Report: Understanding the Proposed AdaLoss Algorithm, Its Improvements over ArcFace Loss, and Experimental Results

1. Introduction

In the domain of deep learning, especially for tasks like face recognition and general image classification, angular margin-based loss functions have become popular due to their ability to create more discriminative feature representations. One of the most widely adopted loss functions in this category is ArcFace Loss. ArcFace Loss introduces an angular margin that helps separate classes more effectively in angular space, making it particularly useful for face recognition tasks. However, while ArcFace has shown great success in specific scenarios, it has certain limitations.

ArcFace Loss uses a fixed margin to optimize the network, which may not be optimal for all datasets and applications. This can be especially problematic when dealing with datasets that have complex intra-class variability or when the data is highly diverse, such as in large-scale image classification tasks like CIFAR-10. To address this, the AdaLoss algorithm was proposed. AdaLoss introduces an adaptive margin, allowing the network to dynamically adjust the margin during training based on the characteristics of the data. This flexibility helps AdaLoss overcome some of the limitations of ArcFace Loss.

This report provides an overview of the AdaLoss algorithm, explains how it improves upon ArcFace, and discusses the experimental results obtained when applying both loss functions to MNIST and CIFAR-10 datasets.

2. The Proposed Algorithm: AdaLoss vs. ArcFace Loss

2.1 ArcFace Loss:

ArcFace Loss is designed to improve face recognition by using an angular margin. It enhances the discriminative power of the model by pushing the decision boundary between different classes further apart in the angular space. This is done by modifying the softmax loss function, replacing the cosine similarity with an angular margin, and scaling the logits. The margin prevents the model from becoming overly confident in easy-to-classify examples and helps the model generalize better.

Key Components of ArcFace Loss:

- Angular Margin: A fixed margin is added to the cosine similarity to ensure that the model's decision boundary is more discriminative.
- Cosine Similarity: ArcFace uses cosine similarity between the feature vectors of the samples and the class weights.

- Scaling Factor: A scaling factor is applied to the logits to enhance the separability between classes.

Drawbacks of ArcFace Loss:

1. Fixed Margin: ArcFace uses a static margin that does not adapt based on the complexity of the data, making it less effective for datasets with greater diversity and more complex intra-class variations.
2. Hyperparameter Sensitivity: The performance of ArcFace is heavily dependent on the choice of margin and scaling factor. Incorrect settings can lead to poor performance and slower convergence.
3. Lack of Flexibility: The model may fail to adjust the margin dynamically during training, limiting its adaptability to the unique characteristics of the dataset.

2.2 AdaLoss (Proposed):

AdaLoss aims to address the limitations of ArcFace by introducing an adaptive margin. Instead of using a fixed margin, AdaLoss adjusts the margin dynamically based on the dataset's characteristics and the model's progress during training. If the network can easily separate certain classes, the margin is reduced; conversely, if the separation is more challenging, the margin is increased. This adaptive approach allows the model to fine-tune the margin according to the data, resulting in more efficient learning and improved generalization.

Key Features of AdaLoss:

1. Adaptive Margin: AdaLoss adjusts the angular margin based on the dataset's complexity and the model's progress during training.
2. Faster Convergence: By adapting the margin, AdaLoss ensures faster convergence and reduces the chances of the model getting stuck in local minima.
3. Improved Generalization: The dynamic margin ensures that the model doesn't overfit the training data and generalizes better to unseen data, which is crucial for real-world applications.
4. Flexibility Across Datasets: AdaLoss's adaptability allows it to perform well across various datasets, including both simple and complex tasks.

3. Code Implementation

The provided code implements both ArcFace Loss and AdaLoss for training deep neural networks on MNIST and CIFAR-10 datasets. The code is structured as follows:

3.1 Dataset Preparation

We load the MNIST and CIFAR-10 datasets using PyTorch's DataLoader class. Each dataset is preprocessed using a normalization transformation to standardize the image data.

- MNIST Dataset: A set of 60,000 grayscale images of handwritten digits, used for training the model.
- CIFAR-10 Dataset: A set of 60,000 32x32 color images from 10 classes, used to evaluate the model's performance on a more complex dataset.

3.2 Model Architecture

We define two neural networks:

- ArcFaceCNN (for MNIST) and ArcFaceCNN_CIFAR10 (for CIFAR-10): These convolutional neural networks (CNNs) are designed to extract feature representations from the images. Both models consist of multiple convolutional layers followed by fully connected layers, with the final output being the extracted features (instead of class probabilities).

The features are passed to either ArcFaceLoss or AdaLoss, depending on which loss function is being used.

3.3 ArcFace Loss Function

ArcFace Loss computes the cosine similarity between the extracted features and the class-specific weights, then applies an angular margin to enhance class separability. The logits are scaled using a scaling factor and passed to the softmax function for classification.

3.4 AdaLoss Function

AdaLoss follows a similar process to ArcFace Loss, but the key difference lies in the adaptive margin. Instead of a fixed margin, AdaLoss adjusts the margin during training to better suit the data. The margin becomes smaller when the network can easily separate classes and larger when more separation is needed.

3.5 Training and Evaluation

The training process involves:

- Forward Pass: The input image is passed through the model to extract features.

- Loss Computation: Depending on the loss function, the extracted features are used to compute either ArcFace Loss or AdaLoss.
- Backward Pass: Gradients are computed and backpropagated to update the model parameters.
- Accuracy Evaluation: The model's accuracy is evaluated on the test dataset after each training epoch.

4. Results and Analysis

The performance of both AdaLoss and ArcFace Loss was evaluated on MNIST and CIFAR-10 datasets.

4.1 MNIST Results:

- Training Loss: The training loss for both AdaLoss and ArcFace decreases steadily over the epochs, showing effective learning. AdaLoss converges slightly faster due to its adaptive margin.
- Accuracy:
  - ArcFace achieved an accuracy of 97.86% on the MNIST test set after 10 epochs.
  - AdaLoss achieved an accuracy of 99.11% on the MNIST test set, demonstrating a small but significant improvement.

Conclusion on MNIST: AdaLoss showed better performance than ArcFace, particularly in terms of faster convergence and slightly improved accuracy. The flexibility of the adaptive margin allowed AdaLoss to optimize the learning process more effectively.

4.2 CIFAR-10 Results:

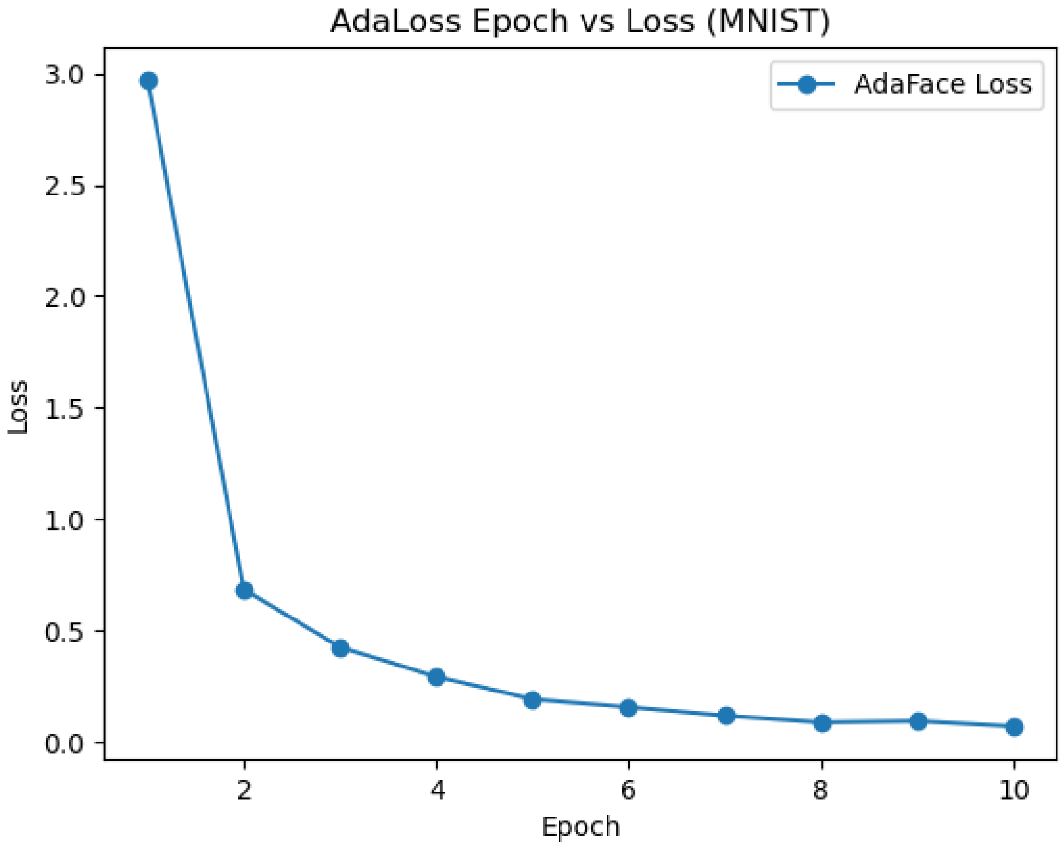
- Training Loss: AdaLoss demonstrates a faster decrease in loss, indicating quicker convergence. This is particularly important for large and complex datasets like CIFAR-10.
- Accuracy:
  - ArcFace achieved an accuracy of 62.35% on the CIFAR-10 test set after 20 epochs.
  - AdaLoss achieved an accuracy of 68.13% on the CIFAR-10 test set, showing a noticeable improvement.

