Hochschule Düsseldorf
University of Applied Sciences
HSD

Project documentation:

Face Recognition Using Convolutional Neural Network & Autoencoder

Repository link : https://github.com/adhamarif/face-recognition

Name : Mohammad Adham bin Mohd Arif
Matriculation number: 888818
Faculty : DAISY
University : University of Applied Science Düsseldorf

Name : Muhammad Syahid bin Husein
Matriculation number: 888783
Faculty : DAISY
University : University of Applied Science Düsseldorf

Reviewer : Prof. Dr. Dennis Müller
Subject : Advances in Artificial Intelligence

Düsseldorf, 12.02.2024

Table of Content

Overview

This project is part of completing the module Advances in Intelligent Systems that we have in our Bachelor's course. The purpose of this project is to design dynamically deployable software in GitHub, applying all the concepts of machine learning and intelligent systems we learned in our lectures. Facial recognition was one of the topics offered for us to choose from and that is what we chose. We drafted out a plan for the project phases that is illustrated below:

First, we designed a data collection procedure to retrieve our dataset according to our needs and standards. Then, we designed and trained our networks, both the convolutional neural network and the autoencoder. We then use the trained model to employ our main script that will run our facial recognition. Even though project documentation is put at the end, realistically, we document our tasks parallel to doing them to avoid forgetting or overlooking anything.

Dataset preparation

Data collection

Data collection is crucial in this project, as we need to collect the facial images of (known) people to be trained on both of our models. The first phase of this project is mainly focused on preparing the suitable datasets for this project and creating the optimal data pipeline for it.

We prepared scripts that could automate this process. This is done by using OpenCV and Haar cascade classifiers. The code is written such that the user can generate their own image datasets and save it in their own folders. The image collection process are as follows:

1. The user gives their name as the input. This is used to create our subdirectory names that we can extract as labels for our prediction.
2. The camera will be turned on and the bounding box will be drawn around the face, whenever the face is detected by the Haar cascade classifier. The capturing process can start as soon as they see the bounding box appear on the screen.

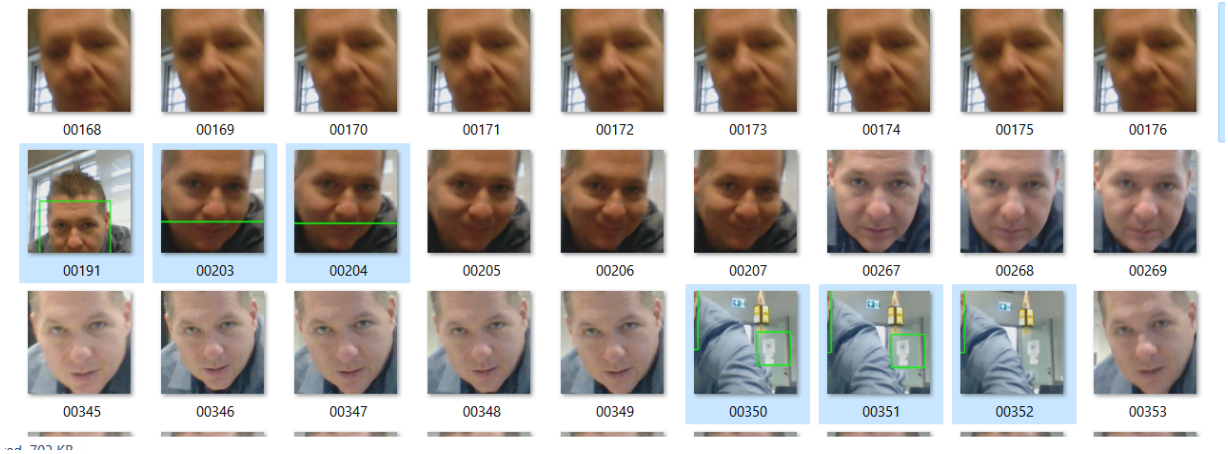
3. They can capture as many images as they need for the training. During this process, 2 images are saved at once, but into different directories:
    a. Original images (for backup)
    b. Region of Interest (ROI) - cropped face from the original images.

At the end of this process, we managed to capture datasets of 9 people, who 3 of those have 2 variations (with or without glasses). In total, we have 12 different labels for the training.

Data cleaning

After the data collection is finished, we inspect the result of the data collection. We figured out that the collected datasets were not meeting our standards. During the inspection process, we noticed the following problems in our datasets:

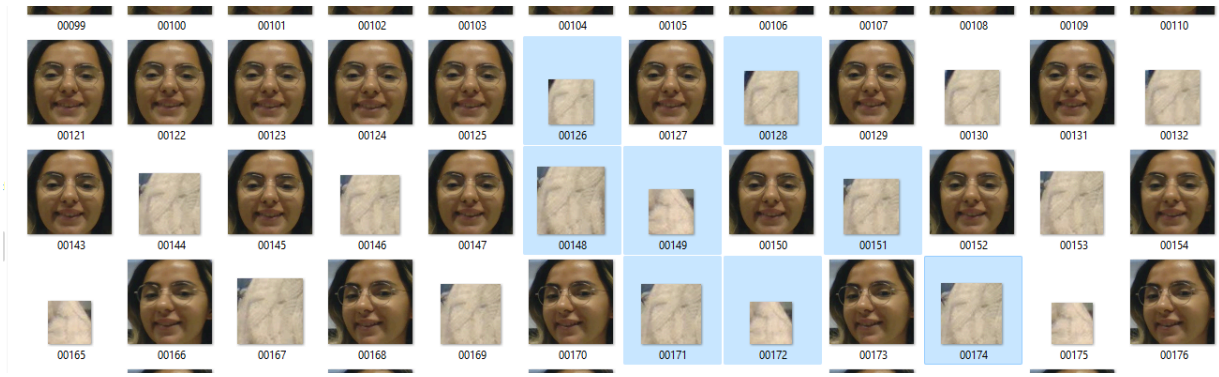
- Sometimes the objects that may have identical features as a human face were also detected in the bounding box, which we don't want to include in the datasets.
- The previous scripts only saved the first index of the detected bounding box. If there are more than one bounding box detected, we can't really know which region is indexed as the first index.
- Some images also appeared with the bounding box line. This happened when 2 bounding boxes were drawn close to each other, causing the overlapped bounding box on the face images.



Bad face datasets

In order to solve this problem, we actually had to write a new script and run it on the backup dataset that we collected (the original images). The solution we did was:

- Loop through all the original images again.
- Instead of only including the first index of the detected bounding box, we include all the indexes of the detected bounding boxes.
- Crop the image according to the region of face detection and remove the green bounding box border.
- Save the new images in the new folder to preserve the original dataset.
- At the end, we manually delete the 'noise' images that exist in the dataset folder. This includes images that are too blurry and unnecessary objects.



Manual deletion process

Metadata management

After the datasets are cleaned, we proceed to create a metadata file for these images. By using generator, we loop through all the images in the *cleaned datasets* folder to get the file path, its folder name (for label) and corresponding IDs based on the name. At the end, we save this information in a .csv file. This file is saved under *label.csv*.

Additional data processing steps were also needed for the **Convolutional Neural Network**. By using pandas, we one-hot-encoded the labels on the metadata file that was created previously. This is then saved under *face_label_encoded.csv*.

After all the images and the metadata files are ready, a custom dataset loader was created by using PyTorch's Dataset module. Since both model architectures have slightly different needs of dataset handling, we decided to create 2 different data pipelines for these 2 different models. More detailed explanations are in the next sections.

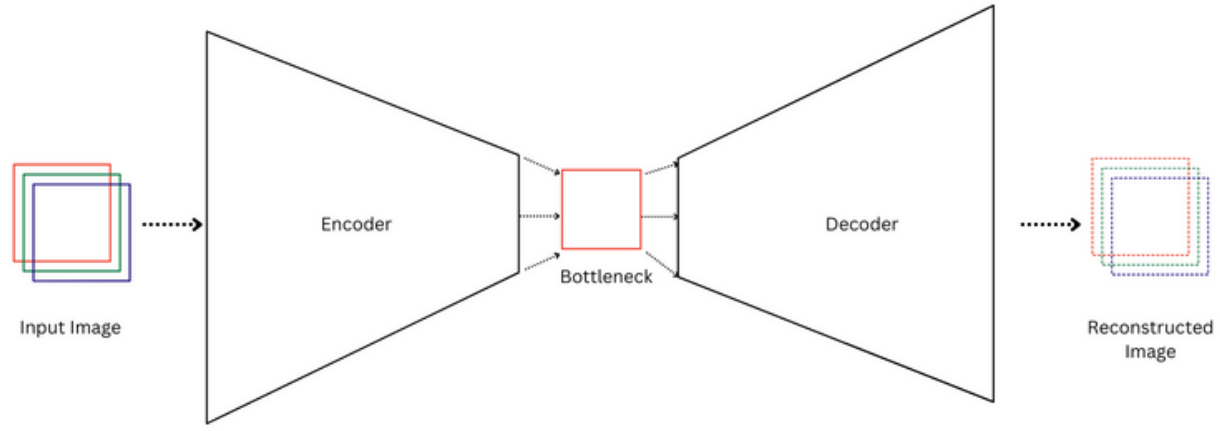
Network architecture

Autoencoder

Goal

The cropped face images are trained with an autoencoder model to reconstruct the original image of known people. At the end, we measure the validation loss to determine the best threshold to distinguish between known and unknown people.

Architecture



Autoencoder architecture

Down Layer
A layer consists of convolution, maxpooling, ReLU and batch normalization. Convolution layer is used to extract features in every pixel while Maxpooling layer reduces its spatial dimension by retaining the maximum value within each pooling region. Batch normalization is used to support better gradient calculation.

Up Layer
A layer consists of convolution transpose, ReLU and batch normalization. Convolution transpose layer is used to up-sample and reconstruct the image from the lower dimensional embeddings from the bottleneck back to its original shape, while recovering the spatial information that was lost during downsampling process. Batch normalization is used to support better gradient calculation.

Encoder

Down layers are repeated 6 times in a sequence to learn the features from the input image (320, 320) to the lower dimensional shape (at the bottleneck). At the end of the encoder, a fully connected layer is added for the model to learn about every information in every pixel.

Decoder

A fully connected layer is added for the model to learn the information in every pixel. Then, the Up layers are repeated 6 times to reconstruct the image back to its original dimension (320, 320). The last layer of Decoder is designed such that it has no batch normalization and uses Sigmoid as the activation function. The reason is because the final output should be retained as raw value and should not be normalized.

Data pipeline

A custom dataset pipeline was created to feed the images to the model by using torch's modules. Since the goal of this network is only to reconstruct the images, we just use the label to specify whose images should be trained on the autoencoder (if we wish to train the model on a specific person). Otherwise, all the images will be given as the input for this model. A few methods are also implemented with these datasets before the training begins to ensure effective model training and evaluation.

Training and validation split

The train_test_split module from scikit-learn is also used to split the datasets into training and validation sets. This is useful to prevent data leakage during the training and also to evaluate the model's performance on the unseen images. At the end, the validation loss is also needed to fine tune the threshold for the face recognition system.

Class imbalance

The balance classes method is also applied in the *training* set, so that images of a minority class label would be oversampled to the same number of images as our majority class label in the dataset.

Data augmentation

Data augmentation is also implemented in the *training* set by using torchvision's transforms module, such that the images will have different lighting factors and random horizontal flip during the training. This is done to ensure the model will be robust enough to adapt to different conditions.

Training & Results

The autoencoder is then trained with these parameters.

Loss function : Mean Squared Error
Optimizer : Adam
Epoch : 20 & 50
Number of batch : 32

MSE is used to measure the reconstruction error from the autoencoder between input and the output images. The losses of *training* and *validation* sets are measured in every epoch to monitor the model performance. The Adam optimizer is also used in training due to its computation performance. It is also known for its adaptive learning rate and momentum capabilities [1]. The optimizer's gradients are zero-ed out every epoch. This is known as 'gradient reset'. This is because we don't want PyTorch to accumulate the gradients from the previous batches as it is inconvenient and undesirable for training [2].

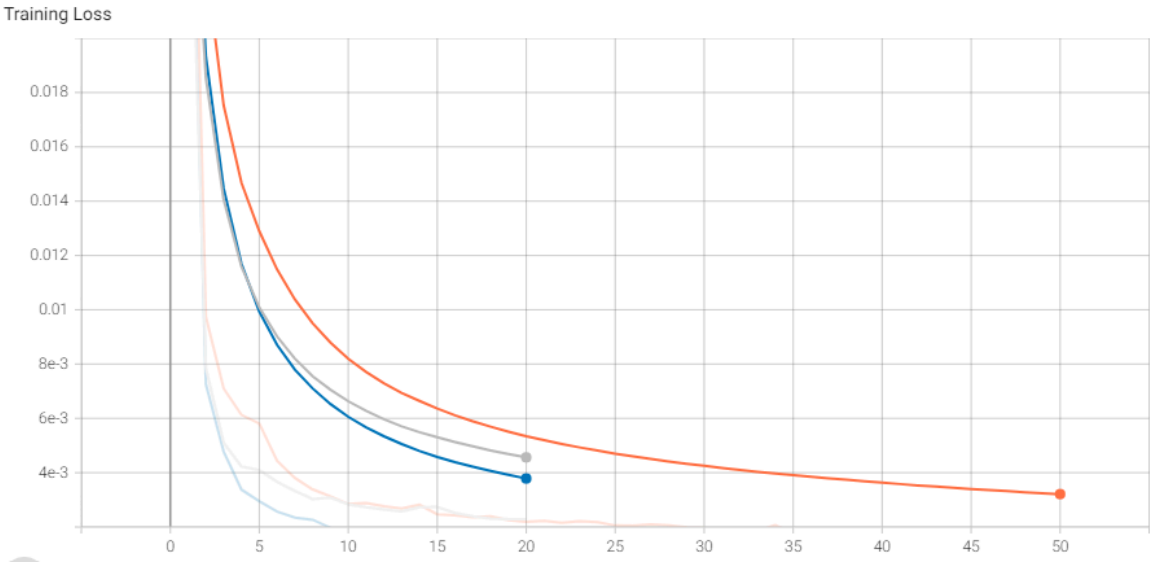
Tensorboard was implemented in the training code, so that the training process can be monitored easily in the web interface. The training loss will be averaged in each epoch, and if the average loss is lower than the best average loss, the model checkpoint will be saved. The optimizer's scheduler was also implemented at the middle of the project phase, but since it reduced the model performance significantly and slowed down hyperparameter fine tuning, we decided to exclude the optimizer's scheduler from the training loop and record it as possible future implementations.

3 different models were trained for 20 and 50 epochs to evaluate their performances. We could observe from the first 2 models that the data augmentation slightly decreased the model performance, but it might be more robust to changes from external factors, since we applied data augmentation during the training phase. This is the trade-off factor that we considered for this model.

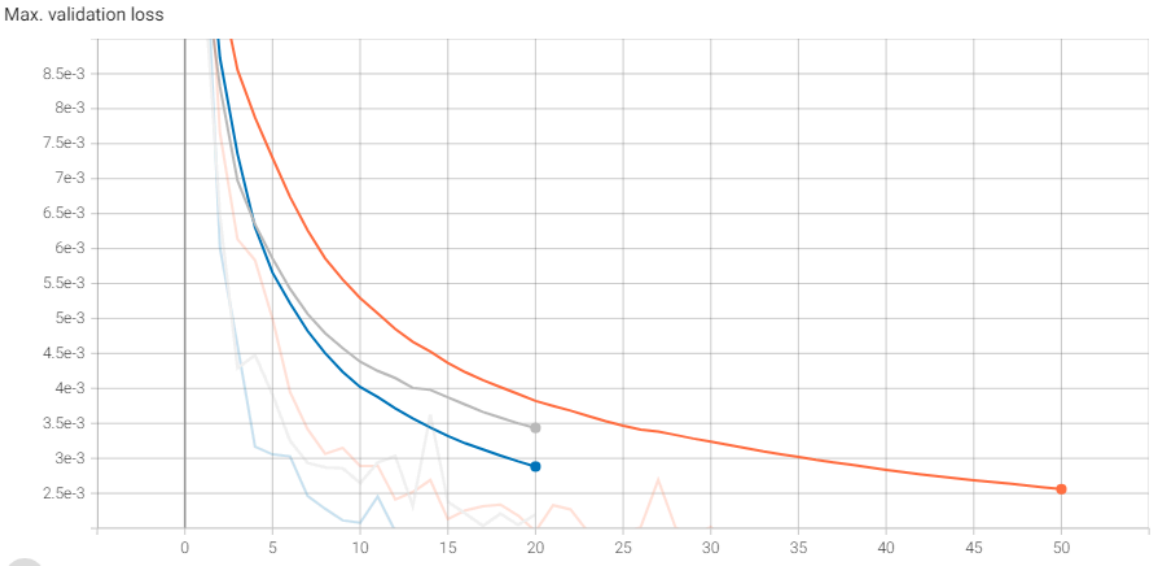
After that, we tried to train the same model for 50 epochs and it recorded slightly better training and max validation loss. Therefore, we proceeded to keep this model for the face recognition system.

Model training summary

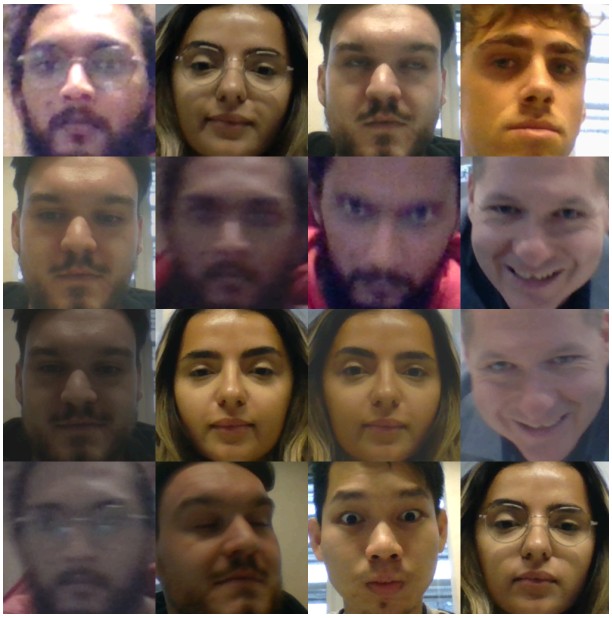
Model	Num. of epoch	Loss	Max validation loss
Without data augmentation	20	0.0012	0.0015
With data augmentation	20	0.0022	0.0022
With data augmentation	50	0.0015	0.0015



Training loss

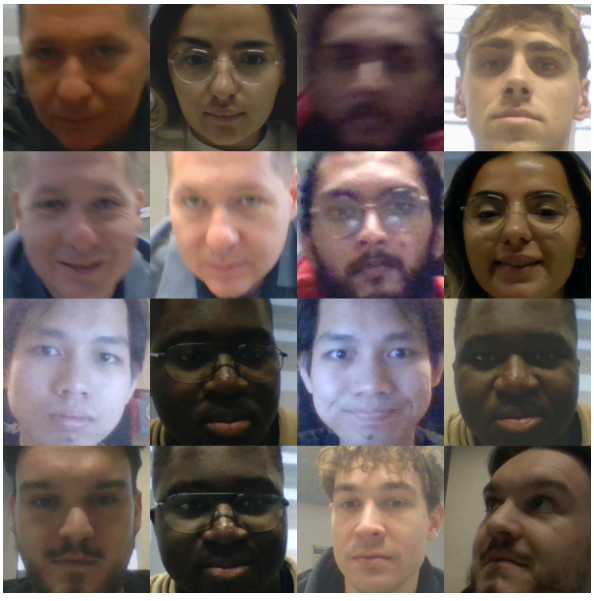


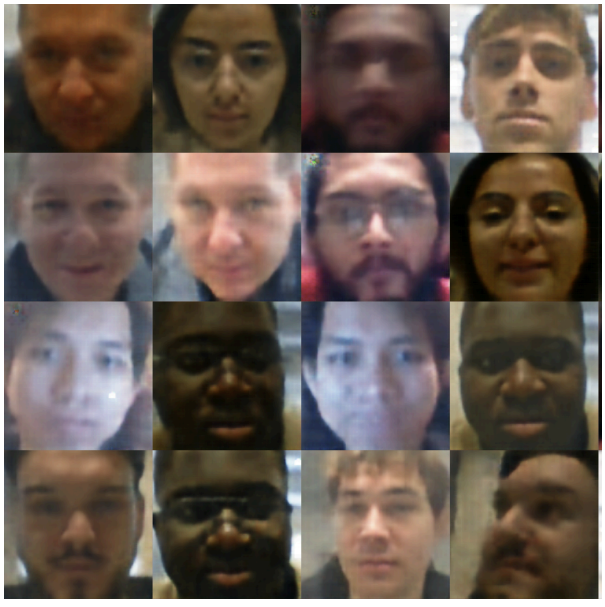
Validation loss



Input image *Reconstructed image*

Face reconstruction at first epoch (1)





Input image *Reconstructed image*

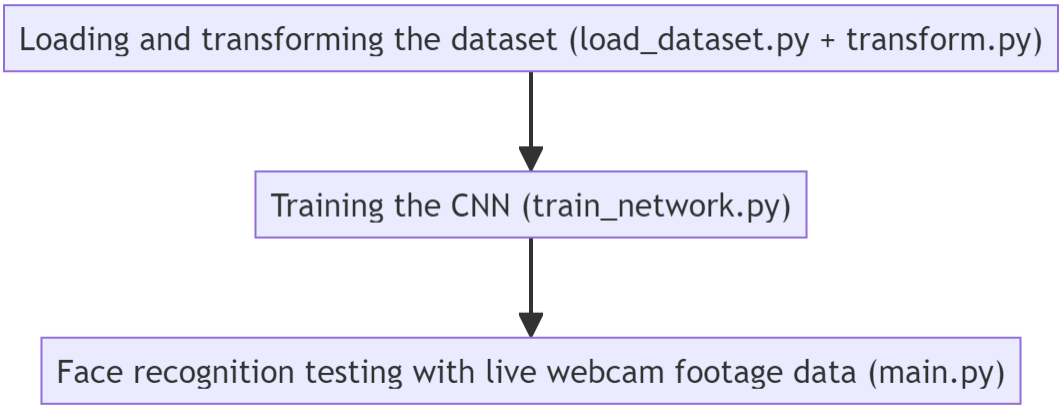
Face reconstruction at last epoch (50)

Convolutional Neural Network

Goal

The cropped, cleaned and labeled face images will be used to train a Convolutional Neural Network as a face recognition model.

Data Pipeline



1. Loading and transforming the dataset

The dataset is generated from the paths of the cleaned dataset and the *label.csv* file. As mentioned previously, we used PyTorch's Dataset module to create a CustomImageDataset class that executes the following tasks:

- Reading the dataset path with Pandas .read_csv() and creating a DataFrame out of it.
- Splitting the data into training (80%) and validation set (20%).
- If it's the training set, class imbalances are handled by oversampling minor classes to the length of the majority class.
- Transform the images according to their respective transformation functions from *transform.py*. The transformation functions are composed from torchvision's transforms module.
  If 'training', the images are :
  - Randomly rotated about a 15 degree angle.
  - Center cropped at a 256x256 pixels frame.
  - Randomly horizontally flipped. There is a 50% chance for every image to be horizontally flipped.
  - Applied Color Jitter. We apply a 30% jitter to each brightness, contrast and saturation.

    - Converted into PyTorch floats, suitable to be passed through a neural network.

  If 'validation', the images are :
    - Center cropped at a 256x256 pixels frame.
    - Converted into PyTorch floats, suitable to be passed through a neural network.
- Return the images' floats and their labels.

The datasets are then flowed into dataset loaders using PyTorch's DataLoader module. The dataset loader is a generator that generates batches of images and labels. We decided on a batch size of 32.

2. Training the CNN

The batches are then flowed into our training script for the CNN. This training script executes the following tasks:

- Checks if there are training checkpoints existing in the 'models' directory. If not, the training begins from scratch. The checkpoint contains the CNN's and the optimizer's state dictionaries that we can load. The script also takes arguments to allow the user to specify if they want to start training from scratch themselves. This is because sometimes there are checkpoints, but the user may have altered the dataset and wants to restart the training process from scratch.
- The script then begins the training process, epoch by epoch, in a while loop. At each epoch :
    - The training data batches are looped through one by one using a for loop and our dataset loader. This is done until all the batches are looped through, marking the end of the epoch.
    - The current training data batch in the for loop is split into inputs and labels, assigned to the user's device. The device variable is in the *device.py* file. If cuda is available, then it takes cuda as the device. If not, it takes the cpu.
    - We use an Adam optimizer for training. The optimizer's gradients are zero-ed out every epoch. This is known as 'gradient reset'. This is because we don't want PyTorch to accumulate the gradients from the previous batches as it is inconvenient and undesirable for training [2].
    - The data is passed through our CNN (The CNN architecture will be explained after).
    - The cross entropy loss between the CNN output and the actual labels is calculated using PyTorch's CrossEntropyLoss module.

    - The top loss values and their corresponding samples are collected. This collection accumulates for every batch. For example, the first batch contains the first top loss values and the second batch contains the first and second top loss values combined and sorted in descending order. They are then added as images in a grid to our logger (Tensorboard's SummaryWriter) for tracking purposes.
    - Backward propagate the loss to our CNN.
    - Adjust the weights/gradients of the optimizer using its .step() method.
    - Calculate the accumulated loss and accuracy values. For example, the loss and accuracy of the first batch represent the performance of the first batch. The loss and accuracy of the second batch represent the average performance of the first and second batches combined.
    - The net is set to evaluation mode by disabling gradient computation because we do not want the CNN to learn from our validation data.
    - The validation data batches are looped through the same way.
    - The training and validation sets' average losses and accuracies are logged into our logger at the end of every epoch.
    - If the average losses and accuracies show improvement compared to the previous epoch, we define them as the new best values and save the model's checkpoint in our 'models' directory. We save the CNN's state dictionary and the optimizer's state dictionary. We only save the model if there's improvement and not at every epoch because we do not want to save a worse model.
- After 5 epochs of showing no improvement, we break the while loop and stop training.

3. Face recognition testing with webcam footage data

The 'best' saved model is saved in our 'model' directory. The *main.py* script uses our best model to run our main face recognition software (Will be explained below). We use the webcam footage as our 'test' set to evaluate our model performance with new unseen data.

Network Architecture

Down Layer:

```
class Down(nn.Module):
    def __init__(self, in_features, out_features):
        super(Down, self).__init__()

        self.seq  = nn.Sequential(
            nn.Conv2d(in_features, out_features, kernel_size=(5,5), padding="same"),
            nn.MaxPool2d(kernel_size=(2,2), stride=(2,2)),
            nn.BatchNorm2d(num_features = out_features),
            nn.ReLU()
            )

    def forward(self, x):
        return self.seq(x)
```

This is the architecture of our single convolutional layer. It runs 2D convolution to the images at a window size of 5x5. The convoluted array is fed into a 2D Max Pooling layer which takes the max value from a window size of 2x2 to be fed into the next layer. We then apply batch normalization to stabilize the training process. Finally, a ReLU activation function is used to allow the network to learn more complex representations of our data.

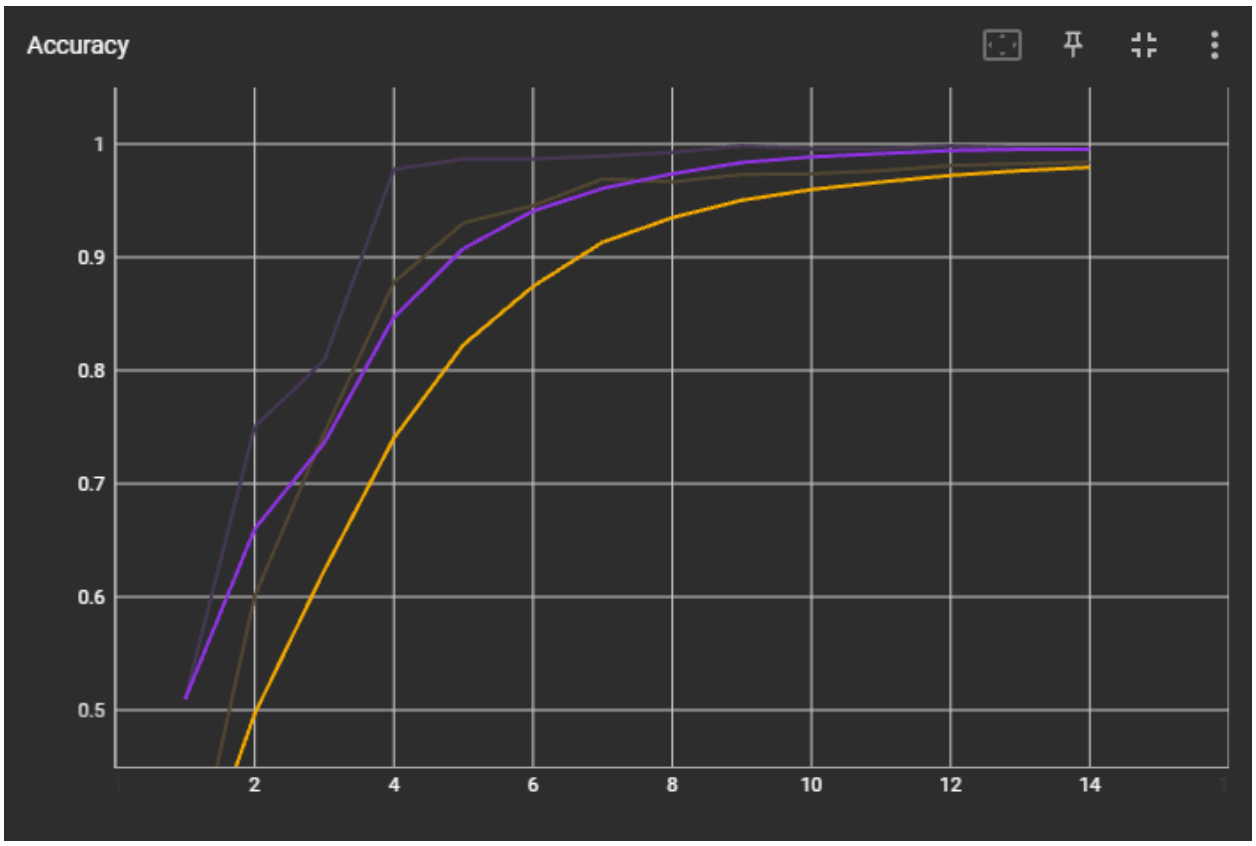
2D Convolution is a process of passing a convolution window through the input image. The window will extract important features from the input image and pass it through to the next convolution layer. At the end, we have an array that contains all the important features of the input image [3].

```python
class NeuralNetwork(torch.nn.Module):
    def __init__(self,num_labels):
        super(NeuralNetwork, self).__init__()

        self.num_labels = num_labels

        self.seq = nn.Sequential(
            Down(in_features =   3, out_features =  16), # 16x128x128
            nn.Dropout2d(0.2),
            Down(in_features =  16, out_features =  32), #  32x64x64
            nn.Dropout2d(0.2),
            Down(in_features =  32, out_features =  64), #  64x32x32
            nn.Dropout2d(0.2),
            Down(in_features =  64, out_features =  128), # 128x16x16
            nn.Dropout2d(0.2),
            Down(in_features =  128, out_features =  256), # 256x8x8
            nn.Dropout2d(0.2),
            Down(in_features =  256, out_features = 512), # 512x4x4
            nn.Dropout2d(0.2),
            nn.Flatten(), # 4096 dimensional
            nn.Linear(8192, 512), # 8192 to 512 dimensional
            nn.ReLU(), # Another ReLU
            nn.Dropout(0.5),
            nn.Linear(512, num_labels)
            # nn.Softmax(dim=-1)- Commented out because CrossEntropyLoss already apply  softmax
        )

    def forward(self, x):
        for i,layer in enumerate(self.seq):
            x = layer(x)
        return x
```

After each Down layer, a 20% Dropout is used to deactivate some of the neurons. This is done to avoid co-adaptation of neurons to recognize features [4]. Doing this will make the training harder and prevent overfitting.

We increase the dimensions of the input array at every layer from the 3 channels (Red,Green,Blue) of 256x256 pixels at the input layer to 512 channels of 4x4 pixels before flattening it to a 1D array. This flattened 1D layer is then fed into a Linear layer to reduce the dimensions even further. Finally, a last ReLU activation function is used and a 50% Dropout is applied before reducing the dimension to our number of labels (12 for 12 faces). 50% provides the highest level of regularization and is the standard machine learning practice at the last layer.

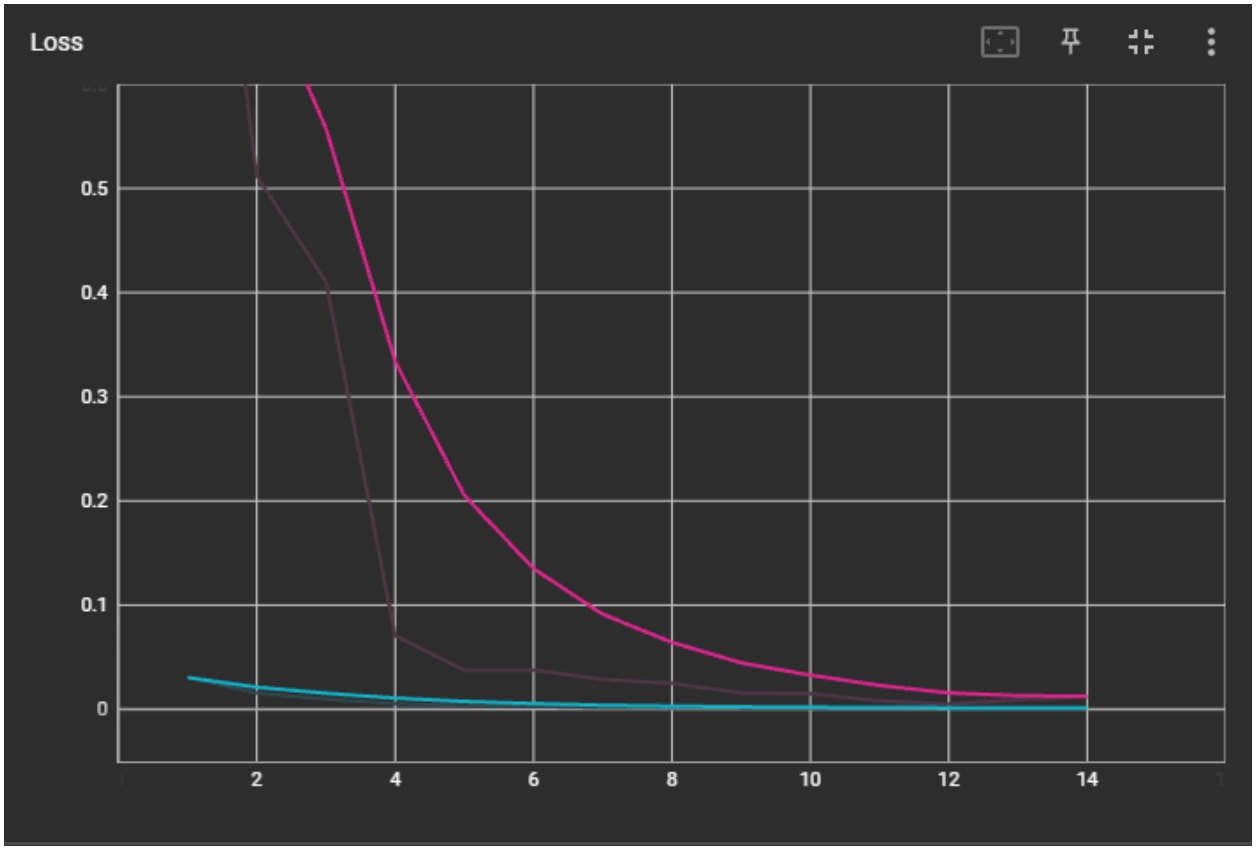
Training Results



This is the model accuracies on training (orange) and validation (purple) across 14 epochs. The training was last saved at epoch 9 because that was when it produced the best accuracy before it stopped improving. The accuracies at epoch 9 are as follows:

- Accuracy : Train = 97.31%, Validation = 99.83%

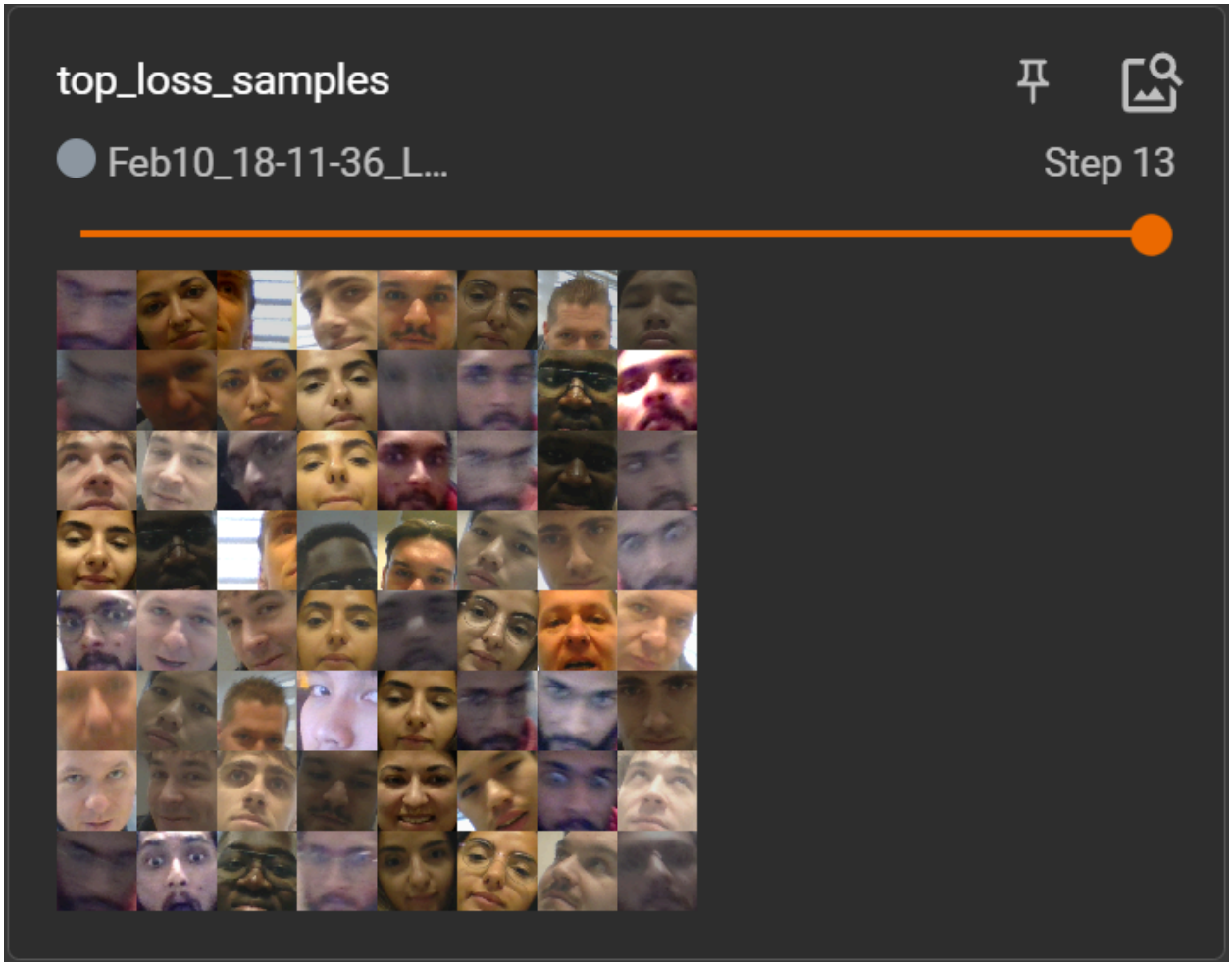
Intuitively, one would expect the model to perform better with training data than validation, however, since we made the training more difficult by jittering the training data and applying dropouts, our training performance is worse than validation most of the time.



This is the model losses on training (turquoise) and validation (magenta) across 14 epochs. The losses at epoch 9 are as follows:

- Loss : Train = 0.0014, Validation = 0.0157

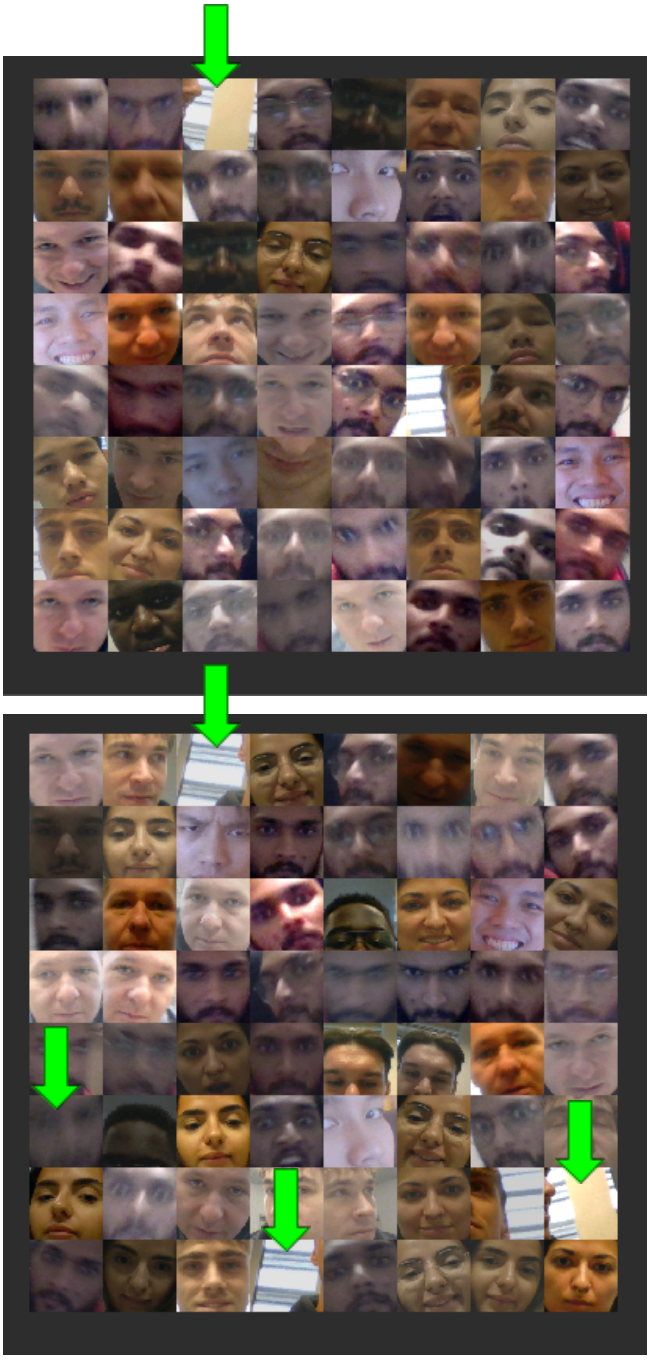
Here are the top loss samples from the above training session :



As we can see here, the inputs that produce the highest loss values are images that are either 1) blurry, 2) only show half of the face, or 3) images that the model simply just got wrong.

The idea behind collecting the top loss samples is to identify if there are images that are intuitively ‘bad’ for training. This is an added measure because we transformed the training data in a way that important features could be lost. Plotting them in a grid lets us know if such ‘bad’ data exist and we can then remove them.

Below are top loss samples from previous training sessions. The images that we removed from the training data are indicated with a green arrow.



We removed these images and retrained the model to obtain the model that you saw above.

Implementation (Face Recognition)

We wrote a *main.py* script for testing the face recognition model with webcam footage data. The script executes the following tasks:

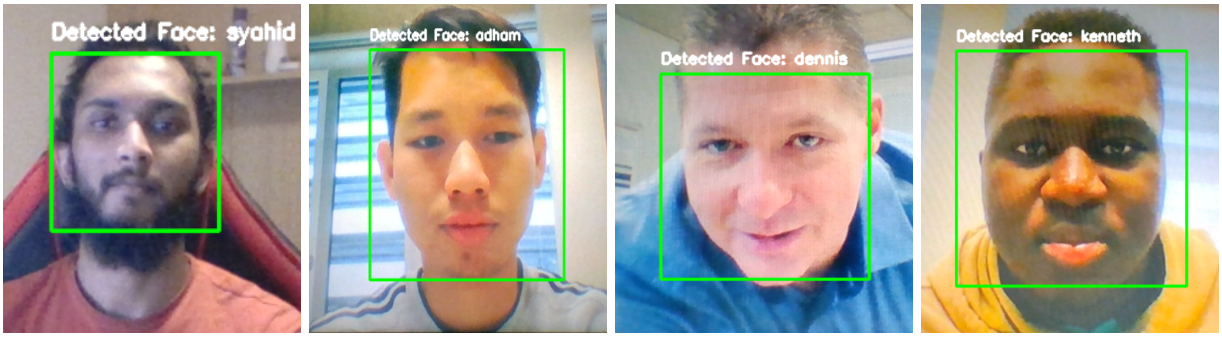
1. Imports the network modules from each of our network's scripts (CNN and Autoencoder).
2. The script takes arguments to allow the user to decide between either the CNN or the Autoencoder.
3. Read the model checkpoint file locally. If it doesn't exist locally, we use GDown's .download() method to retrieve it from our Google Drive online. The checkpoint file contains the model's and the optimizer's state dictionaries. In this case, we only load the model's state dictionary into our network as we are not training.
4. OpenCV's VideoCapture() module is used to open the webcam footage.
5. We then use OpenCV's readily available Haar feature-based cascade classifier for face detection of our webcam footage.
6. We extract the Region of Interest (ROI), or in other words, the face detected by the cascade classifier as a frame.
7. We transform this ROI into a tensor that is suitable for our model to make predictions. We use the transform modules from our respective *transform.py* files.
8. Before making a prediction, we turn on the model's evaluation mode and disable gradient computation. Then we pass the transformed ROI through our model to obtain our outputs.
9. We use PyTorch's softmax activation function to turn the outputs into probability values between 0 and 1 for each of the 12 labels. We use this as a 'confidence' value that we will compare against a predefined confidence threshold.
10. We use PyTorch's .max() method and our self-defined label translator function to turn our outputs into a text corresponding to the identity of the predicted face.
11. If the confidence value of the predicted face exceeds our confidence threshold, we draw a green rectangle around the detected face and display the text of the predicted identity above it. If the confidence value does not exceed our threshold, the rectangle drawn will be red and the text above would be 'Unknown Face'.
12. In the case of our autoencoder, steps 9 to 11 are different. Instead of predicting a label for the ROI, it reconstructs the face and calculates the mean squared error (MSE) loss for the reconstruction. The threshold for 'Known' and 'Unknown' is a loss threshold that we compare to the MSE loss. If the MSE loss is lower than the loss threshold, the rectangle drawn is green and the text that appears is 'Known Face'. Otherwise, the rectangle drawn is red and the text is 'Unknown Face'.

Results

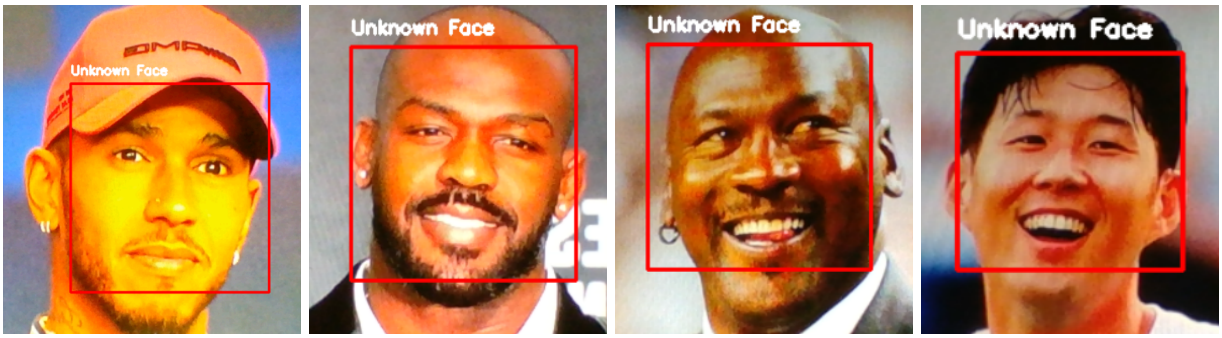
Here are the results from each model:

CNN

Example of a ‘Known Face’ with its corresponding identity label:

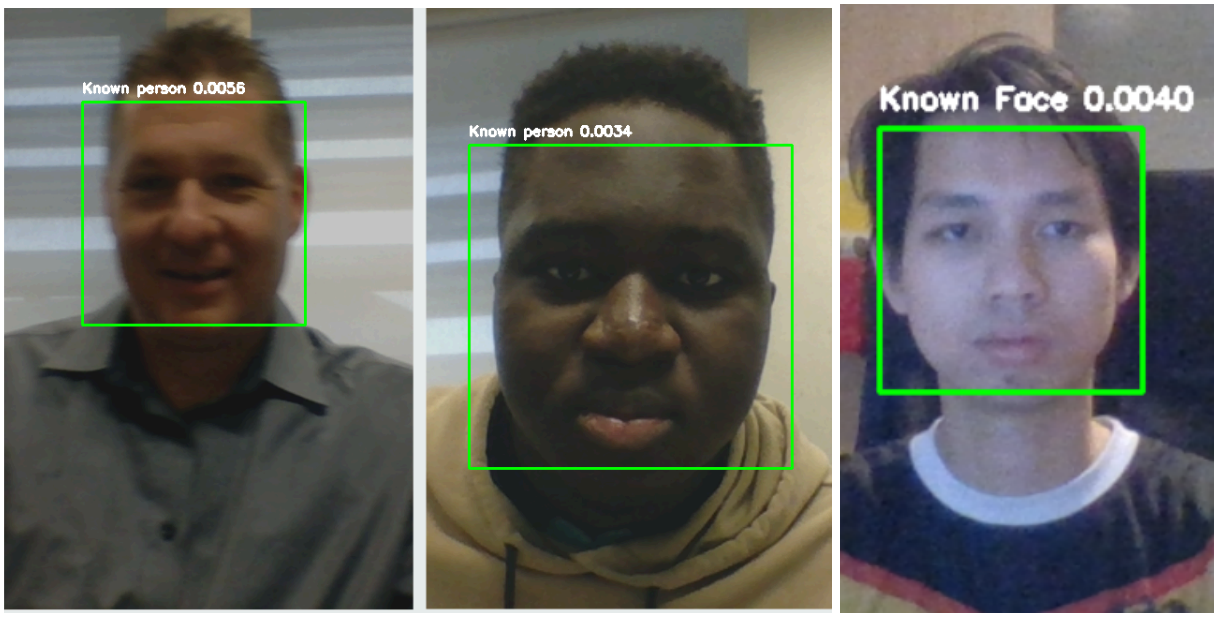


Example of an ‘Unknown Face’:

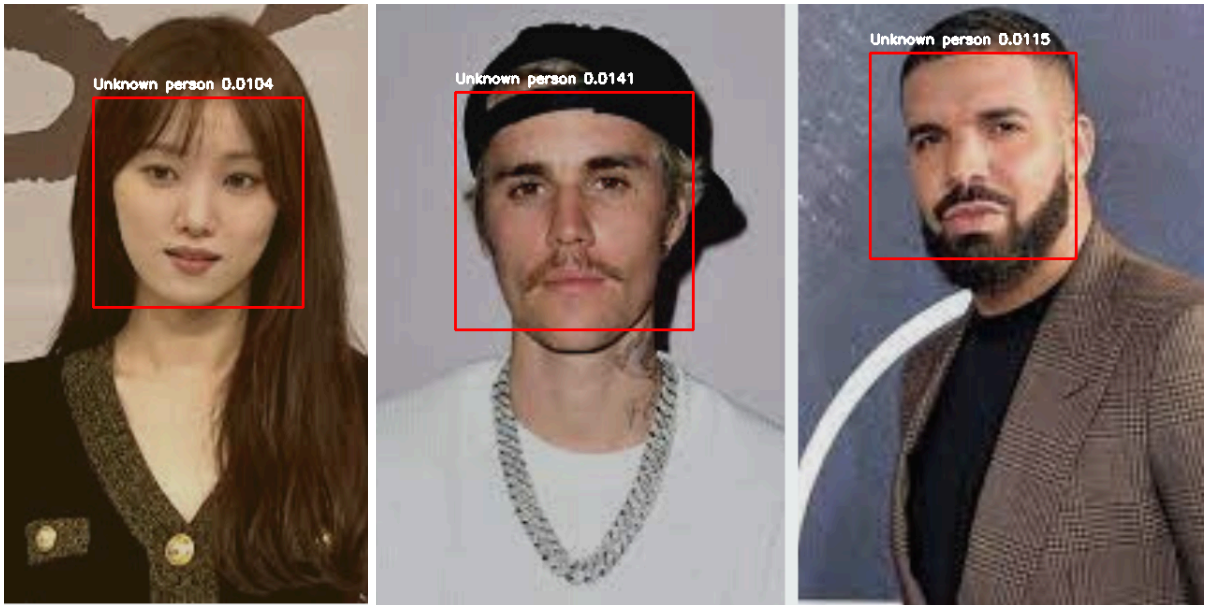


Autoencoder

Example of the ‘Known Face(s)’:



Example of the ‘Unknown Face(s)’:



Key Takeaways

- We learned how to prepare datasets for image training according to our standards and operational framework.
- We learned how to design dynamic and robust networks for image learning.
- We learned industry standard machine learning practices and why they are done.
- We learned the implications of image learning techniques and were able to find trade-offs.

Room for improvement:

- Implement more data augmentation techniques on the Autoencoder model.
- Record predictions from each label in a grid at CNN's validation stage to visualize model performance better.
- Improve data collection procedures by inducing more image variances.
- Standardize dependencies at an early stage to avoid technical errors.

Sources

[1] Kingma, D. P., & Ba, J. (2014, December 22). *[1412.6980] Adam: A Method for Stochastic Optimization*. arXiv. Retrieved February 11, 2024, from https://arxiv.org/abs/1412.6980

[2] (n.d.). *Zeroing out gradients in PyTorch*. https://pytorch.org/tutorials/recipes/recipes/zeroing_out_gradients.html

[3] (n.d.). *2D Convolution Visualizer*. https://adamharley.com/nn_vis/cnn/2d.html

[4] (n.d.). *Improving neural networks by preventing co-adaptation of feature detectors*. https://medium.com/@lipeng2/dropout-is-so-important-e517bbe3ffcc