# Lab

Task :

1: Practice the implementation of optimization algorithms and compare the results of different optimization algorithms.

```python
class DeepNorn(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim): # hidden_dim is a list, input_dim and output_dim are numbers
        super(DeepNorn, self).__init__()
        self.layers = nn.ModuleList()
        self.norm_layers = nn.ModuleList()
        self.residual_linear_layers = nn.ModuleList()  # List to hold residual linear layers

        # Build layers with residual connections and pre-normalization
        for i, hidden_dim in enumerate(hidden_dims):
            # Add normalization layer for each layer
            self.norm_layers.append(nn.LayerNorm(input_dim))

            # Add the linear transformation for the main layer
            self.layers.append(nn.Linear(input_dim, hidden_dim))

            # Add the linear transformation for the residual connection
            self.residual_linear_layers.append(nn.Linear(input_dim, hidden_dim))

            # Update input_dim for the next layer
            input_dim = hidden_dim

        # Final output layer
        self.output_layer = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        relu = nn.ReLU()
```

```
nsor([[-0.3650,  0.3674,  0.5695, -0.2621, -0.3526, -0.5717,  0.6779,  0.0527,
       -0.2888, -0.1106],
      [-0.4134,  0.6926,  0.1105,  0.1416,  0.3184, -0.3293,  0.0285,  0.0369,
       -0.3402,  0.2836],
      [-0.3687,  0.5827, -0.1696,  0.2811, -0.1749, -0.4903,  0.3309, -0.0916,
        0.1088,  0.3598],
      [-0.3590,  0.3398,  0.2070, -0.0387,  0.0523, -0.0810,  0.3103, -0.0899,
       -0.1279, -0.0949],
      [-0.0065,  0.3553,  0.3065, -0.0991, -0.4146, -0.7177,  0.0705, -0.2681,
       -0.2207, -0.1261],
      [-0.1282,  0.4381,  0.1350,  0.5943, -0.0950, -0.3234,  0.0994, -0.1962,
```

2: Practice the implementation and application of the attention mechanism.

```python
    Feed-Forward Network (FFN).
    """

    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, d_model, d_hidden):
        """
        Args:
            d_model: Dimension of input and output features (e.g., 512).
            d_hidden: Dimension of the hidden layer in FFN (e.g., 2048).
        """
        super(FeedForwardNetwork, self).__init__()
        # Define two linear layers and GELU activation
        self.fc1 = nn.Linear(d_model, d_hidden)
        self.gelu = nn.GELU()
        self.fc2 = nn.Linear(d_hidden, d_model)

    Tabnine | Edit | Test | Explain | Document | Ask
    def forward(self, x):
        """
        Args:
            x: Input tensor of shape (batch_size, seq_length, d_model).
        Returns:
            output: Output tensor of shape (batch_size, seq_length, d_model).
        """
        x = self.fc1(x)   # First linear transformation
        x = self.gelu(x)  # GELU activation
        x = self.fc2(x)   # Second linear transformation
        return x


class TransformerEncoderLayer(nn.Module):
    """
    Transformer Encoder Layer.
    """
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, d_model, num_heads, d_hidden, dropout=0.1):
        """
        Args:
            d_model: Dimension of input features (e.g., 512).
            num_heads: Number of attention heads (e.g., 8).
            d_hidden: Dimension of the hidden layer in FFN (e.g., 2048).
            dropout: Dropout probability (e.g., 0.1).
        """
        super(TransformerEncoderLayer, self).__init__()
        self.mha = MultiHeadAttention(d_model, num_heads)  # Multi-head attention module
        self.ffn = FeedForwardNetwork(d_model, d_hidden)   # Feed-forward network
        self.norm1 = nn.LayerNorm(d_model)  # LayerNorm for MHA output
        self.norm2 = nn.LayerNorm(d_model)  # LayerNorm for FFN output
        self.dropout = nn.Dropout(dropout)  # Dropout for regularization

    Tabnine | Edit | Test | Explain | Document | Ask
    def forward(self, x):
        """
        Args:
            x: Input tensor of shape (batch_size, seq_length, d_model).
        Returns:
            output: Output tensor of shape (batch_size, seq_length, d_model).
        """
        # Multi-Head Attention + Residual Connection + LayerNorm
        attn_output = self.mha(x)
        x = x + self.dropout(attn_output)  # Residual connection
        x = self.norm1(x)  # Layer normalization

        # Feed-Forward Network + Residual Connection + LayerNorm
```

BLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL POSTMAN CONSOLE COMMENTS

Task3

```
.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
import time

# Define the CNN model
class CNN_Model(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN_Model, self).__init__()

        # 4+ convolutional layers with ReLU activation
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = self.pool(torch.relu(self.conv3(x)))
        x = self.pool(torch.relu(self.conv4(x)))
```

```
        # Flatten the output from convolutional layers
        x = x.view(-1, 256 * 8 * 8)

        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Define function to load the dataset
def load_dataset(dataset_name):
    transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    if dataset_name == 'CIFAR-10':
        trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
        testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)
        num_classes = 10
    elif dataset_name == 'FashionMNIST':
        trainset = torchvision.datasets.FashionMNIST(root='./data', train=True,
download=True, transform=transform)
        testset = torchvision.datasets.FashionMNIST(root='./data', train=False,
download=True, transform=transform)
        num_classes = 10
    else:
        trainset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
        testset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)
```

```
        num_classes = 10

    trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
    testloader = DataLoader(testset, batch_size=64, shuffle=False)

    return trainloader, testloader, num_classes

# Define function to train the model
def train_model(model, trainloader, testloader, device, epochs=5):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    start_time = time.time()

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
```

```
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        print(f"Epoch {epoch+1}/{epochs} - Loss:
{running_loss/len(trainloader)} - Accuracy: {accuracy}%")

    end_time = time.time()
    elapsed_time = end_time - start_time
    return elapsed_time

# Define function to test the model
def test_model(model, testloader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Test Accuracy: {accuracy}%")
    return accuracy
```

```
# Main function to run the training and testing
def run_experiment(dataset_name, epochs_list=[5, 10, 20], use_gpu=True):
    device = torch.device("cuda" if use_gpu and torch.cuda.is_available() else
"cpu")
    print(f"Training on device: {device}")

    # Load dataset
    trainloader, testloader, num_classes = load_dataset(dataset_name)

    # Initialize model
    model = CNN_Model(num_classes=num_classes).to(device)

    # Train and test for each epoch count
    for epochs in epochs_list:
        print(f"\nTraining for {epochs} epochs:")
        elapsed_time = train_model(model, trainloader, testloader, device,
epochs=epochs)
        print(f"Training time for {epochs} epochs: {elapsed_time:.2f} seconds")
        test_model(model, testloader, device)

# Run experiment for CIFAR-10 dataset (you can change the dataset to
'FashionMNIST' or 'MNIST')
run_experiment(dataset_name='CIFAR-10', epochs_list=[5, 10, 20],
use_gpu=True)
```