Practice School - I Report

Low Compute in-Browser Face Recognition

Submitted in partial fulfillment of the requirements of
BITS C221 Practice School - I

By

Yatharth Singh
ID No. 2022A7PS0146P

Under the supervision of Faculty-in-Charge:
Dr. Ankur Pachauri

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
June 2024

Birla Institute of Technology and Science Pilani, Pilani Campus

Low Compute in-Browser Face Recognition

Practice School - I Report

Submitted by

Yatharth Singh

ID No. 2022A7PS0146P

PS Station:
ParallelDots

Faculty-in-Charge:
Dr. Ankur Pachauri

PS Mentors:
Madhur Sharma (Data Scientist), **Sayalee Bendgude** (Associate Data Scientist), **Muktabh Mayank** (Chief Data Scientist, Co-founder)

Birla Institute of Technology and Science Pilani, Pilani Campus
June 2024

Acknowledgements

I am deeply grateful to everyone who has supported me throughout the first half of my internship at ParallelDots. First and foremost, I would like to sincerely thank **Dr. Ankur Pachauri**, my Faculty-in-Charge, for his continuous engagement and invaluable feedback. His guidance has been crucial in shaping the direction and quality of my work up to this point.

I am also thankful to the entire team at ParallelDots for fostering a collaborative and stimulating environment. I want to extend special gratitude to my project mentors, **Madhur Sharma** and **Sayalee Bendgude**, for their enthusiasm and readiness to share their knowledge, and especially to the ParallelDots's chief data scientist and co-founder, **Muktabh Mayank**, for being as helpful and aproachable as he was, and for going out of his way to help us out with our work. This was a truly wonderful experience working and their support has significantly enhanced my learning involved in my project.

I extend a heartfelt thank you to my fellow interns for their support, camaraderia and the shared learning experiences. Their enthusiasm and collaborative spirit has made this internship both productive and enjoyable. The experiences and skills we have gained will undoubtedly shape my future endeavors as an aspiring machine learning researcher. Thank you once again for the invaluable opportunity and for contributing to our personal and professional growth.

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Low Compute in-Browser Face Recognition

Practice School - I Report

by Yatharth Singh

(2022A7PS0146P)

PS Station: **ParallelDots, Gurgaon**
Faculty-in-Charge: **Dr. Ankur Pachauri**
Duration: **May 28, 2024 – July 23, 2024**
PS Mentors: **Madhur Sharma, Sayalee Bendgude, Muktabh Mayank**
Project Areas: **Deep Learning, Deep Metric Learning, Computer Vision**

This report details a low-compute face recognition project using lightweight models, specifically ConvNeXt v2 Atto and MobileNet v4 Small, with ArcFace loss for improved metric learning. Significant progress has been made in the first four weeks, including data pipeline development, dataset preparation, and model training.

In week one, we set up the timm library, explored the Celebrity Face Dataset, and established a download and face extraction pipeline. Week two involved cleaning the dataset, creating a Train-Val-Test split, and developing a training code skeleton. Week three added validation, testing, checkpointing, dynamic learning rate adjustment, and early stopping, achieving 94% top-1 accuracy on a small dataset subset.

By week four, TensorBoard logging was integrated, hyperparameter tuning was conducted, and multiple models were tested. ConvNeXt Atto achieved 96.1% top-1 accuracy, and MobileNet v4 Small reached 97.2% on the limited dataset. Full dataset training with ConvNeXt Atto achieved 80.18% top-1 accuracy over 15 epochs (still under training).

Future work will focus on refining training, testing additional models, and developing a deployment pipeline. This project shows the potential for high-performance face recognition on low-compute resources, emphasizing systematic testing, model optimization, and robust pipeline development.

Signature of Student

Contents

Chapter 1

Introduction

Face recognition technology is integral to modern applications in security, authentication, and personalized user experiences. The challenge lies in deploying high-performing face recognition systems on devices with limited computational resources. This project addresses this challenge by focusing on deploying lightweight yet high-accuracy models for face recognition.

Our project utilizes state-of-the-art models like ConvNeXt Atto and the newly released MobileNet v4 Small. These models are optimized for computational efficiency without compromising accuracy, making them ideal for deployment on resource-constrained devices. We employ ArcFace loss, a sophisticated deep metric learning technique, to enhance recognition performance by maximizing inter-class variance and minimizing intra-class variance.

