**Pytorch Exercises**

**#build a simple neural network**

import torch

import torch.nn as nn

import torch.nn.functional as F

class SimpleNN(nn.Module):

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

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

self.fc1 = nn.Linear(784,128)

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

self.fc3 = nn.Linear(10,1)

def forward(self,x):

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

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

x = F.log\_softmax(self.fc3(x), dim =1)

return x

model = SimpleNN()

print(model)

**#logisitic sigmoid**

import torch

import matplotlib.pyplot as plt

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

y = torch.sigmoid(x)

plt.plot(x.numpy(), y.numpy(), color = 'purple')

plt.xlabel('input')

plt.ylabel('output')

plt.title("Logisitic Activation Function")

plt.show()

**#TAnh activation function**

y = torch.tanh(x)

plt.plot(x.numpy(), y.numpy(), color = 'purple')

plt.xlabel('input')

plt.ylabel('output')

plt.title("Tanh Activation Function")

plt.show()

**#RELU fucntion**

y = torch.relu(x)

plt.plot(x.numpy(), y.numpy(), color = 'purple')

plt.xlabel('input')

plt.ylabel('output')

plt.title("RELU Activation Function")

plt.show()

**PYTORCH - 2(ADVANCED)**

!pip install torch torchvision

import torch.nn as nn

import torch

class SimpleNN(nn.Module):

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

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

self.fc1 = nn.Linear(784,128)

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

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

def forward(self,x):

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

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

x = torch.softmax(self.fc3(x), dim = 1)

return x

model = SimpleNN()

print(model)

**#TRAIN THE MODEL**

import torch.optim as optim

import torch.nn.functional as F

from torchvision import datasets, transforms

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

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

trainloader = torch.utils.data.DataLoader(trainset, batch\_size = 32, shuffle = True)

**#define the loss function and optimizer**

criterion = nn.CrossEntropyLoss()

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

**#training loop**

for epoch in range(1,6):

running\_loss = 0.0

for images, labels in trainloader:

#flatten images into vectors

images = images.view(images.shape[0], -1)

#zero the parameter gradients

optimizer.zero\_grad()

#forward pass

output = model(images)

#compute loss

loss = criterion(output, labels)

#backward pass and optimisation

loss.backward()

optimizer.step()

#update running loss

running\_loss += loss.item()

print(f'Epoch {epoch}, Loss:{running\_loss/len(trainloader)}')

**##EVALUATE THE MODEL**

import torch # Ensure torch is imported

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

testloader = torch.utils.data.DataLoader(testset, batch\_size = 32, shuffle = True)

correct = 0

total = 0

# Use torch.no\_grad() as a function call instead of a context manager

for images, labels in testloader:

with torch.no\_grad(): # Apply torch.no\_grad() to each iteration

images = images.view(images.shape[0], -1)

outputs = model(images)

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

total += labels.size(0)

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

print(f'Accuracy: {100\* correct/total}%')

**#FUNCTION TO DISPLAY BATCH OF IMAGES ALONG WITH PREDICTIONS**

import matplotlib.pyplot as plt

import numpy as np

# Function to display a batch of test images along with predictions

def visualize\_predictions(model, testloader):

# Get a batch of test data

dataiter = iter(testloader)

images, labels = next(dataiter)

# Flatten images for model input

images\_flattened = images.view(images.shape[0], -1)

# Predict using the model

with torch.no\_grad():

outputs = model(images\_flattened)

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

# Convert images to numpy for display

images = images.numpy()

# Display the first 10 images with predictions and actual labels

fig, axes = plt.subplots(1, 10, figsize=(15, 4))

for idx in range(10):

ax = axes[idx]

ax.imshow(images[idx].squeeze(), cmap='gray')

ax.set\_title(f"P: {predicted[idx].item()}\nT: {labels[idx].item()}")

ax.axis('off')

plt.tight\_layout()

plt.show()

# Call the visualization function

visualize\_predictions(model, testloader)