APPLIED DEEP LEARNING

Project 1

1) Code:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
from torchvision.datasets import MNIST, CIFAR10
from torch.utils.data import DataLoader
import numpy as np

# Define the ArcFace Loss function
class ArcFaceLoss(nn.Module):
    def __init__(self, s=30.0, m=0.50, num_classes=10,
embedding_size=128):
        super(ArcFaceLoss, self).__init__()
        self.s = s  # Scale factor
        self.m = m  # Margin
        self.num_classes = num_classes
        self.embedding_size = embedding_size
        self.W = nn.Parameter(torch.randn(embedding_size, num_classes))
        nn.init.xavier_uniform_(self.W)

    def forward(self, embeddings, labels):
        # Normalize embeddings and weights
        embeddings = F.normalize(embeddings, dim=1)
        W = F.normalize(self.W, dim=0)

        # Cosine of angle between embeddings and weights
        cos_theta = torch.mm(embeddings, W)
        theta = torch.acos(torch.clamp(cos_theta, -1.0 + 1e-7, 1.0 - 1e-
7))

        # Add the margin to the target angle
        target_logit = torch.cos(theta + self.m)

        # One-hot encode labels to only apply margin to correct class
        one_hot = torch.zeros_like(cos_theta)
        one_hot.scatter_(1, labels.view(-1, 1), 1.0)
```

```
        # Combine the adjusted target logits with original logits
        output = one_hot * target_logit + (1.0 - one_hot) * cos_theta
        output *= self.s  # Scale the output

        # Cross-entropy loss
        loss = F.cross_entropy(output, labels)
        return loss, output

# Sample Network for Embedding
class SimpleCNN(nn.Module):
    def __init__(self, embedding_size=128):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, embedding_size)

    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(x.size(0), -1)
        x = self.fc1(x)
        return x

# Evaluation function to calculate accuracy
def evaluate(model, dataloader, arcface_loss):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in dataloader:
            embeddings = model(images)
            _, outputs = arcface_loss(embeddings, labels)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total

# Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor()])
train_data = MNIST(root='./data', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_data = MNIST(root='./data', train=False, download=True,
transform=transform)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

```python
# Initialize model, loss, and optimizer
embedding_size = 128
model = SimpleCNN(embedding_size=embedding_size)
arcface_loss = ArcFaceLoss(num_classes=10, embedding_size=embedding_size)
optimizer = torch.optim.Adam(list(model.parameters()) +
list(arcface_loss.parameters()), lr=0.001)

# Training loop with accuracy calculation
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    for images, labels in train_loader:
        optimizer.zero_grad()
        embeddings = model(images)
        loss, _ = arcface_loss(embeddings, labels)
        loss.backward()
        optimizer.step()

    # Calculate train and test accuracy
    train_accuracy = evaluate(model, train_loader, arcface_loss)
    test_accuracy = evaluate(model, test_loader, arcface_loss)

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}, Train
Accuracy: {train_accuracy:.4f}, Test Accuracy: {test_accuracy:.4f}')
```

Results:

```
Epoch [1/10], Loss: 0.3071, Train Accuracy: 0.9175, Test Accuracy: 0.9159
Epoch [2/10], Loss: 1.7320, Train Accuracy: 0.9408, Test Accuracy: 0.9352
Epoch [3/10], Loss: 0.2352, Train Accuracy: 0.9479, Test Accuracy: 0.9357
Epoch [4/10], Loss: 1.1581, Train Accuracy: 0.9480, Test Accuracy: 0.9387
Epoch [5/10], Loss: 0.1421, Train Accuracy: 0.9530, Test Accuracy: 0.9440
Epoch [6/10], Loss: 0.4246, Train Accuracy: 0.9630, Test Accuracy: 0.9467
Epoch [7/10], Loss: 0.2144, Train Accuracy: 0.9687, Test Accuracy: 0.9502
Epoch [8/10], Loss: 1.4651, Train Accuracy: 0.9623, Test Accuracy: 0.9465
Epoch [9/10], Loss: 0.3099, Train Accuracy: 0.9714, Test Accuracy: 0.9539
Epoch [10/10], Loss: 0.0006, Train Accuracy: 0.9704, Test Accuracy: 0.9491
```

Conclusion:

The model, trained using the ArcFace loss function, achieved strong performance on the MNIST dataset, with the training and test accuracies showing consistent improvement across epochs. Starting from an initial test accuracy of 91.6%, the model steadily increased its classification accuracy, reaching around 95% on the test set by the 10th epoch. This reflects ArcFace's ability to create well-separated class boundaries, enhancing the model's discriminative power.

In conclusion, the ArcFace loss function proved effective for multi-class classification, demonstrating significant improvements in accuracy. With some parameter fine-tuning, this approach has the potential to achieve even more stable convergence, making it well-suited for deep learning applications requiring high inter-class separation, such as face recognition or complex image classification tasks.