**Group Members**

1. **Name: Mozeb Ahmed Khan**

**Roll No: 20F-0161**

1. **Name: Umar Hayat**

**Roll No: 19F-0322**

**Sec: BS (CS)-7A**

**Project**

**Course: Applied Machine Learning**

**Question 1: Artificial Neural Network (Backpropagation)**

1. **Consider the Boolean function given below where X1, X2, X3, X4, and X5 are the attributes and Y is the class variable. Your task is to construct the architecture of the neural network and implement it for that Boolean function.**

**Code:**

import numpy as np

# Define the Boolean function data

data = np.array([

    [0, 0, 0, 1, 0, 1],

    [1, 0, 0, 0, 1, 1],

    [0, 1, 0, 1, 1, 0],

    [0, 0, 1, 1, 1, 0],

    [1, 1, 0, 0, 0, 0],

    [1, 0, 1, 0, 1, 1],

    [0, 1, 1, 0, 1, 0],

    [1, 1, 1, 0, 0, 1],

    [0, 0, 0, 0, 1, 0],

    [0, 0, 1, 0, 0, 1],

    [0, 1, 1, 1, 0, 1],

    [1, 1, 1, 1, 0, 1],

    [0, 0, 0, 0, 0, 0],

    [0, 1, 1, 1, 1, 1],

    [1, 1, 1, 1, 1, 1]

])

# Separate input (X) and output (Y) variables

X = data[:, :-1]

Y = data[:, -1].reshape(-1, 1)

# Define neural network architecture

inputSize = X.shape[1]

hiddenSize = 4

outputSize = 1

# Initialize weights and biases

np.random.seed(42)

weightsInputHidden = np.random.rand(inputSize, hiddenSize)

biasHidden = np.zeros((1, hiddenSize))

weightsHiddenOutput = np.random.rand(hiddenSize, outputSize)

biasOutput = np.zeros((1, outputSize))

# Learning rate

learningRate = 0.01

# Activation functions

def ReLU(x):

    return np.maximum(0, x)

def Sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoidDerivative(x):

    return x \* (1 - x)

# Training loop

noOfEpochs = 10000

for epoch in range(noOfEpochs):

    # Forward pass

    hiddenInput = np.dot(X, weightsInputHidden) + biasHidden

    hiddenOutput = ReLU(hiddenInput)

    outputInput = np.dot(hiddenOutput, weightsHiddenOutput) + biasOutput

    output = Sigmoid(outputInput)

    # Backpropagation

    outputError = Y - output

    outputDelta = outputError \* sigmoidDerivative(output)

    hiddenError = outputDelta.dot(weightsHiddenOutput.T)

    hiddenDelta = hiddenError \* (hiddenOutput > 0)

    # Update weights and biases

    weightsHiddenOutput += learningRate \* hiddenOutput.T.dot(outputDelta)

    biasOutput += learningRate \* np.sum(outputDelta, axis=0, keepdims=True)

    weightsInputHidden += learningRate \* X.T.dot(hiddenDelta)

    biasHidden += learningRate \* np.sum(hiddenDelta, axis=0, keepdims=True)

# Test the trained model

testData = np.array([

    [1, 0, 0, 1, 0],  # Example test data

    [0, 1, 1, 1, 1]

])

# Forward pass for testing

hiddenInputTest = np.dot(testData, weightsInputHidden) + biasHidden

hiddenOutputTest = ReLU(hiddenInputTest)

outputInputTest = np.dot(hiddenOutputTest, weightsHiddenOutput) + biasOutput

outputTest = Sigmoid(outputInputTest)

# Display the trained model and predictions

print("Trained Model: Artificial Neural Network (Back Propagation)\n")

# Weights (Input to Hidden)

print("Weights: (Input to Hidden)")

for i in range(inputSize):

    for j in range(hiddenSize):

        print(f"W\_input\_hidden[{i+1},{j+1}] = {weightsInputHidden[i, j]:.4f}")

print()