The initial phase of this project involved extensive preparation, including setting up a robust data pipeline and handling the Celebrity Face Dataset. Key tasks included developing a face extraction pipeline, cleaning the dataset, and creating a training and evaluation framework.

This project also emphasizes the integration of machine learning operations (MLOps) to streamline the deployment pipeline. By combining systematic model optimization with efficient deployment strategies, we aim to deliver a reliable face recognition system capable of running on low-compute devices.

The following sections will outline our methodology, detailing the steps taken to prepare, train, and deploy our models. We will also discuss the challenges faced and solutions implemented, with the ultimate goal of achieving a deployable, high-performance face recognition system suitable to be deployed in-browser and on custom client datasets.

Chapter 2

Dataset Preparation

This chapter will detail the steps taken to prepare the Celebrity Face Dataset for training. It will cover the process of downloading the dataset, handling connection timeouts, and cleaning the dataset to ensure high-quality data. The chapter will also discuss the importance of a robust data pipeline in the context of face recognition projects.

2.1 Dataset Selection

The dataset used for this project is the Celebrity Face Dataset [2], sourced from GitHub. This dataset comprises images of 1087 different celebrities scraped from Google Images and is available under an open license, allowing for commercial use. The choice of this dataset was driven by its diversity and the licensing flexibility, making it suitable for developing and deploying face recognition models.

The Celebrity Face Dataset contains a wide range and a huge number of images totalling over 8 lakhs, providing a comprehensive basis for training robust face recognition models. However, the nature of web-scraped data introduces challenges such as varying image quality and potential inaccuracies in labels. Addressing these challenges was a critical part of the dataset preparation process.

2.2 Dataset Download

Since we were using a remote server over SSH, we could not just download the files using the Google Drive website, so we decided to automate the downloading of the dataset using the `gdown` Python library as seen in the code below. We use the `zipfile` module to automatically extract the downloaded files and the `os` module to remove the original zip files.

```python
import gdown
import zipfile
import os


def get_download_link(URL):
    file_id = URL.split('/')[-2]
    download_link = f"https://drive.google.com/uc?id={file_id}"
    return download_link


def download_dataset(URL, i):
    URL = get_download_link(URL)
    gdown.download(URL, output=f'data-{i}.zip', quiet=False)

    with zipfile.ZipFile(f'data-{i}.zip', 'r') as zip_ref:
        zip_ref.extractall('.')

    os.remove(f'data-{i}.zip')


URLs = [
    "https://drive.google.com/file/d/0B5G8pYUQMNZnWHJpX0t0XzhNSGM/view?usp=drive_link&resourcekey=0-KhDR2QuKg1Rry5Q3wusJKg", # part1.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnSjhCbWNlb1BXZWs/view?usp=drive_link&resourcekey=0-yVzlvxG007A4R6h7WPWKGw", # part2.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnazFWT2R2b1N5M1k/view?usp=drive_link&resourcekey=0-6IKcx-bXog0QdllPTFGpkA", # part3.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZncjl3ZXVrSGxrY28/view?usp=drive_link&resourcekey=0-pNqvGDu19eByEARmrHg1_w", # part4.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnUW1BZ2ZJWFNkZlU/view?usp=drive_link&resourcekey=0-RCGP25I14Q1vpx2482FpNg", # part5.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnQjJReE00UjhMZFE/view?usp=drive_link&resourcekey=0-PlvVeH22o4SmMTabCv8ouQ", # part6.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZna1ZYeW9YRHpNY1k/view?usp=drive_link&resourcekey=0-Sqzrvbu-KM5hplLzIZ4LmA", # part7.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnN0tPSi16RzYtMGM/view?usp=drive_link&resourcekey=0-D8q6Gbcfox2OvloC6WQ1mA", # part8.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnUUQxUVNDN19nQ0E/view?usp=drive_link&resourcekey=0-lA7VEZ0WAx7xuwmnFmu_jQ", # part9.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZncVVsOTBkNnJITHc/view?usp=drive_link&resourcekey=0-vHjrzFNxBLNiB2sceV5Jog", # part10.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnREFUS1FGZktUejg/view?usp=drive_link&resourcekey=0-RpkScc6MNRhEN_lQHb39ZQ", # part11.zip
    "https://drive.google.com/file/d/0B5G8pYUQMNZnbjlVSVh6V21xcE0/view?usp=drive_link&resourcekey=0-dnDcZ5A9CrXfgLuv1c0sJQ"] # part12.zip


for i, URL in enumerate(URLs):
    download_dataset(URL, i+1)
    print("\033[92mDownloaded data-{i+1}.zip\033[0m")

print("\033[92mAll files downloaded successfully!\033[0m")
```

