**Program 1**

import numpy as np

import matplotlib.pyplot as plt

def plot\_sigmoid():

x = np.linspace(-10, 10, 100)

y = 1 / (1 + np.exp(-x))

plt.plot(x, y)

plt.xlabel('Input')

plt.ylabel('Sigmoid Output')

plt.title('Sigmoid Activation Function')

plt.grid(True)

plt.show()

def plot\_tanh():

x = np.linspace(-10, 10, 100)

tanh = np.tanh(x)

plt.plot(x, tanh)

plt.title("Hyperbolic Tangent (tanh) Activation Function")

plt.xlabel("x")

plt.ylabel("tanh(x)")

plt.grid(True)

plt.show()

def plot\_relu():

x = np.linspace(-10, 10, 100)

relu = np.maximum(0, x)

plt.plot(x, relu)

plt.title("ReLU Activation Function")

plt.xlabel("x")

plt.ylabel("ReLU(x)")

plt.grid(True)

plt.show()

def plot\_leaky\_relu():

x = np.linspace(-10, 10, 100)

def leaky\_relu(x, alpha=0.1):

return np.where(x >= 0, x, alpha \* x)

leaky\_relu\_values = leaky\_relu(x)

plt.plot(x, leaky\_relu\_values)

plt.title("Leaky ReLU Activation Function")

plt.xlabel("x")

plt.ylabel("Leaky ReLU(x)")

plt.grid(True)

plt.show()

def softmax():

def softmax\_act(x):

e\_x = np.exp(x - np.max(x))

return e\_x / np.sum(e\_x, axis=0)

x = np.array([1, 2, 3])

result = softmax\_act(x)

print(result)

def plot\_softmax(probabilities, class\_labels):

plt.bar(class\_labels, probabilities)

plt.xlabel("Class")

plt.ylabel("Probability")

plt.title("Softmax Output")

plt.show()

class\_labels = ["Class A", "Class B", "Class C"]

plot\_softmax(result, class\_labels)

while True:

print("\nMAIN MENU")

print("1. Sigmoid")

print("2. Hyperbolic tangent")

print("3. Rectified Linear Unit")

print("4. Leaky ReLU")

print("5. Softmax")

print("6. Exit")

choice = int(input("Enter the Choice:"))

if choice == 1:

plot\_sigmoid()

elif choice == 2:

plot\_tanh()

elif choice == 3:

plot\_relu()

elif choice == 4:

plot\_leaky\_relu()

elif choice == 5:

softmax()

elif choice == 6:

break

else:

print("Oops! Incorrect Choice.")

**Program 2**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader, Subset

import numpy as np

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,)) # normalization optional

])

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

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

train\_subset = Subset(train\_dataset, range(200))

test\_subset = Subset(test\_dataset, range(50))

train\_loader = DataLoader(train\_subset, batch\_size=10, shuffle=True)

test\_loader = DataLoader(test\_subset, batch\_size=10, shuffle=False)

class SimpleANN(nn.Module):

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

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

self.fc1 = nn.Linear(28\*28, 128)

self.fc2 = nn.Linear(128, 64)

self.fc3 = nn.Linear(64, 10)

def forward(self, x):

x = torch.flatten(x, start\_dim=1)

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

x = torch.relu(self.fc2(x))

x = self.fc3(x)

return x

model = SimpleANN()

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

criterion = nn.CrossEntropyLoss()

def train\_model(num\_epochs):

for epoch in range(num\_epochs):

model.train()

train\_loss = 0

correct\_train = 0

total\_train = 0

for data, target in train\_loader:

optimizer.zero\_grad()

output = model(data)

loss = criterion(output, target)

loss.backward()

optimizer.step()

train\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_train += target.size(0)

correct\_train += (predicted == target).sum().item()

avg\_train\_loss = train\_loss / len(train\_loader)

train\_acc = 100 \* correct\_train / total\_train

model.eval()

test\_loss = 0

correct\_test = 0

total\_test = 0

with torch.no\_grad():

for data, target in test\_loader:

output = model(data)

loss = criterion(output, target)

test\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_test += target.size(0)

correct\_test += (predicted == target).sum().item()

avg\_test\_loss = test\_loss / len(test\_loader)

test\_acc = 100 \* correct\_test / total\_test

print(f'Epoch {epoch+1}, Train Loss: {avg\_train\_loss:.4f}, Train Accuracy: {train\_acc:.8f}%, '

f'Test Loss: {avg\_test\_loss:.4f}, Test Accuracy: {test\_acc:.8f}%')

train\_model(10)

**Program 3**

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

from torch.utils.data import DataLoader, Subset

import matplotlib.pyplot as plt

import numpy as np

class CNNWithBNDropout(nn.Module):

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

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

self.conv\_block1 = nn.Sequential(

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

nn.BatchNorm2d(32),

nn.ReLU(),

nn.MaxPool2d(2)

)

self.conv\_block2 = nn.Sequential(

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

nn.BatchNorm2d(64),

nn.ReLU(),

nn.MaxPool2d(2)

