**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.")

**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(),

])

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)

**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 SimpleCNN(nn.Module):

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

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

self.conv\_block1 = nn.Sequential(

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

nn.ReLU(),

nn.MaxPool2d(2)

)

self.conv\_block2 = nn.Sequential(

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

nn.ReLU(),

nn.MaxPool2d(2)

)

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

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

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)

return x

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, bias=False),

nn.BatchNorm2d(32),

nn.ReLU(),

nn.MaxPool2d(2)

)

self.conv\_block2 = nn.Sequential(

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

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()

])

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}%')

model1 = SimpleCNN()

model2 = CNNWithBNDropout()

criterion = nn.CrossEntropyLoss()

optimizer1 = optim.Adam(model1.parameters(), lr=0.001)

optimizer2 = optim.Adam(model2.parameters(), lr=0.001)

train(model1, optimizer1, criterion, 20)

train(model2, optimizer2, criterion, 30)

**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

from torch.optim import Optimizer

transform = transforms.Compose([

transforms.ToTensor()

])

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

test\_dataset = datasets.MNIST(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)

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)

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\_()

class CustomAdagrad(Optimizer):

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\_()

def zero\_grad(self):

with torch.no\_grad():

for param in self.parameters:

if param.grad is not None:

param.grad.zero\_()

device = torch.device('cpu')

model = SimpleCNN().to(device)

criterion = nn.CrossEntropyLoss()

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)

optimizer.zero\_grad()

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

**5**

import torch

import torch.nn as nn

import torch.nn.functional as F

import torchvision.transforms as transforms

from torchvision.datasets import VOCSegmentation

from torch.utils.data import DataLoader, Subset

class TinyUNet(nn.Module):

def \_\_init\_\_(self, in\_channels=3, out\_channels=21):

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

def conv\_block(in\_channels, out\_channels):

return nn.Sequential(

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

nn.ReLU(),

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

nn.ReLU()

)

self.encoder1 = conv\_block(in\_channels, 16)

self.encoder2 = conv\_block(16, 32)

self.encoder3 = conv\_block(32, 64)

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

self.bottleneck = conv\_block(64, 128)

self.upconv3 = nn.ConvTranspose2d(128, 64, kernel\_size=2, stride=2)

self.decoder3 = conv\_block(128, 64)

self.upconv2 = nn.ConvTranspose2d(64, 32, kernel\_size=2, stride=2)

self.decoder2 = conv\_block(64, 32)

self.upconv1 = nn.ConvTranspose2d(32, 16, kernel\_size=2, stride=2)

self.decoder1 = conv\_block(32, 16)

self.conv\_final = nn.Conv2d(16, out\_channels, kernel\_size=1)

def forward(self, x):

enc1 = self.encoder1(x)

enc2 = self.encoder2(self.pool(enc1))

enc3 = self.encoder3(self.pool(enc2))

bottleneck = self.bottleneck(self.pool(enc3))

dec3 = self.upconv3(bottleneck)

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 self.conv\_final(dec1)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# Define transformations

transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor(),

])

# Load VOC Segmentation dataset

train\_dataset = VOCSegmentation(root='./data', year='2012', image\_set='train', download=True, transform=transform, target\_transform=transform)

test\_dataset = VOCSegmentation(root='./data', year='2012', image\_set='val', download=True, transform=transform, target\_transform=transform)

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

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

# Define DataLoader

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

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

model = TinyUNet().to(device)

criterion = nn.CrossEntropyLoss()

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

# 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

for data, target in train\_loader:

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

outputs = model(data).to(device)

loss = criterion(outputs, target.squeeze(1).long())

optimizer.zero\_grad()

loss.backward()

optimizer.step()

train\_loss += loss.item() \* data.size(0)

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

model.eval()

test\_loss = 0.0

with torch.no\_grad():

for data, target in test\_loader:

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

outputs = model(data).to(device)

loss = criterion(outputs, target.squeeze(1).long())

test\_loss += loss.item() \* data.size(0)

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

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

train(model, optimizer, criterion, 10)

**6**

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()

])

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

test\_dataset = datasets.MNIST(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)

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

model = AlexNet()

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 = 0.0

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\_model(5)

**7**

import torch

from torch import nn

from torch.utils.data import DataLoader, TensorDataset

data = torch.randint(0, 1000, (100, 10))

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

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())

def train(n\_epochs):

for epoch in range(n\_epochs):

model.train()

train\_loss = 0.0

for data, tgts in loader:

outputs = model(data)

loss = criterion(outputs, tgts)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

train\_loss += loss.item()

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

print(f"Epoch {epoch+1}, Loss: {avg\_train\_loss:.4f}")

train(10)