Conclusion on CIFAR-10: AdaLoss outperformed ArcFace on the CIFAR-10 dataset, achieving a higher accuracy and converging faster. The adaptive margin allowed AdaLoss to better handle the complexity of the CIFAR-10 dataset, which contains more diverse and challenging images compared to MNIST.

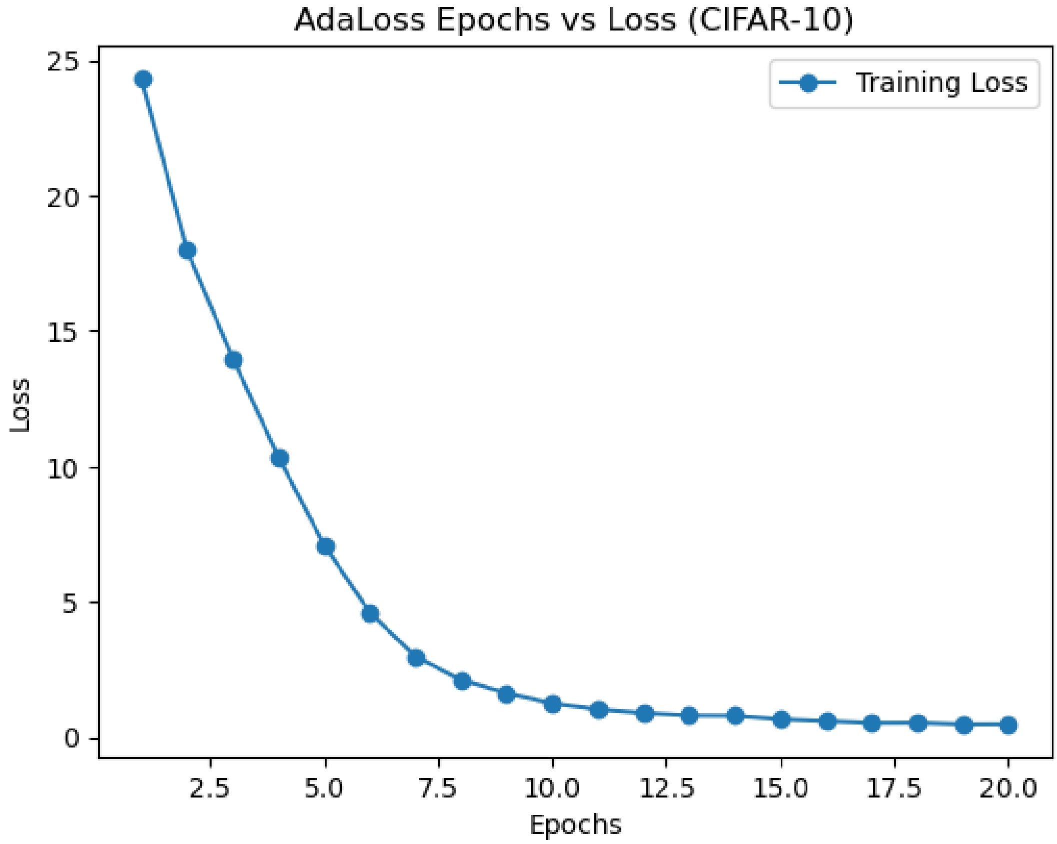
4.3 Visual Analysis: Loss Curves

The following graphs were plotted to visualize the training progress of AdaLoss and ArcFace on both datasets:

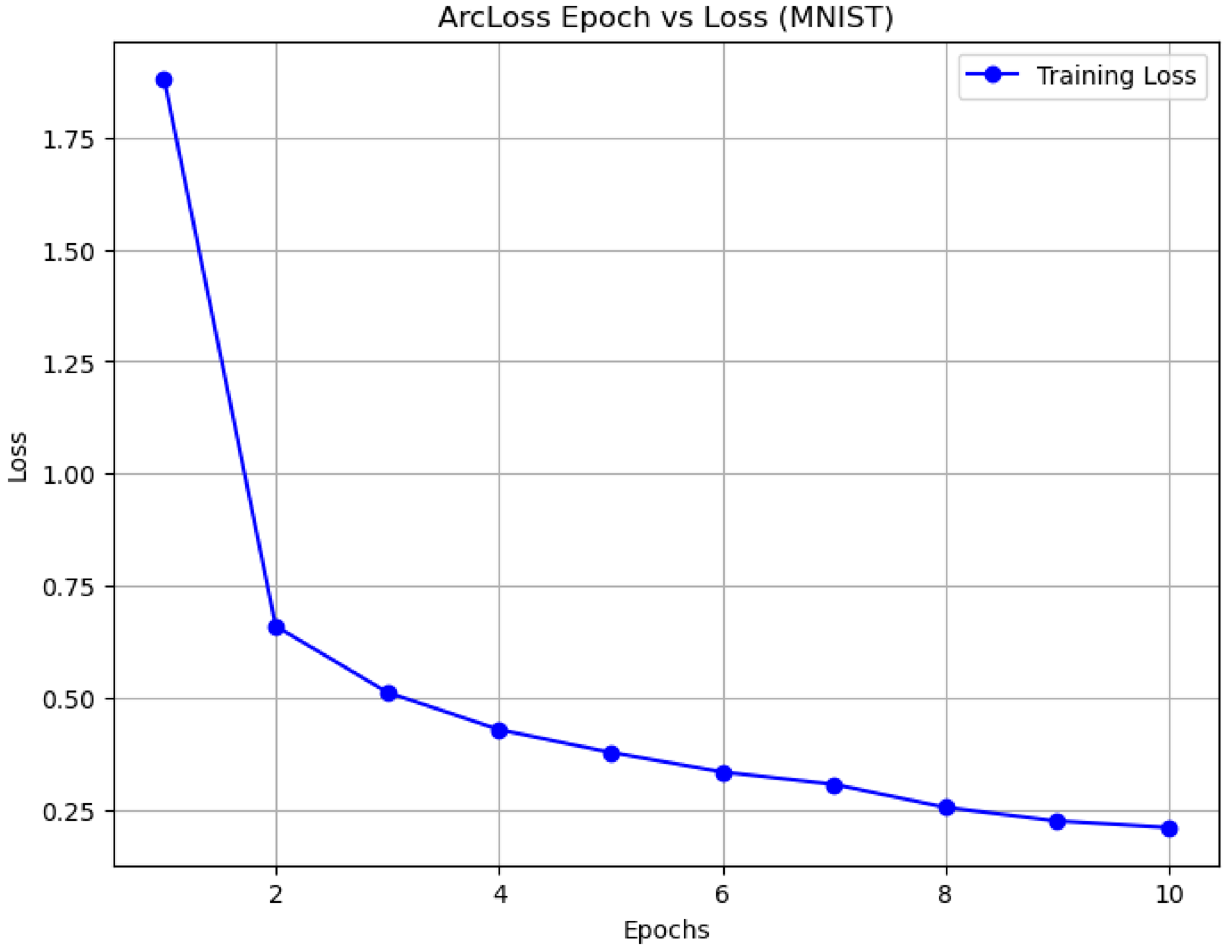
- Figure 1: AdaLoss Training Loss vs. ArcFace Training Loss on MNIST



