Deep Learning Practical

Practical 1a:

Aim: Intro to TensorFlow

- Create tensors with different shapes and data types.
- Perform basic operations like addition, subtraction, multiplication, and division on tensors.
- Reshape, slice, and index tensors to extract specific elements or sections.
- Performing matrix multiplication and finding eigenvectors and eigenvalues using TensorFlow

Solution:

```
import tensorflow as tf

# Creating tensors with different shapes and data types
t1 = tf.constant([1, 2, 3], dtype=tf.int32)   # 1D tensor (vector)
t2 = tf.constant([[1.5, 2.5], [3.5, 4.5]], dtype=tf.float32)   # 2D tensor (matrix)

# Performing basic tensor operations
add_result = tf.add(t1, 2)   # Adding a scalar
sub_result = tf.subtract(t1, 1)   # Subtracting a scalar
mul_result = tf.multiply(t1, 2)   # Element-wise multiplication
div_result = tf.divide(t1, 2)   # Element-wise division

# Reshaping, slicing, and indexing tensors
t3 = tf.reshape(t2, [4, 1])   # Reshape to a column vector
slice_result = t2[:, 1]   # Extracting second column
index_result = t1[0]   # Extracting first element

# Performing matrix multiplication
mat1 = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
mat2 = tf.constant([[5, 6], [7, 8]], dtype=tf.float32)
mat_mul_result = tf.matmul(mat1, mat2)   # Matrix multiplication

# Finding eigenvalues and eigenvectors
eigenvalues, eigenvectors = tf.linalg.eig(mat1)

# Printing results
print("Addition:", add_result.numpy())
print("Subtraction:", sub_result.numpy())
print("Multiplication:", mul_result.numpy())
print("Division:", div_result.numpy())
print("Reshaped tensor:", t3.numpy())
print("Sliced tensor:", slice_result.numpy())
print("Indexed element:", index_result.numpy())
print("Matrix multiplication result:\n", mat_mul_result.numpy())
print("Eigenvalues:\n", eigenvalues.numpy())
print("Eigenvectors:\n", eigenvectors.numpy())
```

Output:

```
Addition: [3 4 5]
Subtraction: [0 1 2]
Multiplication: [2 4 6]
Division: [0.5 1.  1.5]
Reshaped tensor: [[1.5]
 [2.5]
 [3.5]
 [4.5]
 [3.5]
 [4.5]]
Sliced tensor: [2.5 4.5 4.5]
Indexed element: 1
Matrix multiplication result:
 [[19. 22.]
 [43. 50.]]
Eigenvalues:
 [-0.37228122+0.j  5.372281  +0.j]
Eigenvectors:
 [[-0.8245648 +0.j -0.41597357+0.j]
 [ 0.56576747+0.j -0.90937674+0.j]]
```

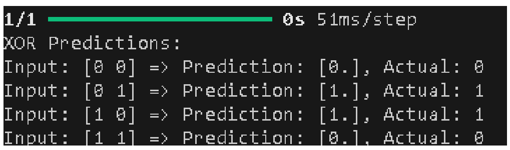
Practical 1b:

Aim: Program to solve the XOR problem.

Solution:

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# XOR dataset (input and expected output)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])   # Input
y = np.array([[0], [1], [1], [0]])   # Output

# Build a neural network model
model = Sequential()
model.add(Dense(4, input_dim=2, activation='relu'))   # Hidden layer with 4 neurons
model.add(Dense(1, activation='sigmoid'))   # Output layer (sigmoid for binary classification)

# Compile the model
model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X, y, epochs=1000, verbose=0)

# Evaluate the model
predictions = model.predict(X)

# Print the XOR predictions
print("XOR Predictions:")
for i in range(len(X)):
	print(f"Input: {X[i]} => Prediction: {np.round(predictions[i])}, Actual: {y[i][0]}")
```

Output:



Practical 2:

Aim: Implement a simple linear regression model using TensorFlow's low level API (or tf. keras).

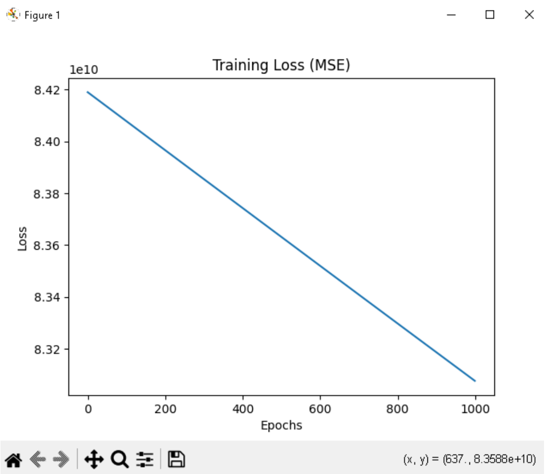
- Train the model on a toy dataset (e.g., housing prices vs. square footage).
- Visualize the loss function and the learned linear relationship.
- Make predictions on new data points.

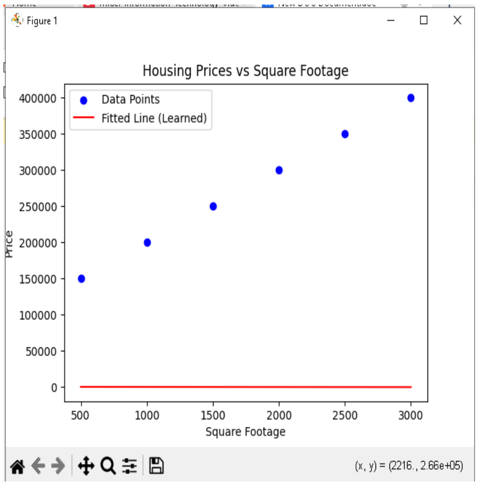
Solution:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# 1. Create a toy dataset: Square footage vs Housing prices
np.random.seed(42)  # For reproducibility
X = np.array([500, 1000, 1500, 2000, 2500, 3000], dtype=np.float32)  #
Square footage
y = np.array([150000, 200000, 250000, 300000, 350000, 400000],
dtype=np.float32)  # Prices in USD

# Reshape X for the model input
X = X.reshape(-1, 1)
y = y.reshape(-1, 1)

# 2. Define the model: A simple linear regression model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_dim=1)  # Linear layer with 1 input and 1
output (slope and intercept)
])

# 3. Compile the model with Mean Squared Error (MSE) loss and Adam
optimizer
model.compile(optimizer='adam', loss='mse')

# 4. Train the model
history = model.fit(X, y, epochs=1000, verbose=0)

# 5. Plot the loss function over training epochs
plt.plot(history.history['loss'])
plt.title('Training Loss (MSE)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()

# 6. Visualize the learned linear relationship
plt.scatter(X, y, color='blue', label='Data Points')
predicted_y = model.predict(X)
plt.plot(X, predicted_y, color='red', label='Fitted Line (Learned)')
plt.title('Housing Prices vs Square Footage')
plt.xlabel('Square Footage')
```

```
plt.ylabel('Price')
plt.legend()
plt.show()

# 7. Make predictions on new data points
new_data = np.array([1200, 1800, 2200], dtype=np.float32).reshape(-1, 1)    #
New square footage values
predictions = model.predict(new_data)

print(f"Predictions for new data points (Square Footage): {new_data.flatten()}")
print(f"Predicted Prices: {predictions.flatten()}")
```