FIGURE 2.1: Python code for downloading and extracting the dataset

However, we were met an issue when we realised the download would always stop at exactly 3605 seconds and give out a `ConnectionError`. I browsed the internet for hints but could not find anything, so I raised an issue on GitHub, but to no avail.

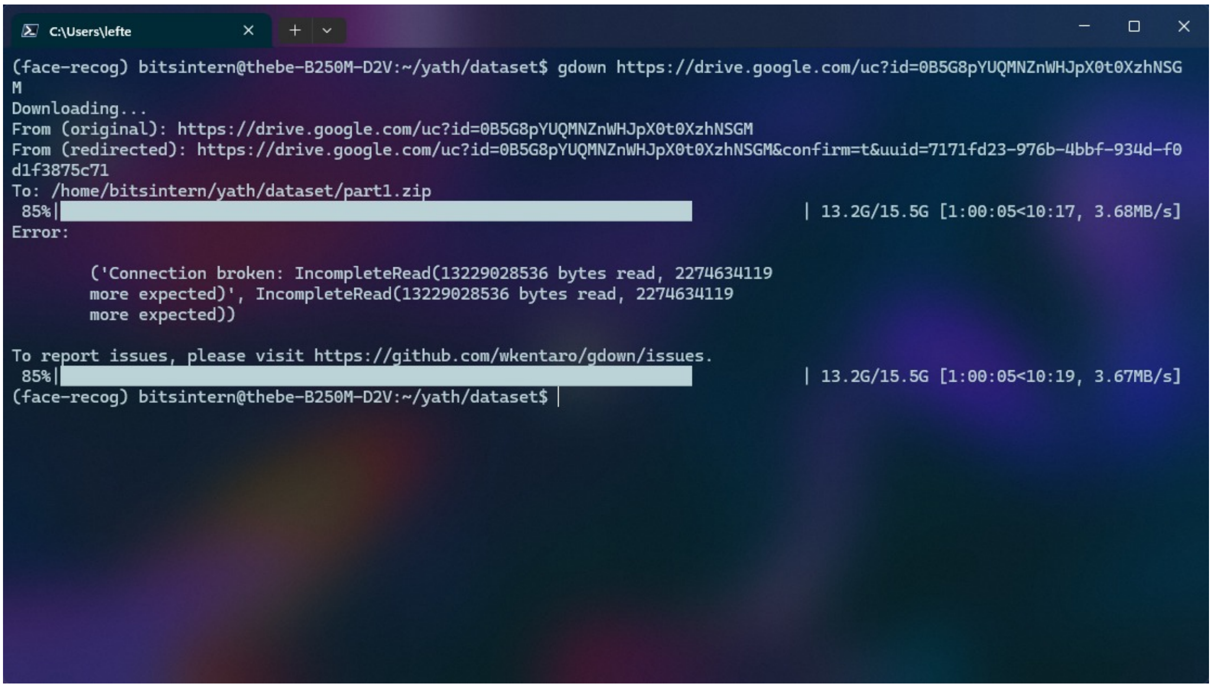

FIGURE 2.2: GDown Connection Error at exactly 3605 seconds

So we decided to manually download each file and once the download stopped, we'd continue the download from the downloaded temp file using the `--resume` flag. Since this forced us to raise too many requests, we were also hit by GDrive API limits ultimately hindering our downloading progress. Thereafter, instead of downloading the complete dataset, we downloaded one part of the dataset and continue developing the face extraction pipeline while we downloaded the rest of the dataset.

2.3 Dataset Cleaning

Upon writing a rudimentary dataloader I found that some files were corrupted and unreadable by the system. So I wrote a short python script to iterate over all the images and log and delete any file that was not a proper image, as seen in the code below.

cleaning.py M

dataset > cleaning.py > ...