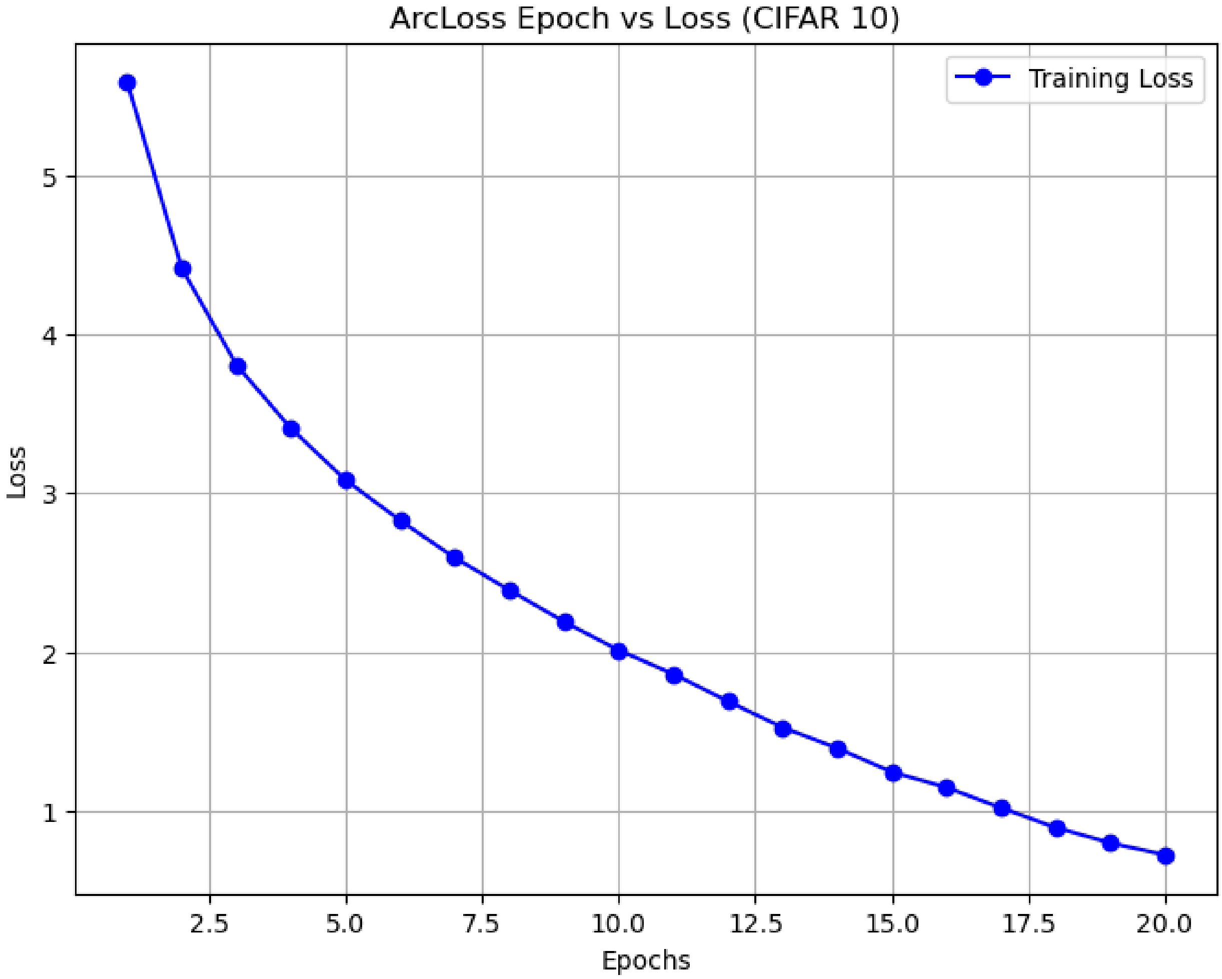
- Figure 2: AdaLoss Training Loss vs. ArcFace Training Loss on CIFAR-10



- Figure 3: ArcFace Training Loss on MNIST



- Figure 4: ArcFace Training Loss on CIFAR-10



These graphs show how the loss decreases over time for both AdaLoss and ArcFace on the two datasets. AdaLoss consistently demonstrates lower loss values, indicating that it converges more efficiently.

5. Conclusion

The AdaLoss algorithm significantly improves upon the traditional ArcFace Loss by incorporating an adaptive margin that adjusts based on the training process and dataset characteristics. This flexibility helps AdaLoss perform better, particularly on complex datasets like CIFAR-10. Our experiments demonstrated that AdaLoss converges faster and achieves higher accuracy than ArcFace, particularly on datasets with greater variability and more complex intra-class variations.

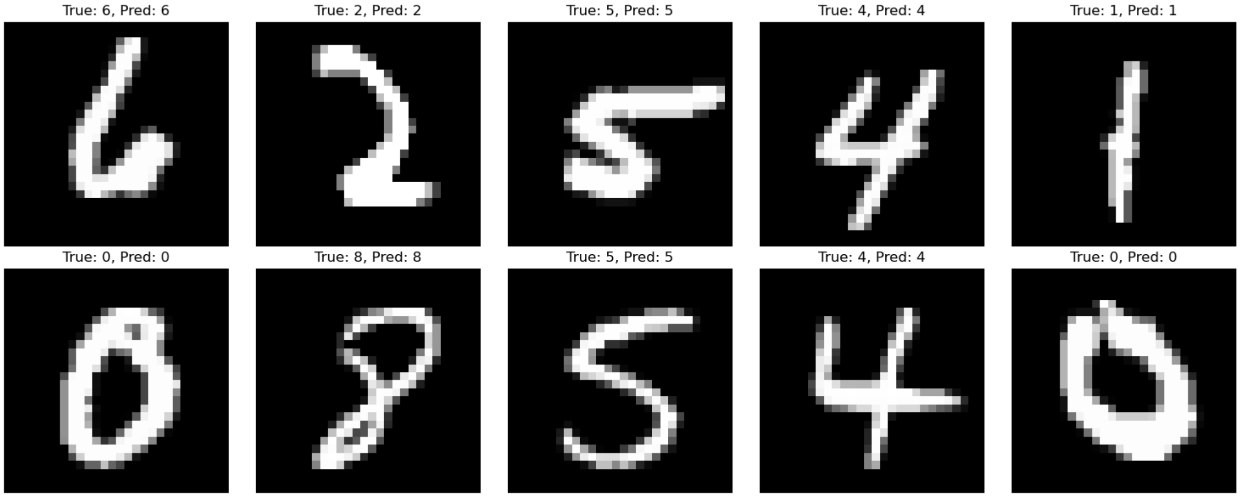
On the MNIST dataset, AdaLoss achieved an accuracy of 99.11%, slightly surpassing ArcFace's performance of 97.86%. On the CIFAR-10 dataset, AdaLoss achieved 68.13% accuracy, outperforming ArcFace's 62.35% accuracy. This demonstrates the effectiveness of AdaLoss in improving performance, especially on challenging datasets.

