and sets the rest to black, resulting in the segmented image showing only the target color.
- **Show Images:** ***cv2_imshow(image)*** and ***cv2_imshow(segmented):*** These lines display the original image and the segmented image using cv2_imshow, which is necessary for displaying OpenCV images in a Colab environment.

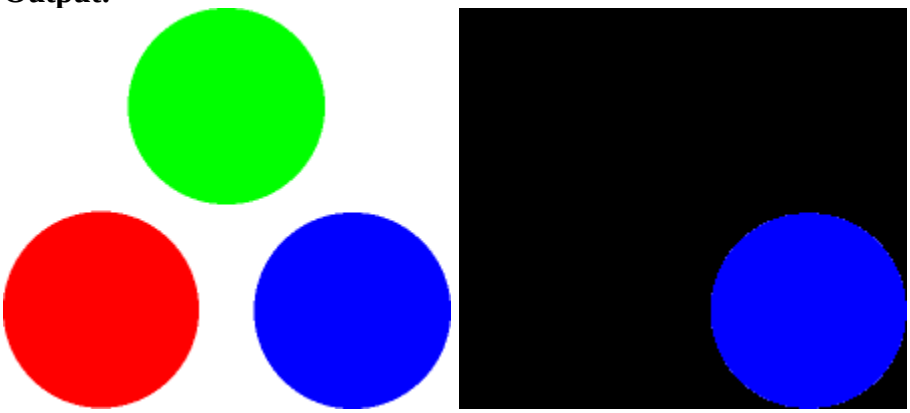
The code repeats the process for blue, green, and red color ranges to demonstrate segmenting different colors.

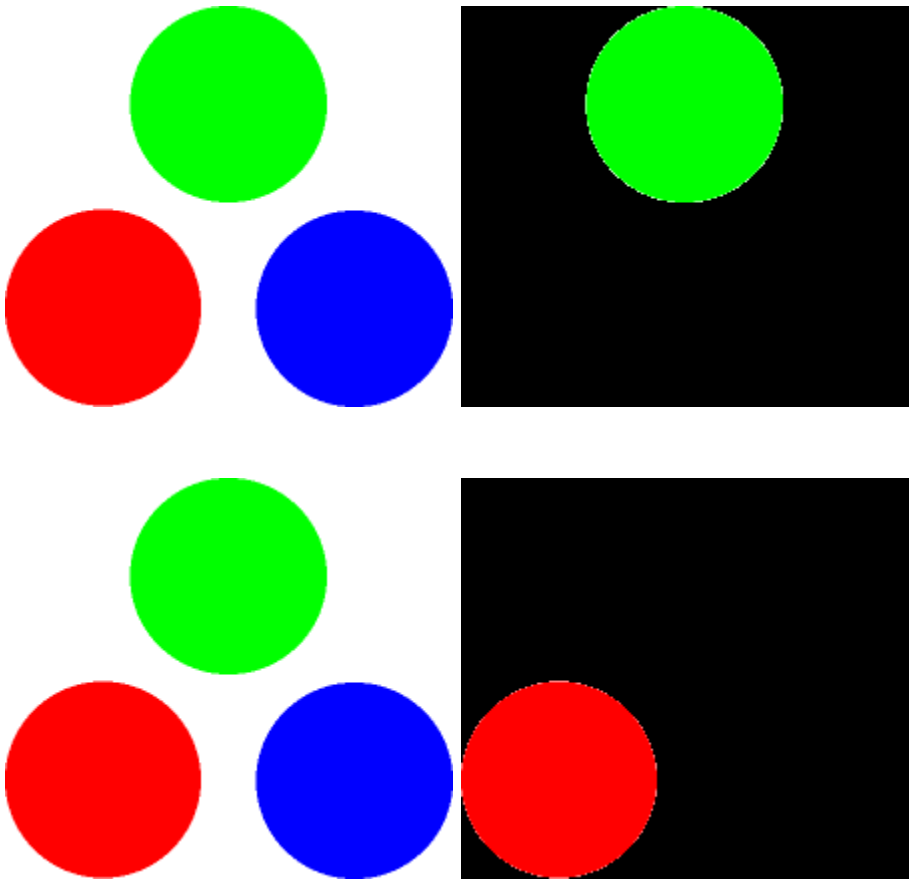
Code:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
# Load the image
```

```
image = cv2.imread('/content/image.png')
# Convert to HSV
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
# Define color range (example: blue)
lower_blue = np.array([100, 150, 0])
upper_blue = np.array([140, 255, 255])
# Create mask
mask = cv2.inRange(hsv, lower_blue, upper_blue)
# Segment image
segmented = cv2.bitwise_and(image, image, mask=mask)
# Show images using cv2_imshow in Colab
cv2_imshow(image)
cv2_imshow(segmented)
#Define color range (example: green)
lower_green = np.array([40, 40, 40])
upper_green = np.array([80, 255, 255])
# Create mask
mask = cv2.inRange(hsv, lower_green, upper_green)
# Segment image
segmented = cv2.bitwise_and(image, image, mask=mask)
# Show images using cv2_imshow in Colab
cv2_imshow(image)
cv2_imshow(segmented)
#Define color range (example: red)
lower_red = np.array([0, 100, 100])
upper_red = np.array([10, 255, 255])
# Create mask
mask = cv2.inRange(hsv, lower_red, upper_red)
# Segment image
segmented = cv2.bitwise_and(image, image, mask=mask)
# Show images using cv2_imshow in Colab
cv2_imshow(image)
cv2_imshow(segmented)
```

Output:





Practical No. 6

Aim: Training a generative adversarial network (GAN) for generating realistic images.

Writeup:

- **Import Libraries:** It imports tensorflow for building and training the neural networks, numpy for numerical operations, and matplotlib.pyplot for visualizing the generated images.
- **Load and Preprocess Data:** The MNIST dataset of handwritten digits is loaded. The images are reshaped to have a channel dimension and normalized to the range [-1, 1]. Normalizing to this range is common in GANs, especially when using tanh activation in the generator's output layer. A tf.data.Dataset is created for efficient batching and shuffling of the training data.
- **Define Hyperparameters:** batch_size, noise_dim (the size of the random noise vector fed to the generator), and epochs (the number of training cycles) are defined.
- **Define the Generator Model:** ***generator = tf.keras.Sequential([...])*** defines the generator network. Its purpose is to take a random noise vector (noise_dim) and transform it into an image that looks like the training data.
    - It starts with a Dense layer to project the noise into a higher dimension.
    - Reshape changes the output shape to a 3D tensor (like a small image with many channels).
    - BatchNormalization helps stabilize training.
    - LeakyReLU is an activation function commonly used in GANs.
    - Conv2DTranspose (also known as deconvolution or upsampling) layers are used to increase the spatial dimensions of the data, gradually building up an image from the smaller representation. The strides greater than 1 (strides=2) are key to upsampling.
    - The final Conv2DTranspose layer outputs a single-channel image (grayscale) with a tanh activation function, which outputs values in the range [-1, 1], matching the normalized input data.
- **Define the Discriminator Model:** ***discriminator = tf.keras.Sequential([...])*** defines the discriminator network. Its purpose is to take an image (either a real image from the dataset or a fake image from the generator) and output a single value indicating whether it thinks the image is real (closer to 1) or fake (closer to 0).
    - Conv2D layers with strides greater than 1 (strides=2) are used to downsample the image and extract features.
    - LeakyReLU and Dropout (to prevent overfitting) are used.
    - Flatten converts the 2D feature maps into a 1D vector.
    - The final Dense layer outputs a single value.
- **Define Loss Function and Optimizers:**
    - ***loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True):*** Binary crossentropy is used as the loss function. from_logits=True is important because the discriminator's final Dense layer does not have an activation function applied (it outputs logits).
    - ***gen_optimizer*** and ***disc_optimizer:*** Separate Adam optimizers are defined for the generator and discriminator, as they are trained adversarially.

- **Training Loop:** The code enters a loop for the specified number of epochs.
  - Inside the epoch loop, it iterates through batches of real images from the dataset.
  - For each batch, random noise is generated.
    - ***Training the Discriminator:***
      - A tf.GradientTape is used to record operations for automatic differentiation.
      - Fake images are generated by the generator.
      - The discriminator is called on both real and fake images.
      - Losses are calculated: real_loss (discriminator trying to classify real images as real) and fake_loss (discriminator trying to classify fake images as fake). The total disc_loss is the sum of these.
      - Gradients of the disc_loss with respect to the discriminator's trainable variables are computed and applied using the disc_optimizer.
    - ***Training the Generator:***
      - Another tf.GradientTape is used.
      - Fake images are generated.
      - The discriminator is called on the fake images.
      - gen_loss is calculated: the generator tries to make the discriminator classify the fake images as real (tf.ones_like(fake_output)).
      - Gradients of the gen_loss with respect to the generator's trainable variables are computed and applied using the gen_optimizer.
- **Visualization:** After each epoch, the generator is used to generate a sample of images from a fixed seed noise vector (to see how the generation improves over epochs). These images are rescaled back to [0, 1] and displayed using matplotlib.pyplot.

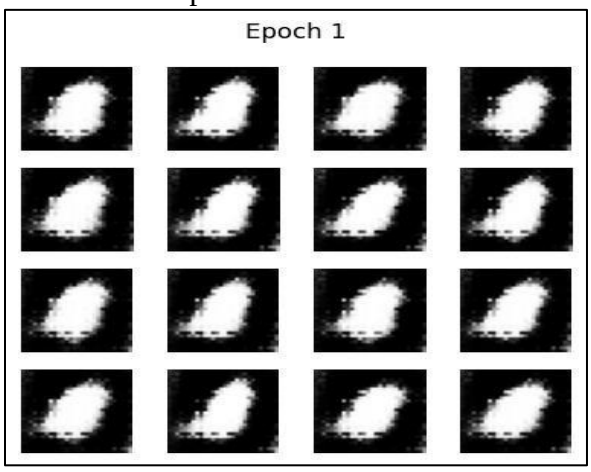
Code:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype("float32")
x_train = (x_train - 127.5) / 127.5
batch_size = 128
dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batch_size)
noise_dim = 100
generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7*7*256, input_shape=(noise_dim,)),
    tf.keras.layers.Reshape((7, 7, 256)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(128, 5, strides=1, padding='same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(64, 5, strides=2, padding='same'),
    tf.keras.layers.BatchNormalization(),
```

