| EX.NO 1 | Simple Neural Network |
| --- | --- |
|  |

**Program :**

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.utils import to\_categorical

# Load the dataset

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

# Normalize the images to [0, 1]

x\_train = x\_train / 255.0

x\_test = x\_test / 255.0

# One-hot encode the labels

y\_train = to\_categorical(y\_train)

y\_test = to\_categorical(y\_test)

# Create the neural network

model = Sequential([

Flatten(input\_shape=(28, 28)), # Input layer (flatten 28x28 images)

Dense(128, activation='relu'), # Hidden layer with 128 neurons

Dense(10, activation='softmax') # Output layer (10 classes)

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(x\_train, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)

# Evaluate on test set

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test accuracy: {test\_acc:.4f}")

| EX.NO 2 | Backpropagation |
| --- | --- |
|  |

**Program :**

import numpy as np

sigmoid = lambda x: 1 / (1 + np.exp(-x))

sigmoid\_derivative = lambda x: x \* (1 - x)

X = np.array([[0,0],[0,1],[1,0],[1,1]])

y = np.array([[0],[1],[1],[0]])

np.random.seed(42)

wh = np.random.uniform(size=(2,2))

bh = np.zeros((1,2))

wo = np.random.uniform(size=(2,1))

bo = np.zeros((1,1))

lr, epochs = 0.1, 10000

for epoch in range(epochs):

h = sigmoid(np.dot(X, wh) + bh)

pred = sigmoid(np.dot(h, wo) + bo)

e = y - pred

d\_pred = e \* sigmoid\_derivative(pred)

d\_hidden = d\_pred.dot(wo.T) \* sigmoid\_derivative(h)

wo += h.T.dot(d\_pred) \* lr

bo += np.sum(d\_pred, axis=0, keepdims=True) \* lr

wh += X.T.dot(d\_hidden) \* lr

bh += np.sum(d\_hidden, axis=0, keepdims=True) \* lr

if epoch % 1000 == 0:

print(f"Epoch {epoch}, Loss: {np.mean(e\*\*2):.4f}")

print("\nFinal output after training:")

print(pred)

| EX.NO 3 | Different Activation Functions |
| --- | --- |
|  |

**Program :**

import numpy as np

# ReLU activation function and its derivative

def relu(x):

return np.maximum(0, x)

def relu\_derivative(x):

return (x > 0).astype(float)

def tanh(x):

return np.tanh(x)

def tanh\_derivative(x):

return 1 - np.tanh(x)\*\*2

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

return x \* (1 - x)

# Training dataset: 4 samples with 2 inputs each (for XOR)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Target output: XOR logic gate

y = np.array([[0], [1], [1], [0]])

# Seed for reproducibility

np.random.seed(42)

# Initialize weights and biases randomly

input\_layer\_neurons = X.shape[1] # 2

hidden\_layer1\_neurons = 3 # 3 neurons in hidden layer 1

hidden\_layer2\_neurons = 2 # 2 neurons in hidden layer 2

output\_neurons = 1 # 1 neuron in output layer

# Weights and biases

wh1 = np.random.uniform(size=(input\_layer\_neurons, hidden\_layer1\_neurons)) # weights for input to hidden layer 1

bh1 = np.zeros((1, hidden\_layer1\_neurons)) # bias for hidden layer 1

wh2 = np.random.uniform(size=(hidden\_layer1\_neurons, hidden\_layer2\_neurons)) # weights for hidden layer 1 to hidden layer 2

bh2 = np.zeros((1, hidden\_layer2\_neurons)) # bias for hidden layer 2

wo = np.random.uniform(size=(hidden\_layer2\_neurons, output\_neurons)) # weights for hidden layer 2 to output

bo = np.zeros((1, output\_neurons)) # bias for output layer

# Training loop

epochs = 10000

learning\_rate = 0.1

for epoch in range(epochs):

# --- Forward pass ---

# Input to hidden layer 1

hidden\_input1 = np.dot(X, wh1) + bh1

hidden\_output1 = relu(hidden\_input1) # ReLU activation in hidden layer 1

# Hidden layer 1 to hidden layer 2

hidden\_input2 = np.dot(hidden\_output1, wh2) + bh2

hidden\_output2 = tanh(hidden\_input2) # Tanh activation in hidden layer 2

# Hidden layer 2 to output layer

final\_input = np.dot(hidden\_output2, wo) + bo

predicted\_output = sigmoid(final\_input) # Sigmoid activation in output layer

# --- Backward pass ---

# Calculate error

error = y - predicted\_output

d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output)

