Lab Programs (Internal 1)

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

- MNIST
- CIFAR10
- VOCSegmentation

General Procedure:

- **Classification**: (Program 2, 3, 4, 6)
    - Load required dataset
    - Create image augmentations (transforms) to apply to dataset
    - Create subset of train and test datasets
    - Create train and test DataLoaders
    - Define required model architecture
    - Define required criterion and optimizer
    - Create model train function calculating model loss and accuracy
    - Train model
- **Segmentation** (Program 5)
    - Load Segmentation dataset
    - Create image augmentations (transforms) to apply to dataset
    - Create subset of train and test datasets
    - Create train and test DataLoaders
    - Define UNet segmentation model
    - Define required criterion and optimizer
    - Create model train function calculating model loss
    - Train model

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

def plot_leaky_relu():
    x = np.linspace(-10, 10, 100)
    def leaky_relu(x, alpha=0.1):
        return np.where(x >= 0, x, alpha * x)
    leaky_relu_values = leaky_relu(x)
    plt.plot(x, leaky_relu_values)
    plt.title("Leaky ReLU Activation Function")
    plt.xlabel("x")
    plt.ylabel("Leaky ReLU(x)")
    plt.grid(True)
    plt.show()

def softmax():
    def softmax_act(x):
        e_x = np.exp(x - np.max(x))
        return e_x / np.sum(e_x, axis=0)
    x = np.array([1, 2, 3])
    result = softmax_act(x)
    print(result)
    def plot_softmax(probabilities, class_labels):
        plt.bar(class_labels, probabilities)
        plt.xlabel("Class")
        plt.ylabel("Probability")
        plt.title("Softmax Output")
        plt.show()
    class_labels = ["Class A", "Class B", "Class C"]
    plot_softmax(result, class_labels)
```

```
while True:
    print("\nMAIN MENU")
    print("1. Sigmoid")
    print("2. Hyperbolic tangent")
    print("3. Rectified Linear Unit")
    print("4. Leaky ReLU")
    print("5. Softmax")
    print("6. Exit")
    choice = int(input("Enter the Choice:"))
    if choice == 1:
        plot_sigmoid()
    elif choice == 2:
        plot_tanh()
    elif choice == 3:
        plot_relu()
    elif choice == 4:
        plot_leaky_relu()
    elif choice == 5:
        softmax()
    elif choice == 6:
        break
    else:
        print("Oops! Incorrect Choice.")
```

Program 2:

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
```

```python
from torch.utils.data import DataLoader, Subset
import numpy as np
```

```python
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # normalization optional
])

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

train_subset = Subset(train_dataset, range(200))
test_subset = Subset(test_dataset, range(50))
train_loader = DataLoader(train_subset, batch_size=10, shuffle=True)
test_loader = DataLoader(test_subset, batch_size=10, shuffle=False)

class SimpleANN(nn.Module):
  def __init__(self):
    super(SimpleANN, self).__init__()
    self.fc1 = nn.Linear(28*28, 128)
    self.fc2 = nn.Linear(128, 64)
    self.fc3 = nn.Linear(64, 10)

  def forward(self, x):
    x = torch.flatten(x, start_dim=1)
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)
    return x


model = SimpleANN()
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

```python
def train_model(num_epochs):
  for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    correct_train = 0
    total_train = 0
    for data, target in train_loader:
      optimizer.zero_grad()
```

```
      output = model(data)
      loss = criterion(output, target)
      loss.backward()
      optimizer.step()
      train_loss += loss.item()
      predicted = torch.argmax(output.data, dim=1)
      total_train += target.size(0)
      correct_train += (predicted==target).sum().item()

    avg_train_loss = train_loss/len(train_loader)
    train_acc = 100 * correct_train/total_train

    model.eval()
    test_loss = 0
    correct_test = 0
    total_test = 0
    with torch.no_grad():
      for data, target in test_loader:
        output = model(data)
        loss = criterion(output, target)
        test_loss += loss.item()
        predicted = torch.argmax(output.data, dim=1)
        total_test += target.size(0)
        correct_test += (predicted==target).sum().item()