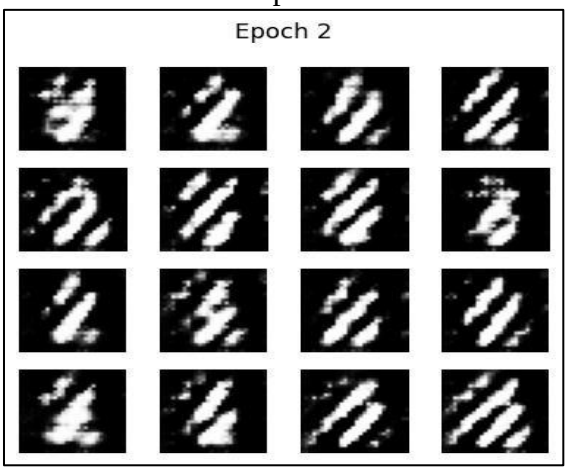
```
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(1, 5, strides=2, padding='same', activation='tanh')
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 5, strides=2, padding='same', input_shape=(28, 28, 1)),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Conv2D(128, 5, strides=2, padding='same'),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1)
])
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
gen_optimizer = tf.keras.optimizers.Adam(1e-4)
disc_optimizer = tf.keras.optimizers.Adam(1e-4)
epochs = 30
seed = tf.random.normal([16, noise_dim])
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    for real_images in dataset:
        noise = tf.random.normal([batch_size, noise_dim])
        # Generate fake images
        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            fake_images = generator(noise, training=True)
            real_output = discriminator(real_images, training=True)
            fake_output = discriminator(fake_images, training=True)
            gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)
            real_loss = loss_fn(tf.ones_like(real_output), real_output)
            fake_loss = loss_fn(tf.zeros_like(fake_output), fake_output)
            disc_loss = real_loss + fake_loss
        gradients_gen = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_disc = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
        gen_optimizer.apply_gradients(zip(gradients_gen, generator.trainable_variables))
        disc_optimizer.apply_gradients(zip(gradients_disc, discriminator.trainable_variables))
    generated_images = generator(seed, training=False)
    generated_images = (generated_images + 1) / 2.0
    fig = plt.figure(figsize=(4, 4))
    for i in range(16):
        plt.subplot(4, 4, i+1)
        plt.imshow(generated_images[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle(f'Epoch {epoch+1}')
    plt.tight_layout()
    plt.show()
```

Output:

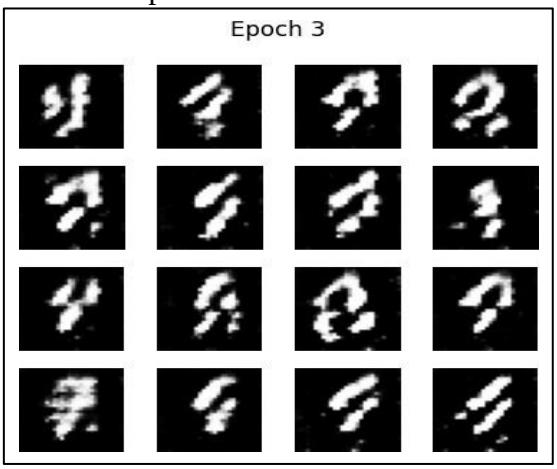
Epoch 1/30



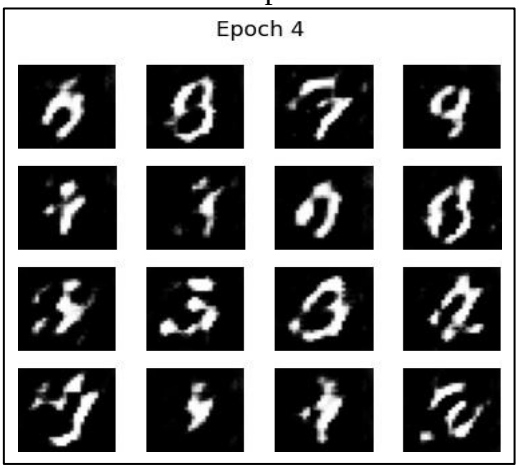
Epoch 2/30



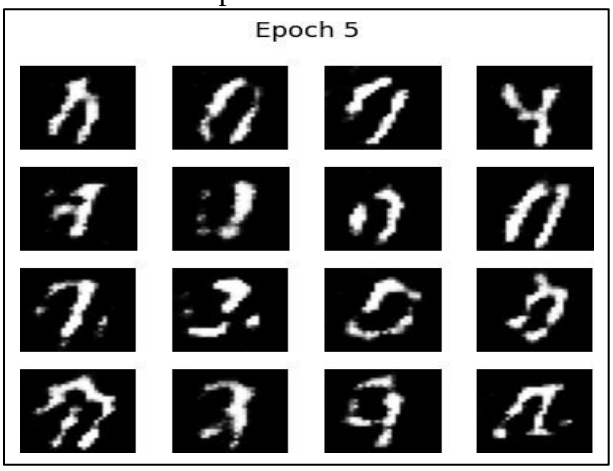
Epoch 3/30



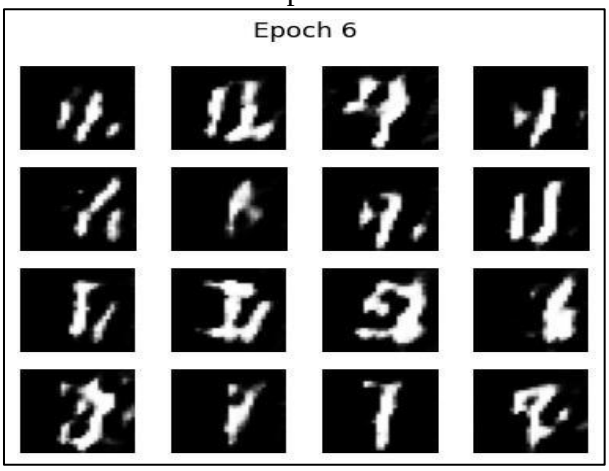
Epoch 4/30



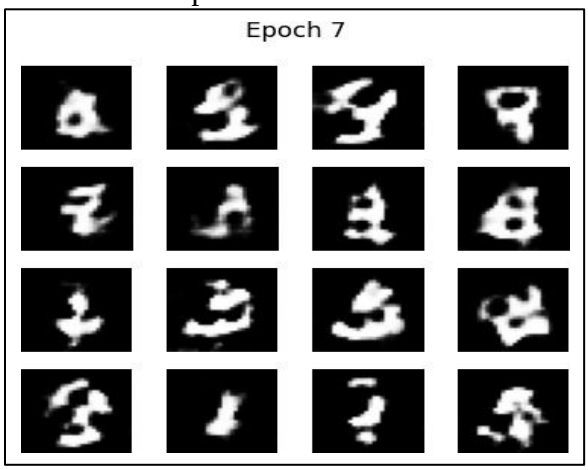
Epoch 5/30



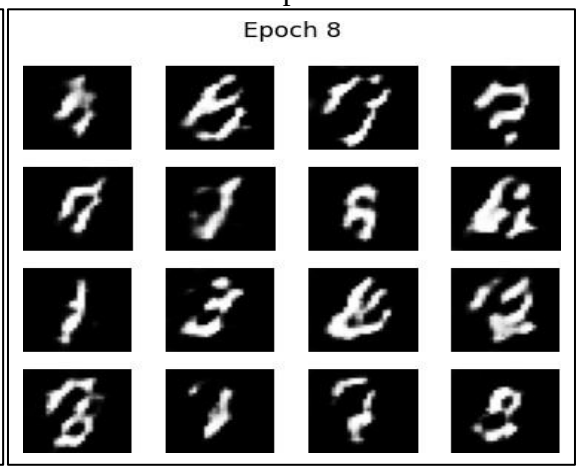
Epoch 6/30



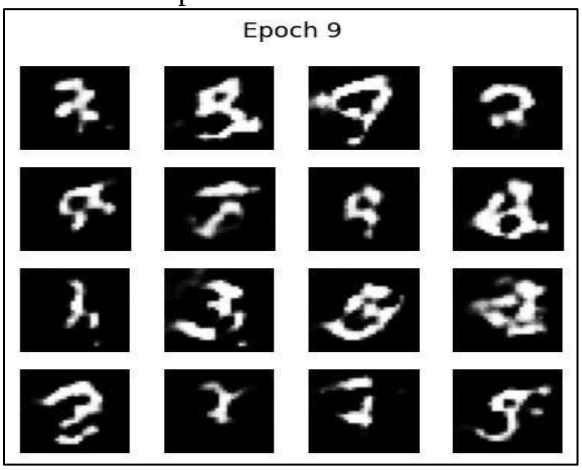
Epoch 7/30



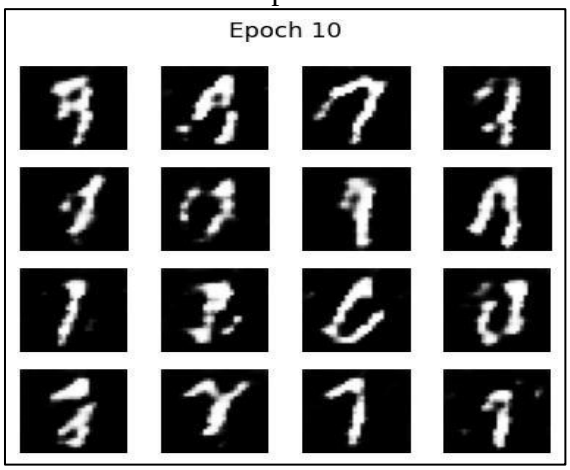
Epoch 8/30



Epoch 9/30



Epoch 10/30



Practical No. 7

Aim: Applying reinforcement learning algorithms to solve complex decision-making problems.

Writeup:

- **Import Libraries:** It imports numpy for numerical operations and random for random action selection.
- **Environment Setup:**
  - ***grid_size = 4:*** Defines the size of the grid (4x4).
  - ***actions = ['up', 'down', 'left', 'right']:*** Defines the possible actions the agent can take.
  - ***action_dict:*** A dictionary mapping numerical action indices (0-3) to their corresponding changes in row and column coordinates.
- **Q-table Initialization:**
  - ***q_table = np.zeros((grid_size, grid_size, len(actions))):*** This is the core of the Q-learning algorithm. It's a NumPy array (the Q-table) that stores the learned Q-values. The dimensions are (number of rows, number of columns, number of actions). q_table[state_row, state_col, action_index] will store the estimated future reward of taking action_index from the state (state_row, state_col). It's initialized with zeros.
- **Hyperparameters:**
  - ***alpha = 0.1:*** The learning rate. It determines how much the newly acquired information overrides the old information. A higher value means the agent learns faster but might be less stable.
  - ***gamma = 0.9:*** The discount factor. It determines the importance of future rewards. A value closer to 1 means the agent considers future rewards more heavily.
  - ***epsilon = 0.2:*** The exploration rate for the epsilon-greedy policy. It determines the probability of the agent taking a random action instead of the action with the highest Q-value. This encourages exploration of the environment.
  - ***episodes = 500:*** The number of training episodes. An episode is a full run from the start state to the goal state.
- **Reward Function:**
  - ***def get_reward(state)::*** This function defines the reward received for being in a given state.
  - ***if state == (grid_size - 1, grid_size - 1):*** return 10: If the agent reaches the goal state (3,3), it receives a reward of +10, as specified in your problem.
  - ***else:*** return -1: For any other state, the agent receives a reward of -1 for each move, as specified.
- **Next State Function:**
  - ***def next_state(state, action)::*** This function calculates the agent's next state given the current state and the chosen action.
  - It uses the action_dict to determine the change in row and column based on the action.
  - It calculates the new_row and new_col, ensuring the agent stays within the grid boundaries using max(0, min(grid_size - 1, ...)).
  - It returns the (new_row, new_col) as the new_state.

- **Training (Q-learning Algorithm):**
  - ***for episode in range(episodes):***: The main training loop runs for the specified number of episodes.
  - ***state = (0, 0):*** Each episode starts at the initial state (0,0).
  - ***while state != (grid_size - 1, grid_size - 1)::*** The episode continues until the agent reaches the goal state.
  - **Epsilon-Greedy Action Selection:**
    - ***if random.random() < epsilon::*** With probability epsilon, the agent chooses a random action (random.randint(0, 3)).
    - ***else::*** With probability 1 - epsilon, the agent chooses the action with the highest Q-value for the current state (np.argmax(q_table[state[0], state[1]])). This is the greedy action.
    - ***new_state = next_state(state, action):*** Calculate the state the agent moves to based on the chosen action.
    - ***reward = get_reward(new_state):*** Get the reward for being in the new_state.
  - **Q-learning Update Rule:** This is the core learning step:
    - ***old_value = q_table[state[0], state[1], action]:*** Get the current Q-value for the (state, action) pair.
    - ***next_max = np.max(q_table[new_state[0], new_state[1]]):*** Find the maximum Q-value for the new_state across all possible actions. This represents the estimated optimal future reward from the next state.
    - ***q_table[state[0], state[1], action] = old_value + alpha * (reward + gamma * next_max - old_value):*** This is the Q-learning update formula. It updates the old_value based on the reward received, the discounted maximum future reward (gamma * next_max), and the difference between this and the old_value (the temporal difference error), scaled by the alpha learning rate.
    - ***state = new_state:*** Update the current state to the new_state for the next iteration of the while loop.
- **Display Learned Policy:** After training, the code iterates through each state in the grid and prints the action that has the highest learned Q-value (np.argmax(q_table[i, j])). This shows the learned optimal policy for each state.