)

self.dense1 = nn.Linear(64 \* 8 \* 8, 512)

self.dense2 = nn.Linear(512, 10)

self.dropout = nn.Dropout(0.5)

self.relu = nn.ReLU()

self.flatten = nn.Flatten()

def forward(self, x):

x = self.conv\_block1(x)

x = self.conv\_block2(x)

x = self.flatten(x)

x = self.dense1(x)

x = self.relu(x)

x = self.dense2(x)

x = self.dropout(x)

return x

# Data preprocessing and loading

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))

])

train\_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

test\_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

train\_subset = Subset(train\_dataset, range(200))

test\_subset = Subset(test\_dataset, range(50))

train\_loader = DataLoader(train\_subset, batch\_size=10, shuffle=True)

test\_loader = DataLoader(test\_subset, batch\_size=10, shuffle=False)

# Function to train and evaluate a model

def train(model, optimizer, criterion, num\_epochs):

for epoch in range(num\_epochs):

model.train()

train\_loss = 0.0

correct\_train = 0

total\_train = 0

for data, target in train\_loader:

output = model(data)

loss = criterion(output, target)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

train\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_train += target.size(0)

correct\_train += (predicted == target).sum().item()

avg\_train\_loss = train\_loss / len(train\_loader)

train\_acc = 100 \* correct\_train / total\_train

model.eval()

test\_loss = 0.0

correct\_test = 0

total\_test = 0

with torch.no\_grad():

for data, target in test\_loader:

output = model(data)

loss = criterion(output, target)

test\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_test += target.size(0)

correct\_test += (predicted == target).sum().item()

avg\_test\_loss = test\_loss / len(test\_loader)

test\_acc = 100 \* correct\_test / total\_test

print(f'Epoch: [{epoch+1}/{num\_epochs}], Train Loss: {avg\_train\_loss:.4f}, '

f'Train Accuracy: {train\_acc:.4f}%, Test Loss: {avg\_test\_loss:.4f}, '

f'Test Accuracy: {test\_acc:.4f}%')

model = CNNWithBNDropout()

criterion = nn.CrossEntropyLoss()

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

train(model, optimizer, criterion, 30)

**Program 4**

import torch

import torch.nn as nn

import torch.nn.functional as F

from torchvision import datasets, transforms

from torch.utils.data import DataLoader, Subset

# Data transformation

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,))

])

# Load datasets

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

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

# Create subsets of datasets

train\_subset = Subset(train\_dataset, range(200))

test\_subset = Subset(test\_dataset, range(50))

# Create data loaders

train\_loader = DataLoader(train\_subset, batch\_size=10, shuffle=True)

test\_loader = DataLoader(test\_subset, batch\_size=10, shuffle=False)

# Define the CNN model

class SimpleCNN(nn.Module):

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

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

self.conv1 = nn.Conv2d(1, 10, kernel\_size=5)

self.conv2 = nn.Conv2d(10, 20, kernel\_size=5)

self.fc1 = nn.Linear(320, 50)

self.fc2 = nn.Linear(50, 10)

def forward(self, x):

x = F.relu(F.max\_pool2d(self.conv1(x), 2))

x = F.relu(F.max\_pool2d(self.conv2(x), 2))

x = x.view(-1, 320)

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

x = self.fc2(x)

return F.log\_softmax(x, dim=1)

# SGD update function

def sgd\_update(parameters, lr):

with torch.no\_grad():

for param in parameters:

if param.grad is not None:

param.data -= lr \* param.grad.data

param.grad.zero\_()

# Custom Adagrad optimizer

class CustomAdagrad():

def \_\_init\_\_(self, parameters, lr=0.01, epsilon=1e-10):

self.parameters = list(parameters)

self.lr = lr

self.epsilon = epsilon

self.sum\_squared\_gradients = [torch.zeros\_like(p) for p in self.parameters]

def step(self):

with torch.no\_grad():

for param, sum\_sq\_grad in zip(self.parameters, self.sum\_squared\_gradients):

if param.grad is not None:

sum\_sq\_grad += param.grad.data \*\* 2

adjusted\_lr = self.lr / (self.epsilon + torch.sqrt(sum\_sq\_grad))

param.data -= adjusted\_lr \* param.grad.data

param.grad.zero\_()

# Set device

device = torch.device('cpu')

# Initialize model and loss function

model = SimpleCNN().to(device)

criterion = nn.CrossEntropyLoss()