**8**

import torch

import torch.nn as nn

import numpy as np

import matplotlib.pyplot as plt

# Generate synthetic time series data

def generate\_data():

t = np.linspace(0, 20, 100)

y = np.sin(t) + np.random.normal(scale=0.5, size=t.shape)

return y

data = generate\_data()

plt.plot(data)

plt.title('Synthetic Time Series Data')

plt.show()

# Prepare the dataset

def create\_inout\_sequences(input\_data, tw): # tw-> time step window

inout\_seq = []

L = len(input\_data)

for i in range(L - tw):

train\_seq = input\_data[i:i + tw]

train\_label = input\_data[i + tw:i + tw + 1]

inout\_seq.append((train\_seq, train\_label))

return inout\_seq

seq\_length = 10 # Number of time steps to look back

data = torch.FloatTensor(data).view(-1)

sequences = create\_inout\_sequences(data, seq\_length)

class RNN(nn.Module):

def \_\_init\_\_(self, input\_size=1, hidden\_layer\_size=50, output\_size=1):

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

self.hidden\_layer\_size = hidden\_layer\_size

self.rnn = nn.RNN(input\_size, hidden\_layer\_size, num\_layers=1, batch\_first=True)

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

def forward(self, input\_seq):

rnn\_out, hidden = self.rnn(input\_seq.view(len(input\_seq), 1, -1))

predictions = self.linear(rnn\_out.view(len(input\_seq), -1))

return predictions[-1]

model = RNN()

loss\_function = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

epochs = 100

for i in range(epochs):

for seq, labels in sequences:

optimizer.zero\_grad()

y\_pred = model(seq)

single\_loss = loss\_function(y\_pred, labels)

single\_loss.backward()

optimizer.step()

if i % 10 == 0:

print(f'Epoch {i} loss: {single\_loss.item()}')

with torch.no\_grad():

preds = []

for seq, \_ in sequences:

preds.append(model(seq).item())

plt.plot(data.numpy(), label='Original Data')

plt.plot(np.arange(seq\_length, seq\_length + len(preds)), preds, label='Predicted')

plt.legend()

plt.show()

**9**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader, Subset

transform = transforms.Compose([

transforms.ToTensor()

])

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

test\_dataset = datasets.MNIST(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)

class AutoEncoder(nn.Module):

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

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

self.encoder = nn.Sequential(

nn.Linear(28 \* 28, 256),

nn.ReLU(inplace=True),

nn.Linear(256, 64),

)

self.decoder = nn.Sequential(

nn.Linear(64, 256),

nn.ReLU(inplace=True),

nn.Linear(256, 28 \* 28),

nn.Sigmoid()

)

def forward(self, x):

x = self.encoder(x)

x = self.decoder(x)

return x

model = AutoEncoder()

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

criterion = nn.MSELoss()

def train\_model(num\_epochs):

for epoch in range(num\_epochs):

model.train()

train\_loss = 0.0

for data in train\_loader:

img, \_ = data

img = img.view(img.size(0), -1)

output = model(img)

loss = criterion(output, img)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

train\_loss += loss.item()

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

model.eval()

test\_loss = 0.0

with torch.no\_grad():

for data in test\_loader:

img, \_ = data

img = img.view(img.size(0), -1)

output = model(img)

loss = criterion(output, img)

test\_loss += loss.item()

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

print(f'Epoch {epoch + 1}, Train Loss: {avg\_train\_loss:.4f}, Test Loss: {avg\_test\_loss:.4f}')

train\_model(10)

**10**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader, Subset

transform = transforms.Compose([

transforms.ToTensor()

])

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

subset = Subset(dataset, range(1000))

dataloader = DataLoader(subset, batch\_size=10, shuffle=True)

class Generator(nn.Module):

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

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

self.gen = nn.Sequential(

nn.Linear(100, 256),

nn.ReLU(),

nn.Linear(256, 28\*28),

nn.Tanh()

)

def forward(self, x):

return self.gen(x)

class Discriminator(nn.Module):

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

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

self.disc = nn.Sequential(

nn.Linear(28\*28, 256),

nn.ReLU(),

nn.Linear(256, 1),

nn.Sigmoid()

)

def forward(self, x):

return self.disc(x)

generator = Generator()

discriminator = Discriminator()

criterion = nn.BCELoss()

optim\_gen = optim.Adam(generator.parameters(), lr=2e-4)

optim\_disc = optim.Adam(discriminator.parameters(), lr=2e-4)

def train(num\_epochs):

for epoch in range(num\_epochs):

generator.train()

discriminator.train()

for real, \_ in dataloader:

real = real.view(-1, 28\*28)

batch\_size = real.size(0)