Code:

```
import numpy as np
import random
# Environment setup
grid_size = 4
actions = ['up', 'down', 'left', 'right']
action_dict = {
   0: (-1, 0),  # up
   1: (1, 0),   # down
   2: (0, -1),  # left
   3: (0, 1)    # right
}
```

```
# Q-table: states are (row, col), actions are 0-3
q_table = np.zeros((grid_size, grid_size, len(actions)))
# Hyperparameters
alpha = 0.1        # Learning rate
gamma = 0.9        # Discount factor
epsilon = 0.2      # Exploration rate
episodes = 500     # Number of episodes
# Reward function
def get_reward(state):
    if state == (grid_size - 1, grid_size - 1):
        return 10
    else:
        return -1
# Next state
def next_state(state, action):
    row, col = state
    dr, dc = action_dict[action]
    new_row = max(0, min(grid_size - 1, row + dr))
    new_col = max(0, min(grid_size - 1, col + dc))
    return (new_row, new_col)
# Training
for episode in range(episodes):
    state = (0, 0)
    while state != (grid_size - 1, grid_size - 1):
        # Epsilon-greedy action selection
        if random.random() < epsilon:
            action = random.randint(0, 3)
        else:
            action = np.argmax(q_table[state[0], state[1]])
        new_state = next_state(state, action)
        reward = get_reward(new_state)
        # Q-learning update
        old_value = q_table[state[0], state[1], action]
        next_max = np.max(q_table[new_state[0], new_state[1]])
        q_table[state[0], state[1], action] = old_value + alpha * (reward + gamma * next_max -
old_value)
        state = new_state
# Display the learned Q-values
for i in range(grid_size):
    for j in range(grid_size):
        best_action = np.argmax(q_table[i, j])
        print(f"({i},{j}): {actions[best_action]}", end=" | ")
    print()
```

Output:

(0,0): down | (0,1): down | (0,2): down | (0,3): down |
(1,0): right | (1,1): down | (1,2): down | (1,3): down |
(2,0): right | (2,1): down | (2,2): down | (2,3): down |
(3,0): right | (3,1): right | (3,2): right | (3,3): up |

Practical No. 8

Aim: Utilizing transfer learning to improve model performance on limited datasets.

Writeup:

- **Import Libraries:** It imports tensorflow and several components from the transformers library, including BertTokenizer, TFBertForQuestionAnswering, and pipeline. These are essential for loading and using pre-trained transformer models.
- **Define Context and Questions:**
  - ***context:*** This is the block of text from which the model will find answers to the questions. It contains information about various Indian national symbols.
  - ***question:*** This is a list of questions about the information provided in the context.
- **Load Model and Tokenizer:**
  - ***AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2"):*** This line loads the tokenizer associated with the "deepset/bert-base-cased-squad2" pre-trained model. The tokenizer is responsible for converting text (both the context and the questions) into numerical IDs that the BERT model can understand.
  - ***TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-squad2",from_pt=True):*** This line loads the pre-trained BERT model specifically fine-tuned for the Question Answering task on the SQuAD2.0 dataset. from_pt=True indicates that the model weights are being loaded from a PyTorch checkpoint.
- **Build the Question Answering Pipeline:**
  - ***pipeline('question-answering',model = model,tokenizer = tokenizer):*** The pipeline utility from transformers is used to create a streamlined workflow for the question answering task. It combines the loaded model and tokenizer into a single object that can easily take a question and context as input and return the predicted answer.
- **Define Chatbot Function:**
  - ***def chatbot(question,context)::*** This function encapsulates the question answering process.
  - Inside the function, ***nlp({"question": question, "context": context})*** uses the question answering pipeline to find the answer to the given question within the provided context.
  - ***return output['answer']:*** The function extracts and returns the predicted answer from the pipeline's output.
- **Ask Questions and Print Answers:** The code then iterates through a few of the questions defined earlier, calls the chatbot function with each question and the context, and prints the question along with the answer returned by the chatbot.

Code:

```
#optional → !pip install transformers torch tensorflow
import tensorflow as tf
import transformers
from transformers import BertTokenizer
```

```
from transformers import TFBertForQuestionAnswering
from transformers import AutoTokenizer
from transformers import pipeline
print("Transformers version:", transformers.__version__)
print("TensorFlow version:", tf.__version__)
print("Model path or name:", "deepset/bert-base-cased-squad2")
# This is the given context
context = """
The Bengal tiger was chosen as the national animal in a meeting of the Indian wildlife board in 1972 and was adopted officially in April 1973. It was chosen over the Asiatic lion due to the wider presence of the tiger across India.
Indian peacock was designated as the national bird of India in February 1963.
Indian elephant is the largest terrestrial mammal in India and a cultural symbol throughout its range, appearing in various religious traditions and mythologies.
Indian banyan is a large tree native to the Indian subcontinent and produces aerial roots from the branches which grow downwards, eventually becoming trunks
Mango is a large fruit tree with many varieties, believed to have originated in northeast India.
Lotus is an aquatic plant adapted to grow in the flood plains. Lotus seeds can remain dormant and viable for many years, therefore the plant is regarded as a symbol of longevity.
"""
# These are some questions which we are going to ask to BERT model
question = [
    "What is the National animal of India?",
    "What is the National bird of India?",
    "What is the largest terrestrial mammal in India?",
    "Which tree produces aerial roots?",
    "Which fruit originated in northeast India?",
    "What is the symbol of longevity?",
]
# Importing the tokenizer of the BERT model
tokenizer = AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2")
# Importing the BERT model
model = TFBertForQuestionAnswering.from_pretrained("deepset/bert-base-cased-
squad2",from_pt=True)
# Tokenizing the first question and print the encoded output
tokenizer.encode(question[0],truncation = True, padding = True)
# Building the BERT "Question - Answering" pipeline
nlp = pipeline('question-answering',model = model,tokenizer = tokenizer)
# Making a chatbot function for question answering
def chatbot(question,context):
  output = nlp({
      "question": question,
      "context": context
  })
  return output['answer']
# Asking the first question to the chatbot
```

```
print("Question: ",question[1])
print("Answer: ",chatbot(question[1],context))
print("Question: ",question[2])
print("Answer: ",chatbot(question[2],context))
print("Question: ",question[3])
print("Answer: ",chatbot(question[3],context))
print("Question: ",question[4])
print("Answer: ",chatbot(question[4],context))
print("Question: ",question[5])
print("Answer: ",chatbot(question[5],context))
```

Output:

```
Transformers version: 4.55.4
TensorFlow version: 2.19.0
Model path or name: deepset/bert-base-cased-squad2

[101, 1327, 1110, 1103, 1305, 3724, 1104, 1726, 136, 102]
Question:  What is the National bird of India?
Answer:  Indian peacock
Question:  What is the largest terrestrial mammal in India?
Answer:  Indian elephant
Question:  Which tree produces aerial roots?
Answer:  Indian banyan
Question:  Which fruit originated in northeast India?
Answer:  Mango
Question:  What is the symbol of longevity?
Answer:  Lotus
```

Practical No. 9

Aim: Building a deep learning model for time series forecasting.