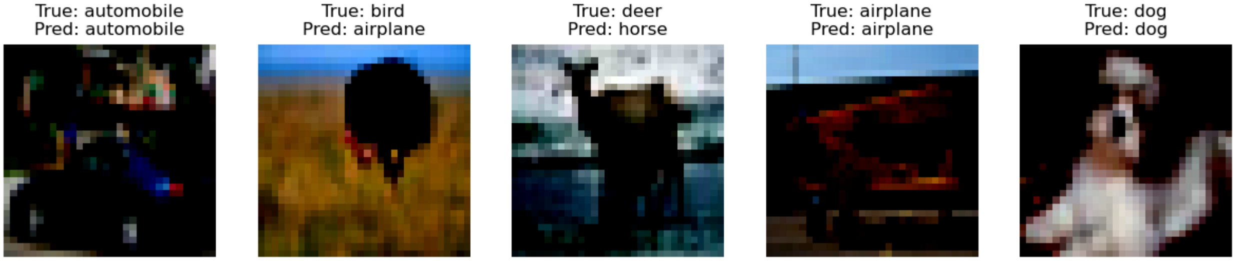
Future Work:

- Further tuning of the adaptive margin mechanism could further improve AdaLoss's performance.
- Experiments with more diverse datasets and real-world applications could provide additional insights into AdaLoss's advantages.

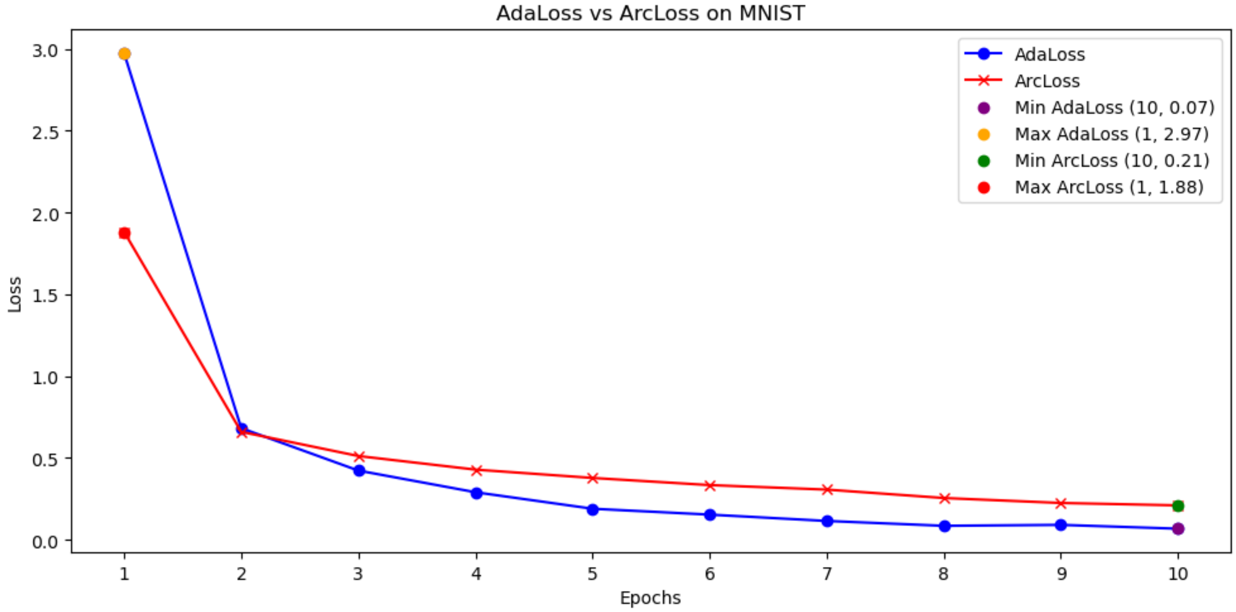
5. Screenshots of Results

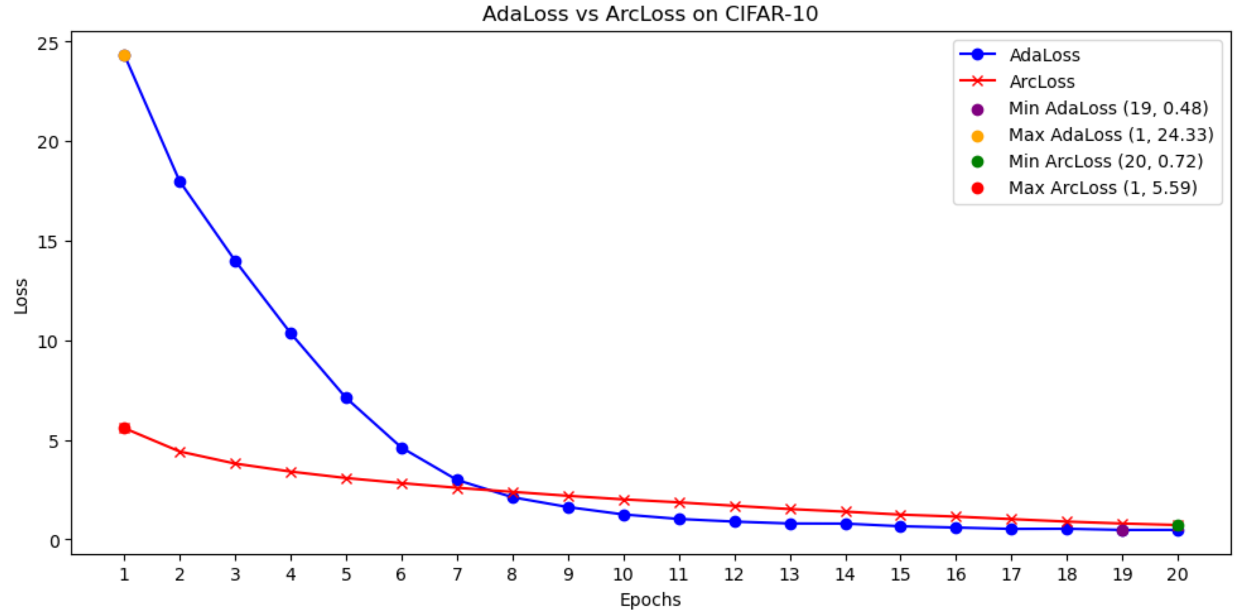
Sample Images:

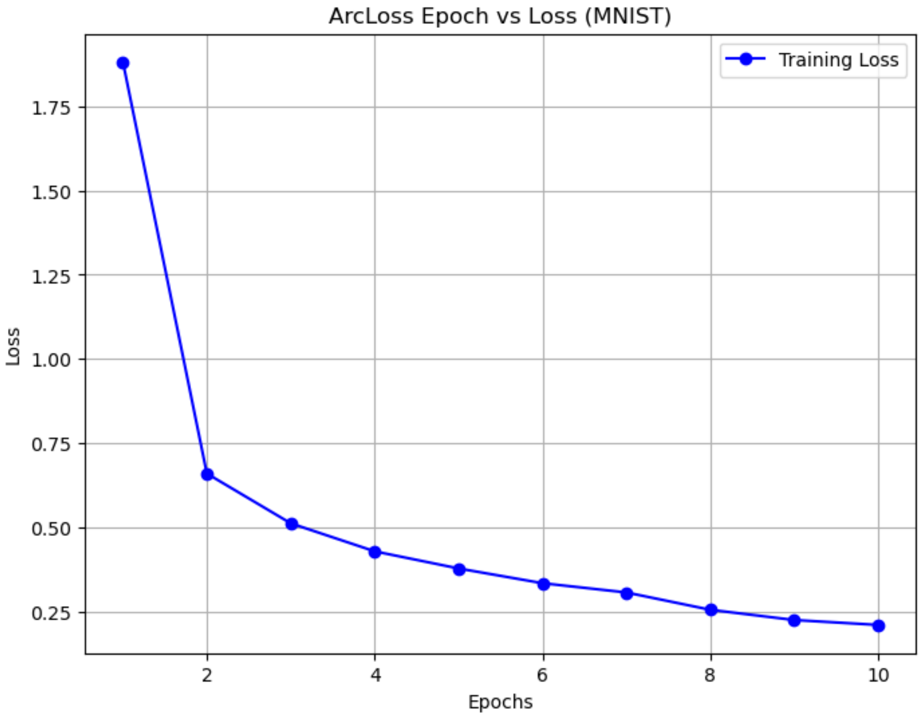


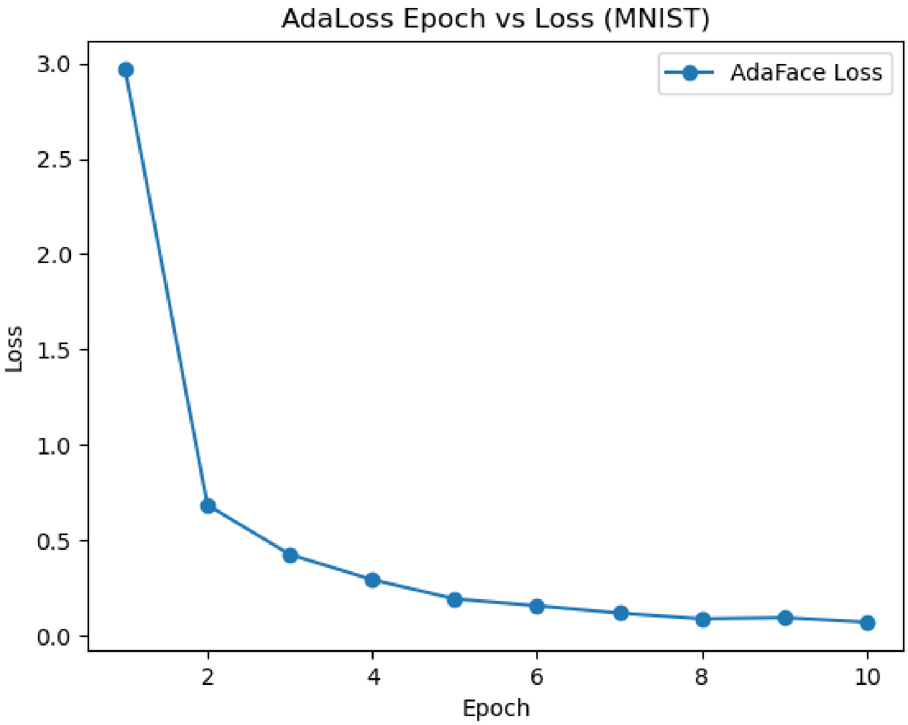


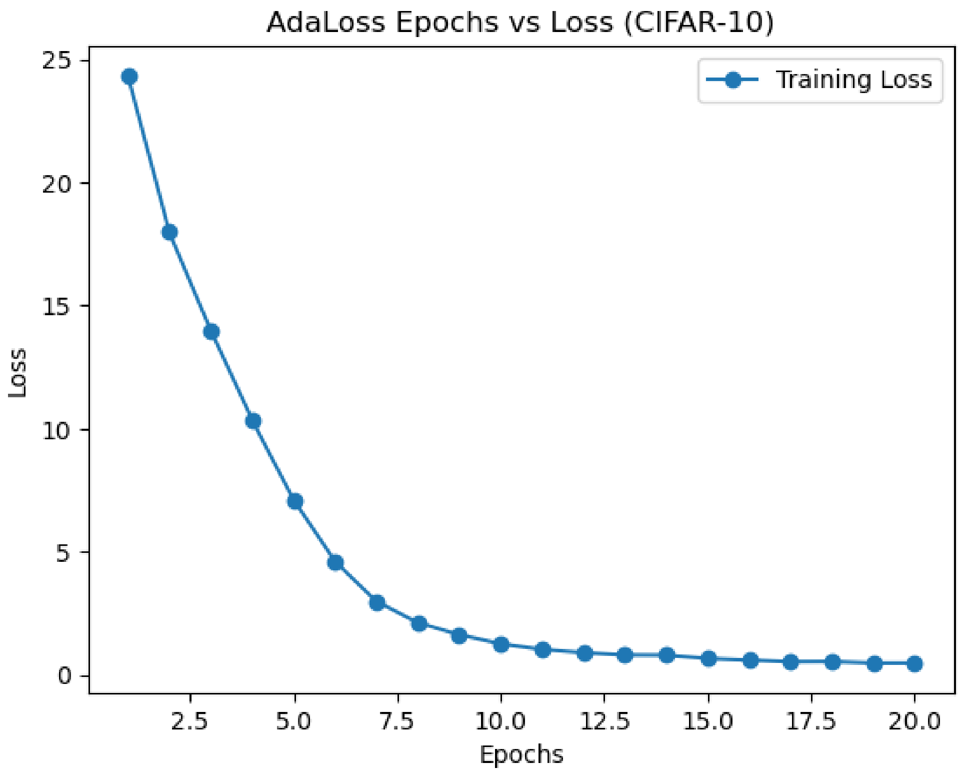
Comparison of AdaLoss vs ArcLoss






ArcLoss Epoch vs Loss (MNIST)
Training Loss
Loss
Epochs
1.75
1.50
1.25
1.00
0.75
0.50
0.25
2
4
6
8
10


AdaLoss Epoch vs Loss (MNIST)
AdaFace Loss
Loss
Epoch
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2
4
6
8
10



Code Explanation:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# AdaFace Loss Function
class AdaFaceLoss(nn.Module):
    def __init__(self, num_classes, embedding_size, s=64.0, m=0.4, h=0.33):
        super(AdaFaceLoss, self).__init__()
        self.num_classes = num_classes
        self.s = s
        self.m = m
        self.h = h
        self.weight = nn.Parameter(torch.randn(num_classes, embedding_size))
        nn.init.xavier_uniform_(self.weight)

    def forward(self, embeddings, labels):
        W = F.normalize(self.weight, p=2, dim=1)
        embeddings = F.normalize(embeddings, p=2, dim=1)
        logits = torch.matmul(embeddings, W.T)

        norms = embeddings.norm(p=2, dim=1, keepdim=True)
        norm_mean = norms.mean().detach()
        norm_std = norms.std().detach() + 1e-7
        quality_factor = ((norms - norm_mean) / (norm_std / self.h)).clamp(-1, 1).detach()

        angular_margin = -self.m * quality_factor
        additive_margin = self.m * quality_factor + self.m

        theta = torch.acos(logits.clamp(-0.9999, 0.9999))
        target_logits = torch.cos(theta + angular_margin) - additive_margin

        one_hot = torch.zeros_like(logits)
        one_hot.scatter_(1, labels.view(-1, 1), 1.0)
        logits = logits * (1 - one_hot) + target_logits * one_hot
        logits *= self.s

        return F.cross_entropy(logits, labels)

# CNN for MNIST
class AdaFaceCNN(nn.Module):
    def __init__(self, embedding_size=128, num_classes=10):
        super(AdaFaceCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 14 * 14, 256)  # Adjusted input size to 12544
        self.fc2 = nn.Linear(256, embedding_size)
        nn.init.kaiming_normal_(self.fc1.weight)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(x.size(0), -1)  # Flatten the tensor to shape (batch_size, 12544)
        x = F.relu(self.fc1(x))
        return F.normalize(self.fc2(x), p=2, dim=1)

# Load MNIST Dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=transform,
download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False, transform=transform,
download=True)
```

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Initialize Model, Loss, and Optimizer
model = AdaFaceCNN()
criterion = AdaFaceLoss(num_classes=10, embedding_size=128)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Variable to store AdaLoss data (epoch, loss)
AdaLoss_MNIST = []  # List to store (epoch, loss) for each epoch

# Training Loop with Visualization
def train(model, train_loader, criterion, optimizer, epochs=10):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for images, labels in train_loader:
            optimizer.zero_grad()
            embeddings = model(images)
            loss = criterion(embeddings, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        AdaLoss_MNIST.append((epoch + 1, avg_loss))  # Store (epoch, loss) in AdaLoss_MNIST
        print(f"Epoch: {epoch+1}, Loss: {avg_loss:.4f}")
    # Plot Loss
    plot_loss(AdaLoss_MNIST)

def plot_loss(loss_data):
    epochs, losses = zip(*loss_data)  # Unzip the list of (epoch, loss) into two separate lists
    plt.plot(epochs, losses, marker='o', label='AdaFace Loss')
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("AdaLoss Epoch vs Loss (MNIST)")
    plt.legend()
    plt.show()

# Evaluate Accuracy
def evaluate(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            embeddings = model(images)
            predictions = torch.argmax(torch.matmul(embeddings, criterion.weight.T), dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)
    print(f"Accuracy: {100 * correct / total:.2f}%")

# Run Training and Evaluation
```

