

ArcFace: Additive Angular Margin Loss for Deep Face Recognition

Team Members

1. Bobba Shruti
2. Varsha Reddy Chinthalapudi

Project Description:

The project, titled "Arc Face: Additive Angular Margin Loss for Deep Face Recognition," focuses on advancing face recognition capabilities through the implementation of innovative techniques. The process initiates with precise face detection and annotation using the Multi-task Cascaded Convolutional Networks (MTCNN), enhancing the quality of the dataset. The organized dataset undergoes deep learning methodologies, incorporating an embedding model to extract crucial facial features and employing the Arc Face loss function to amplify feature discrimination, thereby elevating recognition accuracy. The dataset is meticulously divided for training, validation, and testing, facilitating thorough performance assessment. The model undergoes systematic fine-tuning, optimizing hyperparameters, architecture, and training dynamics to achieve superior recognition outcomes. This comprehensive approach, integrating computer vision and deep learning, results in a robust face recognition system notable for its accuracy and adaptability, applicable across security, access control, and various domains.

Method:

The method facilitates accurate face detection, keypoint extraction, and bounding box determination, thereby augmenting the overall dataset quality. Subsequently, transfer learning is employed by leveraging pre-trained deep learning models for face recognition. This involves fine-tuning the model on the processed dataset. The method also entails the creation of a specialized deep learning network tailored for face recognition, incorporating an embedding model and utilizing the Arc Face loss function for heightened feature discrimination. The dataset is meticulously divided into training, validation, and test sets to rigorously evaluate the performance of the developed face recognition network. The synthesis of these components results in a comprehensive face recognition system, adept at accurate face detection, model optimization through transfer learning, and the creation of a sophisticated network for robust face recognition, applicable across various domains such as security and access control. The provided code performs preprocessing and keypoints extraction for a face recognition dataset using the MTCNN (Multi-task Cascaded Convolutional Networks) for face detection. Here's a breakdown of the method:

1. The necessary libraries are imported, including MTCNN for face detection, OpenCV for image processing, Torch for deep learning, and other related modules.

```
!pip install mtcnn
import os
import cv2
from mtcnn import MTCNN
```

```
import torch
from torchvision import transforms
from PIL import Image
import json
```

2. The MTCNN model is utilized for face detection, and a preprocessing function is defined. The MTCNN model is initialized to detect faces in images. The preprocessing function takes an image path and MTCNN detector, loads the image, detects faces, and extracts keypoints and bounding box information. It then aligns and resizes the face to a standard size, converting it to a PyTorch tensor. The code iterates through an original dataset, applies the preprocessing, and saves the resulting images and keypoints in designated directories. This process is crucial for creating a well-organized and standardized dataset, a foundational step for training accurate face recognition models.

```
detector = MTCNN()
def preprocess_image(image_path, detector):
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    result = detector.detect_faces(image)
    if result:
        keypoints = result[0]['keypoints']
        bounding_box = result[0]['box']
        x, y, width, height = bounding_box
        face = image[y:y+height, x:x+width]
        face_resized = cv2.resize(face, (112, 112))
        face_tensor = transforms.ToTensor()(face_resized)

        return face_tensor, keypoints
    else:
        return None, None

dataset_path = '/content/drive/MyDrive/DL/data/raw'
new_dataset_path = '/content/drive/MyDrive/DL/data/new_dataset'
keypoints_path = '/content/drive/MyDrive/DL/data/keypoints'
```

3. This code processes each image in a given dataset using the MTCNN face detection model. For each image, it loads the image, detects faces, extracts keypoints, crops and aligns the face, resizes it to 112x112 pixels, and converts it to a PyTorch tensor. The preprocessed face and keypoints are then saved in a new dataset directory, and the keypoints are stored in JSON files.

```
for subdir, dirs, files in os.walk(dataset_path):
    for file in files:
        input_image_path = os.path.join(subdir, file)
```

```

try:
    preprocessed_face, keypoints = preprocess_image(input_image_path, detector)
    if preprocessed_face is not None:
        save_subdir = os.path.relpath(subdir, dataset_path)
        save_dir = os.path.join(new_dataset_path, save_subdir)
        os.makedirs(save_dir, exist_ok=True)
        save_image_path = os.path.join(save_dir, file)
        pil_image = transforms.ToPILImage()(preprocessed_face).convert("RGB")
        pil_image.save(save_image_path)
        keypoints_dir = os.path.join(keypoints_path, save_subdir)
        os.makedirs(keypoints_dir, exist_ok=True)
        keypoints_file = os.path.splitext(file)[0] + '.json'
        keypoints_file_path = os.path.join(keypoints_dir, keypoints_file)
        with open(keypoints_file_path, 'w') as f:
            json.dump(keypoints, f)

    except Exception as e:
        print(f"Error processing {input_image_path}: {e}")

print("Preprocessing and keypoints extraction complete.")