Writeup:

- **Import Libraries:** It imports numpy for numerical operations, tensorflow and Keras components (Sequential, Dense) for building the neural network, and MinMaxScaler from sklearn.preprocessing for data scaling.
- **Sample Time Series Data:** ***sales = np.array(...)*** defines your time series data. In this case, it's a simple sequence of sales values over time. The .reshape(-1, 1) is to make it a 2D array, which is often required for scikit-learn's MinMaxScaler.
- **Data Scaling:**
  - ***scaler = MinMaxScaler():*** An instance of MinMaxScaler is created. This scaler will transform the data so that all values are within a specific range, typically [0, 1]. Scaling is important for many neural networks as it can help with training stability and performance.
  - ***sales_scaled = scaler.fit_transform(sales).flatten():*** The scaler is fitted to your sales data (learning the min and max values) and then transforms the data. .flatten() converts the 2D scaled array back into a 1D array.
- **Prepare Supervised Data (Windowing):**
  - ***n_steps = 3:*** This defines the size of the "window" or the number of past time steps used to predict the next time step. In this code, it uses the last 3 time steps to predict the 4th.
  - The for loop iterates through the scaled time series data to create input-output pairs for supervised learning.
  - ***X.append(sales_scaled[i:i+n_steps]):*** For each position i, a sequence of n_steps values starting from i is extracted and added to the input list X. This represents the historical data points in the window.
  - ***y.append(sales_scaled[i+n_steps]):*** The value immediately following the window (i + n_steps) is extracted and added to the output list y. This is the value the model will try to predict.
  - ***X = np.array(X) and y = np.array(y):*** The lists of inputs and outputs are converted into NumPy arrays, which are required for training a Keras model. X will be a 2D array where each row is a window of past values, and y will be a 1D array of the corresponding next values.
- **Build Deep Learning Model:** ***model = Sequential([...])*** defines a simple feedforward neural network.
  - ***Dense(4, activation='relu', input_shape=(n_steps,)):*** The first dense layer has 4 neurons and uses the ReLU activation function. input_shape=(n_steps,) tells the model that the input to this layer will be sequences of length n_steps (the size of your window).
  - ***Dense(1):*** The output layer has 1 neuron, as you are predicting a single future value.
- **Compile Model:** ***model.compile(optimizer='adam', loss='mse'):*** The model is compiled with the adam optimizer and Mean Squared Error (mse) loss function, which is commonly used for regression tasks like forecasting.

- **Train Model:** ***model.fit(X, y, epochs=500, verbose=0):*** The model is trained on the prepared supervised data (X and y) for 500 epochs. verbose=0 keeps the training progress output silent.
- **Make Prediction:**
  - ***x_input = np.array([18, 17, 19]).reshape(-1, 1):*** A new input sequence (the last 3 original sales values) is defined for prediction. It's reshaped for scaling.
  - ***x_input_scaled = scaler.transform(x_input).flatten().reshape(1, n_steps):*** This new input is scaled using the *same* scaler fitted on the training data. It's then flattened and reshaped into the correct input shape for the model (1 sample, n_steps features).
  - ***y_pred_scaled = model.predict(x_input_scaled, verbose=0):*** The trained model makes a prediction on the scaled input. The output is a scaled prediction.
  - ***y_pred = scaler.inverse_transform([[y_pred_scaled[0][0]]])[0][0]:*** The predicted value is currently scaled. This line uses the inverse_transform method of the scaler to bring the predicted value back to the original sales scale. The nested indexing is to handle the shape of the scaler's input and output.
- **Print Prediction:** ***print(f"Predicted sales on day 11: {y_pred:.2f}"):*** The final predicted sales value on the original scale is printed.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import MinMaxScaler
# Original sales data
sales = np.array([10, 12, 13, 12, 14, 15, 16, 18, 17, 19]).reshape(-1, 1)
# Scale sales data to [0,1]
scaler = MinMaxScaler()
sales_scaled = scaler.fit_transform(sales).flatten()
# Prepare supervised data
X = []
y = []
n_steps = 3
for i in range(len(sales_scaled) - n_steps):
    X.append(sales_scaled[i:i+n_steps])
    y.append(sales_scaled[i+n_steps])
X = np.array(X)
y = np.array(y)
# Build model
model = Sequential([
    Dense(4, activation='relu', input_shape=(n_steps,)),
    Dense(1) ])
model.compile(optimizer='adam', loss='mse')
# Train model
model.fit(X, y, epochs=500, verbose=0)
```

```
# Predict the sales for day 11 using last 3 days [18, 17, 19]
x_input = np.array([18, 17, 19]).reshape(-1, 1)
x_input_scaled = scaler.transform(x_input).flatten().reshape(1, n_steps)
y_pred_scaled = model.predict(x_input_scaled, verbose=0)
y_pred_scaled
y_pred = scaler.inverse_transform([[y_pred_scaled[0][0]]])[0][0]
y_pred
print(f"Predicted sales on day 11: {y_pred:.2f}")
```

Output:

```
array([[0.34281263]], dtype=float32)
np.float64(13.08531364798546)
Predicted sales on day 11: 13.09
```

Practical No. 10

Aim: Using advanced optimization techniques like evolutionary algorithms or Bayesian optimization for hyperparameter tuning.

Writeup:

- **Import Libraries:** It imports random for generating random numbers needed for selection, crossover, and mutation.
- **Initial Population:**
  - ***population = [...]:*** This list represents the initial "population" of candidate solutions. Each dictionary in the list represents an "individual" in the population, and its keys (id, hidden, learning rate) represent its "genes" or the specific combination of hyperparameters being tested. The comments indicate assumed fitness values, although the code calculates fitness later.
- **Define Parameters:**
  - ***generations = 5:*** The number of iterations the evolutionary process will run. Each generation involves evaluating the current population, selecting parents, creating offspring, and forming a new population.
  - ***mutation_rate = 0.4:*** The probability that a gene (hyperparameter value) in an offspring will be randomly changed (mutated).
  - ***hidden_bounds = (10, 100)*** and ***learningrate_bounds = (0.001, 0.1):*** These define the valid range of values for the hidden and learning rate hyperparameters. Mutation and crossover results are constrained within these bounds.
- **Training Loop (Generations):** for gen in range(generations): The main loop runs for the specified number of generations.
- **Fitness Calculation:**
  - ***fitness_scores = []:*** An empty list to store the fitness score for each individual in the current population.
  - The inner loop iterates through each ind (individual) in the population.
  - ***fitness = 0.8 + (ind['hidden'] / 200) - abs(ind['learningrate'] - 0.03) * 2:*** This is the fitness function. In a real hyperparameter tuning scenario, this function would represent the performance of a model trained with these hyperparameters (e.g., accuracy on a validation set). Here, it's a simplified mathematical formula that rewards individuals with hidden values closer to 200 and learning rate values closer to 0.03. The goal of the genetic algorithm is to find individuals that maximize this fitness score.
  - ***fitness_scores.append(fitness):*** The calculated fitness for each individual is added to the list.
- **Selection:**
  - ***sorted_indices = sorted(...):*** The indices of the individuals are sorted based on their fitness_scores in descending order (highest fitness first).
  - ***p1 = population[sorted_indices[0]]*** and ***p2 = population[sorted_indices[1]]:*** The two individuals with the highest fitness scores (the "fittest") are selected as "parents" for the next generation. This is a simple form of selection called truncation selection.
- **Crossover (Recombination):**

- ***child_hidden = int((p1['hidden'] + p2['hidden']) / 2)*** and ***child_learningrate = round((p1['learningrate'] + p2['learningrate']) / 2, 4)***: A new individual (the "child") is created by combining the genes (hyperparameter values) of the two selected parents. Here, it's a simple averaging of the hidden and learning rate values. More complex crossover strategies exist in genetic algorithms.

- **Mutation:**
  - ***if random.random() < mutation_rate::*** With a probability determined by mutation_rate, the child's genes are subjected to mutation.
  - ***child_hidden += random.randint(-5, 5)*** and ***child_learningrate += round(random.uniform(-0.01, 0.01), 4):*** Random noise is added to the child's hyperparameter values within a small range.
  - ***max(min(..., bounds[1]), bounds[0]):*** The mutated values are then clamped to stay within the defined hidden_bounds and learningrate_bounds. Mutation introduces diversity into the population and helps the algorithm explore the search space to avoid getting stuck in local optima.
- **Evaluate Child: *child_fitness = 0.8 + (...):*** The fitness of the newly created child is calculated using the same fitness function.
- **Replacement (Forming the Next Generation):**
  - ***worst_idx = fitness_scores.index(min(fitness_scores)):*** The index of the individual with the lowest fitness in the current population is found.
  - ***if child_fitness > fitness_scores[worst_idx]:*** population[worst_idx] = {'id': '*', ...}: If the child's fitness is better than the worst individual in the current population, the child replaces the worst individual. This ensures that the population, on average, becomes fitter over generations.
  - ***else:*** print(...): If the child is not better than the worst, it is not included in the next generation.
- **Final Population:** After all generations, the code prints the hyperparameters and fitness of the individuals in the final population. The individual with the highest fitness in the final population represents the best combination of hyperparameters found by the algorithm.

Code:

```
import random
# Initial population (dataset)
population = [
   {'id': 'A', 'hidden': 20, 'learningrate': 0.01},  # Assume fitness = 0.82
   {'id': 'B', 'hidden': 50, 'learningrate': 0.05},  # Assume fitness = 0.87
   {'id': 'C', 'hidden': 80, 'learningrate': 0.02},  # Assume fitness = 0.78
]
population
# Fitness function calculation  0.8 + (hidden / 200) - abs(lr - 0.03) * 2
generations = 5
mutation_rate = 0.4
hidden_bounds = (10, 100)
learningrate_bounds = (0.001, 0.1)
for gen in range(generations):
   print(f"\n--- Generation {gen + 1} ---")
```

```
    # Calculate fitness
    fitness_scores = []
    for ind in population:
        fitness = 0.8 + (ind['hidden'] / 200) - abs(ind['learningrate'] - 0.03) * 2
        fitness_scores.append(fitness)
    # Print population
    for i, ind in enumerate(population):
        print(f"ID {ind['id']}: hidden={ind['hidden']}, learningrate={ind['learningrate']:.4f},
fitness={fitness_scores[i]:.4f}")
    # Select best two individuals
    sorted_indices = sorted(range(len(fitness_scores)), key=lambda i: fitness_scores[i],
reverse=True)
    p1 = population[sorted_indices[0]]
    p2 = population[sorted_indices[1]]
    print(f"\nSelected parents: {p1['id']} and {p2['id']}")
    # Crossover
    child_hidden = int((p1['hidden'] + p2['hidden']) / 2)
    child_learningrate = round((p1['learningrate'] + p2['learningrate']) / 2, 4)
    # Mutation
    if random.random() < mutation_rate:
        child_hidden += random.randint(-5, 5)
        child_hidden = max(min(child_hidden, hidden_bounds[1]), hidden_bounds[0])
    if random.random() < mutation_rate:
        child_learningrate += round(random.uniform(-0.01, 0.01), 4)
        child_learningrate = max(min(child_learningrate, learningrate_bounds[1]),
learningrate_bounds[0])
    # Evaluate child
    child_fitness = 0.8 + (child_hidden / 200) - abs(child_learningrate - 0.03) * 2
    print(f"\nChild: hidden={child_hidden}, learningrate={child_learningrate:.4f},
fitness={child_fitness:.4f}")
    # Replace worst if better
    worst_idx = fitness_scores.index(min(fitness_scores))
    if child_fitness > fitness_scores[worst_idx]:
        print(f"Child replaces individual {population[worst_idx]['id']}")
        population[worst_idx] = {'id': '*', 'hidden': child_hidden, 'learningrate': child_learningrate}
    else:
        print("Child is not better than the worst individual. No replacement.")
# Final population
print("\n=== Final Population ===")
for ind in population:
    final_fitness = 0.8 + (ind['hidden'] / 200) - abs(ind['learningrate'] - 0.03) * 2
    print(f"ID {ind['id']}: hidden={ind['hidden']}, learningrate={ind['learningrate']:.4f},
fitness={final_fitness:.4f}")
```

Output:

```
[{'id': 'A', 'hidden': 20, 'learningrate': 0.01},
 {'id': 'B', 'hidden': 50, 'learningrate': 0.05},
 {'id': 'C', 'hidden': 80, 'learningrate': 0.02}]

--- Generation 1 ---
ID A: hidden=20, learningrate=0.0100, fitness=0.8600
ID B: hidden=50, learningrate=0.0500, fitness=1.0100
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and B

Child: hidden=65, learningrate=0.0350, fitness=1.1150
Child replaces individual A

--- Generation 2 ---
ID *: hidden=65, learningrate=0.0350, fitness=1.1150
ID B: hidden=50, learningrate=0.0500, fitness=1.0100
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=72, learningrate=0.0312, fitness=1.1576
Child replaces individual B

--- Generation 3 ---
ID *: hidden=65, learningrate=0.0350, fitness=1.1150
ID *: hidden=72, learningrate=0.0312, fitness=1.1576
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=72, learningrate=0.0256, fitness=1.1512
Child replaces individual *

--- Generation 4 ---
ID *: hidden=72, learningrate=0.0256, fitness=1.1512
ID *: hidden=72, learningrate=0.0312, fitness=1.1576
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=76, learningrate=0.0256, fitness=1.1712
Child replaces individual *

--- Generation 5 ---
```

ID *: hidden=76, learningrate=0.0256, fitness=1.1712
ID *: hidden=72, learningrate=0.0312, fitness=1.1576
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Selected parents: C and *

Child: hidden=78, learningrate=0.0228, fitness=1.1756
Child replaces individual *

=== Final Population ===
ID *: hidden=76, learningrate=0.0256, fitness=1.1712
ID *: hidden=78, learningrate=0.0228, fitness=1.1756
ID C: hidden=80, learningrate=0.0200, fitness=1.1800

Practical No. 11

Aim: Use Python libraries such as GPT-2 or textgenrnn to train generative models on a corpus of text data and generate new text based on the patterns it has learned.

Writeup:

- **Import Libraries:** It imports numpy, pandas, tensorflow, and warnings (though these aren't directly used in the text generation itself) and crucially, components from the transformers library: GPT2LMHeadModel, GPT2Tokenizer, and pipeline. These are necessary for working with pre-trained transformer models like GPT-2.
- **Define Prompt:** ***text = "Once upon a time, in a magical forest, there lived a curious elf named Elara. She had":*** This line defines the initial string of text that the GPT-2 model will use as a starting point for generating new text.
- **Define generate_text Function:** This function encapsulates the text generation process.
  - ***generator = pipeline('text-generation', model='gpt2'):*** This line is key. It utilizes the pipeline utility from the transformers library to set up a text generation task. It automatically loads the pre-trained gpt2 model and its corresponding tokenizer. This simplifies the process of using the model for generation.
  - ***generated_text = generator(prompt, max_length=200, num_return_sequences=1, truncation=True):*** This line performs the text generation.
  - ***prompt:*** The starting text provided to the model.
  - ***max_length=200:*** Specifies the maximum length of the generated text (including the prompt).
  - ***num_return_sequences=1:*** Requests only one generated output sequence.
  - ***truncation=True:*** Ensures that the prompt is truncated if it exceeds the model's maximum input length.
  - ***return generated_text[0]['generated_text']:*** The function extracts and returns the generated text string from the pipeline's output.
- **Generate and Display Text:**
  - ***output = generate_text(text):*** The generate_text function is called with the defined starting text, and the generated output is stored in the output variable.
  - ***output:*** In a Colab code cell, having a variable name as the last line displays its value, showing the generated text.

Code:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import warnings
from transformers import GPT2LMHeadModel, GPT2Tokenizer, pipeline
text = "Once upon a time, in a magical forest, there lived a curious elf named Elara. She had"
#text= input("Enter your sentence here\n")
def generate_text(prompt):
    generator = pipeline('text-generation', model='gpt2')
```

```
    generated_text = generator(prompt, max_length=200, num_return_sequences=1,
truncation=True)
    return generated_text[0]['generated_text']
    #print(generated_text[0]['generated_text'])
output=generate_text(text)
output
```

Output:

Once upon a time, in a magical forest, there lived a curious elf named Elara. She had a strange, seemingly random nature and, when she came across the young man she suddenly felt as if she were on a journey to find out more about him.