You, 9 seconds ago | 1 author (You)

```python
import os
import re
from PIL import Image

def remove_corrupt_jpg_files(directory):
    count = 0
    for root, dirs, files in os.walk(directory):
        for file in files:
            if not re.match(r'.*\.jpg', file):
                file_path = os.path.join(root, file)
                # os.remove(file_path)
                print('Removed', file_path)
    print('\nTotal removed:', count)

def is_jpg_corrupt(file):
    try:
        img = Image.open(file)
        img.verify()
    except (IOError, SyntaxError) as e:
        return True
    return False

def remove_corrupt_jpg_files(directory):
    count = 0
    for root, dirs, files in os.walk(directory):
        for file in files:
            if re.match(r'.*\.jpg', file):
                file_path = os.path.join(root, file)
                if is_jpg_corrupt(file_path):
                    os.remove(file_path)
                    count += 1
                    print('Removed', file_path)
    print('\nTotal removed:', count)

directory = "part-1"
remove_corrupt_jpg_files(directory)      You, 3 weeks ago • add dataset cleaning code
```

Figure 2.3: Dataset cleaning code

Chapter 3

Face Extraction

This chapter explains the development of the face extraction pipeline, crucial for preparing the images for training. The process involved using the `dlib` and `face-recognition` libraries to detect and crop faces from the dataset images. The chapter also discusses challenges encountered, such as handling small images and installation issues.

3.1 Face Extraction Pipeline

The face extraction pipeline employed the `face-recognition` library, leveraging its integration with `dlib` for face detection and manipulation tasks. This approach facilitated efficient face extraction without the need for direct use of `dlib` functionalities.

General Steps in the Face Extraction Process:

3.1.1 Face Detection:

- Utilizing the `face-recognition` library, the pipeline performed face detection on each image in the dataset.
- The library's `dlib` backend enabled accurate detection of facial features and bounding boxes.

3.1.2 Face Cropping:

- After detection, faces were cropped from the images based on the computed bounding boxes.

- The pipeline ensured that the original dimensions of the images were maintained to preserve the integrity and quality of the extracted faces.

Note that we do not reduce the resolution of the image at any point since the dataset has images of varying sizes and some face crops might have very small dimensions, which we do not want to lose out any information from.

3.2 Removing Noise

Some images in the dataset had multiple faces (noise due to being scraped from Google Images), but since they had only one class label there was no simple way to extract only the correct face from the image. Hence, we decided to drop any such images, and it didn't affect our dataset much since the frequency of such images was anyways not very high.

Some images also had incorrect attributions and either belonged to some other class or no class in the dataset at all. There was hoewever no simple way to clean them out, and since the frequency of such images was anyways fairly low (empirically about 3%), we left them in the dataset hoping they would not meddle with the feature extraction.

3.3 CMake Installation Bug

During our installation of the `dlib` library which needed `cmake` to build, we noticed that the `pip install cmake` installation could not be used to install `dlib==19.24.4` but it could successfully install `dlib==19.24.2`. We raised this issue on GitHub and it was confirmed by other users. To fix this, we instead resorted to using the `apt cmake` for our `dlib` installation.

```python
import os
from pathlib import Path

import numpy as np
from PIL import Image
import face_recognition

from tqdm import tqdm


image_dir = Path('../dataset')
face_dir = Path('../faces')
os.makedirs(face_dir, exist_ok=True)


total_images = 0
for root, dirs, files in os.walk(image_dir):
    total_images += len(files)

print(f'Total Images: {total_images}.')


count = 1
for root, dirs, files in os.walk(image_dir):
# for root, dirs, files in tqdm(os.walk(image_dir)):
    for filename in files:
        if filename.endswith('.jpg') or filename.endswith('.png') or filename.endswith('.webp'):

            if count%100 == 0:
                print(f"Processing Image: {count}/{total_images}")
            count += 1

            image_path = os.path.join(root, filename)
            image = Image.open(image_path).convert('RGB')

            image = np.array(image)
            face_box = face_recognition.face_locations(image)

            if len(face_box) > 1:
                continue

            for i in range(len(face_box)):
                top, right, bottom, left = face_box[i]
                image_array = np.array(image)
                face_image = image_array[top+1:bottom, left+1:right]
                pil_image = Image.fromarray(face_image)

                base_filename, _ = os.path.splitext(filename)
                class_name = os.path.basename(root)
                class_dir = os.path.join(face_dir, class_name)
                os.makedirs(class_dir, exist_ok=True)

                target_path = os.path.join(class_dir, f"img-{base_filename}-face-{i+1}.jpg")
                pil_image.save(target_path)
```