```

The output shows the progress of the processing, indicating the time taken for each image or batch of images. This represents the processing of one image, where "1/1" indicates one image processed.

The completion message "Preprocessing and keypoints extraction complete" indicates the successful execution of the code. This process is essential for creating a standardized dataset with aligned and resized face images, ready for training face recognition models.

```
1/1 [=====] - 1s 768ms/step
1/1 [=====] - 0s 339ms/step
1/1 [=====] - 0s 191ms/step
1/1 [=====] - 0s 107ms/step
1/1 [=====] - 0s 84ms/step
1/1 [=====] - 0s 59ms/step
1/1 [=====] - 0s 43ms/step
1/1 [=====] - 0s 40ms/step
1/1 [=====] - 0s 91ms/step
1/1 [=====] - 0s 46ms/step
1/1 [=====] - 0s 32ms/step
1/1 [=====] - 0s 58ms/step
1/1 [=====] - 0s 39ms/step
34/34 [=====] - 1s 14ms/step
2/2 [=====] - 0s 29ms/step
1/1 [=====] - 4s 4s/step
1/1 [=====] - 1s 1s/step
1/1 [=====] - 1s 501ms/step
1/1 [=====] - 0s 220ms/step
1/1 [=====] - 0s 126ms/step
1/1 [=====] - 0s 70ms/step
1/1 [=====] - 0s 45ms/step
1/1 [=====] - 0s 35ms/step
1/1 [=====] - 0s 32ms/step
1/1 [=====] - 0s 25ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 22ms/step
```

Transfer Learning:

This code performs fine-tuning on a MobileFaceNet model for a face recognition task. This code demonstrates the process of adapting a pre-trained face recognition model to a specific dataset and task through fine-tuning, including hyperparameter tuning for optimization

1. The code begins by importing necessary libraries and defining a custom dataset along with file paths.

Importing Necessary Libraries

```
import os
import pandas as pd
import torch
from torch.utils.data import DataLoader, Dataset, random_split
from torchvision import transforms
from PIL import Image
```

Define Custom Dataset Class

```
class CustomDataset(Dataset):
    def __init__(self, csv_file, root_dir, transform=None):
```

```

self.dataframe = pd.read_csv(csv_file)
self.root_dir = root_dir
self.transform = transform

def __len__(self):
    return len(self.dataframe)

def __getitem__(self, idx):
    img_name = os.path.join(self.root_dir, self.dataframe.iloc[idx, 0])
    image = Image.open(img_name)
    label = int(self.dataframe.iloc[idx, 1])

    if self.transform:
        image = self.transform(image)

    return image, label

```

Paths to the dataset and labels

```

root_dir = '/content/drive/MyDrive/DL/data/new_dataset'
csv_filename = '/content/drive/MyDrive/DL/data/dataset_labels.csv'

```

2. It begins by defining a set of image transformations, including resizing, converting to tensors, and normalization. Subsequently, a custom dataset is initialized using a CSV file containing image labels and a root directory. The size of the dataset is printed, and a sample image is visualized after undergoing the specified transformations. The dataset is then split into training, validation, and test sets using random splitting techniques. DataLoader instances are created for each set, facilitating efficient batch-wise loading of data during the model training process.

Transform

```

transform = transforms.Compose([
    transforms.Resize((112, 112)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

```

Load the data and transform

```
dataset = CustomDataset(csv_file=csv_filename, root_dir=root_dir, transform=transform)
dataset
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
print("Dataset size:", len(dataset))
```

```
Dataset size: 199
```

```
sample_image, sample_label = dataset[0]
print("Sample label:", sample_label)
if isinstance(sample_image, torch.Tensor):
    sample_image = sample_image.permute(1, 2, 0)
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    sample_image = sample_image * torch.tensor(std) + torch.tensor(mean)
    sample_image = torch.clamp(sample_image, 0, 1)
    sample_image = sample_image.numpy()
    plt.imshow(sample_image)
    plt.title(f"Sample Label: {sample_label}")
    plt.axis('off')
    plt.show()
```



