PROGRAM 1

*# Define the neural network architecture for binary classification*

class NeuralNetwork(nn.Module):

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

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

self.layer1 = nn.Linear(2, 10) *# Input layer to hidden layer*

self.layer2 = nn.Linear(10, 1) *# Hidden layer to output layer*

self.sigmoid = nn.Sigmoid() *# Sigmoid activation function*

def forward(self, x):

x = self.layer1(x)

x = self.sigmoid(x)

x = self.layer2(x)

x = self.sigmoid(x)

return x

*# Instantiate the model, define loss function and optimizer*

model = NeuralNetwork()

criterion = nn.BCELoss() *# Binary Cross Entropy Loss for binary classification*

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

*# Training the neural network*

epochs = 1000

for epoch in range(epochs):

*# Forward pass*

y\_pred = model(X\_tensor)

*# Compute the loss*

loss = criterion(y\_pred, y\_tensor)

*# Backward pass and optimization*

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if (epoch + 1) % 100 == 0:

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

PROGRAM 2

*# Define the neural network architecture*

class DeepNeuralNetwork(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size1, hidden\_size2, output\_size):

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

self.layer1 = nn.Linear(input\_size, hidden\_size1)

self.activation1 = nn.Tanh() *# or nn.ReLU() for ReLU activation*

self.layer2 = nn.Linear(hidden\_size1, hidden\_size2)

self.activation2 = nn.Tanh() *# or nn.ReLU() for ReLU activation*

self.output\_layer = nn.Linear(hidden\_size2, output\_size)

self.output\_activation = nn.Sigmoid() *# or nn.Softmax(dim=1) for multiclass*

def forward(self, x):

x = self.layer1(x)

x = self.activation1(x)

x = self.layer2(x)

x = self.activation2(x)

x = self.output\_layer(x)

x = self.output\_activation(x)

return x

*# Instantiate the model, define loss function and optimizer*

input\_size = X\_train.shape[1]

hidden\_size1 = 64

hidden\_size2 = 32

output\_size = 1 *# Binary classification*

model = DeepNeuralNetwork(input\_size, hidden\_size1, hidden\_size2, output\_size)

criterion = nn.BCELoss() *# Binary Cross Entropy Loss for binary classification*

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

PROGRAM 3

*# Define the neural network architecture*

class DeepNeuralNetwork(nn.Module):

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

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

self.layer1 = nn.Linear(1, 10) *# Input layer to hidden layer*

self.layer2 = nn.Linear(10, 10) *# Hidden layer to hidden layer*

self.layer3 = nn.Linear(10, 1) *# Hidden layer to output layer*

def forward(self, x):

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

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

x = self.layer3(x)

return x

*# Instantiate the model*

model = DeepNeuralNetwork()

*# Define the loss function and optimizer for gradient descent*

criterion = nn.MSELoss()

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

*# Define the loss function and optimizer for stochastic gradient descent*

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

*# Training the neural network with gradient descent*

epochs = 1000

for epoch in range(epochs):

*# Forward pass*

y\_pred = model(X\_tensor)

*# Compute the loss*

loss = criterion(y\_pred, y\_tensor)

*# Backward pass and optimization*

optimizer\_gd.zero\_grad()

loss.backward()

optimizer\_gd.step()

PROGRAM 4

*# Define the CNN architecture*

class CNN(nn.Module):

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

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

self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=16, kernel\_size=3, stride=1, padding=1)

self.conv2 = nn.Conv2d(16, 32, kernel\_size=3, stride=1, padding=1)

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

self.fc1 = nn.Linear(32 \* 7 \* 7, 128)

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

self.relu = nn.ReLU()

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

def forward(self, x):

x = self.pool(self.relu(self.conv1(x)))

x = self.pool(self.relu(self.conv2(x)))

x = x.view(-1, 32 \* 7 \* 7) *# Flatten the tensor*

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

x = self.fc2(x)

x = self.softmax(x)

return x

*# Create an instance of the CNN*

model = CNN()

*# Define loss function and optimizer*

criterion = nn.CrossEntropyLoss()

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

PROGRAM 5

*# 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 = []

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, label))

return np.array([s[0] for s in sequences]), np.array([s[1] for s in sequences])

*# Function to build the GRU model*

def build\_gru\_model(sequence\_length, input\_dim=1):

model = Sequential()

model.add(GRU(50, activation='relu', input\_shape=(sequence\_length, input\_dim)))

model.add(Dropout(0.2))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

return model

PROGRAM 6

*# Add noise to the images*

noise\_factor = 0.5

def add\_noise(images, noise\_factor):

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

return torch.clamp(images + noise, 0., 1.)

*# Define the denoising autoencoder model*

class Autoencoder(nn.Module):

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

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

self.encoder = nn.Sequential(

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

nn.ReLU(),

nn.MaxPool2d(2, 2),

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

nn.ReLU(),

nn.MaxPool2d(2, 2)

)

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

*# Instantiate the model, loss function, and optimizer*

autoencoder = Autoencoder()

criterion = nn.BCELoss()

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