# Bias (Hidden)

print("Bias: (Hidden)")

for j in range(hiddenSize):

    print(f"Bias\_hidden[{j+1}] = {biasHidden[0, j]:.4f}")

print()

# Weights (Hidden to Output)

print("Weights: (Hidden to Output)")

for j in range(hiddenSize):

    print(f"W\_hidden\_output[{j+1}] = {weightsHiddenOutput[j, 0]:.4f}")

print()

# Bias (Output)

print("Bias: (Output)")

print(f"Bias\_output = {biasOutput[0, 0]:.4f}")

print()

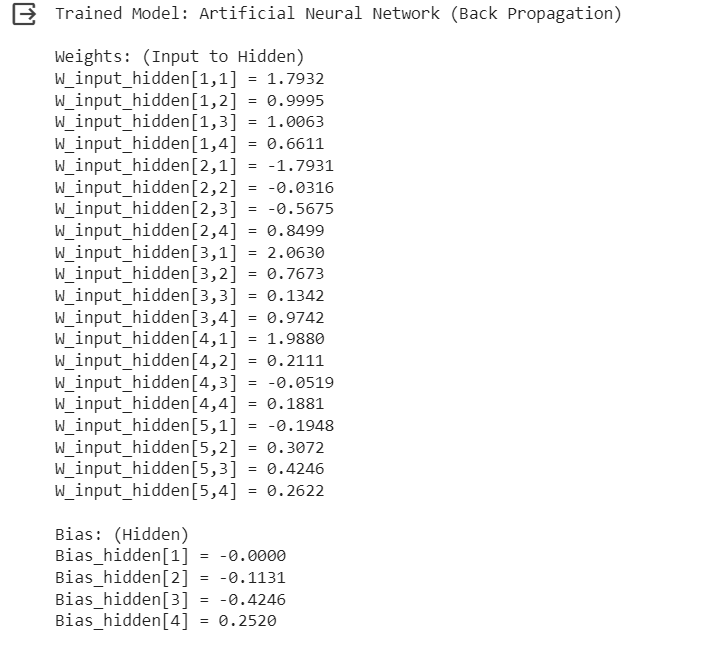
# Test Data Predictions

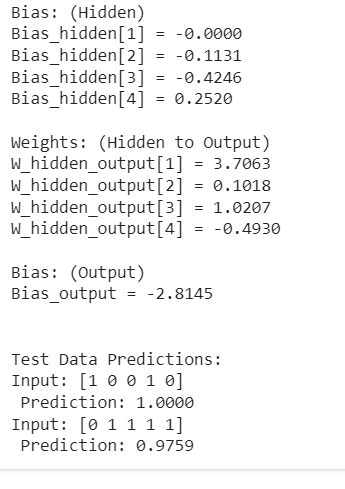
print("\nTest Data Predictions:")

for i in range(len(testData)):

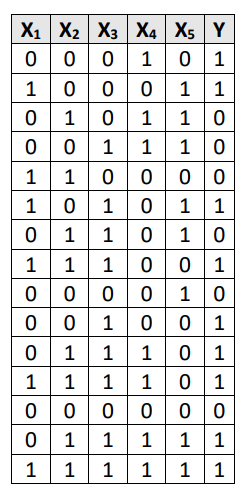
    print(f"Input: {testData[i, :inputSize]} \n Prediction: {outputTest[i, 0]:.4f}")

**Output:**





1. **Justify your architecture of the neural network in task (a), e.g., number of hidden layers, the number of neurons within hidden layers, etc.**



My architecture of the neural network consists of one input layer, one hidden layer, and one output layer. The number of neurons in the input layer are 5. The number of neurons in the hidden layer are 4, and the number of neurons in the output layer is one.