Train,Test and Validation split

```
train_val_split = int(len(dataset) * 0.9)
test_split = len(dataset) - train_val_split
train_val_dataset, test_dataset = random_split(dataset, [train_val_split, test_split])

train_split = int(train_val_split * 0.88)
val_split = train_val_split - train_split
train_dataset, val_dataset = random_split(train_val_dataset, [train_split, val_split])
```

Create DataLoaders for train, test and validation split

```
batch_size = 8
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

```
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

3. The presented code defines a convolutional neural network (CNN) architecture named MobileFaceNet for face recognition. The model is structured with various custom blocks, including Conv_block, Linear_block, Depth_Wise, Residual, and MobileFaceNet. These blocks are designed to capture and process features hierarchically, incorporating depth-wise convolutions and residual connections for improved representational learning. The MobileFaceNet architecture aims to embed facial features into a lower-dimensional space, facilitating face recognition tasks. Additionally, the code includes a Flatten module and an L2 normalization function. The model's weights are loaded from a pre-trained state dictionary file, enabling the reuse of knowledge learned from a prior training session. Overall, this code provides the foundation for a face recognition model with a focus on mobile deployment, given its emphasis on efficiency and lightweight architecture.

Model Loading

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn import Linear, Conv2d, BatchNorm1d, BatchNorm2d, PReLU, ReLU, Sigmoid,
Dropout2d, Dropout, AvgPool2d, MaxPool2d, AdaptiveAvgPool2d, Sequential, Module,
Parameter
class Conv_block(nn.Module):
    def __init__(self, in_c, out_c, kernel=(1, 1), stride=(1, 1), padding=(0, 0), groups=1):
        super(Conv_block, self).__init__()
        self.conv = Conv2d(in_c, out_channels=out_c, kernel_size=kernel, groups=groups,
stride=stride, padding=padding, bias=False)
        self.bn = BatchNorm2d(out_c)
        self.prelu = PReLU(out_c)
    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.prelu(x)
        return x

class Linear_block(nn.Module):
    def __init__(self, in_c, out_c, kernel=(1, 1), stride=(1, 1), padding=(0, 0), groups=1):
        super(Linear_block, self).__init__()
        self.conv = Conv2d(in_c, out_channels=out_c, kernel_size=kernel, groups=groups,
stride=stride, padding=padding, bias=False)
        self.bn = BatchNorm2d(out_c)
    def forward(self, x):
```

```

x = self.conv(x)
x = self.bn(x)
return x

```

```

class Depth_Wise(nn.Module):
    def __init__(self, in_c, out_c, residual = False, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=1):
        super(Depth_Wise, self).__init__()
        self.conv = Conv_block(in_c, out_c=groups, kernel=(1, 1), padding=(0, 0), stride=(1, 1))
        self.conv_dw = Conv_block(groups, groups, groups=groups, kernel=kernel,
padding=padding, stride=stride)
        self.project = Linear_block(groups, out_c, kernel=(1, 1), padding=(0, 0), stride=(1, 1))
        self.residual = residual
    def forward(self, x):
        if self.residual:
            short_cut = x
            x = self.conv(x)
            x = self.conv_dw(x)
            x = self.project(x)
        if self.residual:
            output = short_cut + x
        else:
            output = x
        return output

```

```

class Residual(nn.Module):
    def __init__(self, c, num_block, groups, kernel=(3, 3), stride=(1, 1), padding=(1, 1)):
        super(Residual, self).__init__()
        modules = []
        for _ in range(num_block):
            modules.append(Depth_Wise(c, c, residual=True, kernel=kernel, padding=padding,
stride=stride, groups=groups))
        self.model = Sequential(*modules)
    def forward(self, x):
        return self.model(x)

```

```

class MobileFaceNet(nn.Module):
    def __init__(self, embedding_size):
        super(MobileFaceNet, self).__init__()
        self.conv1 = Conv_block(3, 64, kernel=(3, 3), stride=(2, 2), padding=(1, 1))
        self.conv2_dw = Conv_block(64, 64, kernel=(3, 3), stride=(1, 1), padding=(1, 1),
groups=64)
        self.conv_23 = Depth_Wise(64, 64, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=128)

```