Ouput:






```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
Predictions for new data points (Square Footage): [1200. 1800. 2200.]
Predicted Prices: [-172.85994 -259.7892  -317.7421 ]
```

Practical 3A:

Aim:Implementing deep neural network for performing binary classification task.

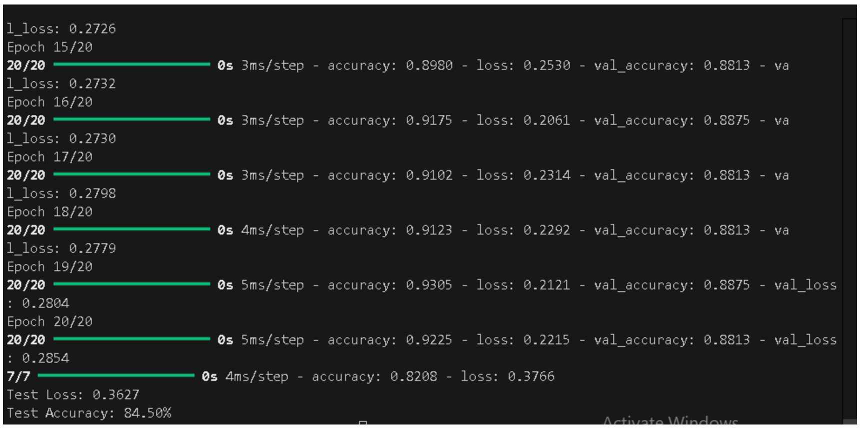
Solution :-

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification

# 1. Prepare the Data (You can replace this with your dataset)
# Let's generate a synthetic dataset for binary classification
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Feature scaling (important for neural networks)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 2. Build the Model
model = Sequential()

# Input layer (input shape matches the number of features in the data)
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))

# Hidden layer
model.add(Dense(32, activation='relu'))

# Output layer (for binary classification, use sigmoid activation)
model.add(Dense(1, activation='sigmoid'))

# 3. Compile the Model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',   # Binary classification loss
function
metrics=['accuracy'])
```

# 4. Train the Model
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2)

5. Evaluate the Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy * 100:.2f}%")

Output:



```
l_loss: 0.2726
Epoch 15/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8980 - loss: 0.2530 - val_accuracy: 0.8813 - va
l_loss: 0.2732
Epoch 16/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9175 - loss: 0.2061 - val_accuracy: 0.8875 - va
l_loss: 0.2730
Epoch 17/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9102 - loss: 0.2314 - val_accuracy: 0.8813 - va
l_loss: 0.2798
Epoch 18/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9123 - loss: 0.2292 - val_accuracy: 0.8813 - va
l_loss: 0.2779
Epoch 19/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9305 - loss: 0.2121 - val_accuracy: 0.8875 - val_loss
: 0.2804
Epoch 20/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9225 - loss: 0.2215 - val_accuracy: 0.8813 - val_loss
: 0.2854
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8208 - loss: 0.3766
Test Loss: 0.3627
Test Accuracy: 84.50%
```

Practical 3B:-

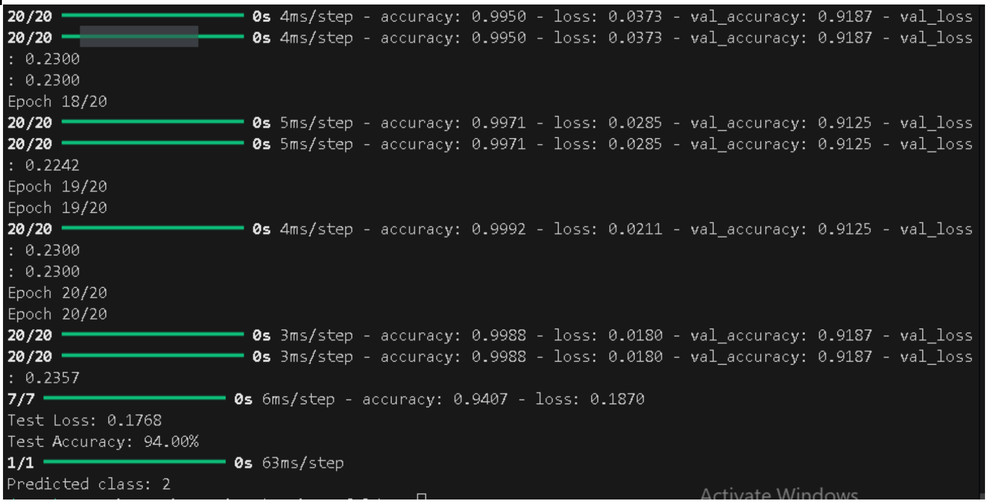
Aim: Using a deep feed-forward network with two hidden layers for performing multi-class classification and predicting the class.

Solution:-

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification
from tensorflow.keras.utils import to_categorical

# 1. Prepare the Data (Replace this with your dataset)
# Let's generate a synthetic dataset for multiclass classification
# Generate a synthetic multiclass classification dataset
X, y = make_classification(
    n_samples=1000,          # Number of samples
    n_features=20,           # Number of features
    n_classes=3,             # Number of classes
    n_clusters_per_class=1,  # Reducing clusters per class
    n_informative=5,         # Increase number of informative features
    random_state=42
)
# One-hot encode the labels for multi-class classification
y = to_categorical(y, num_classes=3)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Feature scaling (important for neural networks)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 2. Build the Model
model = Sequential()

# Input layer (input shape matches the number of features in the data)
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))

# Hidden layer 1
model.add(Dense(32, activation='relu'))

# Hidden layer 2
model.add(Dense(16, activation='relu'))
```

```
# Output layer (for multiclass classification, use softmax activation)
model.add(Dense(3, activation='softmax'))   # 3 classes in the output

# 3. Compile the Model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',   # Categorical cross-entropy loss
              metrics=['accuracy'])

# 4. Train the Model
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2)

# 5. Evaluate the Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy * 100:.2f}%")

# 6. Predicting the class for new data
# Example: Predicting the class for a new data point
new_data = np.random.randn(1, X_train.shape[1])   # Random example
new_data_scaled = scaler.transform(new_data)   # Scale the new data point
prediction = model.predict(new_data_scaled)
predicted_class = np.argmax(prediction)   # Convert prediction probabilities to the class with highest probability
print(f"Predicted class: {predicted_class}")
```