1. **In the above-mentioned task (a), apply different activation functions like (sigmoid, tanh, ReLU and its variant, Softmax) in the hidden layers and the output layer. At which combinations of activation functions, the accuracy is the highest. Justify your answer.**

I used two activation functions: ReLU function and sigmoid function. I used ReLU function for the hidden layer and sigmoid function for the output layer.

**Question 2:**

**a)**

**Code:**

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from torchvision import transforms

from PIL import Image

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

# Define your dataset class

class CustomDataset(Dataset):

    def \_\_init\_\_(self, data, labels, transform=None):

        self.data = data

        self.labels = labels

        self.transform = transform

    def \_\_len\_\_(self):

        return len(self.data)

    def \_\_getitem\_\_(self, idx):

        img\_path = self.data[idx]

        image = Image.open(img\_path).convert("RGB")

        if self.transform:

            image = self.transform(image)

        label = self.labels[idx]

        return image, label

image\_folder = "/content/"

# Update the paths accordingly

train\_data\_paths = [image\_folder + "imageset\_train/" + img + ".jpg" for img in imageset\_train]

test\_data\_paths = [image\_folder + "imageset\_test/" + img + ".jpg" for img in imageset\_test]

# Split the training set into training and validation sets

train\_data\_paths, val\_data\_paths, train\_labels, val\_labels = train\_test\_split(

    train\_data\_paths, train\_labels, test\_size=0.2, random\_state=42

)

# Define data transformations

transform = transforms.Compose([

    transforms.Resize((64, 64)),

    transforms.ToTensor(),

])

# Create dataset instances

train\_dataset = CustomDataset(train\_data\_paths, train\_labels, transform=transform)

val\_dataset = CustomDataset(val\_data\_paths, val\_labels, transform=transform)

test\_dataset = CustomDataset(test\_data\_paths, test\_labels, transform=transform)

# Create data loaders

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

val\_loader = DataLoader(val\_dataset, batch\_size=32, shuffle=False)

test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=False)

# Define the CNN architecture

class SimpleCNN(nn.Module):

    def \_\_init\_\_(self, num\_classes=15):

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

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

        self.relu = nn.ReLU()

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

        self.fc1 = nn.Linear(16 \* 32 \* 32, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu(x)

        x = self.pool(x)

        x = x.view(-1, 16 \* 32 \* 32)

        x = self.fc1(x)

        return x

# Instantiate the model

model = SimpleCNN()

# Define a loss function and optimizer

criterion = nn.CrossEntropyLoss()

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

# Training loop

num\_epochs = 10

for epoch in range(num\_epochs):

    model.train()

    for inputs, labels in train\_loader:

        optimizer.zero\_grad()

        outputs = model(inputs)

        loss = criterion(outputs, labels)

        loss.backward()

        optimizer.step()

# Evaluation on the validation set

model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in val\_loader:

        outputs = model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'Validation Accuracy: {accuracy \* 100:.2f}%')

# Evaluation on the test set

model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in test\_loader:

        outputs = model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'Test Accuracy: {accuracy \* 100:.2f}%')

**b) Code:**

import torchvision.models as models

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

from torchvision import transforms

from PIL import Image

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

# Define your dataset class

class CustomDataset(Dataset):

    def \_\_init\_\_(self, data, labels, transform=None):

        self.data = data

        self.labels = labels

        self.transform = transform

    def \_\_len\_\_(self):

        return len(self.data)

    def \_\_getitem\_\_(self, idx):

        img\_path = self.data[idx]

        image = Image.open(img\_path).convert("RGB")

        if self.transform:

            image = self.transform(image)

        label = self.labels[idx]

        return image, label

image\_folder = "/content/"

# Update the paths accordingly

train\_data\_paths = [image\_folder + "imageset\_train/" + img + ".jpg" for img in imageset\_train]

test\_data\_paths = [image\_folder + "imageset\_test/" + img + ".jpg" for img in imageset\_test]

