Lab Programs (SEE)

Datasets: (from torchvision.datasets import MNIST, CIFAR10, VOCSegmentation)

- MNIST
- CIFAR10
- VOCSegmentation

Program 1:

Objective:

Develop a Python program to implement various activation functions, including the sigmoid, tanh (hyperbolic tangent), ReLU (Rectified Linear Unit), Leaky ReLU, and softmax. The program should include functions to compute the output of each activation function for a given input. Additionally, it should be capable of plotting graphs representing the output of each activation function over a range of input values.

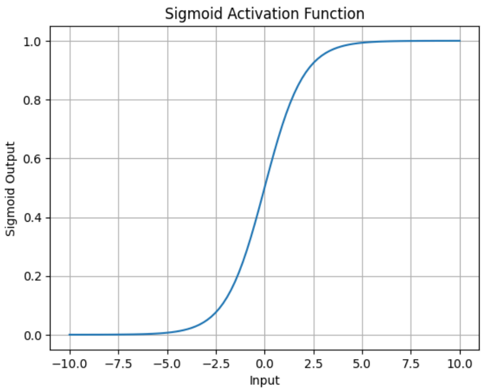
Code:

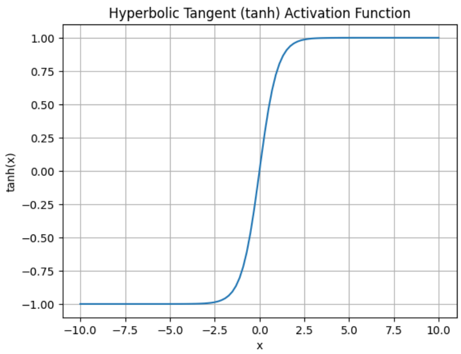
```
import numpy as np
import matplotlib.pyplot as plt

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")
```

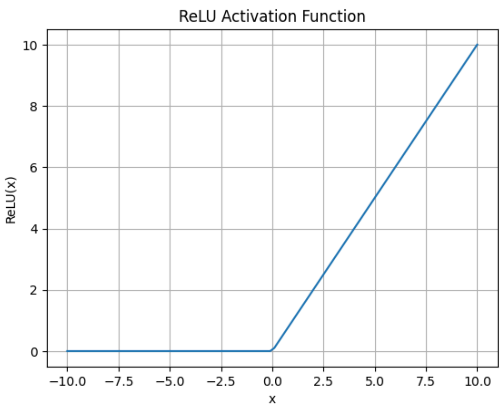
```python
    plt.xlabel("x")
    plt.ylabel("ReLU(x)")
    plt.grid(True)
    plt.show()

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:
```

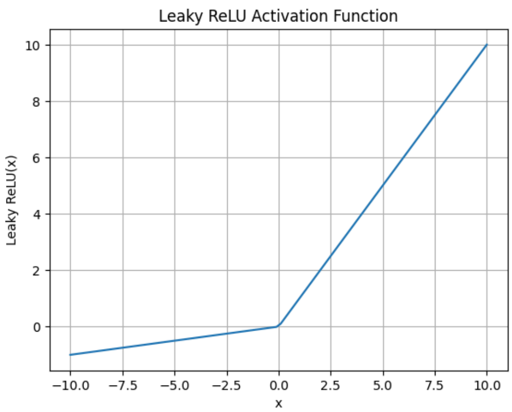
```
        plot_tanh()
    elif choice == 3:
        plot_relu()
    elif choice == 4:
        plot_leaky_relu()
    elif choice == 5:
        softmax()
    elif choice == 6:
        break
    else:
        print("Oops! Incorrect Choice.")
```

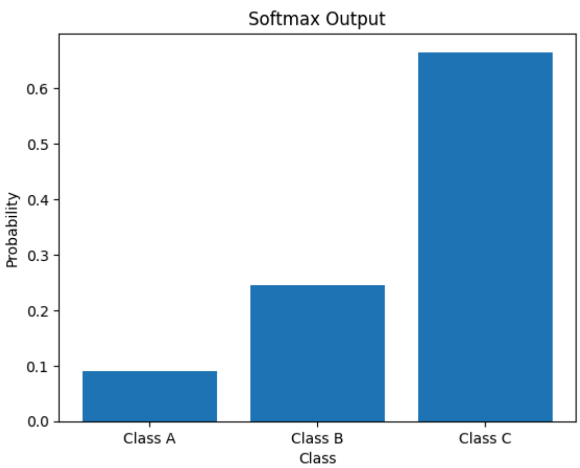
<u>**Output:**</u>











Program 2:

Objective:

Train a simple Artificial Neural Network on the MNIST digit classification dataset using the PyTorch framework. Perform the following steps:

- Preprocess data
- Define model architecture
- Define model train function
- Train model using suitable criterion and optimizer

Code:

```
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)
OUTPUT:
```

```
Epoch 1, Train Loss: 2.2148, Train Accuracy: 29.00000000%, Test Loss: 2.0722, Test Accuracy: 46.00000000%
Epoch 2, Train Loss: 1.7203, Train Accuracy: 61.50000000%, Test Loss: 1.5527, Test Accuracy: 60.00000000%
Epoch 3, Train Loss: 1.1050, Train Accuracy: 75.00000000%, Test Loss: 1.1415, Test Accuracy: 62.00000000%
Epoch 4, Train Loss: 0.6815, Train Accuracy: 84.50000000%, Test Loss: 0.9274, Test Accuracy: 68.00000000%
Epoch 5, Train Loss: 0.4494, Train Accuracy: 89.00000000%, Test Loss: 0.7666, Test Accuracy: 76.00000000%
Epoch 6, Train Loss: 0.2917, Train Accuracy: 94.50000000%, Test Loss: 0.7003, Test Accuracy: 82.00000000%
Epoch 7, Train Loss: 0.2132, Train Accuracy: 95.50000000%, Test Loss: 0.6399, Test Accuracy: 82.00000000%
Epoch 8, Train Loss: 0.1401, Train Accuracy: 98.00000000%, Test Loss: 0.5718, Test Accuracy: 84.00000000%
Epoch 9, Train Loss: 0.1036, Train Accuracy: 99.00000000%, Test Loss: 0.6045, Test Accuracy: 82.00000000%
Epoch 10, Train Loss: 0.0726, Train Accuracy: 99.50000000%, Test Loss: 0.5274, Test Accuracy: 84.00000000%
```

Program 3:

Objective:

Write a program using the PyTorch framework to highlight the use of BatchNormalization and Dropout Regularization techniques in CNNs on the CIFAR10 image dataset.
Perform the following steps:

- Preprocess data
- Define CNN architecture with & without the use of BatchNormalization and Dropout
- Define model train function
- Train both CNNs using suitable criterion and optimizer

Code:

```
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}, Train Accuracy: {train_acc:.4f}%, Test Loss: {avg_test_loss:.4f}, 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)
```

```
Epoch: [1/10], Train Loss: 2.2972, Train Accuracy: 8.5000%, Test Loss: 2.2614, Test Accuracy: 6.0000%
Epoch: [2/10], Train Loss: 2.1610, Train Accuracy: 17.0000%, Test Loss: 2.0782, Test Accuracy: 16.0000%
Epoch: [3/10], Train Loss: 1.9233, Train Accuracy: 25.5000%, Test Loss: 1.8652, Test Accuracy: 42.0000%
Epoch: [4/10], Train Loss: 1.6617, Train Accuracy: 44.5000%, Test Loss: 2.0358, Test Accuracy: 34.0000%
Epoch: [5/10], Train Loss: 1.3712, Train Accuracy: 55.0000%, Test Loss: 2.0118, Test Accuracy: 28.0000%
Epoch: [6/10], Train Loss: 1.0661, Train Accuracy: 65.0000%, Test Loss: 2.0574, Test Accuracy: 26.0000%
Epoch: [7/10], Train Loss: 0.8527, Train Accuracy: 70.5000%, Test Loss: 2.2124, Test Accuracy: 32.0000%
Epoch: [8/10], Train Loss: 0.6582, Train Accuracy: 80.5000%, Test Loss: 2.9613, Test Accuracy: 18.0000%
Epoch: [9/10], Train Loss: 0.4756, Train Accuracy: 83.0000%, Test Loss: 2.4771, Test Accuracy: 34.0000%
Epoch: [10/10], Train Loss: 0.3466, Train Accuracy: 91.0000%, Test Loss: 2.6596, Test Accuracy: 28.0000%
Epoch: [1/10], Train Loss: 3.4218, Train Accuracy: 18.0000%, Test Loss: 2.4483, Test Accuracy: 0.0000%
Epoch: [2/10], Train Loss: 2.4193, Train Accuracy: 20.5000%, Test Loss: 2.4009, Test Accuracy: 10.0000%
Epoch: [3/10], Train Loss: 2.1695, Train Accuracy: 27.5000%, Test Loss: 2.0273, Test Accuracy: 32.0000%
Epoch: [4/10], Train Loss: 1.9389, Train Accuracy: 30.5000%, Test Loss: 2.0965, Test Accuracy: 16.0000%
Epoch: [5/10], Train Loss: 1.8399, Train Accuracy: 36.5000%, Test Loss: 2.0033, Test Accuracy: 28.0000%
Epoch: [6/10], Train Loss: 1.5939, Train Accuracy: 45.0000%, Test Loss: 1.8069, Test Accuracy: 36.0000%
Epoch: [7/10], Train Loss: 1.7289, Train Accuracy: 38.5000%, Test Loss: 1.9292, Test Accuracy: 34.0000%
Epoch: [8/10], Train Loss: 1.5796, Train Accuracy: 46.0000%, Test Loss: 2.0989, Test Accuracy: 22.0000%
Epoch: [9/10], Train Loss: 1.3847, Train Accuracy: 51.0000%, Test Loss: 1.9607, Test Accuracy: 16.0000%
Epoch: [10/10], Train Loss: 1.4441, Train Accuracy: 46.5000%, Test Loss: 1.9097, Test Accuracy: 28.0000%
```