Output:-



```
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9950 - loss: 0.0373 - val_accuracy: 0.9187 - val_loss
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9950 - loss: 0.0373 - val_accuracy: 0.9187 - val_loss
: 0.2300
: 0.2300
Epoch 18/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9971 - loss: 0.0285 - val_accuracy: 0.9125 - val_loss
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9971 - loss: 0.0285 - val_accuracy: 0.9125 - val_loss
: 0.2242
Epoch 19/20
Epoch 19/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9992 - loss: 0.0211 - val_accuracy: 0.9125 - val_loss
: 0.2300
: 0.2300
Epoch 20/20
Epoch 20/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9988 - loss: 0.0180 - val_accuracy: 0.9187 - val_loss
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9988 - loss: 0.0180 - val_accuracy: 0.9187 - val_loss
: 0.2357
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9407 - loss: 0.1870
Test Loss: 0.1768
Test Accuracy: 94.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step
Predicted class: 2
```

Practical 4:-

Aim:- Write a program to implement deep learning Techniques for image segmentation

Solution :-

```
import os
import random
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from PIL import Image
import numpy as np
import albumentations as A   # Added Albumentations import

# 1. Define the U-Net Model
class UNet(nn.Module):
	def __init__(self, in_channels=3, out_channels=1, init_features=32):
		super(UNet, self).__init__()
		features = init_features
		self.encoder1 = UNet._block(in_channels, features, name="enc1")
		self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
		self.encoder2 = UNet._block(features, features * 2, name="enc2")
		self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
		self.encoder3 = UNet._block(features * 2, features * 4, name="enc3")
		self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
		self.encoder4 = UNet._block(features * 4, features * 8, name="enc4")
		self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)
		self.bottleneck = UNet._block(features * 8, features * 16,
name="bottleneck")
		self.upconv4 = nn.ConvTranspose2d(features * 16, features * 8,
kernel_size=2, stride=2)
		self.decoder4 = UNet._block((features * 8) * 2, features * 8,
name="dec4")
		self.upconv3 = nn.ConvTranspose2d(features * 8, features * 4,
kernel_size=2, stride=2)
		self.decoder3 = UNet._block((features * 4) * 2, features * 4,
name="dec3")
		self.upconv2 = nn.ConvTranspose2d(features * 4, features * 2,
kernel_size=2, stride=2)
		self.decoder2 = UNet._block((features * 2) * 2, features * 2,
name="dec2")
		self.upconv1 = nn.ConvTranspose2d(features * 2, features,
kernel_size=2, stride=2)
		self.decoder1 = UNet._block(features * 2, features, name="dec1")
		self.conv = nn.Conv2d(in_channels=features,
out_channels=out_channels, kernel_size=1)
```

```
    @staticmethod
    def _block(in_channels, features, name):
        return nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=features,
kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(num_features=features),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=features, out_channels=features,
kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(num_features=features),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        enc1 = self.encoder1(x)
        enc2 = self.encoder2(self.pool1(enc1))
        enc3 = self.encoder3(self.pool2(enc2))
        enc4 = self.encoder4(self.pool3(enc3))
        bottleneck = self.bottleneck(self.pool4(enc4))
        dec4 = self.upconv4(bottleneck)
        dec4 = torch.cat((dec4, enc4), dim=1)
        dec4 = self.decoder4(dec4)
        dec3 = self.upconv3(dec4)
        dec3 = torch.cat((dec3, enc3), dim=1)
        dec3 = self.decoder3(dec3)
        dec2 = self.upconv2(dec3)
        dec2 = torch.cat((dec2, enc2), dim=1)
        dec2 = self.decoder2(dec2)
        dec1 = self.upconv1(dec2)
        dec1 = torch.cat((dec1, enc1), dim=1)
        dec1 = self.decoder1(dec1)
        return torch.sigmoid(self.conv(dec1))

# 2. Custom Dataset for Loading Images and Masks
class SegmentationDataset(Dataset):
    def __init__(self, image_paths, mask_paths, transform=None):
        self.image_paths = image_paths
        self.mask_paths = mask_paths
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        mask_path = self.mask_paths[idx]
        image = np.array(Image.open(img_path).convert("RGB"))
        mask = np.array(Image.open(mask_path).convert("L"),
dtype=np.float32)
```

```
        mask[mask == 255.0] = 1.0  # Normalize mask to [0, 1]

        if self.transform:
            # Apply Albumentations transforms to both image and mask
            augmented = self.transform(image=image, mask=mask)
            image = augmented['image']
            mask = augmented['mask']

        return image, mask

# 3. Training Function
def train_fn(loader, model, optimizer, loss_fn, device):
    model.train()
    running_loss = 0.0
    for images, masks in loader:
        images = images.to(device)
        masks = masks.unsqueeze(1).to(device)  # Add channel dimension
        outputs = model(images)