# Train Discriminator

noise = torch.randn(batch\_size, 100)

fake = generator(noise)

disc\_real = discriminator(real)

loss\_disc\_real = criterion(disc\_real, torch.ones\_like(disc\_real))

disc\_fake = discriminator(fake)

loss\_disc\_fake = criterion(disc\_fake, torch.zeros\_like(disc\_fake))

loss\_disc = (loss\_disc\_real + loss\_disc\_fake) / 2

# Backprop

optim\_disc.zero\_grad()

loss\_disc.backward()

optim\_disc.step()

# Train Generator

noise = torch.randn(batch\_size, 100)

fake = generator(noise)

disc\_fake = discriminator(fake)

loss\_gen = criterion(disc\_fake, torch.ones\_like(disc\_fake))

# Backprop

optim\_gen.zero\_grad()

loss\_gen.backward()

optim\_gen.step()

print(f'Epoch {epoch+1}, Loss D: {loss\_disc.item():.4f}, Loss G: {loss\_gen.item():.4f}')

train(15)

**11**

import torch

import torch.nn as nn

import torch.nn.functional as F

class SelfAttention(nn.Module):

def \_\_init\_\_(self, embed\_dim, num\_heads):

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

self.embed\_dim = embed\_dim

self.num\_heads = num\_heads

assert embed\_dim % num\_heads == 0, "Embedding dimension must be divisible by number of heads"

self.head\_dim = embed\_dim // num\_heads

self.scale = self.head\_dim \*\* -0.5

self.query = nn.Linear(embed\_dim, embed\_dim)

self.key = nn.Linear(embed\_dim, embed\_dim)

self.value = nn.Linear(embed\_dim, embed\_dim)

self.out = nn.Linear(embed\_dim, embed\_dim)

def forward(self, x):

batch\_size, seq\_len, embed\_dim = x.size()

# Compute Q, K, V matrices

Q = self.query(x) # (batch\_size, seq\_len, embed\_dim)

K = self.key(x) # (batch\_size, seq\_len, embed\_dim)

V = self.value(x) # (batch\_size, seq\_len, embed\_dim)

# Split the embedding into multiple heads

Q = Q.view(batch\_size, seq\_len, self.num\_heads, self.head\_dim).transpose(1, 2)

K = K.view(batch\_size, seq\_len, self.num\_heads, self.head\_dim).transpose(1, 2)

V = V.view(batch\_size, seq\_len, self.num\_heads, self.head\_dim).transpose(1, 2)

# Compute attention scores

attn\_scores = torch.matmul(Q, K.transpose(-2,-1)) \* self.scale

attn\_weights = F.softmax(attn\_scores, dim=-1)

# Compute the weighted sum of the values

attn\_output = torch.matmul(attn\_weights, V)

# Concatenate the multiple heads

attn\_output = attn\_output.transpose(1, 2).contiguous().view(batch\_size, seq\_len, embed\_dim)

# Apply the final linear layer

output = self.out(attn\_output)

return output

embed\_dim = 128

num\_heads = 8

seq\_len = 10

batch\_size = 32

x = torch.randn(batch\_size, seq\_len, embed\_dim)

self\_attention = SelfAttention(embed\_dim, num\_heads)

output = self\_attention(x)

print(output.shape) # Output shape will be (batch\_size, seq\_len, embed\_dim)

**12**

import numpy as np

def sigmoid(x):

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

def sigmoid\_derivative(x):

return x \* (1 - x)

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

expected\_output = np.array([[0], [1], [1], [0]])

# Initialize weights for the two layers

weights1 = np.random.rand(2, 4)

weights2 = np.random.rand(4, 1)

bias1 = np.random.rand(1, 4)

bias2 = np.random.rand(1, 1)

learning\_rate = 0.1

for epoch in range(10000):

# First layer

hidden\_layer\_input = np.dot(inputs, weights1) + bias1

hidden\_layer\_output = sigmoid(hidden\_layer\_input)

# Second layer

final\_output = sigmoid(np.dot(hidden\_layer\_output, weights2) + bias2)

# Backpropagation

error = expected\_output - final\_output

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

error\_hidden\_layer = d\_predicted\_output.dot(weights2.T)

d\_hidden\_layer = error\_hidden\_layer \* sigmoid\_derivative(hidden\_layer\_output)

weights2 += hidden\_layer\_output.T.dot(d\_predicted\_output) \* learning\_rate

bias2 += np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learning\_rate

weights1 += inputs.T.dot(d\_hidden\_layer) \* learning\_rate

bias1 += np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* learning\_rate

if epoch % 1000 == 0:

print(f'Epoch {epoch} Loss: {np.mean(np.abs(error))}')

print("Final outputs after training:")

print(final\_output)