```

        self.conv_3 = Residual(64, num_block=4, groups=128, kernel=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv_34 = Depth_Wise(64, 128, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=256)
        self.conv_4 = Residual(128, num_block=6, groups=256, kernel=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv_45 = Depth_Wise(128, 128, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=512)
        self.conv_5 = Residual(128, num_block=2, groups=256, kernel=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv_6_sep = Conv_block(128, 512, kernel=(1, 1), stride=(1, 1), padding=(0, 0))
        self.conv_6_dw = Linear_block(512, 512, groups=512, kernel=(7,7), stride=(1, 1),
padding=(0, 0))
        self.conv_6_flatten = Flatten()
        self.linear = Linear(512, embedding_size, bias=False)
        self.bn = BatchNorm1d(embedding_size)

def forward(self, x):
    out = self.conv1(x)

    out = self.conv2_dw(out)

    out = self.conv_23(out)

    out = self.conv_3(out)

    out = self.conv_34(out)

    out = self.conv_4(out)

    out = self.conv_45(out)

    out = self.conv_5(out)

    out = self.conv_6_sep(out)

    out = self.conv_6_dw(out)

    out = self.conv_6_flatten(out)

    out = self.linear(out)

    out = self.bn(out)
    return l2_norm(out)

```

```

class Flatten(Module):
    def forward(self, input):
        return input.view(input.size(0), -1)

def l2_norm(input,axis=1):
    norm = torch.norm(input,2,axis,True)
    output = torch.div(input, norm)
    return output
model_weights = torch.load("/content/drive/MyDrive/DL/model_mobilefacenet.pth",
map_location=torch.device('cpu'))

model = MobileFaceNet(embedding_size=512)
model.load_state_dict(model_weights)

<All keys matched successfully>

```

4. The code modifies the pre-trained face recognition model, MobileFaceNet, for a specific classification task with 10 classes. It adjusts the final linear layer and adds batch normalization to match the new classification requirements. The training setup includes defining a cross-entropy loss function and using Stochastic Gradient Descent (SGD) with a learning rate of 0.001 and momentum of 0.9. This prepares the model for fine-tuning on a new dataset, ensuring it can adapt to the classification task's demands.

Model Training

```

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import transforms
num_classes=10
in_features = model.linear.in_features
new_linear_layer = nn.Linear(in_features, num_classes)
model.linear = new_linear_layer
new_bn_layer = nn.BatchNorm1d(num_classes)
model.bn = new_bn_layer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

```

5. The code snippet encompasses the training loop for fine-tuning a MobileFaceNet model on a custom dataset. It iterates through multiple epochs, optimizing the model's parameters using a specified criterion and SGD optimizer. Training accuracy and loss are monitored and printed for each epoch. The fine-tuned model is then saved. Subsequently, the saved model is evaluated on a test dataset, and the test loss and accuracy are calculated, providing insights into the model's performance on unseen data.

Training Loop

```
num_epochs=10
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print(f'Epoch {epoch + 1}/{num_epochs}, '
          f'Training Loss: {total_loss / len(train_loader):.4f}, '
          f'Training Accuracy: {(100 * correct / total):.2f} %')
```

Saving the trained Model

```
torch.save(model.state_dict(), '/content/drive/MyDrive/DL/fine_tuned_mobilefacenet.pth')
```

```
print("Fine-tuning completed.")
```



Fine-tuning completed.

Evaluate the Model

```
model.eval()
test_loss = 0.0
correct = 0
total = 0
```

```

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
average_test_loss = test_loss / len(test_loader)
accuracy = 100 * correct / total
print(f'Test Loss: {average_test_loss:.4f}')
print(f'Test Accuracy: {accuracy:.2f}%')

Test Loss: 2.0997
Test Accuracy: 35.00%

```

6. The provided code involves hyperparameter tuning using a grid search approach on learning rates and momentums. It iterates through combinations of specified learning rates and momentums, training the model for a fixed number of epochs on the training set. The model's performance is then evaluated on the validation set, and the combination of hyperparameters resulting in the lowest validation loss is recorded. After the grid search, the best learning rate and momentum are printed. Following this, the model is retrained using the identified optimal hyperparameters, and the training statistics, including loss and accuracy, are printed for each epoch. This process aims to fine-tune the model's hyperparameters for improved performance on the given dataset.

HyperParameter Tuning from Validation set

```

import torch
import torch.nn as nn
import torch.optim as optim
learning_rates = [1e-5, 1e-4, 1e-3, 1e-2]
momentums = [0.9, 0.95, 0.99]
best_lr = None
best_momentum = None
best_validation_loss = float('inf')
for lr in learning_rates:
    for momentum in momentums:
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=momentum)
        num_epochs = 10

```

```

for epoch in range(num_epochs):
    model.train()
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    # Validation loop
    model.eval()
    with torch.no_grad():
        validation_loss = 0.0
        for inputs, labels in val_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            validation_loss += loss.item()

    # Update the best hyperparameters if validation loss is lower
    if validation_loss < best_validation_loss:
        best_validation_loss = validation_loss
        best_lr = lr
        best_momentum = momentum

# Print the best hyperparameters
print(f'Best Learning Rate: {best_lr}')
print(f'Best Momentum: {best_momentum}')

```



```

Best Learning Rate: 0.01
Best Momentum: 0.95

```

Retrain the Model with Best Hyperparameters

```

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.95)
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

```

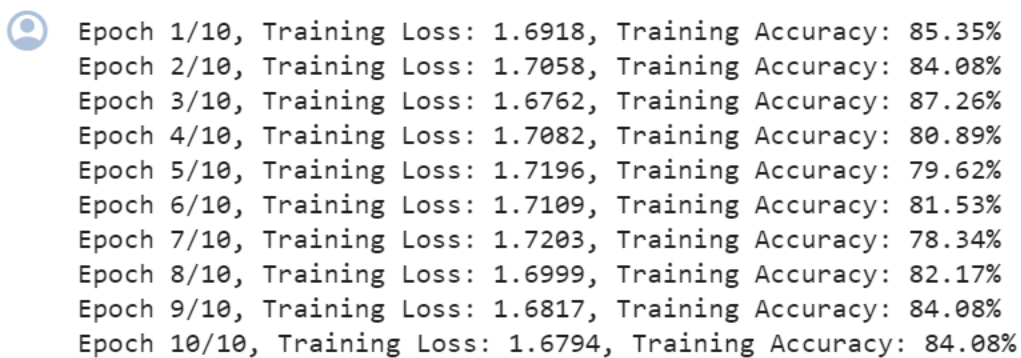
```

total_loss += loss.item()

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

# Print training statistics
print(f'Epoch {epoch + 1}/{num_epochs}, '
      f'Training Loss: {total_loss / len(train_loader):.4f}, '
      f'Training Accuracy: {(100 * correct / total):.2f}%')

```



```

Epoch 1/10, Training Loss: 1.6918, Training Accuracy: 85.35%
Epoch 2/10, Training Loss: 1.7058, Training Accuracy: 84.08%
Epoch 3/10, Training Loss: 1.6762, Training Accuracy: 87.26%
Epoch 4/10, Training Loss: 1.7082, Training Accuracy: 80.89%
Epoch 5/10, Training Loss: 1.7196, Training Accuracy: 79.62%
Epoch 6/10, Training Loss: 1.7109, Training Accuracy: 81.53%
Epoch 7/10, Training Loss: 1.7203, Training Accuracy: 78.34%
Epoch 8/10, Training Loss: 1.6999, Training Accuracy: 82.17%
Epoch 9/10, Training Loss: 1.6817, Training Accuracy: 84.08%
Epoch 10/10, Training Loss: 1.6794, Training Accuracy: 84.08%

```

- The provided code implements a machine learning pipeline for fine-tuning the MobileFaceNet model on a custom dataset. It covers data preprocessing, model loading, training, hyperparameter tuning, and evaluation. The reported results indicate a test loss of 1.9509 and a test accuracy of 50.00%. While there is room for improvement, these results serve as a starting point for refining the model and its performance.

Save the final Model

```

torch.save(model.state_dict(), '/content/drive/MyDrive/DL/final_fine_tuned_mobilefacenet.pth')
print("Fine-tuning completed.")

```

Evaluate with the final model

```

model.eval()

```

```

test_loss = 0.0
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
average_test_loss = test_loss / len(test_loader)
accuracy = 100 * correct / total

print(f'Test Loss: {average_test_loss:.4f}')
print(f'Test Accuracy: {accuracy:.2f} %')

```

End – End ArcFace Model:

1. The provided code initializes a custom dataset class, `CustomDataset`, to load images and labels from a CSV file. It applies transformations to the dataset, such as resizing and normalization. Paths to the dataset and labels are specified, and the dataset is split into training, validation, and test sets.

Import Necessary Libraries

```

import os
import pandas as pd
import math
import torch
from torch.utils.data import DataLoader, Dataset, random_split
from torchvision import transforms

```

```
from PIL import Image
import torch.nn as nn
import torch.optim as optim
from torch.nn import Linear, Conv2d, BatchNorm1d, BatchNorm2d, PReLU, ReLU, Sigmoid,
Dropout2d, Dropout, AvgPool2d, MaxPool2d, AdaptiveAvgPool2d, Sequential, Module,
Parameter
```

Create Custom DataSet Class

```
class CustomDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = Image.open(img_path)
        label = self.img_labels.iloc[idx, 1]

        if self.transform:
            image = self.transform(image)

        return image, label
```

Transform

```
transform = transforms.Compose([
    transforms.Resize((112, 112)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

Paths to the dataset and labels

```
dataset = CustomDataset(annotations_file='/content/drive/MyDrive/DL/data/dataset_labels.csv',
img_dir='/content/drive/MyDrive/DL/data/new_dataset', transform=transform)
```

Train, Test, and Validation Split

```
train_val_split = int(len(dataset) * 0.9)
test_split = len(dataset) - train_val_split
train_val_dataset, test_dataset = random_split(dataset, [train_val_split, test_split])
train_split = int(train_val_split * 0.88)
val_split = train_val_split - train_split
```



```
train_dataset, val_dataset = random_split(train_val_dataset, [train_split, val_split])
```

2. The provided code defines a custom ArcFace model for face recognition. It utilizes a feature extractor, MobileFaceNet, followed by an Arcface module for classification. Additionally, Conv_block, Linear_block, Depth_Wise, Residual, and Flatten modules are implemented for building the architecture.

The ArcFace model incorporates an additive margin softmax loss, aiming to enhance the discriminative power of the learned features. The model is equipped with parameters such as embedding size, class number, margin (m), and scalar (s).

The training process involves defining data loaders for the training, validation, and test sets. The model is trained using a specified criterion (Arcface loss), optimizer (SGD), and a learning rate scheduler

The MobileFaceNet, responsible for feature extraction, consists of various convolutional and depth-wise layers to capture facial features effectively.

Define Data Loaders for train, test and validation dataset

```
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

Define Model

```
class Arcface(nn.Module):
    def __init__(self, embedding_size=512, classnum=10, s=64., m=0.5):
        super(Arcface, self).__init__()
        self.classnum = classnum
        self.kernel = Parameter(torch.Tensor(embedding_size, classnum))
        self.kernel.data.uniform_(-1, 1).renorm_(2, 1, 1e-5).mul_(1e5)
        self.m = m
        self.s = s
        self.cos_m = math.cos(m)
        self.sin_m = math.sin(m)
        self.mm = self.sin_m * m
        self.threshold = math.cos(math.pi - m)
    def forward(self, embeddings, label):
        nB = len(embeddings)
        kernel_norm = l2_norm(self.kernel, axis=0)
        cos_theta = torch.mm(embeddings, kernel_norm)
        cos_theta = cos_theta.clamp(-1, 1)
        cos_theta_2 = torch.pow(cos_theta, 2)
        sin_theta_2 = 1 - cos_theta_2
```

```

sin_theta = torch.sqrt(sin_theta_2)
cos_theta_m = (cos_theta * self.cos_m - sin_theta * self.sin_m)

cond_v = cos_theta - self.threshold
cond_mask = cond_v <= 0
keep_val = (cos_theta - self.mm)
cos_theta_m[cond_mask] = keep_val[cond_mask]
output = cos_theta * 1.0
idx_ = torch.arange(0, nB, dtype=torch.long)
output[idx_, label] = cos_theta_m[idx_, label]
output *= self.s
return output
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.nn import Linear, Conv2d, BatchNorm1d, BatchNorm2d, PReLU, ReLU, Sigmoid,
Dropout2d, Dropout, AvgPool2d, MaxPool2d, AdaptiveAvgPool2d, Sequential, Module,
Parameter
class CustomArcfaceModel(nn.Module):
    def __init__(self, num_classes, embedding_size, s, m, feature_extractor):
        super(CustomArcfaceModel, self).__init__()
        self.num_classes = num_classes
        self.embedding_size = embedding_size
        self.s = s
        self.m = m
        self.feature_extractor = feature_extractor
        self.arcface = Arcface(embedding_size, num_classes, s, m)
        self.linear = nn.Linear(1024, embedding_size, bias=False)

    def forward(self, x, labels):
        embeddings = self.feature_extractor(x)
        logits = self.arcface(embeddings, labels)
        return logits
class Conv_block(nn.Module):
    def __init__(self, in_c, out_c, kernel=(1, 1), stride=(1, 1), padding=(0, 0), groups=1):
        super(Conv_block, self).__init__()
        self.conv = Conv2d(in_c, out_channels=out_c, kernel_size=kernel, groups=groups,
stride=stride, padding=padding, bias=False)
        self.bn = BatchNorm2d(out_c)
        self.prelu = PReLU(out_c)
    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.prelu(x)

```

```

        return x

class Linear_block(nn.Module):
    def __init__(self, in_c, out_c, kernel=(1, 1), stride=(1, 1), padding=(0, 0), groups=1):
        super(Linear_block, self).__init__()
        self.conv = Conv2d(in_c, out_channels=out_c, kernel_size=kernel, groups=groups,
stride=stride, padding=padding, bias=False)
        self.bn = BatchNorm2d(out_c)
    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        return x

class Depth_Wise(nn.Module):
    def __init__(self, in_c, out_c, residual = False, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=1):
        super(Depth_Wise, self).__init__()
        self.conv = Conv_block(in_c, out_c=groups, kernel=(1, 1), padding=(0, 0), stride=(1, 1))
        self.conv_dw = Conv_block(groups, groups, groups=groups, kernel=kernel,
padding=padding, stride=stride)
        self.project = Linear_block(groups, out_c, kernel=(1, 1), padding=(0, 0), stride=(1, 1))
        self.residual = residual
    def forward(self, x):
        if self.residual:
            short_cut = x
            x = self.conv(x)
            x = self.conv_dw(x)
            x = self.project(x)
            if self.residual:
                output = short_cut + x
            else:
                output = x
        return output

class Residual(nn.Module):
    def __init__(self, c, num_block, groups, kernel=(3, 3), stride=(1, 1), padding=(1, 1)):
        super(Residual, self).__init__()
        modules = []
        for _ in range(num_block):
            modules.append(Depth_Wise(c, c, residual=True, kernel=kernel, padding=padding,
stride=stride, groups=groups))
        self.model = Sequential(*modules)
    def forward(self, x):
        return self.model(x)

```

MobileFaceNet for feature extraction

```

class MobileFaceNet(nn.Module):
    def __init__(self, embedding_size):
        super(MobileFaceNet, self).__init__()
        self.conv1 = Conv_block(3, 64, kernel=(3, 3), stride=(2, 2), padding=(1, 1))
        self.conv2_dw = Conv_block(64, 64, kernel=(3, 3), stride=(1, 1), padding=(1, 1),
groups=64)
        self.conv_23 = Depth_Wise(64, 64, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=128)
        self.conv_3 = Residual(64, num_block=4, groups=128, kernel=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv_34 = Depth_Wise(64, 128, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=256)
        self.conv_4 = Residual(128, num_block=6, groups=256, kernel=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv_45 = Depth_Wise(128, 128, kernel=(3, 3), stride=(2, 2), padding=(1, 1),
groups=512)
        self.conv_5 = Residual(128, num_block=2, groups=256, kernel=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv_6_sep = Conv_block(128, 512, kernel=(1, 1), stride=(1, 1), padding=(0, 0))
        self.conv_6_dw = Linear_block(512, 512, groups=512, kernel=(7,7), stride=(1, 1),
padding=(0, 0))
        self.conv_6_flatten = Flatten()
        self.linear = Linear(512, embedding_size, bias=False)
        self.bn = BatchNorm1d(embedding_size)

    def forward(self, x):
        out = self.conv1(x)

        out = self.conv2_dw(out)

        out = self.conv_23(out)

        out = self.conv_3(out)

        out = self.conv_34(out)

        out = self.conv_4(out)

        out = self.conv_45(out)

        out = self.conv_5(out)

        out = self.conv_6_sep(out)

```

```

        out = self.conv_6_dw(out)

        out = self.conv_6_flatten(out)

        out = self.linear(out)

        out = self.bn(out)

    return out

class Flatten(nn.Module):
    def forward(self, input):
        return input.view(input.size(0), -1)

def l2_norm(input,axis=1):
    norm = torch.norm(input,2,axis,True)
    output = torch.div(input, norm)
    return output

num_classes = 10
embedding_size = 512
s = 64.0
m = 0.5

```

3. The provided code establishes an end-to-end ArcFace model for face recognition. The model architecture incorporates a MobileFaceNet feature extractor with an embedding size of 512. It employs the Arcface loss function and is optimized using Stochastic Gradient Descent with specified learning rate and momentum. The training loop executes over 50 epochs, iteratively updating the model's parameters and printing training statistics. Following training, the model is evaluated on a test dataset, yielding an accuracy percentage. The output tensor showcases the raw predictions for each class in a batch of test inputs. This approach encapsulates an end-to-end pipeline for training and evaluating an ArcFace model, with the effectiveness contingent on dataset quality, hyperparameter tuning, and feature extractor performance.

Define optimizer

```

# Define optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.95)
in_features = model.linear.in_features
new_linear_layer = nn.Linear(in_features, num_classes)
model.linear = new_linear_layer

new_bn_layer = nn.BatchNorm1d(num_classes)

```

```
model.bn = new_bn_layer
```

Training Loop

```
# Training loop
num_epochs = 50
from torchvision.io import read_image
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs, labels)
        print(outputs)
        print(labels)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

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

# Print training statistics
print(f'Epoch {epoch + 1}/{num_epochs}, '
      f'Training Loss: {total_loss / len(train_loader):.4f}, '
      f'Training Accuracy: {(100 * correct / total):.2f} %')
```

```

tensor([[ -79.3416,  43.0896, -33.5216, -64.0000, -64.0000,  64.0000, -64.0000,
          29.0857,  64.0000, -56.8434],
 [ 64.0000,  64.0000, 11.8680,  56.1653,  64.0000, -64.0000,  64.0000,
 -64.0000, -64.0000, -64.0000],
 [ 64.0000,  64.0000,  64.0000,  64.0000, -64.0000,  64.0000,   7.6077,
  64.0000, -64.0000,  56.1653],
 [ 64.0000,  56.1653, -5.8880, -20.6224,   7.2180,  64.0000, -64.0000,
  64.0000, -64.0000, -32.3123],
 [ 64.0000, -64.0000, -37.6514, -64.0000,  64.0000, -64.0000,  56.1653,
 -64.0000, -64.0000, -64.0000],
 [ 61.9605,  64.0000,  64.0000,  64.0000, -44.2315,  60.1712, -58.1705,
 44.0137, -64.0000,  56.1653],
 [ 64.0000, -64.0000, -64.0000, -79.3416,  64.0000, -64.0000,  64.0000,
 -64.0000,  64.0000, -64.0000],
 [-64.0000, -79.3416,  64.0000,  64.0000, -64.0000, -24.6774, -64.0000,
 -8.3893,  64.0000,  64.0000],
 [   0.7787,  64.0000,   8.8904, 19.2436, -64.0000,  56.1653, -64.0000,
  64.0000, -34.4095, -53.3075],
 [-64.0000, -64.0000,  64.0000, -30.3779, -64.0000, -64.0000, -64.0000,
 -54.1781,  56.1653, 22.4963],
 [-64.0000, 21.3014, -48.5249, -64.0000, -27.3497,  64.0000, -64.0000,
  64.0000, -57.5923, -35.6693],
 [-60.6681, -47.7761,  64.0000, -20.6293, -64.0000,  56.1653, -64.0000,
  64.0000,  64.0000, -8.1000],
 [-10.9135,  64.0000, -51.3464, -24.8996,  56.1653,  64.0000, 17.5154,
  64.0000, -64.0000,   6.5692],
 [ 50.3605,  64.0000,  64.0000,  64.0000,  64.0000,  64.0000,  56.1653,
  64.0000, -64.0000, -64.0000],
 [ 20.9939,  64.0000,  40.5093, -14.0741,  64.0000,  56.1653, -64.0000,
  64.0000, -61.9624, -64.0000],
 [ 64.0000,  64.0000,  56.1653,  64.0000,  64.0000,  64.0000,  64.0000,
  60.4282, -64.0000, -64.0000],
 [-57.4443,  64.0000,  64.0000,  64.0000, -64.0000,  64.0000, -64.0000,
  64.0000,  56.1653, -57.3539],
 [-64.0000,  50.4222,  64.0000, -64.0000, -64.0000,  56.1653, -64.0000,
  64.0000,  64.0000, -60.6627],
 [ 64.0000,  64.0000,  64.0000,  64.0000,  56.1653,  64.0000, 47.5099,
  64.0000, -64.0000, -62.6082]], grad_fn=<MulBackward0>)
tensor([9, 4, 3, 9, 9, 5, 1, 3, 9, 7, 0, 5, 2, 6, 5, 0, 3, 7, 8, 6, 3, 9, 9, 9,
        2, 5, 0, 5, 2, 8, 5, 4])

```

```

model.eval()
total_correct = 0
total_samples = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        logits = model(inputs, labels)
        _, predicted = torch.max(logits, 1)
        total_samples += labels.size(0)
        total_correct += (predicted == labels).sum().item()

test_accuracy = total_correct / total_samples
print(f'Test Accuracy: {test_accuracy * 100:.2f} %')

```