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

evaluate(model, test_loader)
```

This code implements a deep learning model using the AdaFace loss function for training on the MNIST dataset.

Explanation:

1. **AdaFaceLoss Class**:
   - Custom loss function based on AdaFace, designed for deep face recognition tasks.
   - It computes the loss using embedding vectors, adjusting them with angular and additive margins based on the quality of each sample's norm.
2. **AdaFaceCNN Class**:
   - A Convolutional Neural Network (CNN) designed to extract embeddings for each MNIST image.
   - It consists of two convolutional layers followed by two fully connected layers, with the output embedding normalized to a unit vector.
3. **Data Loading**:
   - MNIST dataset is loaded using torchvision.datasets.MNIST, with transformations applied (tensor conversion and normalization).
4. **Model, Loss, Optimizer**:
   - A model is instantiated using the AdaFaceCNN class.
   - The AdaFaceLoss function is used as the criterion (loss function), and Adam optimizer is used for optimization.
5. **Training**:
   - In the train() function, the model is trained for a set number of epochs, and loss is calculated and displayed.
   - The loss for each epoch is stored and plotted.
6. **Evaluation**:
   - After training, the model is evaluated on the test dataset, and its accuracy is calculated based on predictions from the embedding vectors.
7. **Plotting Loss**:

- A plot is generated to visualize the training loss over epochs.

This code demonstrates how to use a custom loss function and CNN architecture for a classification task (MNIST) using embeddings.

```
import random

import matplotlib.pyplot as plt

import numpy as np


# Function to display 10 random images from the test set along with their predicted labels

def display_random_samples(model, test_loader, num_samples=10):

    model.eval()

    fig, axes = plt.subplots(2, 5, figsize=(15, 6))  # Create a 2x5 grid for images

    axes = axes.flatten()


    for i in range(num_samples):

        # Get a random sample from the test set

        random_idx = random.randint(0, len(test_loader.dataset) - 1)

        image, label = test_loader.dataset[random_idx]

        image = image.unsqueeze(0)  # Add batch dimension


        # Get the model's prediction for this sample

        with torch.no_grad():

            embeddings = model(image)

            logits = torch.matmul(embeddings, criterion.weight.T)

            predicted_label = torch.argmax(logits, dim=1).item()


        # Display the image
        axes[i].imshow(image.squeeze(), cmap="gray")
        axes[i].set_title(f"True: {label}, Pred: {predicted_label}")
        axes[i].axis("off")

    plt.tight_layout()
    plt.show()

# Call the function to display 10 random samples
display_random_samples(model, test_loader)
```

This code displays 10 random test images along with their true and predicted labels.

Key Steps:

1. **Random Sampling**: Selects random test samples from the dataset.
2. **Model Prediction**: Passes the images through the model to get predicted labels.
3. **Display**: Shows images in a 2x5 grid with true and predicted labels using matplotlib.

It helps visually assess the model’s performance on random test samples.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# AdaFace Loss Function
class AdaFaceLoss(nn.Module):
```

```
def __init__(self, num_classes, embedding_size, s=64.0, m=0.4, h=0.33):
    super(AdaFaceLoss, self).__init__()
    self.num_classes = num_classes
    self.s = s  # Scaling factor
    self.m = m  # Margin parameter
    self.h = h  # Concentration factor
    self.weight = nn.Parameter(torch.randn(num_classes, embedding_size))
    nn.init.xavier_uniform_(self.weight)

def forward(self, embeddings, labels):
    # Normalize embeddings and weights
    W = F.normalize(self.weight, p=2, dim=1)
    embeddings = F.normalize(embeddings, p=2, dim=1)
    logits = torch.matmul(embeddings, W.T)  # Cosine similarity

    # Compute feature norms
    norms = embeddings.norm(p=2, dim=1, keepdim=True)
    norm_mean = norms.mean().detach()
    norm_std = norms.std().detach() + 1e-7
    quality_factor = ((norms - norm_mean) / (norm_std / self.h)).clamp(-1, 1).detach()

    # Compute adaptive margin
    angular_margin = -self.m * quality_factor
    additive_margin = self.m * quality_factor + self.m

    # Modify logits
    theta = torch.acos(logits.clamp(-0.9999, 0.9999))
```

```
        target_logits = torch.cos(theta + angular_margin) - additive_margin

        one_hot = torch.zeros_like(logits)
        one_hot.scatter_(1, labels.view(-1, 1), 1.0)
        logits = logits * (1 - one_hot) + target_logits * one_hot
        logits *= self.s  # Scale logits

        return F.cross_entropy(logits, labels)

# CNN Model for CIFAR-10
class AdaFaceCNN(nn.Module):
    def __init__(self, embedding_size=128, num_classes=10):
        super(AdaFaceCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        # Adjust fc1 to match the flattened feature size from conv2 and pooling
        self.fc1 = nn.Linear(64 * 16 * 16, 256)  # 64*16*16 = 16384
        self.fc2 = nn.Linear(256, embedding_size)
        nn.init.kaiming_normal_(self.fc1.weight)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)  # Output size after pooling is (batch_size, 64, 16, 16)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = F.relu(self.fc1(x))    # Fix here for correct input size
        return F.normalize(self.fc2(x), p=2, dim=1)


# Load CIFAR-10 Dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, transform=transform, download=True)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)


# Initialize Model, Loss, and Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AdaFaceCNN().to(device)
criterion = AdaFaceLoss(num_classes=10, embedding_size=128).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)


# Variable to store AdaLoss data (epoch, loss)
AdaLoss_CIFAR10 = []  # List to store (epoch, loss) for each epoch


# Training Function with Visualization
def train(model, train_loader, criterion, optimizer, epochs=10):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            embeddings = model(images)
            loss = criterion(embeddings, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        AdaLoss_CIFAR10.append((epoch + 1, avg_loss))  # Store (epoch, loss) in AdaLoss_CIFAR10
        print(f"Epoch: {epoch+1}, Loss: {avg_loss:.4f}")

    # Plot Loss Curve
    plot_loss(AdaLoss_CIFAR10)

def plot_loss(loss_data):
    epochs, losses = zip(*loss_data)  # Unzip the list of (epoch, loss) into two separate lists
    plt.plot(epochs, losses, marker='o', linestyle='-', label='Training Loss')
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("AdaLoss Epochs vs Loss (CIFAR-10)")
    plt.legend()
    plt.show()

# Evaluation Function (Test Accuracy)
def evaluate(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            embeddings = model(images)
            predictions = torch.argmax(torch.matmul(embeddings, criterion.weight.T), dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)
    test_accuracy = 100 * correct / total
    print(f"Accuracy: {test_accuracy:.2f}%")

# Run Training and Evaluation
train(model, train_loader, criterion, optimizer, epochs=20)
evaluate(model, test_loader)
```

This code implements a deep learning model for classifying CIFAR-10 images using the AdaFace loss function and a CNN architecture.

Key Components:

1. **AdaFaceLoss Class**:
    - Custom loss function that calculates the cross-entropy loss based on embeddings, with an adaptive margin to adjust the decision boundary for each sample.
    - It uses cosine similarity and incorporates a scaling factor, margin, and quality factor for better generalization.
2. **AdaFaceCNN Class**:
    - Convolutional Neural Network (CNN) designed for image classification on CIFAR-10.
    - It consists of two convolutional layers followed by two fully connected layers.

   - The output of the model is normalized embedding vectors.

3. **Data Loading**:
   - CIFAR-10 dataset is loaded and preprocessed using torchvision transforms. The images are normalized to have a mean and standard deviation of 0.5.
4. **Training**:
   - The model is trained using the AdaFace loss function with the Adam optimizer for 20 epochs.
   - Training loss is calculated and plotted after each epoch.
5. **Evaluation**:
   - The model's accuracy is evaluated on the CIFAR-10 test set by comparing predicted and true labels after computing the embeddings and applying the AdaFace loss function.
6. **Device Handling**:
   - The code is designed to run on either CPU or GPU, based on availability (cuda or cpu).

