PROGRAM 1

import torch

import torch.nn as nn

import torch.optim as optim

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

from sklearn.preprocessing import StandardScaler

# Generate some random data for demonstration purposes

torch.manual\_seed(42)

X = torch.rand((1000, 10))

y = torch.randint(2, (1000, 1))

# Split the data into training and testing sets

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

# Standardize the data

scaler = StandardScaler()

X\_train = torch.from\_numpy(scaler.fit\_transform(X\_train)).float()

X\_test = torch.from\_numpy(scaler.transform(X\_test)).float()

y\_train = torch.from\_numpy(y\_train).float()

y\_test = torch.from\_numpy(y\_test).float()

# Define the neural network architecture

class SimpleNN(nn.Module):

def \_\_init\_\_(self):

super(SimpleNN, self).\_\_init\_\_()

self.layer1 = nn.Linear(X\_train.shape[1], 64)

self.layer2 = nn.Linear(64, 32)

self.layer3 = nn.Linear(32, 1)

self.sigmoid = nn.Sigmoid()

def forward(self, x):

x = self.sigmoid(self.layer1(x))

x = self.sigmoid(self.layer2(x))

x = self.sigmoid(self.layer3(x))

return x

model = SimpleNN()

# Define the loss function and optimizer

criterion = nn.BCELoss()

optimizer = optim.Adam(model.parameters())

# Train the model

for epoch in range(10):

optimizer.zero\_grad()

outputs = model(X\_train)

loss = criterion(outputs, y\_train)

loss.backward()

optimizer.step()

# Evaluate the model on the test set

with torch.no\_grad():

model.eval()

test\_outputs = model(X\_test)

test\_loss = criterion(test\_outputs, y\_test)

test\_accuracy = ((test\_outputs > 0.5) == y\_test.byte()).float().mean()

print(f'Test Loss: {test\_loss.item()}, Test Accuracy: {test\_accuracy.item()}')

PROGRAM 2

import torch

import torch.nn as nn

import torch.optim as optim

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

from sklearn.preprocessing import StandardScaler

# Generate some random data for demonstration purposes

torch.manual\_seed(42)

X = torch.rand((1000, 10))

y = torch.randint(2, (1000, 1))

# Split the data into training and testing sets

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

# Standardize the data

scaler = StandardScaler()

X\_train = torch.from\_numpy(scaler.fit\_transform(X\_train)).float()

X\_test = torch.from\_numpy(scaler.transform(X\_test)).float()

y\_train = torch.from\_numpy(y\_train).float()

y\_test = torch.from\_numpy(y\_test).float()

# Define the neural network architecture

class CustomModel(nn.Module):

def \_\_init\_\_(self):

super(CustomModel, self).\_\_init\_\_()

self.layer1 = nn.Linear(X\_train.shape[1], 1)

self.layer2 = nn.Linear(1, 64)

self.layer3 = nn.Linear(64, 32)

self.layer4 = nn.Linear(32, 1)

self.tanh = nn.Tanh()

self.relu = nn.ReLU()

self.sigmoid = nn.Sigmoid()

def forward(self, x):

x = self.layer1(x)

x = self.tanh(x)

x = self.relu(self.layer2(x))

x = self.relu(self.layer3(x))

x = self.sigmoid(self.layer4(x))

return x

model = CustomModel()

# Define the loss function and optimizer

criterion = nn.BCELoss()

optimizer = optim.Adam(model.parameters())

# Train the model

for epoch in range(10):

optimizer.zero\_grad()

outputs = model(X\_train)

loss = criterion(outputs, y\_train)

loss.backward()

optimizer.step()

# Evaluate the model on the test set

with torch.no\_grad():

model.eval()

test\_outputs = model(X\_test)

test\_loss = criterion(test\_outputs, y\_test)

test\_accuracy = ((test\_outputs > 0.5) == y\_test.byte()).float().mean()

print(f'Test Loss: {test\_loss.item()}, Test Accuracy: {test\_accuracy.item()}')

PROGRAM 3

import torch

import torch.nn as nn

import torch.optim as optim

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

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.datasets import load\_iris

# Load Iris dataset

iris = load\_iris()

X = iris.data

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

# One-hot encode the target variable for multi-class classification

encoder = OneHotEncoder(sparse=False)

y\_onehot = encoder.fit\_transform(y)

# Split the data into training and testing sets

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

# Standardize the data

scaler = StandardScaler()

X\_train = torch.from\_numpy(scaler.fit\_transform(X\_train)).float()

X\_test = torch.from\_numpy(scaler.transform(X\_test)).float()

y\_train = torch.from\_numpy(y\_train).float()

y\_test = torch.from\_numpy(y\_test).float()