# Training function

def train\_model(num\_epochs, optimizer\_choice='adagrad'):

if optimizer\_choice == 'sgd':

optimizer = None

else:

optimizer = CustomAdagrad(model.parameters(), lr=0.01)

for epoch in range(num\_epochs):

model.train()

train\_loss = 0

correct\_train = 0

total\_train = 0

for data, target in train\_loader:

data, target = data.to(device), target.to(device)

output = model(data)

loss = criterion(output, target)

loss.backward()

if optimizer\_choice == 'sgd':

sgd\_update(model.parameters(), lr=0.01)

else:

optimizer.step()

train\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_train += target.size(0)

correct\_train += (predicted == target).sum().item()

avg\_train\_loss = train\_loss / len(train\_loader)

train\_acc = 100 \* correct\_train / total\_train

model.eval()

test\_loss = 0

correct\_test = 0

total\_test = 0

with torch.no\_grad():

for data, target in test\_loader:

output = model(data)

loss = criterion(output, target)

test\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_test += target.size(0)

correct\_test += (predicted == target).sum().item()

avg\_test\_loss = test\_loss / len(test\_loader)

test\_acc = 100 \* correct\_test / total\_test

print(f'Epoch {epoch+1}, Train Loss: {avg\_train\_loss:.4f}, Train Accuracy: {train\_acc:.8f}%, '

f'Test Loss: {avg\_test\_loss:.4f}, Test Accuracy: {test\_acc:.8f}%')

# Train the model

train\_model(5, optimizer\_choice='adagrad')

**Program 5**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader, Subset

import numpy as np

# Data transformation

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,)) # normalization optional

])

# Load datasets

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

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

# Create subsets of datasets

train\_subset = Subset(train\_dataset, range(200))

test\_subset = Subset(test\_dataset, range(50))

# Create data loaders

train\_loader = DataLoader(train\_subset, batch\_size=10, shuffle=True)

test\_loader = DataLoader(test\_subset, batch\_size=10, shuffle=False)

# Define the AlexNet model

class AlexNet(nn.Module):

def \_\_init\_\_(self, num\_classes=10):

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

self.features = nn.Sequential(

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

nn.ReLU(inplace=True),

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

nn.Conv2d(64, 192, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

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

nn.Conv2d(192, 384, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(384, 256, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(256, 256, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

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

)

self.classifier = nn.Sequential(

nn.Dropout(),

nn.Linear(256 \* 3 \* 3, 4096),

nn.ReLU(inplace=True),

nn.Dropout(),

nn.Linear(4096, 4096),

nn.ReLU(inplace=True),

nn.Linear(4096, num\_classes),

)

def forward(self, x):

x = self.features(x)

x = torch.flatten(x, 1)

x = self.classifier(x)

return x

# Initialize model, optimizer, and loss function

model = AlexNet()

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

criterion = nn.CrossEntropyLoss()

# Training function

def train\_model(num\_epochs):

for epoch in range(num\_epochs):

model.train()

train\_loss = 0.0

correct\_train = 0

total\_train = 0

for data, target in train\_loader:

optimizer.zero\_grad()

output = model(data)

loss = criterion(output, target)

loss.backward()

optimizer.step()

train\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_train += target.size(0)

correct\_train += (predicted == target).sum().item()

avg\_train\_loss = train\_loss / len(train\_loader)

train\_acc = 100 \* correct\_train / total\_train

model.eval()

test\_loss = 0.0

correct\_test = 0

total\_test = 0

with torch.no\_grad():

for data, target in test\_loader:

output = model(data)

loss = criterion(output, target)

test\_loss += loss.item()

predicted = torch.argmax(output.data, dim=1)

total\_test += target.size(0)

correct\_test += (predicted == target).sum().item()

avg\_test\_loss = test\_loss / len(test\_loader)

test\_acc = 100 \* correct\_test / total\_test

print(f'Epoch {epoch+1}, Train Loss: {avg\_train\_loss:.4f}, Train Accuracy: {train\_acc:.4f}%, '

f'Test Loss: {avg\_test\_loss:.4f}, Test Accuracy: {test\_acc:.4f}%')

# Train the model

train\_model(15)

**Program 6**

def forward(self, x):

x = x.long() # Convert input tensor to Long type

x = self.embedding(x)

\_, (hidden, \_) = self.lstm(x)

return self.fc(hidden.squeeze(0))

import torch

from torch import nn

from torch.utils.data import DataLoader, TensorDataset

# Assuming `data` and `labels` are already defined tensors

# Replace these with your actual data

data = torch.randn(100, 10) # Example data

labels = torch.randint(0, 2, (100,)) # Example labels

dataset = TensorDataset(data, labels)

loader = DataLoader(dataset, batch\_size=10, shuffle=True)

class LSTMClassifier(nn.Module):

def \_\_init\_\_(self, vocabsize, embeddingdim, hiddendim, outputdim):

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

self.embedding = nn.Embedding(vocabsize, embeddingdim)

self.lstm = nn.LSTM(embeddingdim, hiddendim, batch\_first=True)

self.fc = nn.Linear(hiddendim, outputdim)

def forward(self, x):

x = self.embedding(x)

\_, (hidden, \_) = self.lstm(x)

return self.fc(hidden.squeeze(0))

model = LSTMClassifier(1000, 50, 100, 2)

criterion = nn.CrossEntropyLoss()

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

for epoch in range(10):

for inputs, tgts in loader:

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, tgts)

loss.backward()

optimizer.step()

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