        loss = loss_fn(outputs, masks)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    return running_loss / len(loader)

# 4. Evaluation Function
def evaluate_fn(loader, model, loss_fn, device):
    model.eval()
    total_loss = 0.0
    with torch.no_grad():
        for images, masks in loader:
            images = images.to(device)
            masks = masks.unsqueeze(1).to(device)
            outputs = model(images)
            loss = loss_fn(outputs, masks)
            total_loss += loss.item()
    return total_loss / len(loader)

# 5. Save Model Function
def save_model(model, path):
    torch.save(model.state_dict(), path)
    print(f"Model saved to {path}")

# 6. Load Model Function
def load_model(model, path, device):
    model.load_state_dict(torch.load(path, map_location=device))
    model.to(device)
    model.eval()
    print(f"Model loaded from {path}")
```

```
# 7. Prediction Function
def predict_image(model, image_path, transform, device, threshold=0.5):
    model.eval()
    image = np.array(Image.open(image_path).convert("RGB"))
    original_shape = image.shape[:2]  # Save original shape for resizing
later

    if transform:
        # Apply Albumentations transforms (only image needed here)
        augmented = transform(image=image)
        image = augmented['image']

    # Add batch dimension and move to device
    image = image.unsqueeze(0).to(device)

    with torch.no_grad():
        output = model(image)

    # Convert output to binary mask
    output = (output.squeeze().cpu().numpy() > threshold).astype(np.uint8)
    # Resize mask back to original image size
    output = np.array(Image.fromarray(output).resize(original_shape[::-1],
Image.NEAREST))
    return output

# 8. Main Script
if __name__ == "__main__":
    # Hyperparameters
    LEARNING_RATE = 1e-4
    BATCH_SIZE = 8
    NUM_EPOCHS = 1
    IMAGE_HEIGHT = 128
    IMAGE_WIDTH = 128
    DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
    PIN_MEMORY = True
    TRAIN_VAL_SPLIT = 0.8  # 80% for training, 20% for validation
    MODEL_SAVE_PATH = "unet_model.pth"

    # Directories
    IMAGES_DIR = "/content/data/sub_images"
    MASKS_DIR = "/content/data/sub_masks"

    # Albumentations Transform (Corrected)
    transform = A.Compose([
        A.Resize(IMAGE_HEIGHT, IMAGE_WIDTH),
        A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
        A.ToTensorV2(),
    ], additional_targets={'mask': 'mask'})

    # Load all image and mask paths
```

```
    image_files = sorted([f for f in os.listdir(IMAGES_DIR) if f.endswith(('.jpg',
'.png'))])
    mask_files = sorted([f for f in os.listdir(MASKS_DIR) if f.endswith(('.jpg',
'.png'))])

    # Ensure filenames match
    image_names = [os.path.splitext(f)[0] for f in image_files]
    mask_names = [os.path.splitext(f)[0] for f in mask_files]
    assert set(image_names) == set(mask_names), "Mismatch between
image and mask filenames"

    # Combine full paths
    image_paths = [os.path.join(IMAGES_DIR, f) for f in image_files]
    mask_paths = [os.path.join(MASKS_DIR, f) for f in mask_files]

    # Shuffle and split into train/validation
    combined = list(zip(image_paths, mask_paths))
    random.shuffle(combined)
    split_index = int(len(combined) * TRAIN_VAL_SPLIT)
    train_data, val_data = combined[:split_index], combined[split_index:]
    train_image_paths, train_mask_paths = zip(*train_data)
    val_image_paths, val_mask_paths = zip(*val_data)

    # Create datasets and dataloaders
    train_dataset = SegmentationDataset(train_image_paths,
train_mask_paths, transform=transform)
    val_dataset = SegmentationDataset(val_image_paths, val_mask_paths,
transform=transform)
    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
shuffle=True, pin_memory=PIN_MEMORY)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE,
shuffle=False, pin_memory=PIN_MEMORY)

    # Initialize Model, Loss, Optimizer
    model = UNet(in_channels=3, out_channels=1).to(DEVICE)
    loss_fn = nn.BCEWithLogitsLoss()  # Binary Cross Entropy Loss
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

    # Training Loop
    for epoch in range(NUM_EPOCHS):
        train_loss = train_fn(train_loader, model, optimizer, loss_fn, DEVICE)
        val_loss = evaluate_fn(val_loader, model, loss_fn, DEVICE)
        print(f"Epoch [{epoch+1}/{NUM_EPOCHS}] | Train Loss:
{train_loss:.4f} | Val Loss: {val_loss:.4f}")

   # Save the trained model
save_model(model, MODEL_SAVE_PATH)

    # Load the model for prediction
loaded_model = UNet(in_channels=3, out_channels=1).to(DEVICE)
```

```
load_model(loaded_model, MODEL_SAVE_PATH, DEVICE)

    # Perform prediction on a new image
test_image_path = "my_car.webp"    # Replace with your test image path
predicted_mask = predict_image(loaded_model, test_image_path, transform, DEVICE)

    # Save the predicted mask as an image
predicted_mask_image = Image.fromarray((predicted_mask * 255).astype(np.uint8))
predicted_mask_image.save("predicted_mask.png")
print("Prediction complete! Predicted mask saved as 'predicted_mask.png'")
```

output:-

Input Image

Output Image

Practical 5:-
Aim:- Write a program to predict a caption for a sample image using LSTM.