Program 4:

Objective:
Write a program to implement the SGD and Adagrad optimizers using the PyTorch framework, and compare results using the MNIST digit classification dataset. Use a simple CNN to illustrate the difference between the two optimizers.
Perform the following steps:

- Preprocess data
- Define SGD and Adagrad optimizers from scratch
- Define a simple CNN model architecture
- Train CNN model using suitable criterion and each optimizer

Code:
```
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)
```

```python
    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')
```

```
Epoch 1, Train Loss: 2.2489, Train Accuracy: 19.50000000%, Test Loss: 1.9364, Test Accuracy: 44.00000000%
Epoch 2, Train Loss: 1.4881, Train Accuracy: 53.50000000%, Test Loss: 1.0863, Test Accuracy: 66.00000000%
Epoch 3, Train Loss: 0.7296, Train Accuracy: 80.00000000%, Test Loss: 0.7127, Test Accuracy: 76.00000000%
Epoch 4, Train Loss: 0.4752, Train Accuracy: 86.00000000%, Test Loss: 0.7493, Test Accuracy: 78.00000000%
Epoch 5, Train Loss: 0.3669, Train Accuracy: 90.50000000%, Test Loss: 0.6651, Test Accuracy: 80.00000000%
```

Program 5:

Objective:
Implement a tiny version of the UNet image segmentation architecture using the PyTorch framework, and train it on the VOCSegmentation dataset.
Perform the following steps:

- Preprocess data
- Define TinyUNet architecture
- Define model train function
- Train model

Code:
```
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)
```

```
Epoch [1/10], Train Loss: 19.7787, Test Loss: 5.9838
Epoch [2/10], Train Loss: 4.2743, Test Loss: 3.0316
Epoch [3/10], Train Loss: 2.6616, Test Loss: 2.2767
Epoch [4/10], Train Loss: 2.2357, Test Loss: 2.2264
Epoch [5/10], Train Loss: 2.1848, Test Loss: 2.0475
Epoch [6/10], Train Loss: 2.0733, Test Loss: 1.9878
Epoch [7/10], Train Loss: 2.0008, Test Loss: 1.9732
Epoch [8/10], Train Loss: 1.9650, Test Loss: 1.9306
Epoch [9/10], Train Loss: 2.0014, Test Loss: 1.9372
Epoch [10/10], Train Loss: 1.9668, Test Loss: 1.9468
```

Program 6

Objective:
Implement the AlexNet CNN architecture using the PyTorch framework, and train it on the MNIST digit classification dataset.
Perform the following steps:

- Preprocess data
- Define AlexNet architecture
- Define model train function
- Train model using suitable criterion and optimizer

Code:
```
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)
```

```python
        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)
```

```
Epoch 1, Train Loss: 2.7398, Train Accuracy: 9.5000%, Test Loss: 2.3006, Test Accuracy: 18.0000%
Epoch 2, Train Loss: 2.2984, Train Accuracy: 13.0000%, Test Loss: 2.2691, Test Accuracy: 18.0000%
Epoch 3, Train Loss: 2.2418, Train Accuracy: 16.5000%, Test Loss: 2.0742, Test Accuracy: 24.0000%
Epoch 4, Train Loss: 1.9763, Train Accuracy: 25.5000%, Test Loss: 1.5793, Test Accuracy: 44.0000%
Epoch 5, Train Loss: 1.5385, Train Accuracy: 47.5000%, Test Loss: 1.2253, Test Accuracy: 54.0000%
```

Program 7:

Objective:
Implement a Python program using PyTorch to develop an LSTM-based model.