Elara had long since become a magical girl and was known for being a very good at reading and writing. By the time she was twenty-two years old, she had found a place to live in a small village called Welt. The village was located on an island in the middle of the continent and Elara had been taken there by the spirit of the elves she had met on the island.

In the short time she had been there, Elara had learned that the village was haunted. The spirit of the elves she had met on the island had been the one that had made it so that no one found her.

So she was sent to the village of Welt, and to the world she had come to live in. The spirit of the elves she had met on the island had been that of a young boy named Elara.

The magical elves were not the only ones who had witnessed the spirit of the elves on the island. It was obvious that the spirit of the elves had also been in the village. The elves had come to the village as a group.

Practical No. 12

Aim: Experiment with neural networks like GANs (Generative Adversarial Networks) using Python libraries like TensorFlow or PyTorch to generate new images based on a dataset of images.

Writeup:

- **Import Libraries**: It imports tensorflow for building and training the neural networks, numpy for numerical operations, and matplotlib.pyplot for visualizing the generated images.
- **Load and Preprocess Data**: The MNIST dataset of handwritten digits is loaded. The images are reshaped to have a channel dimension and normalized to the range [-1, 1]. Normalizing to this range is common in GANs, especially when using tanh activation in the generator's output layer. A tf.data.Dataset is created for efficient batching and shuffling of the training data.
- **Define Hyperparameters:** batch_size, noise_dim (the size of the random noise vector fed to the generator), and epochs (the number of training cycles) are defined.
- **Define the Generator Model:** ***generator = tf.keras.Sequential([...])*** defines the generator network. Its purpose is to take a random noise vector (noise_dim) and transform it into an image that looks like the training data.
    - It starts with a Dense layer to project the noise into a higher dimension.
    - Reshape changes the output shape to a 3D tensor (like a small image with many channels).
    - BatchNormalization helps stabilize training.
    - LeakyReLU is an activation function commonly used in GANs.
    - Conv2DTranspose (also known as deconvolution or upsampling) layers are used to increase the spatial dimensions of the data, gradually building up an image from the smaller representation. The strides greater than 1 (strides=2) are key to upsampling.
    - The final Conv2DTranspose layer outputs a single-channel image (grayscale) with a tanh activation function, which outputs values in the range [-1, 1], matching the normalized input data.
- **Define the Discriminator Model:** ***discriminator = tf.keras.Sequential([...])*** defines the discriminator network. Its purpose is to take an image (either a real image from the dataset or a fake image from the generator) and output a single value indicating whether it thinks the image is real (closer to 1) or fake (closer to 0).
    - Conv2D layers with strides greater than 1 (strides=2) are used to downsample the image and extract features.
    - LeakyReLU and Dropout (to prevent overfitting) are used.
    - Flatten converts the 2D feature maps into a 1D vector.
    - The final Dense layer outputs a single value.
- **Define Loss Function and Optimizers:**
    - ***loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True):*** Binary crossentropy is used as the loss function. from_logits=True is important because the discriminator's final Dense layer does not have an activation function applied (it outputs logits).

- ***gen_optimizer*** and ***disc_optimizer:*** Separate Adam optimizers are defined for the generator and discriminator, as they are trained adversarially.

- **Training Loop**: The code enters a loop for the specified number of epochs.
  - Inside the epoch loop, it iterates through batches of real images from the dataset.
  - For each batch, random noise is generated.
  - **Training the Discriminator:**
    - A tf.GradientTape is used to record operations for automatic differentiation.
    - Fake images are generated by the generator.
    - The discriminator is called on both real and fake images.
    - Losses are calculated: real_loss (discriminator trying to classify real images as real) and fake_loss (discriminator trying to classify fake images as fake). The total disc_loss is the sum of these.
    - Gradients of the disc_loss with respect to the discriminator's trainable variables are computed and applied using the disc_optimizer.
  - **Training the Generator:**
    - Another tf.GradientTape is used.
    - Fake images are generated.
    - The discriminator is called on the fake images.
    - gen_loss is calculated: the generator tries to make the discriminator classify the fake images as real (tf.ones_like(fake_output)).
    - Gradients of the gen_loss with respect to the generator's trainable variables are computed and applied using the gen_optimizer.
- **Visualization:** After each epoch, the generator is used to generate a sample of images from a fixed seed noise vector (to see how the generation improves over epochs). These images are rescaled back to [0, 1] and displayed using matplotlib.pyplot.