Solution:-

```
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.models import load_model, Model    # <-- Import Model
here
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np
import pickle
from PIL import Image

# Load the preprocessed mapping and tokenizer
with open("mapping.pkl", "rb") as f:
    mapping = pickle.load(f)

def all_captions(mapping):
    return [caption for key in mapping for caption in mapping[key]]

all_captions = all_captions(mapping)

def create_token(all_captions):
    from tensorflow.keras.preprocessing.text import Tokenizer
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(all_captions)
    return tokenizer

tokenizer = create_token(all_captions)
max_length = 35

def idx_to_word(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None

def predict_caption(model, image, tokenizer, max_length):
    in_text = 'startseq'
    repeated_word_count = 0
    previous_word = None
    for i in range(max_length):
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        sequence = pad_sequences([sequence], maxlen=max_length)
        yhat = model.predict([image, sequence], verbose=0)
        yhat = np.argmax(yhat)
        word = idx_to_word(yhat, tokenizer)
```

```
            if word is None or word == 'endseq' or (word == previous_word and
repeated_word_count > 2):
                break

            in_text += " " + word

            if word == previous_word:
                repeated_word_count += 1
            else:
                repeated_word_count = 0
            previous_word = word

        return in_text.strip('startseq ').strip()

# Load the VGG16 model and the captioning model
vgg_model = VGG16()
vgg_model = Model(inputs=vgg_model.inputs, outputs=vgg_model.layers[-
2].output)
model = load_model("model.keras")

def generate_caption(image_path):
    try:
        # Load and preprocess the image
        image = Image.open(image_path)
        image = image.resize((224, 224))
        image = img_to_array(image)
        image = image.reshape((1, image.shape[0], image.shape[1],
image.shape[2]))
        image = preprocess_input(image)
        feature = vgg_model.predict(image, verbose=0)

        # Generate caption
        caption = predict_caption(model, feature, tokenizer, max_length)

        return caption
    except Exception as e:
        print(f"Error: {str(e)}")
        return None

# Example usage
if __name__ == "__main__":
    image_path = "imgdog.jpg"   # Replace with your image path
    caption = generate_caption(image_path)
    if caption:
        print(f"Generated Caption: {caption}")
    else:
        print("Failed to generate caption.")
```

output:-

```
 `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
I0000 00:00:1745143758.633080    7324 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at most once for the lifetime
 of the process.
Generated Caption: he on side group group cliff cliff and through playing on snow in in in in
PS C:\Users\RPIMS\Desktop\dl_p\5>
```

Practical 6:-

Aim : Applying the Autoencoder algorithms for encoding real-world data

Solution :

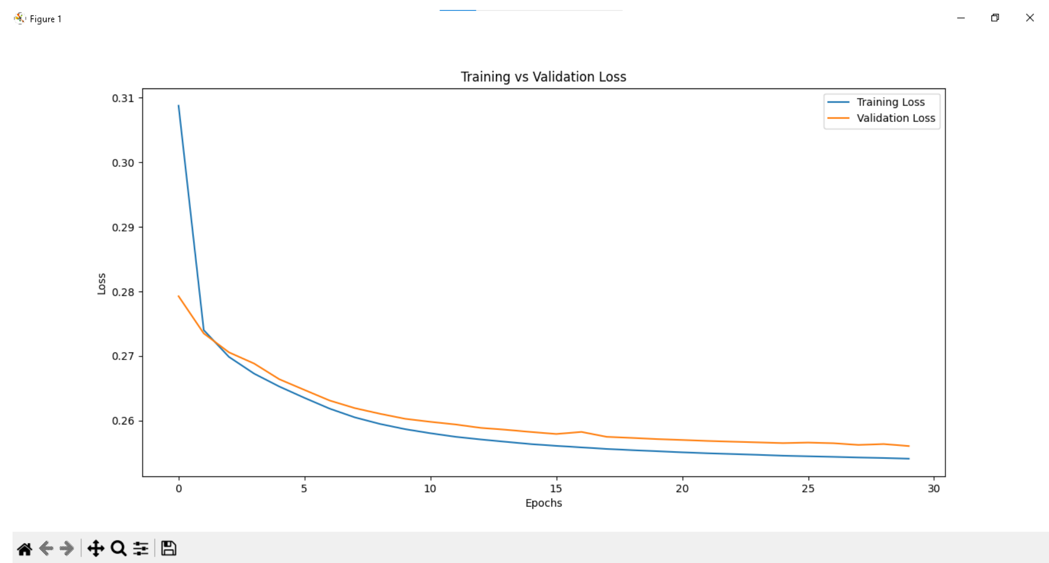
```
# Step 1: Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam

# Step 2: Load Real-World Data
data = load_wine()
X = data.data    # Features
feature_names = data.feature_names

print(f"Dataset shape: {X.shape}")

# Step 3: Preprocessing
# Scale the data between 0 and 1
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Train-test split
X_train, X_test = train_test_split(X_scaled, test_size=0.2, random_state=42)

print(f"Training set shape: {X_train.shape}")
print(f"Testing set shape: {X_test.shape}")

# Step 4: Build Autoencoder Model
input_dim = X_train.shape[1]    # Number of features
encoding_dim = 8    # Bottleneck size

# Input layer
input_layer = Input(shape=(input_dim,))

# Encoder
encoded = Dense(16, activation='relu')(input_layer)
encoded = Dense(encoding_dim, activation='relu')(encoded)

# Decoder
decoded = Dense(16, activation='relu')(encoded)
decoded = Dense(input_dim, activation='sigmoid')(decoded)
```

```
# Full Autoencoder
autoencoder = Model(input_layer, decoded)

# Encoder model
encoder = Model(input_layer, encoded)

# Step 5: Compile the Model
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

# Step 6: Train the Model
history = autoencoder.fit(
    X_train, X_train,
    epochs=100,
    batch_size=16,
    shuffle=True,
    validation_data=(X_test, X_test)
)

# Step 7: Encode Real-World Data
X_encoded = encoder.predict(X_test)

print(f"Encoded data shape: {X_encoded.shape}")
print("\nSample Encoded Data:\n", X_encoded[:5])

# Step 8: Plot Loss Curves
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss During Training')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
# Step 9: Save Encoded Data (Optional)
encoded_df = pd.DataFrame(X_encoded, columns=[f'encoded_{i+1}' for i in
range(encoding_dim)])
encoded_df.to_csv('encoded_wine_data.csv', index=False)
print("\nEncoded data saved to 'encoded_wine_data.csv'")
```