# Define the neural network architecture

class IrisModel(nn.Module):

def \_\_init\_\_(self):

super(IrisModel, self).\_\_init\_\_()

self.layer1 = nn.Linear(X\_train.shape[1], 64)

self.layer2 = nn.Linear(64, 32)

self.layer3 = nn.Linear(32, 3) # Multi-class classification with 3 classes

self.relu = nn.ReLU()

self.softmax = nn.Softmax(dim=1)

def forward(self, x):

x = self.relu(self.layer1(x))

x = self.relu(self.layer2(x))

x = self.softmax(self.layer3(x))

return x

model = IrisModel()

# Define the loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=0.01)

# Train the model with Gradient Descent

print("Training with Gradient Descent...")

for epoch in range(10):

optimizer.zero\_grad()

outputs = model(X\_train)

loss = criterion(outputs, torch.max(y\_train, 1)[1])

loss.backward()

optimizer.step()

print(f'Epoch {epoch + 1}/{10}, Loss: {loss.item()}')

# Train the model with Stochastic Gradient Descent (SGD)

print("\nTraining with Stochastic Gradient Descent (SGD)...")

optimizer = optim.SGD(model.parameters(), lr=0.01)

for epoch in range(50):

optimizer.zero\_grad()

outputs = model(X\_train)

loss = criterion(outputs, torch.max(y\_train, 1)[1])

loss.backward()

optimizer.step()

print(f'Epoch {epoch + 1}/{50}, Loss: {loss.item()}')

PROGRAM 4

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision.transforms as transforms

from torch.utils.data import DataLoader

from torchvision.datasets import MNIST

# Load and preprocess the MNIST dataset

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

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

test\_dataset = MNIST(root='./data', train=False, download=True, transform=transform)

train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=64, shuffle=False)

# Define the CNN architecture

class CNN(nn.Module):

def \_\_init\_\_(self):

super(CNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 32, kernel\_size=3, padding=1)

self.pool1 = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(32, 64, kernel\_size=3, padding=1)

self.pool2 = nn.MaxPool2d(2, 2)

self.conv3 = nn.Conv2d(64, 64, kernel\_size=3, padding=1)

self.pool3 = nn.MaxPool2d(2, 2)

self.flatten = nn.Flatten()

self.fc1 = nn.Linear(64 \* 3 \* 3, 128)

self.fc2 = nn.Linear(128, 10)

def forward(self, x):

x = self.pool1(torch.relu(self.conv1(x)))

x = self.pool2(torch.relu(self.conv2(x)))

x = self.pool3(torch.relu(self.conv3(x)))

x = self.flatten(x)

x = torch.relu(self.fc1(x))

x = self.fc2(x)

return x

model = CNN()

# Define the loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters())

# Train the model

for epoch in range(5):

for images, labels in train\_loader:

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

# Evaluate the model on the test set

correct = 0

total = 0

with torch.no\_grad():

for images, labels in test\_loader:

outputs = model(images)

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

total += labels.size(0)

correct += (predicted == labels).sum().item()

test\_accuracy = correct / total

print(f'Test accuracy: {test\_accuracy}')

PROGRAM 5

!pip install tensorflow yfinance pandas numpy

import yfinance as yf

import pandas as pd

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, TensorDataset

from sklearn.preprocessing import MinMaxScaler # Add this line

import matplotlib.pyplot as plt

# Rest of your code...

# Function to fetch historical stock data

def fetch\_stock\_data(ticker, start\_date, end\_date):

stock\_data = yf.download(ticker, start=start\_date, end=end\_date)

return stock\_data

# Function to preprocess stock data

def preprocess\_data(stock\_data):

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

# Extracting the closing prices

closing\_prices = stock\_data['Close'].values.reshape(-1, 1)

# Scaling the closing prices

scaled\_prices = scaler.fit\_transform(closing\_prices)

return scaled\_prices, scaler

# Function to create sequences for training the GRU model

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:i+sequence\_length+1]

sequences.append(seq)

labels.append(label)

return torch.Tensor(sequences), torch.Tensor(labels)

# GRU model definition

class GRUModel(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

super(GRUModel, self).\_\_init\_\_()

self.gru = nn.GRU(input\_size, hidden\_size, batch\_first=True)

self.fc = nn.Linear(hidden\_size, output\_size)

def forward(self, x):

\_, h\_n = self.gru(x)

out = self.fc(h\_n[-1])

return out

# Function to plot actual vs predicted prices

def plot\_predictions(actual, predicted, ticker):

plt.figure(figsize=(10, 6))

plt.plot(actual, label='Actual Prices', color='blue')

plt.plot(predicted, label='Predicted Prices', color='red')

plt.title(f'{ticker} Stock Price Prediction')

plt.xlabel('Time')

plt.ylabel('Stock Price')

plt.legend()

plt.show()