    avg_test_loss = test_loss/len(test_loader)
    test_acc = 100 * correct_test/total_test
    print(f'Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}, Train
Accuracy: {train_acc:.8f}%, '
          f'Test Loss: {avg_test_loss:.4f}, Test Accuracy: {test_acc:.8f}%')


train_model(10)
```

Program 3:

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 CNNWithBNDropout(nn.Module):
 def __init__(self):
  super(CNNWithBNDropout, self).__init__()
  self.conv_block1 = nn.Sequential(
      nn.Conv2d(3, 32, kernel_size=3, padding=1),
      nn.BatchNorm2d(32),
      nn.ReLU(),
      nn.MaxPool2d(2)
  )

  self.conv_block2 = nn.Sequential(
      nn.Conv2d(32, 64, kernel_size=3, padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.MaxPool2d(2)
  )

  self.dense1 = nn.Linear(64 * 8 * 8, 512)
  self.dense2 = nn.Linear(512, 10)

  self.dropout = nn.Dropout(0.5)
  self.relu = nn.ReLU()
  self.flatten = nn.Flatten()

 def forward(self, x):
   x = self.conv_block1(x)
   x = self.conv_block2(x)
   x = self.flatten(x)
   x = self.dense1(x)
```

```
    x = self.relu(x)
    x = self.dense2(x)
    x = self.dropout(x)
    return x
# Data preprocessing and loading
transform = transforms.Compose([
 transforms.ToTensor(),
 transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)

train_subset = Subset(train_dataset, range(200))
test_subset = Subset(test_dataset, range(50))
train_loader = DataLoader(train_subset, batch_size=10, shuffle=True)
test_loader = DataLoader(test_subset, batch_size=10, shuffle=False)

# Function to train and evaluate a model
def train(model, optimizer, criterion, num_epochs):
 for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
    correct_train = 0
    total_train = 0

    for data, target in train_loader:
      output = model(data)
      loss = criterion(output, target)

      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      train_loss += loss.item()
      predicted = torch.argmax(output.data, dim=1)
      total_train += target.size(0)
      correct_train += (predicted == target).sum().item()

    avg_train_loss = train_loss / len(train_loader)
    train_acc = 100 * correct_train / total_train

    model.eval()
```

```
    test_loss = 0.0
    correct_test = 0
    total_test = 0

    with torch.no_grad():
      for data, target in test_loader:
        output = model(data)
        loss = criterion(output, target)
        test_loss += loss.item()
        predicted = torch.argmax(output.data, dim=1)
        total_test += target.size(0)
        correct_test += (predicted==target).sum().item()

    avg_test_loss = test_loss/len(test_loader)
    test_acc = 100 * correct_test/total_test
    print(f'Epoch: [{epoch+1}/{num_epochs}], Train Loss: {avg_train_loss:.4f}, Train Accuracy: {train_acc:.4f}%, Test Loss: {avg_test_loss:.4f}, Test Accuracy: {test_acc:.4f}%')

model = CNNWithBNDropout()

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

train(model, optimizer, criterion, 30)
```

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

transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

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():
    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_()


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)
            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')

Program 5

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

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
import numpy as np
transform = transforms.Compose([
   transforms.ToTensor(),
   transforms.Normalize((0.1307,), (0.3081,))  # normalization optional
])
train_dataset = datasets.MNIST(root="./data", train=True, download=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(15)
```

Program 6:

Lab Exam: Text Classification with LSTM

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

Tasks:

1. **LSTM Model Setup:**
   - Define an LSTM classifier with embedding, LSTM, and fully connected layers. Adjust the model to handle a hypothetical vocabulary size and embedding dimensions.
2. **Data Preparation:**
   - Simulate a dataset where "data" represents text indices and "labels" are binary classification targets. Set up data loading and batching using DataLoader.
3. **Model Training:**
   - Train the LSTM model using the CrossEntropyLoss and Adam optimizer, monitoring the loss over epochs.

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
```

```python
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())
for epoch in range(10):
    for inputs, tgts in loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, tgts)
        loss.backward()
        optimizer.step()

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