# Error in hidden layer 2

error\_hidden\_layer2 = d\_predicted\_output.dot(wo.T)

d\_hidden\_layer2 = error\_hidden\_layer2 \* tanh\_derivative(hidden\_output2)

# Error in hidden layer 1

error\_hidden\_layer1 = d\_hidden\_layer2.dot(wh2.T)

d\_hidden\_layer1 = error\_hidden\_layer1 \* relu\_derivative(hidden\_output1)

# --- Update weights and biases ---

# Update weights and biases for output layer

wo += hidden\_output2.T.dot(d\_predicted\_output) \* learning\_rate

bo += np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learning\_rate

# Update weights and biases for hidden layer 2

wh2 += hidden\_output1.T.dot(d\_hidden\_layer2) \* learning\_rate

bh2 += np.sum(d\_hidden\_layer2, axis=0, keepdims=True) \* learning\_rate

# Update weights and biases for hidden layer 1

wh1 += X.T.dot(d\_hidden\_layer1) \* learning\_rate

bh1 += np.sum(d\_hidden\_layer1, axis=0, keepdims=True) \* learning\_rate

# Optional: print error every 1000 epochs

if epoch % 1000 == 0:

loss = np.mean(np.square(error))

print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Final output after training

print("\nFinal output after training:")

print(predicted\_output)

**Program :**

| EX.NO 4 | Stochastic Gradient Descent |
| --- | --- |
|  |

from sklearn.datasets import load\_iris

from sklearn.linear\_model import SGDClassifier

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

from sklearn.preprocessing import StandardScaler

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

import matplotlib.pyplot as plt

import seaborn as sns

# Step 1: Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

target\_names = iris.target\_names

# Step 2: Convert it into a binary classification problem

# Use only Setosa (0) and Versicolor (1), discard Virginica (2)

X = X[y != 2]

y = y[y != 2]

# Step 3: Split 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)

# Step 4: Feature scaling (important for SGD)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Step 5: Create and train the SGD classifier

sgd\_model = SGDClassifier(loss='hinge', learning\_rate='optimal', max\_iter=1000, random\_state=42)

sgd\_model.fit(X\_train\_scaled, y\_train)

# Step 6: Make predictions

y\_pred = sgd\_model.predict(X\_test\_scaled)

# Step 7: Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

print("Test Accuracy:", round(accuracy, 4))

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred, target\_names=target\_names[:2]))

# Step 8: Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=target\_names[:2], yticklabels=target\_names[:2])

plt.title("Confusion Matrix")

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.show()

**Program :**

| EX.NO 5 | Momentum Based Gradient Descent |
| --- | --- |
|  |

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import fetch\_california\_housing

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error

# Load dataset

data = fetch\_california\_housing()

X = data.data

y = data.target.reshape(-1, 1)

# Standardize features and target

scaler\_X = StandardScaler()

scaler\_y = StandardScaler()

X = scaler\_X.fit\_transform(X)

y = scaler\_y.fit\_transform(y)

# Train-test split

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

# Initialize parameters

n\_samples, n\_features = X\_train.shape

weights = np.zeros((n\_features, 1))

bias = 0

v\_w = np.zeros\_like(weights)

v\_b = 0

# Hyperparameters

learning\_rate = 0.01

momentum = 0.9

epochs = 100

losses = []

# Training loop

for epoch in range(epochs):

y\_pred = X\_train @ weights + bias

error = y\_pred - y\_train

dw = (2 / n\_samples) \* X\_train.T @ error

db = (2 / n\_samples) \* np.sum(error)

v\_w = momentum \* v\_w - learning\_rate \* dw

v\_b = momentum \* v\_b - learning\_rate \* db

weights += v\_w

bias += v\_b

loss = np.mean(error \*\* 2)

losses.append(loss)

if epoch % 20 == 0:

print(f"Epoch {epoch}: Loss = {loss:.4f}")

# Plot training loss

plt.plot(range(epochs), losses, marker='o', color='blue')

plt.title("Training Loss Curve")

plt.xlabel("Epochs")

plt.ylabel("MSE Loss")

plt.grid(True)

plt.show()

| EX.NO 6 | VGG pretrained model |
| --- | --- |
|  |

**Program :**

import torch

import torchvision

from torchvision import transforms

from PIL import Image

import requests

import json

# Load a pretrained VGG16 model

model = torchvision.models.vgg16(pretrained=True)

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

# Define the same transformation as before

transform = transforms.Compose([

transforms.Resize(256), # Resize the shorter side to 256 pixels

transforms.CenterCrop(224), # Crop the central 224x224 portion

transforms.ToTensor(), # Convert image to PyTorch tensor

transforms.Normalize(mean=[0.485, 0.456, 0.406],

std=[0.229, 0.224, 0.225]) # Normalize using ImageNet stats

])

# Load and preprocess the input image

img\_path = '/content/DOG.webp' # Replace with your image path

image = Image.open(img\_path).convert('RGB') # Convert the image to RGB

image = transform(image).unsqueeze(0) # Add batch dimension (1, C, H, W)

# Make the prediction

with torch.no\_grad():

outputs = model(image)

\_, predicted\_class = torch.max(outputs, 1)

# Load the ImageNet class labels

labels\_url = "https://storage.googleapis.com/download.tensorflow.org/data/imagenet\_class\_index.json"

labels = json.loads(requests.get(labels\_url).text)

# Get the predicted label

predicted\_label = labels[str(predicted\_class.item())][1]

print(f"Predicted class: {predicted\_label}")

| EX.NO 7 | Encoder Decoder |
| --- | --- |
|  |

**Program :**

import torch

import torch.nn as nn

import torchvision.transforms as transforms

from torchvision.datasets import MNIST

from torch.utils.data import DataLoader

import matplotlib.pyplot as plt

# ====== Load MNIST dataset ======

transform = transforms.ToTensor()

dataset = MNIST(root='./data', train=True, download=True, transform=transform)

loader = DataLoader(dataset, batch\_size=64, shuffle=True)

# ====== Define encoder and decoder layers ======

fc1 = nn.Linear(784, 128)

fc2 = nn.Linear(128, 32)

fc3 = nn.Linear(32, 128)

fc4 = nn.Linear(128, 784)

activation = nn.ReLU()

sigmoid = nn.Sigmoid()

# ====== Optimizer and loss ======

params = list(fc1.parameters()) + list(fc2.parameters()) + list(fc3.parameters()) + list(fc4.parameters())

optimizer = torch.optim.Adam(params, lr=0.01)

loss\_fn = nn.MSELoss()

# ====== Train the autoencoder ======

for epoch in range(5): # 5 epochs for quick test

total\_loss = 0

for images, \_ in loader:

images = images.view(-1, 784)

# Encode

x = activation(fc1(images))

encoded = activation(fc2(x))

# Decode

x = activation(fc3(encoded))

decoded = sigmoid(fc4(x))

# Loss

loss = loss\_fn(decoded, images)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

total\_loss += loss.item()

print(f"Epoch {epoch+1}, Loss: {total\_loss/len(loader):.4f}")

# ====== Test on one image ======

test\_image, \_ = dataset[0]

test\_image\_flat = test\_image.view(-1, 784)

# Encode + Decode

with torch.no\_grad():

x = activation(fc1(test\_image\_flat))

encoded = activation(fc2(x))

x = activation(fc3(encoded))

decoded\_flat = sigmoid(fc4(x))

decoded\_image = decoded\_flat.view(1, 28, 28)

# ====== Show original vs reconstructed ======

fig, axes = plt.subplots(1, 2)

axes[0].imshow(test\_image.squeeze(), cmap='gray')

axes[0].set\_title('Original')

axes[0].axis('off')

axes[1].imshow(decoded\_image.squeeze(), cmap='gray')

axes[1].set\_title('Reconstructed')

axes[1].axis('off')

plt.show()

| EX.NO 8 | Convolutional Neural Network |
| --- | --- |
|  |

**Program :**

import tensorflow as tf

import tensorflow\_datasets as tfds

import matplotlib.pyplot as plt

import numpy as np

# Dataset & preprocessing same as before

(ds\_train, ds\_val), ds\_info = tfds.load(

'cats\_vs\_dogs',

split=['train[:80%]', 'train[80%:]'],

shuffle\_files=True,

as\_supervised=True,

with\_info=True,

)

IMG\_SIZE = 128

BATCH\_SIZE = 32

def preprocess(image, label):

image = tf.image.resize(image, (IMG\_SIZE, IMG\_SIZE))

image = image / 255.0

return image, label

ds\_val = ds\_val.map(preprocess).batch(BATCH\_SIZE)

# Assume model is already trained as before, or load saved model

# For demonstration, create a new model (or load from your trained model)

model = tf.keras.Sequential([

tf.keras.layers.Conv2D(32, (3,3), activation='relu', input\_shape=(IMG\_SIZE, IMG\_SIZE, 3)),

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

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

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

tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

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

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(512, activation='relu'),

tf.keras.layers.Dense(1, activation='sigmoid')

])

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

# Normally you train the model, but here for example let's just load weights if you have them

# model.load\_weights('your\_model\_weights.h5')

# Or train for 1 epoch quickly for demo

# (Remove if you want to use pre-trained)

# (Use your trained model for real predictions)

history = model.fit(ds\_val.take(5), epochs=1)

# Get some images and labels from validation set

images, labels = next(iter(ds\_val))

# Predict

predictions = model.predict(images)

pred\_labels = (predictions > 0.5).astype("int32")

# Plot images with predicted and true labels

class\_names = ['Cat', 'Dog']

plt.figure(figsize=(12, 12))

for i in range(9):

plt.subplot(3, 3, i+1)

plt.imshow(images[i])

title = f"True: {class\_names[labels[i]]}\nPred: {class\_names[pred\_labels[i][0]]}"

plt.title(title)

plt.axis('off')

plt.show()

| EX.NO 9 | Long Short Term Memory |
| --- | --- |
|  |

**Program :**

import torch

import torch.nn as nn

import numpy as np

# Sample text

corpus = "good mouring, hello how are you doing today? hope you are having a good day!"

# Character mappings

chars = sorted(list(set(corpus)))

char2idx = {ch: i for i, ch in enumerate(chars)}

idx2char = {i: ch for ch, i in char2idx.items()}

vocab\_size = len(chars)

# Hyperparameters

seq\_length = 10

embedding\_dim = 8

hidden\_size = 128

num\_layers = 1

num\_epochs = 200

learning\_rate = 0.01

# Prepare dataset

def create\_dataset(text, seq\_length):

X, y = [], []

for i in range(len(text) - seq\_length):

seq = text[i:i+seq\_length]

target = text[i+seq\_length]

X.append([char2idx[c] for c in seq])

y.append(char2idx[target])

return torch.tensor(X), torch.tensor(y)

X, y = create\_dataset(corpus, seq\_length)

# Layers (no class!)

embedding = nn.Embedding(vocab\_size, embedding\_dim)

lstm = nn.LSTM(embedding\_dim, hidden\_size, num\_layers, batch\_first=True)

fc = nn.Linear(hidden\_size, vocab\_size)

# Loss and optimizer

params = list(embedding.parameters()) + list(lstm.parameters()) + list(fc.parameters())

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(params, lr=learning\_rate)

# Training loop

for epoch in range(num\_epochs):

embedded = embedding(X) # (batch, seq\_len, embed\_dim)

h0 = torch.zeros(num\_layers, X.size(0), hidden\_size)

c0 = torch.zeros(num\_layers, X.size(0), hidden\_size)

lstm\_out, \_ = lstm(embedded, (h0, c0)) # (batch, seq\_len, hidden)

logits = fc(lstm\_out[:, -1, :]) # last time step output

loss = criterion(logits, y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if (epoch + 1) % 20 == 0:

print(f"Epoch [{epoch+1}/{num\_epochs}], Loss: {loss.item():.4f}")

# Text generation (manual logic)

def generate(seed\_text, gen\_len):

generated = seed\_text

input\_seq = [char2idx[c] for c in seed\_text]

for \_ in range(gen\_len):

input\_tensor = torch.tensor([input\_seq[-seq\_length:]], dtype=torch.long)

embedded = embedding(input\_tensor)

h0 = torch.zeros(num\_layers, 1, hidden\_size)

c0 = torch.zeros(num\_layers, 1, hidden\_size)

lstm\_out, \_ = lstm(embedded, (h0, c0))

logits = fc(lstm\_out[:, -1, :])

next\_idx = torch.argmax(logits, dim=1).item()

next\_char = idx2char[next\_idx]

generated += next\_char

input\_seq.append(next\_idx)

return generated

# Try it!

seed = "hello how "

print("\nGenerated Text:\n", generate(seed, 52))

| EX.NO 10 | Recurrent Neural Network |
| --- | --- |
|  |

**Program :**

import torch

import torch.nn as nn

import numpy as np

# Toy dataset (positive/negative sequences)

corpus = [

("I love this product!", 1), # Positive

("This is amazing!", 1), # Positive

("Absolutely terrible.", 0), # Negative

("I hate it.", 0), # Negative

("Not bad, but could be better.", 1), # Positive

("Worst purchase ever.", 0) # Negative

]

# Character mappings

chars = sorted(list(set(' '.join([x[0] for x in corpus])))) # Unique characters from the dataset

char2idx = {ch: i for i, ch in enumerate(chars)}

idx2char = {i: ch for ch, i in char2idx.items()}

vocab\_size = len(chars)

# Hyperparameters

seq\_length = 10 # Fixed sequence length for training

embedding\_dim = 8

hidden\_size = 16

num\_layers = 1

num\_epochs = 100

learning\_rate = 0.01

# Prepare data

def create\_sequences(data, seq\_length):

X, y = [], []

for text, label in data:

text = text.lower() # Case normalization

text = text[:seq\_length] # Cut text to fit into sequence length

X.append([char2idx.get(ch, 0) for ch in text]) # Use 0 as the index for unknown chars

y.append(label)

return torch.tensor(X), torch.tensor(y)

X, y = create\_sequences(corpus, seq\_length)

# RNN model

rnn = nn.RNN(embedding\_dim, hidden\_size, num\_layers, batch\_first=True)

embedding = nn.Embedding(vocab\_size, embedding\_dim)

fc = nn.Linear(hidden\_size, 1)

# Loss and optimizer

params = list(embedding.parameters()) + list(rnn.parameters()) + list(fc.parameters())

criterion = nn.BCEWithLogitsLoss() # Binary Cross Entropy Loss

optimizer = torch.optim.Adam(params, lr=learning\_rate)

# Training loop

for epoch in range(num\_epochs):

embedded = embedding(X) # (batch, seq\_len, embed\_dim)

h0 = torch.zeros(num\_layers, X.size(0), hidden\_size)

out, \_ = rnn(embedded, h0) # (batch, seq\_len, hidden\_size)

logits = fc(out[:, -1, :]) # Only use the last hidden state

output = torch.sigmoid(logits).squeeze() # Apply sigmoid for binary classification

loss = criterion(output, y.float())

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if (epoch + 1) % 20 == 0:

print(f"Epoch [{epoch+1}/{num\_epochs}], Loss: {loss.item():.4f}")

# Test the model

def predict(text):

text = text.lower()[:seq\_length] # Truncate or pad the sequence

input\_seq = torch.tensor([char2idx.get(ch, 0) for ch in text]).unsqueeze(0) # (1, seq\_len)

embedded = embedding(input\_seq)

h0 = torch.zeros(num\_layers, 1, hidden\_size)

out, \_ = rnn(embedded, h0)

logits = fc(out[:, -1, :])

output = torch.sigmoid(logits).squeeze().item()

return "Positive" if output > 0.5 else "Negative"

# Test the model on some examples

print("\nTest Predictions:")

print("Input: 'I love this!' -> Prediction:", predict("I love this!"))

print("Input: 'This is awful' -> Prediction:", predict("This is awful"))

| EX.NO 11 | LSTM FOR STOCK PRICE PREDICTION |
| --- | --- |
|  |

**Program :**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

# Load the dataset

df = pd.read\_csv("/content/data/tesla-stock-price.csv")

df = df[['date', 'close']]

df['date'] = pd.to\_datetime(df['date'])

df = df.set\_index('date')

# Normalize the data

scaler = MinMaxScaler(feature\_range=(0, 1))

df\_scaled = scaler.fit\_transform(df)

# Create sequences for LSTM

def create\_sequences(data, sequence\_length):

sequences, labels = [], []

for i in range(len(data) - sequence\_length):

seq = data[i:i + sequence\_length]

label = data[i + sequence\_length]

sequences.append(seq)

labels.append(label)

return np.array(sequences), np.array(labels)

# Sequence length

sequence\_length = 10

X, y = create\_sequences(df\_scaled, sequence\_length)

# Split the data into training and testing sets

train\_size = int(len(X) \* 0.80)

X\_train, X\_test = X[:train\_size], X[train\_size:]

y\_train, y\_test = y[:train\_size], y[train\_size:]

# Build the LSTM model

model = Sequential([

LSTM(units=50, activation='relu', input\_shape=(X\_train.shape[1], X\_train.shape[2])),

Dense(units=1)

])

model.compile(optimizer='adam', loss='mean\_squared\_error')

# Train the model

model.fit(X\_train, y\_train, epochs=50, batch\_size=32, verbose=1)

# Make predictions on the test set

predictions = model.predict(X\_test)

# Inverse transform the predictions and actual values

predictions = scaler.inverse\_transform(predictions)

y\_test\_actual = scaler.inverse\_transform(y\_test.reshape(-1, 1))

# Plot the results

plt.figure(figsize=(14, 7))

dates = df.index[sequence\_length + train\_size:] # Match the length of test data

plt.plot(dates, y\_test\_actual, label='Actual Stock Price')

plt.plot(dates, predictions, label='Predicted Stock Price')

plt.title('Stock Price Prediction using LSTM')

plt.xlabel('Date')

plt.ylabel('Stock Price')

plt.legend()

plt.show()