# Main function

def main():

ticker = 'AAPL'

start\_date = '2022-01-01'

end\_date = '2023-01-01'

sequence\_length = 10 # Number of previous days to consider for prediction

# Fetch historical stock data

stock\_data = fetch\_stock\_data(ticker, start\_date, end\_date)

# Preprocess the data

scaled\_prices, scaler = preprocess\_data(stock\_data)

# Create sequences for training the model

X, y = create\_sequences(scaled\_prices, sequence\_length)

# Split the data into training and testing sets

split\_ratio = 0.8

split\_index = int(len(X) \* split\_ratio)

X\_train, X\_test = X[:split\_index], X[split\_index:]

y\_train, y\_test = y[:split\_index], y[split\_index:]

# Define PyTorch DataLoader for training and testing sets

train\_dataset = TensorDataset(X\_train, y\_train)

test\_dataset = TensorDataset(X\_test, y\_test)

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=1, shuffle=False)

# Model parameters

input\_size = 1

hidden\_size = 50

output\_size = 1

# Build and train the GRU model

model = GRUModel(input\_size, hidden\_size, output\_size)

criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

num\_epochs = 50

for epoch in range(num\_epochs):

for inputs, labels in train\_loader:

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

# Predict on the test set

model.eval()

predictions = []

with torch.no\_grad():

for inputs, labels in test\_loader:

outputs = model(inputs)

predictions.append(outputs.numpy())

predicted\_prices = scaler.inverse\_transform(np.array(predictions).reshape(-1, 1))

actual\_prices = scaler.inverse\_transform(y\_test.numpy().reshape(-1, 1))

# Plot the results

plot\_predictions(actual\_prices, predicted\_prices, ticker)

if \_\_name\_\_ == "\_\_main\_\_":

main()

PROGRAM 7

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, TensorDataset

import torchvision.transforms as transforms

from torchvision.datasets import MNIST

import numpy as np

import matplotlib.pyplot as plt

# Define the denoising autoencoder model in PyTorch

class DenoisingAutoencoder(nn.Module):

def \_\_init\_\_(self):

super(DenoisingAutoencoder, self).\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Conv2d(1, 32, kernel\_size=3, padding=1),

nn.ReLU(),

nn.MaxPool2d(kernel\_size=2, stride=2, padding=0),

nn.Conv2d(32, 32, kernel\_size=3, padding=1),

nn.ReLU(),

nn.MaxPool2d(kernel\_size=2, stride=2, padding=0)

)

self.decoder = nn.Sequential(

nn.Conv2d(32, 32, kernel\_size=3, padding=1),

nn.ReLU(),

nn.Upsample(scale\_factor=2, mode='nearest'),

nn.Conv2d(32, 32, kernel\_size=3, padding=1),

nn.ReLU(),

nn.Upsample(scale\_factor=2, mode='nearest'),

nn.Conv2d(32, 1, kernel\_size=3, padding=1),

nn.Sigmoid()

)

def forward(self, x):

x = self.encoder(x)

x = self.decoder(x)

return x

# Load MNIST dataset and add noise

transform = transforms.Compose([transforms.ToTensor()])

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

test\_dataset = MNIST(root='./data', train=False, download=True, transform=transform)

def add\_noise(img, noise\_factor=0.5):

noise = torch.randn\_like(img) \* noise\_factor

noisy\_img = img + noise

return torch.clamp(noisy\_img, 0., 1.)

# Initialize the denoising autoencoder model

model = DenoisingAutoencoder()

# Loss function and optimizer

criterion = nn.BCELoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the denoising autoencoder

num\_epochs = 10

batch\_size = 128

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)

for epoch in range(num\_epochs):

model.train()

for data in train\_loader:

img, \_ = data

noisy\_img = add\_noise(img)

optimizer.zero\_grad()

outputs = model(noisy\_img)

loss = criterion(outputs, img)

loss.backward()

optimizer.step()

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

# Denoise some test images

model.eval()

denoised\_images = []

with torch.no\_grad():

for data in test\_loader:

img, \_ = data

noisy\_img = add\_noise(img)

outputs = model(noisy\_img)

denoised\_images.append(outputs)

# Display some examples

n = 10 # Number of images to display

plt.figure(figsize=(20, 4))

for i in range(n):

# Display original images

ax = plt.subplot(2, n, i + 1)

plt.imshow(noisy\_img[i].squeeze().numpy(), cmap='gray')

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

# Display denoised images

ax = plt.subplot(2, n, i + 1 + n)

plt.imshow(denoised\_images[0][i].squeeze().numpy(), cmap='gray')

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()