FIGURE 3.1: Dataset cleaning code

Chapter 4

Training & Testing

This chapter provides a detailed account of the training phase in the development of a high-performance, lightweight face recognition system. It covers the utilization of the timm library for model management, the implementation of ConvNeXt Atto [3] and MobileNet v4 Small [4] models, integration of ArcFace loss for deep metric learning, and leveraging the PyTorch Metric Learning library. The chapter emphasizes the critical role of systematic model optimization and evaluation in achieving superior face recognition accuracy on low-compute devices.

4.1 Model Selection and Configuration

The choice of ConvNeXt v2 Atto (3.3M params) and MobileNet v4 Small (2.2M params) models was driven by their suitability for deployment in resource-constrained environments, balancing computational efficiency with high-performance face recognition capabilities, and ready availability on the `timm` library. We use the ImageNet-pretrained versions of the models since we expect better feature extraction and hence faster convergence with them.

We start training with these but in case they tend to overfit, we remain open to using their heavier siblings like ConvNeXt v2 femto/pico and MobileNet v4 Medium.

```
You, 16 seconds ago | 1 author (You)
class ConvNeXtArcFace(nn.Module):
    def __init__(self, model_name, pretrained=True):          You, 1 second ago •
        super(ConvNeXtArcFace, self).__init__()
        self.convnext = timm.create_model(model_name, pretrained=pretrained)
        self.convnext.reset_classifier(num_classes=0, global_pool='avg')

    def forward(self, x):
        x = self.convnext.forward_features(x)
        embeddings = F.avg_pool2d(x, 7).flatten(1)
        return embeddings
```

FIGURE 4.1: Creating Model using Timm

Here, when we reset the final classifier, we have an output of a batch_size $\times$ 960 $\times$ 7 $\times$ 7 tensor, which we pool and squeeze to a batch_size $\times$ 960 tensor and pass to the `pytorch-metric-learning` ArcFace Loss constructor. (Note that 960 is for MobileNet v4 small. For ConvNeXt v2 Atto, the dimension is 320.)

4.2 ArcFace Loss

We use ArcFace loss due to its discriminative power of face recognition models by optimizing feature embeddings to maximize inter-class separability. This section delves into the theoretical underpinnings of ArcFace loss and its practical implementation within the training framework.

ArcFace loss addresses the limitations of traditional softmax-based classification by directly modeling angular margins between feature vectors. It achieves this by mapping each class label to a unique angular region on the hypersphere of the embedding space. This approach encourages the model to learn embeddings that are not only class-discriminative but also maintain a large angular distance between different classes, thereby improving classification accuracy.

The ArcFace loss function is given by:

$$\mathcal{L}_{\text{ArcFace}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\cdot\cos(\theta_{y_i,i}+m)}}{e^{s\cdot\cos(\theta_{y_i,i}+m)}+\sum_{j\neq y_i}e^{s\cdot\cos(\theta_{j,i})}}$$

Where:

- N is the batch size,
- s is the scaling factor,

- m is the margin parameter,
- y_i is the true class label of the i-th sample,
- $\theta_{j,i}$ is the angle between the embeddings of sample i and class j.

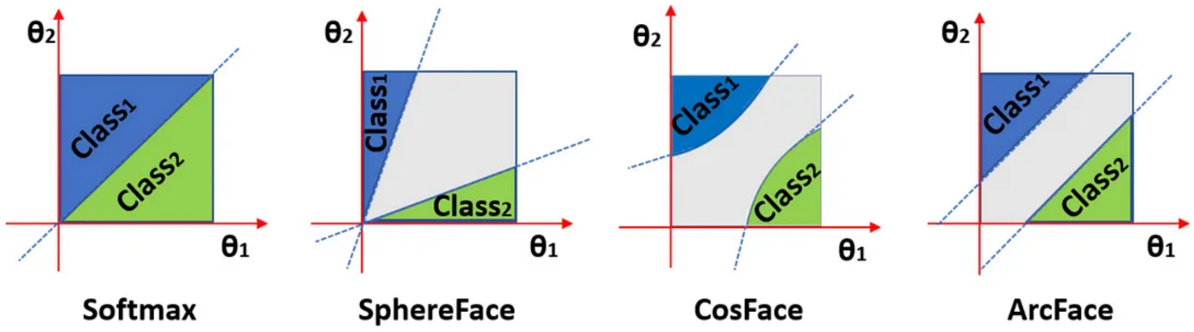

FIGURE 4.2: ArcFace Loss Decision Boundary as compared to other loss functions

Since ArcFace Loss allows us to learn sort of a similarity between two images, we can use it for one-shot learning when dealing with custom client dataset. We use the `pytorch-metric-learning` [6] implementation of ArcFace loss.

4.3 Adding More Features

After the training skeleton is ready, we move on and implement some features that would help us in our training and testing process.

4.3.1 Checkpointing

We store the number of epochs passed, state dictionaries of the model, the optimizer, scheduler etc. so that we can not only use perform inference from a model checkpoint but also continue training from any state. We do not checkpoint at every epoch, but at a predefined list of epochs and when the current accuracy is the best accuracy.

```
if (epoch) in ckpt:
    torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'loss_optimizer_state_dict': loss_optimizer.state_dict(),
                'loss_scheduler_state_dict': loss_scheduler.state_dict(),
                'criterion_state_dict': criterion.state_dict(),
                'loss': running_loss,
                }, f"checkpoints/epoch_{epoch}.pth")
```

FIGURE 4.3: Saving Checkpoint

We also make a `load_checkpoint()` function that can load from a `pth` checkpoint file.

```python
def load_checkpoint(filepath, model, optimizer, scheduler, loss_optimizer, loss_scheduler, criterion):
    checkpoint = torch.load(filepath)
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    scheduler.load_state_dict(checkpoint['scheduler_state_dict'])
    criterion.load_state_dict(checkpoint['criterion_state_dict'])
    loss_optimizer.load_state_dict(checkpoint['loss_optimizer_state_dict'])
    loss_scheduler.load_state_dict(checkpoint['loss_scheduler_state_dict'])
    epoch = checkpoint['epoch'] + 1
    loss = checkpoint['loss']
    return model, optimizer, scheduler, loss_optimizer, loss_scheduler, criterion, epoch, loss

checkpoint = None
if checkpoint:
    model, optimizer, scheduler, loss_optimizer, loss_scheduler, criterion, start_epoch, loss = load_checkpoint(
        f"checkpoints/{checkpoint}", model, optimizer, scheduler, loss_optimizer, loss_scheduler, criterion
        )
```

Figure 4.4: Loading Checkpoint

4.3.2 Logging

We use TensorBoard for logging. We log all the hyperparameters each run as well as the Testing Accuracy, Validation Accuracy and Loss. Since we might not want to log every epoch, we have a `log` parameter that we can toggle on or off.

Figure 4.5: Logging Hyperparameters

```python
if log:
    writer.add_scalar('Accuracy/Training', training_accuracy, epoch)
    writer.add_scalar('Accuracy/Validation', validation_accuracy, epoch)
```

Figure 4.6: Logging Accuracy

4.3.3 Dynamic Learning Rate

We use PyTorch's `ReduceLRonPleateau()` scheduler to implement a dynamic learning rate. We use gamma=0.3 and patience=5 since these tend to give us optimal results.

```python
model_name = 'mobilenetv4_conv_small'
model = ConvNeXtArcFace(model_name, embedding_size)
model = model.to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
scheduler = ReduceLROnPlateau(optimizer, mode='max', factor=factor, patience=patience, verbose=True, threshold=delta)
criterion = losses.ArcFaceLoss(num_classes=num_classes, embedding_size=embedding_size, margin=4).to(device)
loss_optimizer = optim.Adam(criterion.parameters(), lr=loss_lr)
loss_scheduler = ReduceLROnPlateau(loss_optimizer, mode='max', factor=factor, patience=patience, verbose=True, threshold=delta)
```

Figure 4.7: Schedulers

4.3.4 Early Stopping

We implement Early Stopping when accuracy plateaus. We keep early stopping patience thrice that of scheduler patience so that the model has sufficient time to try to increase accuracy. In most cases, the model tend to stop at around 50 epochs.

```python
class EarlyStopping:
    def __init__(self, patience=3*patience, verbose=False, delta=delta_early):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta

    def __call__(self, val_loss, model, epoch, optimizer, scheduler, criterion, loss_optimizer, loss_scheduler, running_loss):

        score = -val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model, epoch, optimizer, scheduler, criterion, loss_optimizer, loss_scheduler, running_loss)
        elif score < self.best_score + self.delta:
            self.counter += 1
            print(f'EarlyStopping counter: {self.counter} out of {self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model, epoch, optimizer, scheduler, criterion, loss_optimizer, loss_scheduler, running_loss)
            self.counter = 0

    def save_checkpoint(self, val_loss, model, epoch, optimizer, scheduler, criterion, loss_optimizer, loss_scheduler, running_loss):
        if self.verbose:
            print(f'Validation loss decreased ({self.val_loss_min:.6f} --> {val_loss:.6f}).  Saving model ...')
        torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'loss_optimizer_state_dict': loss_optimizer.state_dict(),
                'criterion_state_dict': criterion.state_dict(),
                'loss_scheduler_state_dict': loss_scheduler.state_dict(),
                'loss': running_loss,
                }, f"checkpoints/best_{epoch}.pth")
        self.val_loss_min = val_loss
```

Figure 4.8: Enter Caption

4.4 Hyperparameter Tuning

We do comprehensive testing on a subset of the complete dataset (num_classes=20) using different hyperparameters and finally settle on parameters that we find to be performing the best in the given conditions.

```python
batch_size = 64
epochs = 100
learning_rate = 1e-3
loss_lr = 1e-4
factor = 0.3
patience = 5
delta = 0
delta_early = 0
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
printg(f"Using device: {device}")
```

Figure 4.9: Final Hyperparameters

4.5 Comparing Models

We first compared ConvNeXt Atto against MobileNet v4 Small over the subset of the dataset. MobileNet (97.2% accuracy) outperforms ConvNeXt (96.1% accuracy) despite being rougly 33% smaller.

Coming to heavier architectures, we pit ConvNeXt Atto against Femto and Pico, both of which seemed not to perform as well on our data and seem to underfit and stop early, probably requiring an even bigger dataset and/or more patience and much more training epochs. When MobileNet Medium is compared to MobileNet Small, similar results are seen. In all cases, the bigger models tend to stop early at around 30 epochs and between 94.5% (ConvNeXt Pico) to 96.4% (MobileNet Medium) accuracy,

Hence, we decide to proceed with ConvNeXt Atto and MobileNet Small training them over the complete dataset. We will move on to heavier models in case these struggle with the complete dataset, however that doesn't seem to be the case as currently in-training ConvNeXt atto which has completed 15 epochs yet has already achieved 80.22% top-1 accuracy over 1087 classes in the complete dataset. This is even higher than the expected 76.7% that was achieved by this architecture in the ImageNet challenge over 1,000 classes.

All these results can be found as TensorBoard plots in our observations repository [7],

Chapter 5

Future Work

Building upon the foundational work presented in this project, we expect to move on with the deployment next.

The deployment pipeline would depend on whether we finally decide to go with ConVNexT or MobileNet. In case we go with ConvNeXt, we will have to convert the model parameters into the ONNX format and then export it using TensorFlow.js and then deploy on the browser. In case of MobileNet however this would be much simpler since the MobileNet v4 models are already available on the Transormfers.js library, and we'd be able to eliminate the TensorFlow.js pipeline completely, and directly proceed to deploying it on the browser.

We next expect to have to set up a one-shot learning pipeline based on this pretrained model and tune this model over client-specific custom employee datasets.

Chapter 6

Conclusion

This project effectively addresses the challenge of deploying high-accuracy face recognition systems on resource-constrained devices. By leveraging state-of-the-art models such as ConvNeXt Atto and MobileNet v4 Small, we optimize for computational efficiency without compromising on performance. The utilization of ArcFace loss further enhances our model's recognition capabilities, ensuring high accuracy.

Throughout the project's lifecycle, extensive efforts were made in dataset preparation, including the development of a robust face extraction pipeline and comprehensive dataset cleaning. We facilitate a streamlined deployment pipeline, emphasizing systematic model optimization and efficient deployment strategies.

The results indicate that both ConvNeXt Atto and MobileNet v4 Small are suitable for deployment on low-compute devices, with MobileNet v4 Small showing a slight edge in performance metrics. Moving forward, the deployment pipeline will be finalized based on the chosen model, with considerations for client-specific customizations and one-shot learning pipeline setups.

In summary, this project demonstrates the feasibility of deploying advanced face recognition systems on resource-limited devices and builds such an end-to-end pipeline right from dataset preparation and model training to deployment. This was a great project and a great learning experience for me.

- Yatharth Singh

Appendix A

References

1. Yatharth Singh, GitHub.
ConvNeXt Face Recognition Repository.

2. Prateek Mehta, GitHub.
Celebrity Face Recognition Dataset. Repository.

3. Sanghyun Woo, Shoubhik Debnath, Ronghang Hu.
ConvNeXt V2: Co-designing and Scaling ConvNets with Masked Autoencoders.
arXiv:2301.00808.

4. Danfeng Qin, Chas Leichner, Manolis Delakis et al.
MobileNetV4 - Universal Models for the Mobile Ecosystem.
arXiv:2404.10518.

5. Jiankang Deng, Jia Guo, Jing Yang et al.
ArcFace: Additive Angular Margin Loss for Deep Face Recognition.
arXiv:1801.07698.

6. Kevin Musgrave.
pytorch-metric-learining.

7. Yatharth Singh. Observations

Appendix B

Glossary

ArcFace Loss A loss function designed for face recognition tasks that optimizes feature embeddings by maximizing inter-class separability through angular margins between feature vectors.

ConvNeXt A lightweight convolutional neural network architecture family designed for efficient computation on low-resource devices, optimized for performance in tasks such as image classification and face recognition.

MobileNet Another lightweight neural network architecture family optimized for mobile and edge device deployment, known for its efficiency and performance in image recognition tasks.

ONNX (Open Neural Network Exchange) An open format built to represent machine learning models. It allows models to be transferred between various frameworks like PyTorch, TensorFlow, and others.

TensorFlow.js A JavaScript library for training and deploying machine learning models in the browser and on Node.js.

Transformers.js A JavaScript library that provides pre-trained models for natural language processing tasks, which can also be used for image recognition tasks.

Face Extraction Pipeline A series of processes designed to detect and extract faces from images, preparing them for further analysis or recognition tasks.

Checkpointing A technique in machine learning where the state of the model (including weights and biases) is saved at certain intervals or conditions during training, allowing for recovery and continuation of training from those points.

Early Stopping A method used to halt the training process if the model's performance on a validation set does not improve for a pre-specified number of iterations, thus preventing overfitting.

Patience A parameter that defines the number of epochs with no improvement after which training will be stopped or learning rate will be adjusted.

Dynamic Learning Rate An adaptive learning rate mechanism that adjusts the learning rate based on the performance of the model during training.

Scheduler A component that adjusts the learning rate during training to improve convergence and performance of the model.

Gamma A hyperparameter for the learning rate scheduler that multiplies the learning rate by a factor (usually less than 1) when a plateau in performance is detected.

MLOps (Machine Learning Operations) Practices that aim to deploy and maintain machine learning models in production reliably and efficiently.

Hyperparameter Tuning The process of adjusting the parameters that control the training process of a machine learning model to improve its performance.

Learning Rate A hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated.

Batch Size The number of samples processed before the model is updated.

One-shot Learning A machine learning paradigm where the model is trained to recognize objects from a single example, useful in scenarios where data availability is limited.

Feature Embeddings A representation of data in a continuous vector space where similar items are mapped to nearby points, used extensively in tasks like face recognition.

Top-1 Accuracy A metric that measures the accuracy of a classification model by checking if the top predicted label matches the true label.

Log Parameter A toggle used to control whether certain data (e.g., accuracy, loss) is logged during training for monitoring purposes.

Transfer Learning A machine learning technique where a model developed for a task is reused as the starting point for a model on a second task.

PyTorch An open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing.

TensorBoard A suite of web applications for inspecting and understanding deep learning models through visualizations.

pytorch-metric-learning A library that provides various loss functions and miners for metric learning tasks, enabling models to learn embeddings that are more discriminative.

gdown A Python tool used to download files from Google Drive, particularly useful for managing large datasets.