Output :

```
Epoch 1/30
2025-04-26 18:22:43.851863: E tensorflow/core/util/util.cc:131] oneDNN supports DT_INT32 only on platforms with AVX-512. Fa
lling back to the default Eigen-based implementation if present.
469/469 ━━━━━━━━━━━━━━━━━━━━ 13s 26ms/step - loss: 0.3722 - val_loss: 0.2793
Epoch 2/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - loss: 0.2752 - val_loss: 0.2735
Epoch 3/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.2708 - val_loss: 0.2706
Epoch 4/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.2680 - val_loss: 0.2688
Epoch 5/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.2654 - val_loss: 0.2664
Epoch 6/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.2639 - val_loss: 0.2648
Epoch 7/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.2620 - val_loss: 0.2631
Epoch 8/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 24ms/step - loss: 0.2609 - val_loss: 0.2619
Epoch 9/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 24ms/step - loss: 0.2593 - val_loss: 0.2611
Epoch 10/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 24ms/step - loss: 0.2589 - val_loss: 0.2603
Epoch 11/30
```

```
Epoch 25/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 13s 28ms/step - loss: 0.2546 - val_loss: 0.2565
Epoch 26/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 13s 27ms/step - loss: 0.2540 - val_loss: 0.2566
Epoch 27/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - loss: 0.2546 - val_loss: 0.2565
Epoch 28/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - loss: 0.2540 - val_loss: 0.2563
Epoch 29/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.2534 - val_loss: 0.2564
Epoch 30/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.2542 - val_loss: 0.2561
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Encoded images shape: (10000, 7, 7, 16)
```



Practical 7:

Aim : Write a program for character recognition using RNN and compare it with CNN

Solution :

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist

# Load and preprocess MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Reshape for RNN (samples, timesteps, features)
x_train_rnn, x_test_rnn = x_train.reshape(-1, 28, 28), x_test.reshape(-1, 28,
28)
# Reshape for CNN (samples, height, width, channels)
x_train_cnn, x_test_cnn = x_train.reshape(-1, 28, 28, 1), x_test.reshape(-1, 28,
28, 1)

# RNN Model
def create_rnn():
    model = models.Sequential([
        layers.SimpleRNN(128, activation='relu', input_shape=(28, 28)),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
    return model

# CNN Model
def create_cnn():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
    return model

# Train and evaluate RNN
rnn_model = create_rnn()
rnn_model.fit(x_train_rnn, y_train, epochs=3, batch_size=64,
validation_data=(x_test_rnn, y_test))
rnn_loss, rnn_acc = rnn_model.evaluate(x_test_rnn, y_test)
```

```
# Train and evaluate CNN
cnn_model = create_cnn()
cnn_model.fit(x_train_cnn, y_train, epochs=3, batch_size=64,
validation_data=(x_test_cnn, y_test))
cnn_loss, cnn_acc = cnn_model.evaluate(x_test_cnn, y_test)