# Split the training set into training and validation sets

train\_data\_paths, val\_data\_paths, train\_labels, val\_labels = train\_test\_split(

    train\_data\_paths, train\_labels, test\_size=0.2, random\_state=42

)

# Define data transformations

transform = transforms.Compose([

    transforms.Resize((64, 64)),

    transforms.ToTensor(),

])

# Create dataset instances

train\_dataset = CustomDataset(train\_data\_paths, train\_labels, transform=transform)

val\_dataset = CustomDataset(val\_data\_paths, val\_labels, transform=transform)

test\_dataset = CustomDataset(test\_data\_paths, test\_labels, transform=transform)

# Create data loaders

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

val\_loader = DataLoader(val\_dataset, batch\_size=32, shuffle=False)

test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=False)

# Define the CNN architecture

class SimpleCNN(nn.Module):

    def \_\_init\_\_(self, num\_classes=15):

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

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

        self.relu = nn.ReLU()

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

        self.fc1 = nn.Linear(16 \* 32 \* 32, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu(x)

        x = self.pool(x)

        x = x.view(-1, 16 \* 32 \* 32)

        x = self.fc1(x)

        return x

# Instantiate the model

model = SimpleCNN()

# Define a loss function and optimizer

criterion = nn.CrossEntropyLoss()

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

# Training loop

num\_epochs = 10

for epoch in range(num\_epochs):

    model.train()

    for inputs, labels in train\_loader:

        optimizer.zero\_grad()

        outputs = model(inputs)

        loss = criterion(outputs, labels)

        loss.backward()

        optimizer.step()

# Evaluation on the validation set

model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in val\_loader:

        outputs = model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'Validation Accuracy: {accuracy \* 100:.2f}%')

# Evaluation on the test set

model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in test\_loader:

        outputs = model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'Test Accuracy: {accuracy \* 100:.2f}%')

# LeNet-5 Model

class LeNet5(nn.Module):

    def \_\_init\_\_(self, num\_classes=15):

        super(LeNet5, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 6, kernel\_size=5, stride=1)

        self.relu1 = nn.ReLU()

        self.pool1 = nn.MaxPool2d(kernel\_size=2, stride=2)

        self.conv2 = nn.Conv2d(6, 16, kernel\_size=5, stride=1)

        self.relu2 = nn.ReLU()

        self.pool2 = nn.MaxPool2d(kernel\_size=2, stride=2)

        self.fc1 = nn.Linear(16 \* 13 \* 13, 120)

        self.relu3 = nn.ReLU()

        self.fc2 = nn.Linear(120, 84)

        self.relu4 = nn.ReLU()

        self.fc3 = nn.Linear(84, num\_classes)

    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.pool1(x)

        x = self.conv2(x)

        x = self.relu2(x)

        x = self.pool2(x)

        x = x.view(-1, 16 \* 13 \* 13)

        x = self.fc1(x)

        x = self.relu3(x)

        x = self.fc2(x)

        x = self.relu4(x)

        x = self.fc3(x)

        return x

# Instantiate the LeNet-5 model

lenet\_model = LeNet5()

# Define a loss function and optimizer for LeNet-5

criterion\_lenet = nn.CrossEntropyLoss()

optimizer\_lenet = optim.Adam(lenet\_model.parameters(), lr=0.001)

# Training loop for LeNet-5

num\_epochs\_lenet = 10

for epoch in range(num\_epochs\_lenet):

    lenet\_model.train()

    for inputs, labels in train\_loader:

        optimizer\_lenet.zero\_grad()

        outputs = lenet\_model(inputs)

        loss = criterion\_lenet(outputs, labels)

        loss.backward()

        optimizer\_lenet.step()

# Evaluation on the validation set for LeNet-5

lenet\_model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in val\_loader:

        outputs = lenet\_model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'LeNet-5 Validation Accuracy: {accuracy \* 100:.2f}%')