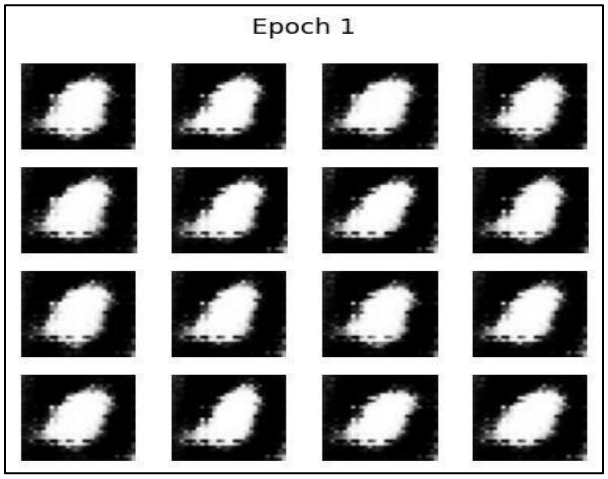
Code:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype("float32")
x_train = (x_train - 127.5) / 127.5
batch_size = 128
dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batch_size)
noise_dim = 100
generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7*7*256, input_shape=(noise_dim,)),
    tf.keras.layers.Reshape((7, 7, 256)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(128, 5, strides=1, padding='same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(64, 5, strides=2, padding='same'),
    tf.keras.layers.BatchNormalization(),
```

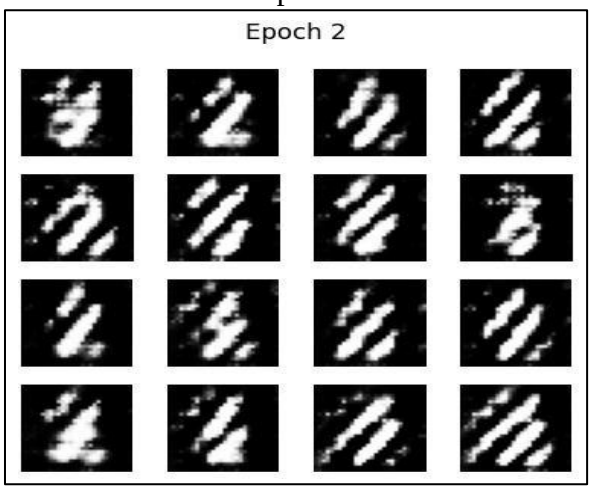
```
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(1, 5, strides=2, padding='same', activation='tanh')
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 5, strides=2, padding='same', input_shape=(28, 28, 1)),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Conv2D(128, 5, strides=2, padding='same'),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1)
])
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
gen_optimizer = tf.keras.optimizers.Adam(1e-4)
disc_optimizer = tf.keras.optimizers.Adam(1e-4)
epochs = 30
seed = tf.random.normal([16, noise_dim])
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    for real_images in dataset:
        noise = tf.random.normal([batch_size, noise_dim])
        # Generate fake images
        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            fake_images = generator(noise, training=True)
            real_output = discriminator(real_images, training=True)
            fake_output = discriminator(fake_images, training=True)
            gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)
            real_loss = loss_fn(tf.ones_like(real_output), real_output)
            fake_loss = loss_fn(tf.zeros_like(fake_output), fake_output)
            disc_loss = real_loss + fake_loss
        gradients_gen = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_disc = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
        gen_optimizer.apply_gradients(zip(gradients_gen, generator.trainable_variables))
        disc_optimizer.apply_gradients(zip(gradients_disc, discriminator.trainable_variables))
    generated_images = generator(seed, training=False)
    generated_images = (generated_images + 1) / 2.0
    fig = plt.figure(figsize=(4, 4))
    for i in range(16):
        plt.subplot(4, 4, i+1)
        plt.imshow(generated_images[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle(f'Epoch {epoch+1}')
    plt.tight_layout()
    plt.show()
```

Output:

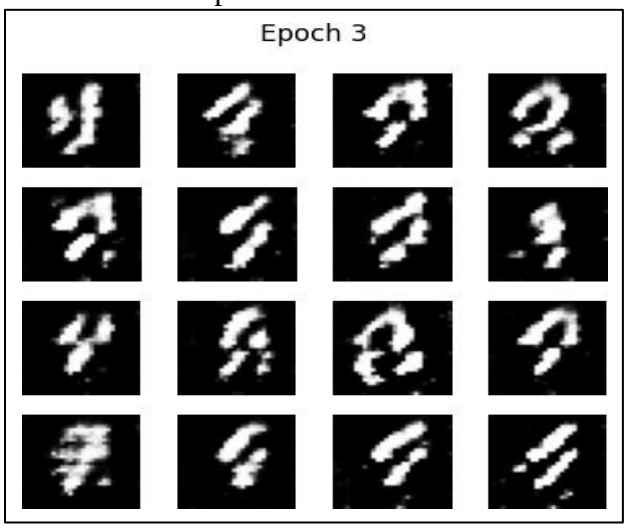
Epoch 1/30



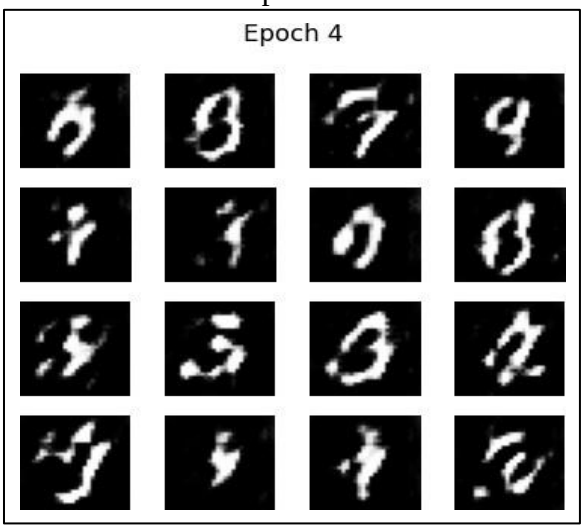
Epoch 2/30



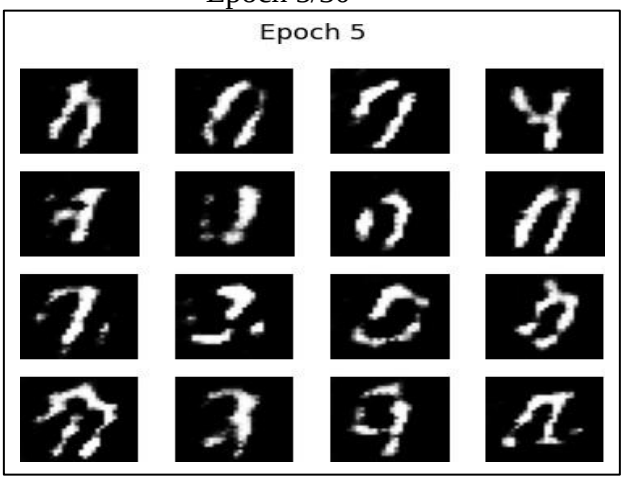
Epoch 3/30



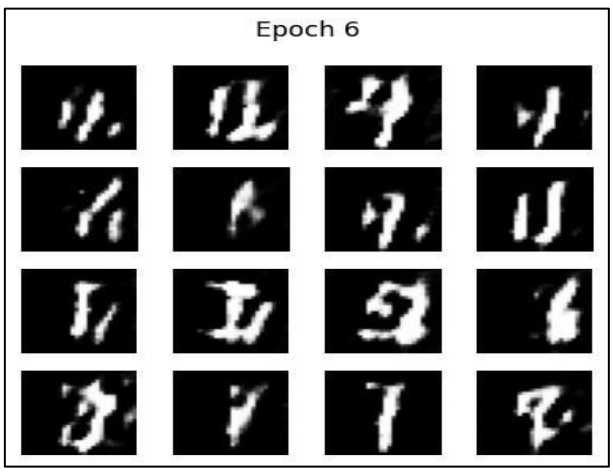
Epoch 4/30



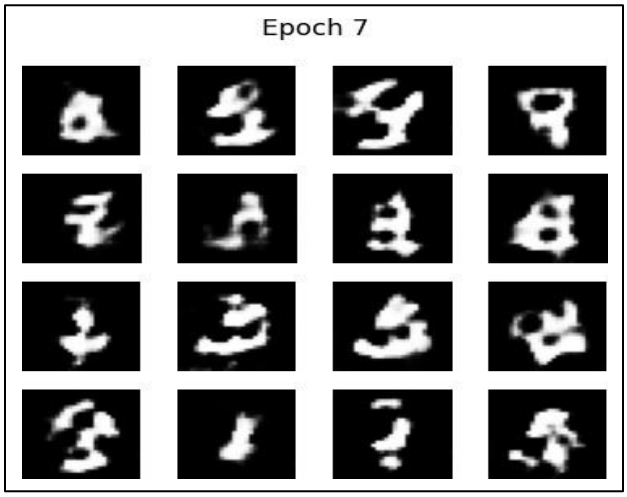
Epoch 5/30



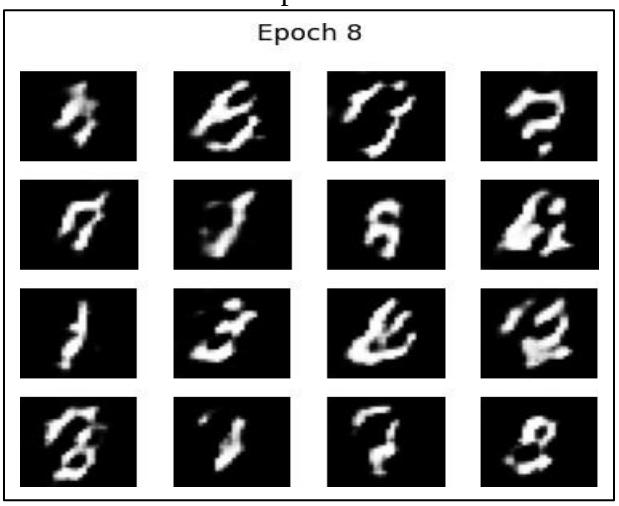
Epoch 6/30



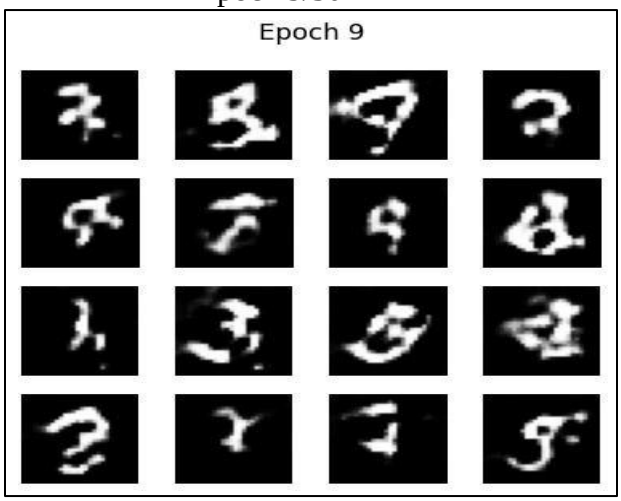
Epoch 7/30



Epoch 8/30



Epoch 9/30



Epoch 10/30