# Compare results
print(f'RNN Accuracy: {rnn_acc:.4f}, CNN Accuracy: {cnn_acc:.4f}')
```

Output :

```
C:\Users\RPIMS\Desktop\dl_p\.venv\Lib\site-packages\keras\src\layers\rnn\rnn.py:200: UserWarning: Do not pass an `input_sha
pe`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer
 in the model instead.
  super().__init__(**kwargs)
Epoch 1/3
938/938 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7219 - loss: 0.8263 - val_accuracy: 0.9389 - val_loss: 0.2003
Epoch 2/3
938/938 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9365 - loss: 0.2115 - val_accuracy: 0.9543 - val_loss: 0.1553
Epoch 3/3
938/938 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.9528 - loss: 0.1582 - val_accuracy: 0.9581 - val_loss: 0.1476
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9537 - loss: 0.1674
C:\Users\RPIMS\Desktop\dl_p\.venv\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pa
ss an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as
 the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/3
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 9ms/step - accuracy: 0.8980 - loss: 0.3571 - val_accuracy: 0.9821 - val_loss: 0.0591
Epoch 2/3
938/938 ━━━━━━━━━━━━━━━━━━━━ 9s 9ms/step - accuracy: 0.9818 - loss: 0.0578 - val_accuracy: 0.9842 - val_loss: 0.0456
Epoch 3/3
938/938 ━━━━━━━━━━━━━━━━━━━━ 9s 9ms/step - accuracy: 0.9895 - loss: 0.0369 - val_accuracy: 0.9839 - val_loss: 0.0499
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9809 - loss: 0.0581
RNN Accuracy: 0.9581, CNN Accuracy: 0.9839
PS C:\Users\RPIMS\Desktop\dl_p>
```

Practical 8 :

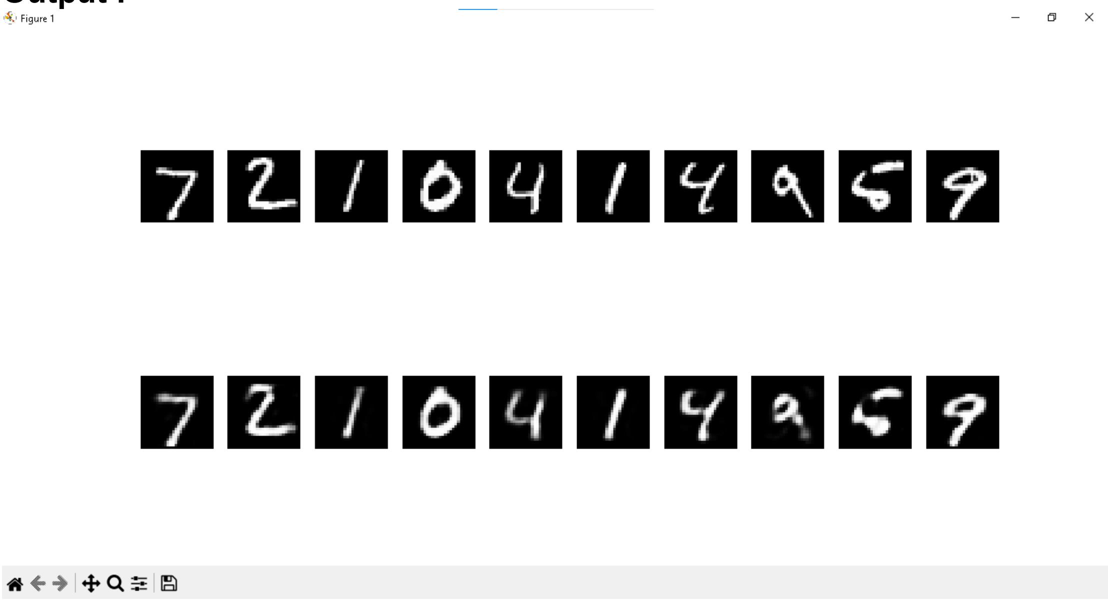
Aim : Write a program to develop Autoencoders using MNIST Handwritten Digits

Solution :

```
# Step 1: Import Libraries
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten, Reshape
from tensorflow.keras.optimizers import Adam

# Step 2: Load and Preprocess Data
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize pixel values between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the images (28x28 -> 784)
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

print(f"x_train shape: {x_train.shape}")
print(f"x_test shape: {x_test.shape}")

# Step 3: Build Autoencoder
input_dim = x_train.shape[1]    # 784
encoding_dim = 32    # Size of the bottleneck

# Input layer
input_img = Input(shape=(input_dim,))

# Encoder
encoded = Dense(encoding_dim, activation='relu')(input_img)

# Decoder
decoded = Dense(input_dim, activation='sigmoid')(encoded)

# Autoencoder Model
autoencoder = Model(input_img, decoded)

# Encoder Model (for getting encoded data separately)
encoder = Model(input_img, encoded)

# Step 4: Compile the Autoencoder
autoencoder.compile(optimizer=Adam(learning_rate=0.001),
loss='binary_crossentropy')
```

```
# Step 5: Train the Autoencoder
history = autoencoder.fit(
    x_train, x_train,
    epochs=50,
    batch_size=256,
    shuffle=True,
    validation_data=(x_test, x_test)
)

# Step 6: Encode and Decode Some Digits
encoded_imgs = encoder.predict(x_test)
decoded_imgs = autoencoder.predict(x_test)

print(f"Encoded images shape: {encoded_imgs.shape}")

# Step 7: Visualize the Original and Reconstructed Images
n = 10   # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original digits
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.axis('off')

    # Display reconstructed digits
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.axis('off')

plt.show()

# Step 8: Plot Training History (Loss curve)
plt.figure(figsize=(8, 4))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Autoencoder Training vs Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Output :



Practical 9 :

Aim : Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.(google stock price)

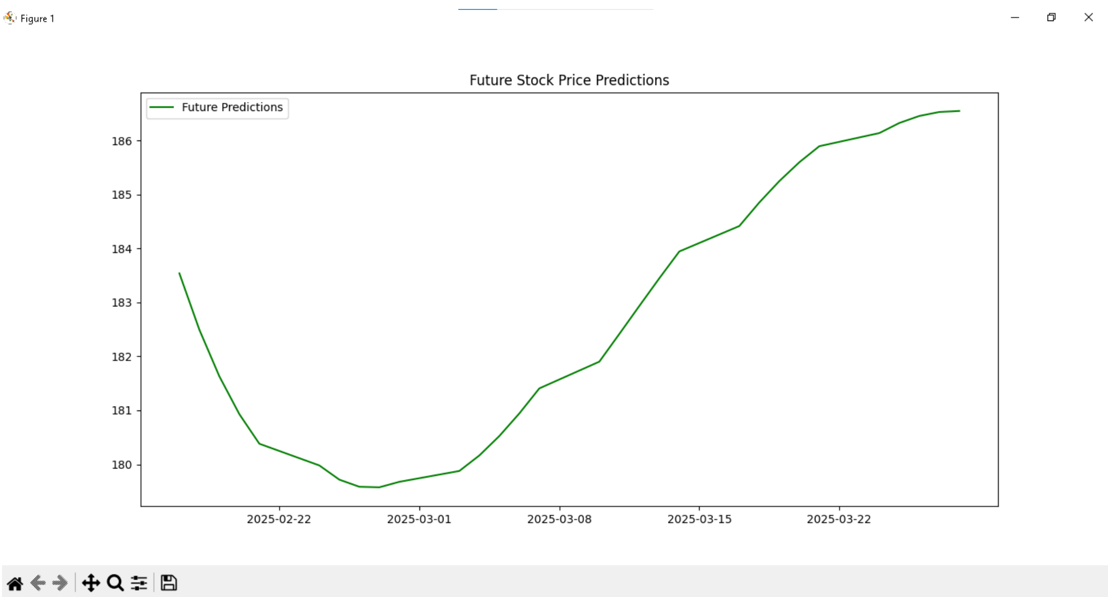
Solution :