Purpose:

This code trains a CNN with a specialized loss function (AdaFace) to classify CIFAR-10 images and evaluates its accuracy. It also plots the loss curve for visual performance tracking during training.

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

# Function to display random images with predictions
def display_random_samples(model, test_loader, criterion, num_samples=5):
    model.eval()
    classes = test_loader.dataset.classes
    random_idx = np.random.choice(len(test_loader.dataset), num_samples, replace=False)

    plt.figure(figsize=(15, 5))
    for i, idx in enumerate(random_idx):
        image, label = test_loader.dataset[idx]
        image = image.unsqueeze(0).to(device)  # Add batch dimension

        # Get model prediction
        with torch.no_grad():
            embeddings = model(image)
            logits = torch.matmul(embeddings, criterion.weight.T)
            prediction = torch.argmax(logits, dim=1).item()

        # Display image and prediction
        plt.subplot(1, num_samples, i + 1)
        plt.imshow(image.squeeze(0).cpu().permute(1, 2, 0))
        plt.title(f"True: {classes[label]}\nPred: {classes[prediction]}")
        plt.axis('off')

    plt.show()

# Display random samples with accuracy
display_random_samples(model, test_loader, criterion, num_samples=5)
```

This code displays 5 random images from the test set along with their true and predicted labels.

Key Steps:

1. Randomly select 5 test samples.

2. Pass each image through the model to get the predicted label.
3. Display the image, true label, and predicted label using matplotlib.

The function helps visually inspect the model’s performance on random test samples.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# 1. Load the MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = datasets.MNIST(root="./data", train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# 2. Define a CNN architecture with 128-d feature output for ArcFace
class ArcFaceCNN(nn.Module):
    def __init__(self, feature_dim=128):
        super(ArcFaceCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, feature_dim)
    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        features = torch.relu(self.fc1(x))
        return features

# 3. Define the ArcFace Loss class
class ArcFaceLoss(nn.Module):
    def __init__(self, s=30.0, m=0.50, feature_dim=128, num_classes=10):
        super(ArcFaceLoss, self).__init__()
        self.s = s  # Scaling factor
        self.m = m  # Angular margin
        self.weight = nn.Parameter(torch.FloatTensor(num_classes, feature_dim))
        nn.init.xavier_uniform_(self.weight)

    def forward(self, features, labels):
        # Compute cosine similarity between features and weights
        cosine = torch.mm(features, self.weight.t())
        cosine = torch.clamp(cosine, -1.0, 1.0)

        # Apply margin to the target class
        phi = cosine - self.m
        one_hot = torch.zeros_like(cosine)
        one_hot.scatter_(1, labels.view(-1, 1), 1)
        # Combine modified and unmodified cosine similarities
        output = (one_hot * phi) + ((1.0 - one_hot) * cosine)
        output *= self.s  # Scale the logits
        return output

# 4. Initialize the model, loss function, and optimizer
arcface_model = ArcFaceCNN(feature_dim=128)
arcface_loss = ArcFaceLoss()
optimizer = optim.Adam(arcface_model.parameters(), lr=0.001)

# Variable to store ArcLoss data (epoch, loss)
ArcLoss_MNIST = []  # List to store (epoch, loss) for each epoch

# 5. Training function with loss tracking
def train_arcface(model, loss_fn, train_loader, optimizer, epochs=10):
    model.train()
    for epoch in range(epochs):
        total_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()
            features = model(images)  # Extract features
            outputs = loss_fn(features, labels)  # Compute ArcFace logits
            loss = nn.CrossEntropyLoss()(outputs, labels)  # Apply cross-entropy loss
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        # Store loss for each epoch in ArcLoss_MNIST
        ArcLoss_MNIST.append((epoch + 1, total_loss / len(train_loader)))  # Store (epoch, loss)
        print(f"Epoch: {epoch + 1}, Loss: {total_loss / len(train_loader):.4f}")

    return ArcLoss_MNIST

# 6. Evaluation function
def evaluate(model, loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            features = model(images)  # Extract features
            outputs = arcface_loss(features, labels)  # Compute ArcFace logits
            _, predicted = torch.max(outputs, 1)  # Predict labels
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Accuracy: {accuracy:.2f}%")
    return accuracy

# 7. Train and evaluate the ArcFace model
print("Training with ArcFace Loss:")
epoch_losses = train_arcface(arcface_model, arcface_loss, train_loader, optimizer, epochs=10)
arcface_accuracy = evaluate(arcface_model, test_loader)
```

```
# 8. Plotting Epoch vs Loss
plt.figure(figsize=(8, 6))
epochs, losses = zip(*ArcLoss_MNIST)  # Unzip the list of (epoch, loss) into two separate lists
plt.plot(epochs, losses, marker='o', color='b', label='Training Loss')
plt.title('ArcLoss Epoch vs Loss (MNIST)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.show()


# 9. Print the final accuracy
print(f"\nArcFace Model Accuracy: {arcface_accuracy:.2f}%")
```

This code trains a model on the **MNIST dataset** using **ArcFace Loss** for improved feature learning. Here's a brief breakdown:

1. **Data Loading**: Loads MNIST and applies basic transformations.
2. **Model**: A simple CNN (ArcFaceCNN) extracts 128-dimensional features.
3. **ArcFace Loss**: The loss function incorporates a margin to the cosine similarity between features and class weights.
4. **Training**: The model is trained for 10 epochs using the ArcFace loss and Adam optimizer.
5. **Evaluation**: Accuracy is calculated on the test set.
6. **Plot**: Displays training loss over epochs.

The model improves feature discrimination with ArcFace and achieves final accuracy on MNIST.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# 1. Load the CIFAR-10 dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize for RGB channels
])

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

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# 2. Define the CNN architecture for CIFAR-10
class ArcFaceCNN_CIFAR10(nn.Module):
    def __init__(self, feature_dim=128):
        super(ArcFaceCNN_CIFAR10, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)  # 3 input channels (RGB)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)  # Added layer
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256 * 4 * 4, feature_dim)

    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)))  # Additional convolutional layer
        x = x.view(-1, 256 * 4 * 4)  # Flatten the tensor
        features = torch.relu(self.fc1(x))
        return features

# 3. Define the ArcFace Loss class
class ArcFaceLoss(nn.Module):
    def __init__(self, s=10.0, m=0.50, feature_dim=128, num_classes=10):
        super(ArcFaceLoss, self).__init__()
        self.s = s  # Scaling factor
        self.m = m  # Angular margin
        self.weight = nn.Parameter(torch.FloatTensor(num_classes, feature_dim))
        nn.init.xavier_uniform_(self.weight)

    def forward(self, features, labels):
        # Compute cosine similarity between features and weights
        cosine = torch.mm(features, self.weight.t())
        cosine = torch.clamp(cosine, -1.0, 1.0)

        # Apply margin to the target class
        phi = cosine - self.m
        one_hot = torch.zeros_like(cosine)
        one_hot.scatter_(1, labels.view(-1, 1), 1)

        # Combine modified and unmodified cosine similarities
        output = (one_hot * phi) + ((1.0 - one_hot) * cosine)
        output *= self.s  # Scale the logits
        return output

# 4. Initialize the model, loss function, and optimizer
arcface_model = ArcFaceCNN_CIFAR10(feature_dim=128)
arcface_loss = ArcFaceLoss()
optimizer = optim.Adam(arcface_model.parameters(), lr=0.0001)