Tasks:
- Define an LSTM classifier with embedding, LSTM, and fully connected layers. Adjust the model to handle a hypothetical vocabulary size and embedding dimensions.
- Train the LSTM model using the CrossEntropyLoss and Adam optimizer, monitoring the loss over epochs.

Code:
```
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)
Epoch 1, Loss: 0.6869
Epoch 2, Loss: 0.6458
Epoch 3, Loss: 0.6051
Epoch 4, Loss: 0.5524
Epoch 5, Loss: 0.4667
Epoch 6, Loss: 0.3410
Epoch 7, Loss: 0.2034
Epoch 8, Loss: 0.0927
Epoch 9, Loss: 0.0272
Epoch 10, Loss: 0.0065
```

Program 8:

Objective: Implement a Recurrent Neural Network (RNN) using the PyTorch framework.

Tasks:

- Define an RNN classifier to make predictions on a synthetic time series dataset.
- Train the classifier using suitable criterion and optimizer.

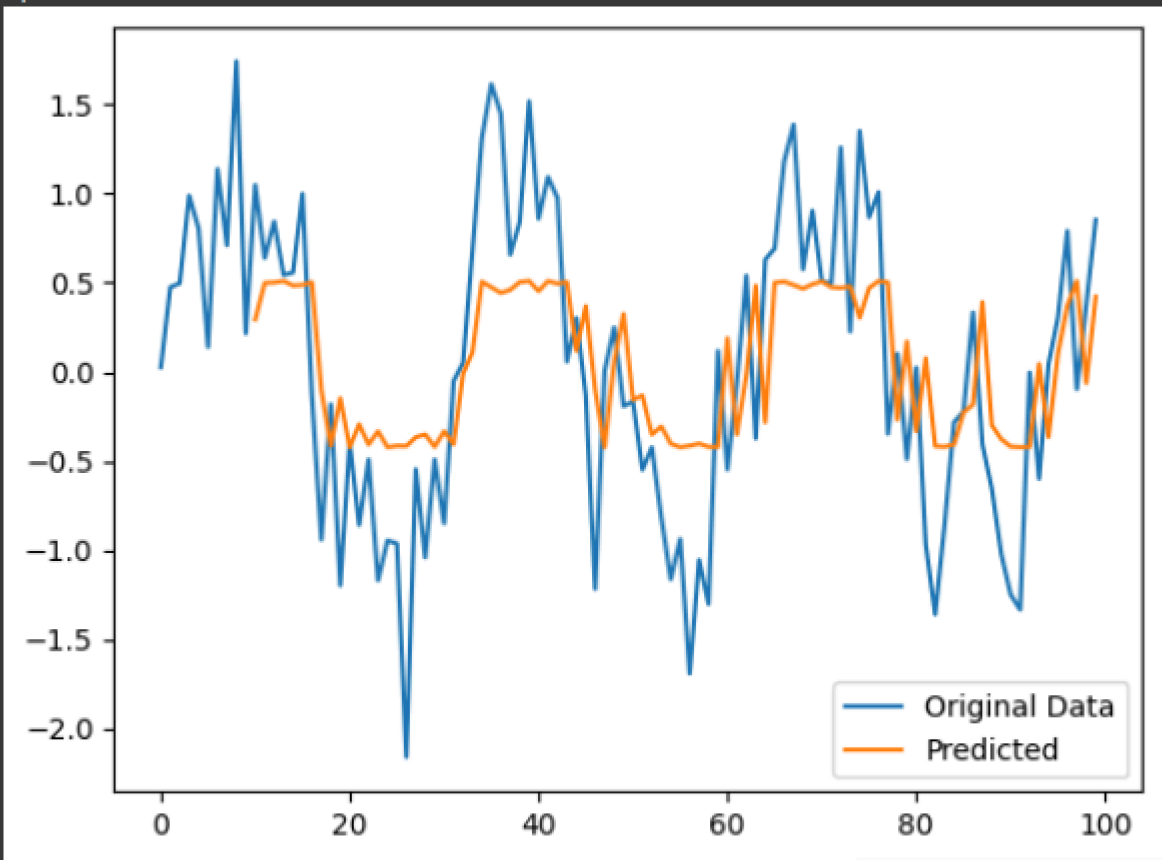
Code:

```
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()
```

```
Epoch 0 loss: 0.6432890892028809
Epoch 10 loss: 0.3645578920841217
Epoch 20 loss: 0.28580573201179504
Epoch 30 loss: 0.25052085518836975
Epoch 40 loss: 0.21080927550792694
Epoch 50 loss: 0.20252841711044312
Epoch 60 loss: 0.16980504989624023
Epoch 70 loss: 0.1840655654668808
Epoch 80 loss: 0.15116114914417267
Epoch 90 loss: 0.15354889631271362
```