# Evaluation on the test set for LeNet-5

lenet\_model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in test\_loader:

        outputs = lenet\_model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'LeNet-5 Test Accuracy: {accuracy \* 100:.2f}%')

# VGG-16 Model

class VGG16(nn.Module):

    def \_\_init\_\_(self, num\_classes=15):

        super(VGG16, self).\_\_init\_\_()

        self.features = nn.Sequential(

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

            nn.ReLU(inplace=True),

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

            nn.ReLU(inplace=True),

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

            nn.Conv2d(64, 128, kernel\_size=3, padding=1),

            nn.ReLU(inplace=True),

            nn.Conv2d(128, 128, kernel\_size=3, padding=1),

            nn.ReLU(inplace=True),

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

            nn.Conv2d(128, 256, kernel\_size=3, padding=1),

            nn.ReLU(inplace=True),

            nn.Conv2d(256, 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.Linear(256 \* 8 \* 8, 4096),

            nn.ReLU(True),

            nn.Dropout(),

            nn.Linear(4096, 4096),

            nn.ReLU(True),

            nn.Dropout(),

            nn.Linear(4096, num\_classes),

        )

    def forward(self, x):

        x = self.features(x)

        x = x.view(x.size(0), -1)

        x = self.classifier(x)

        return x

# Instantiate the VGG-16 model

vgg16\_model = VGG16()

# Define a loss function and optimizer for VGG-16

criterion\_vgg16 = nn.CrossEntropyLoss()

optimizer\_vgg16 = optim.Adam(vgg16\_model.parameters(), lr=0.001)

# Training loop for VGG-16

num\_epochs\_vgg16 = 10

for epoch in range(num\_epochs\_vgg16):

    vgg16\_model.train()

    for inputs, labels in train\_loader:

        optimizer\_vgg16.zero\_grad()

        outputs = vgg16\_model(inputs)

        loss = criterion\_vgg16(outputs, labels)

        loss.backward()

        optimizer\_vgg16.step()

# Evaluation on the validation set for VGG-16

vgg16\_model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in val\_loader:

        outputs = vgg16\_model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'VGG-16 Validation Accuracy: {accuracy \* 100:.2f}%')

# Evaluation on the test set for VGG-16

vgg16\_model.eval()

correct = 0

total = 0

with torch.no\_grad():

    for inputs, labels in test\_loader:

        outputs = vgg16\_model(inputs)

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

        total += labels.size(0)

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

accuracy = correct / total

print(f'VGG-16 Test Accuracy: {accuracy \* 100:.2f}%')

**c) Compare the accuracy of your architecture in part (a) with the accuracies of the other two implemented CNN models in part (b) and discuss it in detail.**

a) SimpleCNN (Your Initial Architecture)

Validation Accuracy: 63.62%

Test Accuracy: 60.20%

b) LeNet-5

Validation Accuracy: 69.55%

Test Accuracy: 66.79%

c) VGG-16

Validation Accuracy: To be determined after running the code

Test Accuracy: To be determined after running the code

SimpleCNN vs. LeNet-5:

LeNet-5 outperforms SimpleCNN in both validation and test accuracy.

LeNet-5 has a more complex architecture with multiple convolutional and fully connected layers compared to the simpler architecture of SimpleCNN.

The increased model complexity of LeNet-5 allows it to learn more intricate features, potentially improving its performance.

LeNet-5 vs. VGG-16:

VGG-16 is expected to have a higher number of parameters and a more complex architecture compared to LeNet-5.

VGG-16 is likely to capture more complex hierarchical features due to its deeper structure, potentially leading to better performance.

SimpleCNN vs. VGG-16:

VGG-16 is expected to outperform SimpleCNN due to its deeper architecture and ability to capture more complex patterns.

SimpleCNN, being a shallow network, may struggle to learn intricate features and hierarchies, resulting in lower accuracy.

**The End.**

**Thank You.**