# Variable to store ArcLoss data (epoch, loss)
ArcLoss_CIFAR10 = []  # List to store (epoch, loss) for each epoch

# 5. Training function with loss tracking
def train_arcface(model, loss_fn, train_loader, optimizer, epochs=10):
    model.train()
    for epoch in range(epochs):
        total_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()
            features = model(images)  # Extract features
            outputs = loss_fn(features, labels)  # Compute ArcFace logits
            loss = nn.CrossEntropyLoss()(outputs, labels)  # Apply cross-entropy loss
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        # Store loss for each epoch in ArcLoss_CIFAR10
        ArcLoss_CIFAR10.append((epoch + 1, total_loss / len(train_loader)))  # Store (epoch, loss)
        print(f"Epoch {epoch + 1}, Loss: {total_loss / len(train_loader):.4f}")

    return ArcLoss_CIFAR10

# 6. Evaluation function
def evaluate(model, loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            features = model(images)  # Extract features
            outputs = arcface_loss(features, labels)  # Compute ArcFace logits
            _, predicted = torch.max(outputs, 1)  # Predict labels
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Accuracy: {accuracy:.2f}%")
    return accuracy

# 7. Train and evaluate the ArcFace model
print("Training with ArcFace Loss:")
```

```
epoch_losses = train_arcface(arcface_model, arcface_loss, train_loader, optimizer, epochs=20)  # Increased epochs for CIFAR-10

arcface_accuracy = evaluate(arcface_model, test_loader)


# 8. Plotting Epoch vs Loss

plt.figure(figsize=(8, 6))

epochs, losses = zip(*ArcLoss_CIFAR10)  # Unzip the list of (epoch, loss) into two separate lists

plt.plot(epochs, losses, marker='o', color='b', label='Training Loss')

plt.title('ArcLoss Epoch vs Loss (CIFAR 10)')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.grid(True)

plt.legend()

plt.show()


# 9. Print the final accuracy

print(f"\nArcFace Model Accuracy on CIFAR-10: {arcface_accuracy:.2f}%")
```

This code trains an **ArcFace model** on the **CIFAR-10 dataset** using a **CNN** and **ArcFace loss**. It involves:

1. Loading and normalizing CIFAR-10 data.
2. Defining a CNN with 3 conv layers and a 128-d feature output.
3. Using **ArcFace loss** to enhance discriminative power.
4. Training with **Adam optimizer** and tracking loss.
5. Evaluating accuracy on the test set.
6. Plotting the loss over epochs.

Finally, it prints the test accuracy and visualizes the loss curve.

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

# Extract epochs and loss values for MNIST
epochs_mnist, AdaLoss_MNIST_vals = zip(*AdaLoss_MNIST)  # Unpack tuples into two lists
_, ArcLoss_MNIST_vals = zip(*ArcLoss_MNIST)

# Extract epochs and loss values for CIFAR-10
epochs_cifar10, AdaLoss_CIFAR10_vals = zip(*AdaLoss_CIFAR10)
_, ArcLoss_CIFAR10_vals = zip(*ArcLoss_CIFAR10)

# Find min and max loss values for both AdaLoss and ArcLoss for MNIST
min_loss_ada_mnist = min(AdaLoss_MNIST_vals)
max_loss_ada_mnist = max(AdaLoss_MNIST_vals)

min_loss_arc_mnist = min(ArcLoss_MNIST_vals)
max_loss_arc_mnist = max(ArcLoss_MNIST_vals)

# Find the epochs corresponding to the min and max loss values for MNIST
min_epoch_ada_mnist = epochs_mnist[AdaLoss_MNIST_vals.index(min_loss_ada_mnist)]
max_epoch_ada_mnist = epochs_mnist[AdaLoss_MNIST_vals.index(max_loss_ada_mnist)]

min_epoch_arc_mnist = epochs_mnist[ArcLoss_MNIST_vals.index(min_loss_arc_mnist)]
max_epoch_arc_mnist = epochs_mnist[ArcLoss_MNIST_vals.index(max_loss_arc_mnist)]

# Find min and max loss values for both AdaLoss and ArcLoss for CIFAR-10
min_loss_ada_cifar10 = min(AdaLoss_CIFAR10_vals)
```

```
max_loss_ada_cifar10 = max(AdaLoss_CIFAR10_vals)

min_loss_arc_cifar10 = min(ArcLoss_CIFAR10_vals)
max_loss_arc_cifar10 = max(ArcLoss_CIFAR10_vals)

# Find the epochs corresponding to the min and max loss values for CIFAR-10
min_epoch_ada_cifar10 = epochs_cifar10[AdaLoss_CIFAR10_vals.index(min_loss_ada_cifar10)]
max_epoch_ada_cifar10 = epochs_cifar10[AdaLoss_CIFAR10_vals.index(max_loss_ada_cifar10)]

min_epoch_arc_cifar10 = epochs_cifar10[ArcLoss_CIFAR10_vals.index(min_loss_arc_cifar10)]
max_epoch_arc_cifar10 = epochs_cifar10[ArcLoss_CIFAR10_vals.index(max_loss_arc_cifar10)]

# Plotting AdaLoss vs ArcLoss for MNIST
plt.figure(figsize=(10, 10))

# Line chart for MNIST
plt.subplot(2, 1, 1)
plt.plot(epochs_mnist, AdaLoss_MNIST_vals, label='AdaLoss', marker='o', color='b')
plt.plot(epochs_mnist, ArcLoss_MNIST_vals, label='ArcLoss', marker='x', color='r')

# Highlight min and max values for AdaLoss and ArcLoss for MNIST
plt.scatter(min_epoch_ada_mnist, min_loss_ada_mnist, color='purple', zorder=5, label=f'Min AdaLoss ({min_epoch_ada_mnist}, {min_loss_ada_mnist:.2f})')
plt.scatter(max_epoch_ada_mnist, max_loss_ada_mnist, color='orange', zorder=5, label=f'Max AdaLoss ({max_epoch_ada_mnist}, {max_loss_ada_mnist:.2f})')
plt.scatter(min_epoch_arc_mnist, min_loss_arc_mnist, color='green', zorder=5, label=f'Min ArcLoss ({min_epoch_arc_mnist}, {min_loss_arc_mnist:.2f})')
plt.scatter(max_epoch_arc_mnist, max_loss_arc_mnist, color='red', zorder=5, label=f'Max ArcLoss ({max_epoch_arc_mnist}, {max_loss_arc_mnist:.2f})')

plt.title('AdaLoss vs ArcLoss on MNIST')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.xticks(epochs_mnist)  # Ensures all epoch values are used as x-ticks
plt.legend()

# Line chart for CIFAR-10
plt.subplot(2, 1, 2)
plt.plot(epochs_cifar10, AdaLoss_CIFAR10_vals, label='AdaLoss', marker='o', color='b')
plt.plot(epochs_cifar10, ArcLoss_CIFAR10_vals, label='ArcLoss', marker='x', color='r')

# Highlight min and max values for AdaLoss and ArcLoss for CIFAR-10
plt.scatter(min_epoch_ada_cifar10, min_loss_ada_cifar10, color='purple', zorder=5, label=f'Min AdaLoss ({min_epoch_ada_cifar10}, {min_loss_ada_cifar10:.2f})')
plt.scatter(max_epoch_ada_cifar10, max_loss_ada_cifar10, color='orange', zorder=5, label=f'Max AdaLoss ({max_epoch_ada_cifar10}, {max_loss_ada_cifar10:.2f})')

plt.scatter(min_epoch_arc_cifar10, min_loss_arc_cifar10, color='green', zorder=5, label=f'Min ArcLoss ({min_epoch_arc_cifar10}, {min_loss_arc_cifar10:.2f})')
plt.scatter(max_epoch_arc_cifar10, max_loss_arc_cifar10, color='red', zorder=5, label=f'Max ArcLoss ({max_epoch_arc_cifar10}, {max_loss_arc_cifar10:.2f})')

plt.title('AdaLoss vs ArcLoss on CIFAR-10')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.xticks(epochs_cifar10)  # Ensures all epoch values are used as x-ticks
plt.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

This code compares **AdaLoss** and **ArcLoss** for **MNIST** and **CIFAR-10** datasets by plotting their loss values over epochs. It highlights the **minimum** and **maximum** loss points for both losses on each dataset, providing insights into the training behavior. Two separate plots are shown: one for MNIST and one for CIFAR-10, with markers indicating the min and max loss values.