Program 9:

Objective: Implement an AutoEncoder using the PyTorch framework.

Tasks:

- Implement an AutoEncoder architecture
- Preprocess the dataset
- Define model train function
- Train model using suitable criterion and optimizer

Code:

```
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)
```

```
Epoch 1, Train Loss: 0.1280, Test Loss: 0.0746
Epoch 2, Train Loss: 0.0719, Test Loss: 0.0710
Epoch 3, Train Loss: 0.0683, Test Loss: 0.0672
Epoch 4, Train Loss: 0.0623, Test Loss: 0.0623
Epoch 5, Train Loss: 0.0565, Test Loss: 0.0548
Epoch 6, Train Loss: 0.0496, Test Loss: 0.0502
Epoch 7, Train Loss: 0.0437, Test Loss: 0.0478
Epoch 8, Train Loss: 0.0394, Test Loss: 0.0436
Epoch 9, Train Loss: 0.0354, Test Loss: 0.0418
Epoch 10, Train Loss: 0.0325, Test Loss: 0.0404
```

Program 10:

Objective: Implement a Generative Adversarial Network (GAN) on the MNIST dataset using the PyTorch framework.

Tasks:

- Define a GAN architecture
- Preprocess the MNIST dataset
- Define the model train function
- Train model using suitable criterion and optimizer

Code:

```
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)
```

```
Epoch 1, Loss D: 0.5456, Loss G: 1.2197
Epoch 2, Loss D: 0.3198, Loss G: 1.8416
Epoch 3, Loss D: 0.4994, Loss G: 1.4539
Epoch 4, Loss D: 0.5980, Loss G: 1.3020
Epoch 5, Loss D: 0.5102, Loss G: 1.5466
Epoch 6, Loss D: 0.2883, Loss G: 1.7432
Epoch 7, Loss D: 0.4820, Loss G: 1.2551
Epoch 8, Loss D: 0.4728, Loss G: 1.3451
Epoch 9, Loss D: 0.3598, Loss G: 1.2704
Epoch 10, Loss D: 0.3948, Loss G: 1.0495
Epoch 11, Loss D: 0.4262, Loss G: 0.9937
Epoch 12, Loss D: 0.4280, Loss G: 0.8331
Epoch 13, Loss D: 0.5069, Loss G: 0.7705
Epoch 14, Loss D: 0.4888, Loss G: 0.6464
Epoch 15, Loss D: 0.4822, Loss G: 0.6916
```

Program 11:

Objective: Implement the Self Attention mechanism using the PyTorch framework.

Tasks:

- Define the Self Attention mechanism
- Show the forward pass

Code:

```
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 0
modulo 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)
torch.Size([32, 10, 128])
```

Program 12:

Objective: Implement a 2 layer Artificial Neural Network using Numpy.

Tasks:

- Implement the forward pass of the network.
- Implement the backward pass of the network
- Train the network

Code:

```
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)
```

```
Epoch 0 Loss: 0.4982982718778417
Epoch 1000 Loss: 0.49639332948919923
Epoch 2000 Loss: 0.46435192880802534
Epoch 3000 Loss: 0.37280071563890294
Epoch 4000 Loss: 0.23950935220019134
Epoch 5000 Loss: 0.14333780993188655
Epoch 6000 Loss: 0.1020383709534976
Epoch 7000 Loss: 0.08073433008073985
```

```
Epoch 8000 Loss: 0.06778248883956983
Epoch 9000 Loss: 0.059035286680823675
Final outputs after training:
[[0.05075362]
 [0.95318224]
 [0.94334761]
 [0.05658842]]
```