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
import yfinance as yf

# Fetch historical stock price data using yfinance
ticker = 'GOOG'
df = yf.download(ticker, start='2010-01-01', end='2025-02-16')
data = df[['Close']].values   # Use closing prices

# Normalize data
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data)

# Prepare training data
X_train, y_train = [], []
time_steps = 60   # Use last 60 days to predict next day
for i in range(time_steps, len(data_scaled)):
	X_train.append(data_scaled[i-time_steps:i, 0])
	y_train.append(data_scaled[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))   #
Reshape for LSTM

# Build RNN model
model = Sequential([
	LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],
1)),
	LSTM(units=50),
	Dense(units=1)
])

# Compile and train model
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=20, batch_size=32)

# Predict stock prices
predicted_stock_price = model.predict(X_train)
predicted_stock_price = scaler.inverse_transform(predicted_stock_price)
```

```
print("Stock Price Prediction Completed!", predicted_stock_price)

#-----------------------------------------------------------------------------#
#   Training and testing data splitted and visulization of future predictions

import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
import yfinance as yf

# Fetch historical stock price data using yfinance
ticker = 'GOOG'
df = yf.download(ticker, start='2010-01-01', end='2025-02-16')
data = df[['Close']].values   # Use closing prices

# Normalize data
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data)

# Split data into training and test sets
train_size = int(len(data_scaled) * 0.8)
train_data, test_data = data_scaled[:train_size], data_scaled[train_size:]

# Prepare training data
X_train, y_train = [], []
time_steps = 60   # Use last 60 days to predict next day
for i in range(time_steps, len(train_data)):
	X_train.append(train_data[i-time_steps:i, 0])
	y_train.append(train_data[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))   #
Reshape for LSTM

# Prepare test data
X_test, y_test = [], []
for i in range(time_steps, len(test_data)):
	X_test.append(test_data[i-time_steps:i, 0])
	y_test.append(test_data[i, 0])
X_test, y_test = np.array(X_test), np.array(y_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

# Build RNN model
model = Sequential([
	LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],
1)),
	LSTM(units=50),
```

```
        Dense(units=1)
])

# Compile and train model
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=20, batch_size=32)

# Predict future stock prices
future_steps = 30
future_input = test_data[-time_steps:].reshape(1, time_steps, 1)
future_predictions = []
for _ in range(future_steps):
        pred = model.predict(future_input)
        future_predictions.append(pred[0, 0])
        future_input = np.append(future_input[:, 1:, :], pred.reshape(1, 1, 1),
axis=1)
future_predictions =
scaler.inverse_transform(np.array(future_predictions).reshape(-1, 1))

# Visualization of future predictions
plt.figure(figsize=(14, 5))
future_dates = pd.date_range(df.index[-1], periods=future_steps + 1,
freq='B')[1:]
plt.plot(future_dates, future_predictions, label='Future Predictions',
color='green')
plt.legend()
plt.title("Future Stock Price Predictions")
plt.show()

print("Future Stock Price Prediction Completed!")
```

Output :



Practical 10

Aim: Applying Generative Adversarial Networks for image generation and unsupervised tasks.

Solution:

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

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Hyperparameters
latent_dim = 100
batch_size = 64
lr = 0.0002
epochs = 50

# DataLoader for MNIST
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])  # Normalize between [-1, 1]
])
train_data = datasets.MNIST(root='.', train=True, transform=transform,
download=True)
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)

# Generator
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 784),
            nn.Tanh()
        )

    def forward(self, z):
        return self.model(z).view(-1, 1, 28, 28)

# Discriminator
```

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        return self.model(img)

# Initialize models
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Loss and optimizers
criterion = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=lr)
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr)

# Training loop
for epoch in range(epochs):
    for batch_idx, (real_imgs, _) in enumerate(train_loader):
        real_imgs = real_imgs.to(device)
        batch_size = real_imgs.size(0)

        # Labels
        real_labels = torch.ones(batch_size, 1).to(device)
        fake_labels = torch.zeros(batch_size, 1).to(device)

        # Train Discriminator
        z = torch.randn(batch_size, latent_dim).to(device)
        fake_imgs = generator(z)
        real_loss = criterion(discriminator(real_imgs), real_labels)
        fake_loss = criterion(discriminator(fake_imgs.detach()), fake_labels)
        d_loss = real_loss + fake_loss
        optimizer_D.zero_grad()
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        z = torch.randn(batch_size, latent_dim).to(device)
        fake_imgs = generator(z)
        g_loss = criterion(discriminator(fake_imgs), real_labels)
        optimizer_G.zero_grad()
        g_loss.backward()
        optimizer_G.step()
```

```
    print(f"Epoch [{epoch+1}/{epochs}]  D_loss: {d_loss.item():.4f}  G_loss: {g_loss.item():.4f}")

    # Show sample generated image
    if (epoch + 1) % 10 == 0:
        with torch.no_grad():
            z = torch.randn(16, latent_dim).to(device)
            samples = generator(z).cpu().numpy()
            fig, axs = plt.subplots(4, 4, figsize=(4, 4))
            for i in range(4):
                for j in range(4):
                    axs[i, j].imshow(samples[i * 4 + j][0], cmap='gray')
                    axs[i, j].axis('off')
            plt.show()
```

Output: