Build New Dataset and Classify Faces by Using Transfer Learning

Batel Shuminov
batelsh@campus.technion.ac.il

Amit Kadosh
amitkadosh@campus.technion.ac.il

January 2023

Abstract

Face recognition and classification is an important capability that can have a wide variety of uses. For example, in a smart home, helping the blind, building personal albums and more.
In this project we will first improve a standard face recognition algorithm in Python which is based on classic ML and then we will train a deep learning model for the purpose of classify the face in the image into one of the 15 people on which the model was trained.
We created the dataset for the project ourselves from photos of family and friends. In order to select the images for this project, there is no need to pre-process the images. The dataset for the project is very small and contains about 50 images per user for 15 peoples.
During the project we examined the use of Transfer Learning using two methods - Feature extraction and Fine tuning.
For the purpose of improving the generalization of the model, we used augmentations of various types.
GitHub Repository link: https://github.com/batelshuminov/Build-New-Dataset-And-Classify-Faces-By-Using-Transfer-Learning.git

1 Introduction

Face classification is an important capability that can have a wide variety of uses. for example:

- The system can be used to help blind people know who is at the front door of the house.
- The system can also be integrated into the smart home field - reading the name of the person standing at the entrance to the house.
- Creating personal albums - you can locate the photos in which a certain person appears and thus build a personal album for him easily.

This area was not tested due to the lack of a suitable dataset.

In addition, face classification is an individual problem of each person because each person knows different people and wants to classify them.
Therefore, we found it appropriate to prove the feasibility of implementing a system based on Deep Learning for the purpose of face classification for a person from a group of people that the model will learn.
The main difficulty expected in this project is that Deep Learning works well when there are many examples and in this project the number of examples is very few.
In order to overcome this difficulty, we will use transfer learning which was trained on another dataset for a different purpose.

2 Dataset

2.1 Original Dataset

The name "Original Dataset" represents the original images loaded into the array before any processing.
We built the dataset for this project from images from our personal smartphones and social networks. We collected data of 15 classes (different people) with each having only 50 images.
We saved the images in a folder called "Original Dataset" which contains 15 folders for each of the people we want to teach our model.
The images in the Dataset can have different characteristics, for example different situation, different lighting, taken from different cameras and more. Because there is no Dataset like this around the internet so we had to build it.
In order for this project to be able after further hardware development to be used by anyone, we need to require a relatively low minimum threshold of images for each person. Therefore, we chose to use 50

images for each person.

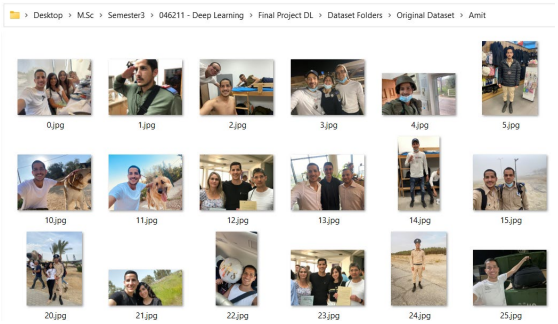

Figure 1: Example of images from "Amit" class in "Original Dataset".

2.2 Prepressing Dataset

2.1.1 Face Recognition

Now we will move to the second stage - the face recognition stage in the image.

We used a built-in algorithm in Python called Cascade Classifier from a GitHub project which is based on classic Machine Learning. This algorithm has been widely used in many web projects.

OpenCV uses machine learning algorithms to search for faces within a picture. For something like a face, might have 6,000 or more classifiers, all of which must match for a face to be detected (within error limits). the algorithm starts at the top left of a picture and moves down across small blocks of data, looking at each block, and check if is this a face. Since there are 6,000 or more tests per block, might have millions of calculations to do. To get around this, OpenCV uses cascades. The OpenCV cascade breaks the problem of detecting faces into multiple stages. For each block, it does a very rough and quick test. If that passes, it does a slightly more detailed test, and so on. The algorithm may have 30 to 50 of these stages or cascades, and it will only detect a face if all stages pass. The advantage is that the majority of the picture will return a negative during the first few stages, which means the algorithm won't waste time testing all 6,000 features on it. Instead of taking hours, face detection can now be done in real time.

This algorithm works with grayscale images, so in order to run the algorithm, we first converted the images to grayscale.

After receiving the face location from the algorithm, we cut the face from the original color image. Finally, we resized it to 256x256 so that all the face photos would be the same size.

Figure 2: Face Recognition. (a) The original image, (b) The image in gray scale, (c) Face recognition performed by the Cascade Classifier algorithm, (d) Cropping the face from original image and resize to 256x256.

2.1.1 Improving Face Recognition Algorithm by Rotation

Unfortunately, we noticed that the algorithm we described does not give good enough results - there are many images in which it fails to recognize faces. We delved into the images and found that the algorithm has more difficulty in images in which the faces are slightly rotated. In Figure 3 in the top row of images, you can see the result of the basic algorithm for the two images on the left - the algorithm did not find the face in both images. Following the importance of this algorithm in our case, in a situation where every image is important, we decided to improve the algorithm - we implemented a cyclic process to rotate the image by 15 degrees at each step. In Figure 3 in the bottom images row, we can see the result after our improvement - the improved algorithm manages to recognize the face in both images. The improved algorithm provides about a 20% improvement in facial recognition. It's a lot in this project.

Figure 3: (a) Original images, (b) Results of facial recognition algorithm – it can be seen that no faces are received in this case, (c) Results of improved algorithmm – it can be seen that the new algorithm find the two faces in the image.

3 Reference Model

3.1 Reference model architecture

In order to see the effect of Transfer Learning and its contribution to improving the results in the case of a small dataset, we compared several Deep Learning models for image classification without using Transfer Learning at all.

In this article we will present the model with which we got the "best" results for this series of models (which does not use Transfer Learning).

This model is used as a starting point of reference for examining the contribution of Transfer Learning. Therefore, we will call this model the "Reference Model".

The "Reference Model" is built in the standard way for image classification in Deep Learning where the

first part consist of convolution layers and the second part consist of fully connected layers. Of course, activation functions are also used to break the linearity, and pooling is used for reduce the dimensions of the image when it passing through the model, which will be explained in detail later.
The architecture of this model is shown in Figure 4.

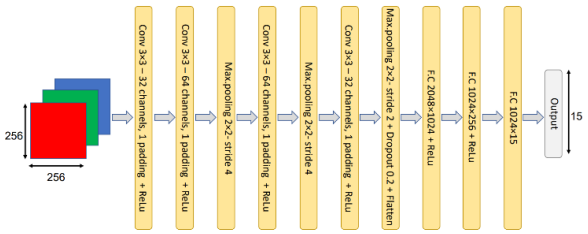

Figure 3: **Our architecture**

Figure 4: The architecture of Reference Model.

3.2 Activation, Dropout, Pooling and Loss Function

- Activation Function

The key change made to the Perceptron that brought upon the era of deep learning is the addition of activation functions to the output of each neuron. These allow the learning of non-linear functions.
In reference model we used ReLU (Rectified Linear Unit) activation functions. It is the most common activation function as it is fast to compute and does not bound the output (which helps with some issues during Gradient Descent).

- Dropout layer

Dropout is a regularization technique for reducing overfitting in neural networks by randomly setting a fraction of input units to 0 during the forward pass. This helps to prevent complex co-adaptations on training data. The dropout rate is the fraction of units that are dropped out, and is a hyperparameter that can be tuned. The dropout layer is usually used after the convolutional layers and before the fully connected layers, as it is more effective in preventing overfitting in these layers.
In "Reference Model", we add dropout layer after the block of the convolutional layers and set it probability to 0.2.

- Pooling

Pooling is a sliding window type technique, but instead of applying weights, which can be trained, it applies a statistical function of some type over the contents of its window.
The most common type of pooling is called max pooling, and it applies the function over the contents of the window. Pooling can assist with this higher level, generalized feature selection.

- Loss Function

We used cross-entropy loss. It is a common loss function used in supervised learning problems, particularly for training models for classification tasks.
CrossEntropyLoss is mainly used for multi-class classification and it already included a softmax layer inside.
The softmax activation function in the output layer of the model, ensure that the predicted probabilities are valid probability distributions (sum of all predicted probabilities is equal to 1).

3.3 Results of the "Reference Model"

Models of this type produced very disappointing results of about 50% on the Test. In particular "Reference Model" yielded an accuracy of only 52.9412% on the Test set.
Although these results are disappointing, they are not so surprising, it is 15 classes with very few images for training each class and as we said Deep Learning works well when there is a lot of data.

4 Transfer Learning

4.1 Main Principle

Transfer learning is a technique in machine learning that involves using a pre-trained model to solve a different, but related, problem. The idea is to take a model that has already been trained on a large dataset, and use its learned features as a starting point for a new task, rather than training a model from scratch.
Transfer learning has been widely used in computer vision, natural language processing, and other fields and have shown remarkable results.
There are two main types of transfer learning which we will explain later: Fine-tuning and Feature extraction.
We will use in models the trained on ImageNet1K_V1 dataset.
The ImageNet1K_V1 dataset contains 1,000 classes and over 1.2 million images. ImageNet1K_V1 contains images of various objects, animals, and scenes. The images are organized into 1,000 classes, with each class containing several images. Some examples of classes in the dataset include "dog", "cat", "car", "flower", "mountain", "building", "person", "food", etc. The images are taken from a wide range of sources, including professional photographers and amateurs, and they vary in terms of resolution, quality, and angle.
The size of images in the ImageNet1K_V1 dataset is 224x224 pixels.
We tested 5 architectures: Resnet18, AlexNet, VGG16, SqueezeNet and Densenet. In order to use Transfer Learning with these models, we converted the image size from 256×256 to 224×224.

4.2 Feature Extraction

In feature extraction, a pre-trained model is used as a fixed feature extractor. In the method, instead of training the entire model on the new dataset, we freeze all the layers of the pre-trained model except the output layer, and only the output layer of the pre-trained model is trained on the new dataset.

The feature extraction process is useful when the new dataset is large and similar to the original dataset. It allows the use of the pre-trained model's features and avoid overfitting, and also allows the new model to achieve good performance with less data.

4.3 Fine Tuning

In fine-tuning, instead of random initialization, a pre-trained model is used as the starting point. The pre-trained model has already been trained on a large dataset, and has learned useful features that can be used for a new, related task. By starting with the pre-trained model, the fine-tuning process can converge faster and achieve better performance compared to training a model from scratch.

The goal is to adjust the model to the new dataset while preserving the useful features learned from the previous dataset.

5 Architectures for Transfer Learning

5.1 Resnet18

ResNet18 is a deep convolutional neural network architecture that publish in 2015. It is a smaller version of the popular ResNet50 and ResNet101 architectures, with 18 layers in total.

The architecture of ResNet18 is based on the concept of residual blocks, which are designed to alleviate the problem of vanishing gradients in deep neural networks. A residual block contains several convolutional layers with shortcut connections, which allow the gradients to flow directly from the input to the output of the block, bypassing the intermediate layers. This helps to preserve the information from the input and allows the network to learn deeper representations.

The architecture of ResNet18 is designed to be relatively lightweight, making it more suitable for deployment on resource-constrained devices. However, it still provides good performance on a variety of image classification tasks.

5.2 AlexNet

AlexNet is a deep convolutional neural network architecture that publish in 2012. It was the first architecture that demonstrated the power of deep convolutional neural networks for image classification tasks.

The architecture of AlexNet is composed of several convolutional layers, max pooling layers, and fully connected layers. It has a total of 8 layers, with 5 convolutional layers and 3 fully connected layers. The input image is passed through the convolutional layers, which extract features from the image. The max pooling layers are used to reduce the spatial resolution of the feature maps and to increase the robustness of the features to small translations in the input image.

AlexNet was a groundbreaking architecture and it set the foundation for the development of deeper architectures like VGG, Inception, and ResNet.

5.3 VGG16

VGG16 is a deep convolutional neural network architecture that publish in 2014. It is known for its very deep architecture, which contains 16 layers in total.

The architecture of VGG16 is based on a simple and uniform design, where multiple convolutional layers are stacked on top of each other, followed by max pooling layers to reduce the spatial resolution of the feature maps. The architecture can be divided into two main parts: the convolutional layers and the fully connected layers.

The VGG16 architecture is known for its high performance on image classification tasks, although its large number of parameters and high computational cost make it less suitable for deployment on resource-constrained devices.

5.4 Squeezenet1.0

SqueezeNet 1.0 is a version of the SqueezeNet architecture that publish in 2016. The main difference between SqueezeNet 1.0 and other versions is the number of layers in the architecture.

SqueezeNet 1.0 architecture is contain:

3 stem layers (2 convolutional layers and 1 max-pooling layer), 13 fire modules,

3 final layers.

The stem layers are the first layers of the architecture, they are designed to reduce the spatial resolution of the input image and extract the initial features.

The fire module layers are the building blocks of the architecture. Each fire module contains two layers: a 1x1 convolutional layer (also called "squeeze layer") and a 3x3 convolutional layer (also called "expand layer"). The squeeze layer reduces the number of filters in the feature maps, while the expand layer increases the number of filters in the feature maps.

SqueezeNet 1.0 has a very small model size, making it suitable for deployment on resource-constrained devices. Additionally, it has comparable accuracy to

much larger architectures like AlexNet and VggNet, but with a much smaller number of parameters.

5.5 DenseNet-121

DenseNet is a convolutional neural network architecture that connects all layers in a feed-forward fashion. It utilizes the concept of feature reuse by concatenating the output of one layer to the input of the next layer. This allows the network to learn more efficient representations of the input data, and can also help to mitigate the vanishing gradient problem. DenseNet is composed of dense blocks and transition layers. A dense block is a group of layers where each layer's output is concatenated to the input of all following layers within the block. The transition layers are responsible for reducing the spatial resolution of the feature maps and increasing the number of channels, this is done by applying a convolutional layer followed by a pooling layer.
DenseNet-121 is a specific implementation of the DenseNet architecture. It has 121 layers in total, with a growth rate of 32, and the number of filters in the first convolutional layer is set to 64.

6 Augmentation

Data augmentation is a technique used to artificially increase the size of a dataset by applying random modifications to the images in the dataset. These modifications can include things like rotation, translation, flipping, cropping, and changing the brightness or contrast of the images. The goal of data augmentation is to expose a model to a wider range of variations during training, which can help to make the model more robust and less prone to overfitting.

6.1 Random Rotation

The RandomRotation class (in torchvision.transforms module) applies a random rotation in the range [-degrees, degrees] to the images. The rotation is applied in the counter-clockwise direction. The class takes one argument, the degrees parameter, which controls the range of rotation angles that will be applied to the images.
We used RandomRotation with a degree's parameter of 20.

6.2 Random Resized Crop

The RandomResizedCrop class (in torchvision.transforms module) applies a random cropping to the images. It randomly crops an area with the specified size and aspect ratio.
The class takes several arguments, including the size of the output image.
It's worth noting that, RandomResizedCrop is useful for object detection, image classification, semantic segmentation, and other computer vision tasks where the position of the object in the image is not important.
We will use RandomResizedCrop with a size of 224 to perform a random crop of size 224 from the images we extracted which are of size 256. When we use this augmentation, we will remove the resize(224) transformation.

6.3 Gaussian Noise

Gaussian noise is a type of image augmentation that adds random noise to an image. The gaussian noise simulate the effect of image capture under different lighting conditions or sensor noise. The noise is generated from a Gaussian distribution, which is a probability distribution characterized by a mean and a standard deviation. This can help to make a model more robust to small variations in the input images, as the model will have seen similar variations during training.
We will use a standard normal distribution (mean 0 and standard deviation 1) and multiply it by a power of 0.01.

6.4 Normalize

Normalization is a technique used in transfer learning to standardize the input data before it is fed into the model. The pre-trained models, such as those trained on ImageNet, have been trained on a large dataset of images with a wide range of pixel values. Normalization helps to ensure that the input data has similar characteristics to the data the model was trained on, making it easier for the model to extract useful features from the input data. In the case of images, this typically involves subtracting the mean pixel value from each pixel and then dividing by the standard deviation.
We use Normalization according to the guidelines from Transfer Learning from models trained on ImageNet which is mean = [0.485,0.456,0.406], and std = [0.229,0.224,0.225].

7 Results

7.1 Reference Model

The model we made yielded a result of 52.9412%, these results are little disappointing, but not so surprising, because it is 15 classes with very few images for training each class, and as we know Deep Learning works well when there is a lot of data. In figure 5 you can see the accuracy of "Reference Model" on train and validation sets as function of epoch number during the training.

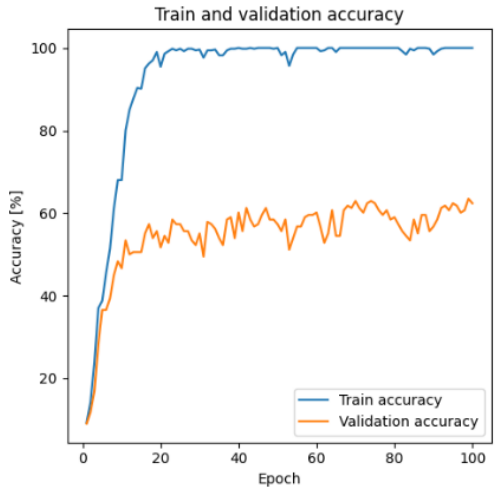

Figure 5: The accuracy of "Reference Model" on train and validation sets as function of epoch number during the training.

7.2 Transfer Learning - Feature Extraction

First, we examined the case of Feature extraction, this method is faster and converges faster than Fine tuning, the results for this case can be seen in Table 1.

model name	train accuracy	validation accuracy	test accuracy
resnet18	100.0000%	83.1461%	77.0588%
AlexNet	100.0000%	73.0337%	68.8235%
vgg16	100.0000%	74.1573%	61.7647%
squeezenet1_0	100.0000%	82.0225%	79.4118%
densenet121	100.0000%	81.4607%	75.2941%

Table 1: The results when using Transfer Learning in the case of Feature Extraction for 5 different architectures: resnet18, AlexNet, vgg16, squeezenet1_0, densent121.

The improvement is very impressive - the best result was obtained from squeezenet1_0 and is 79.4118% on the Test. In Figure 6 you can see the accuracy of squeezenet1_0 model on train and validation sets as function of epoch number during the training.

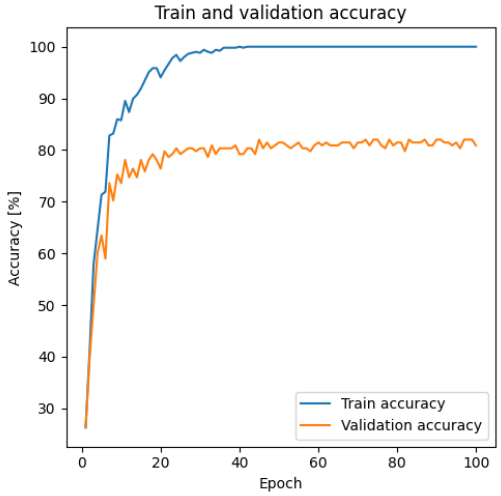

Figure 6: The accuracy of squeezenet1_0 model on train and validation sets as function of epoch number during the training.

7.3 Transfer Learning - Fine Tuning

We examined the training for the case of Fine Tuning. The results for this case can be seen in Table 2. We got a huge improvement of 14% from Feature extraction method, and an accuracy of 94.1176% on the test set for the ResNet18 architecture. **These are already very impressive results.**

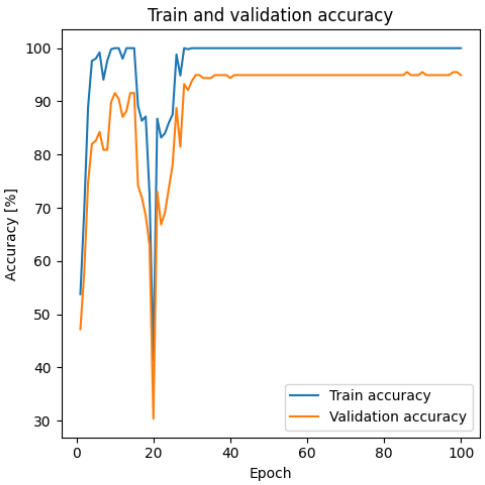

Figure 7: The accuracy of ResNet18 in fine tuning method on train and validation sets as function of epoch number during the training.

model name	train accuracy	validation accuracy	test accuracy
ResNet18	100.0000%	95.5056%	94.1176%
AlexNet	Stuck on low percentages and does not release even with a different learning rate		
vgg16	Out of Memory - Unable to train with our resources		
squeezenet1_0	99.2095%	61.2360%	52.9412%
densenet121	100.0000%	96.6292%	93.5294%

Table 2: The results when using Transfer Learning in the case of Fine Tuning for 5 different architectures: resnet18, AlexNet, vgg16, squeezenet1_0, densent121.

We note that for the AlexNet architecture, we accepted that the model is stuck at low percentages and does not improve even when changing the learning rate, and for vgg16 architecture, we accepted that it can't train with our resources since it requires a lot of memory.

7.4 Augmentation

We tested the augmentation described above on the best model - transfer learning with ResNet18 in fine tuning method. In table 3 you can see the results for each case. It can be seen that only using RandomRotation augmentation improved the results from the situation where we did not use augmentation at all.

Augmentation	train accuracy	validation accuracy	test accuracy
Random Rotation	100.0000%	96.6292%	96.4706%
Random Resized	94.6640%	91.0112%	80.5882%
Gaussian Noise	100.0000%	96.6292%	92.9412%
Normalize	100.0000%	96.6292%	94.1176%

Table 3: The results when using Augmentation for 4 different types: random rotation, random resize, gaussian noise, normalize, for the case of transfer learning with ResNet18 in fine tuning method.

In Figure 8 you can see the accuracy as a function of the epoch number for ResNet18 with augmentation - random rotation.

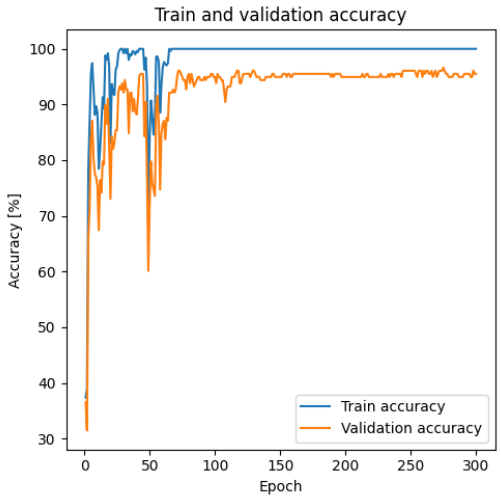

Figure 8: The accuracy of ResNet18 in fine tuning method and random rotation augmentation on train and validation sets as function of epoch number during the training.

We note that the number of epochs we did in the augmentation test is 300 compared to 100 when we tested the different between the architectures.
In order to prove that the improvement is not obtained because of increasing the epochs but because of the augmentation, we retrain ResNet18 without augmentations, but this time for 300 epochs instead of 100.
In this situation we get accuracy of 92.9412% (even less good result than 100 epochs). Thus, we proved that the improvement does not increase because of a larger number of iterations but because of the use of augmentation.

7.4 Best Model – Confusion Matrix.

A confusion matrix is a table that is used to define the performance of a classification algorithm. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class. The name stems from the fact that it makes it easy to see if the system is confusing two classes.

In Figure 9 you can see the confusion matrix for this case.

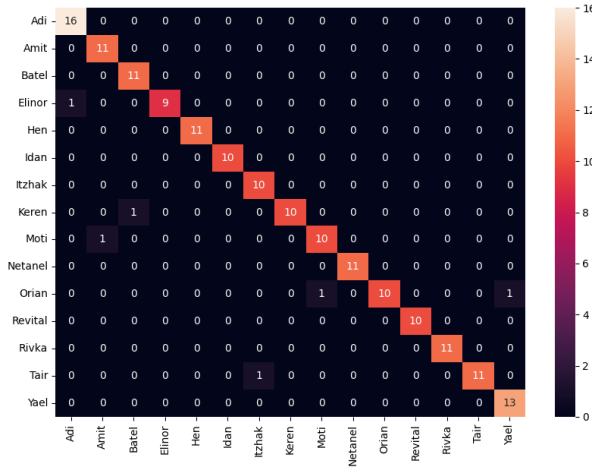

Figure 9: Confusion matrix of ResNet18 in fine tuning method and random rotation augmentation.

8 Conclusion

In this project we will first improve a standard face recognition algorithm in that based on classic ML, by implementing a loop on this algorithm when each time the image in the input of the algorithm rotates by 15 degrees.
We created the dataset for this project ourselves from photos of family and friends. We collect 50 images per user for 15 people.
After that we tested a number of Deep Learning models and trained them on the dataset we collected. We have shown the correctness of the claim that regular deep learning does not provide good results on a small dataset.
Later, we used Transfer Learning with two methods - Feature extraction and Fine tuning. We tested 5 different architectures (resnet18, AlexNet, vgg16, squeezenet1_0, dense121) with both methods.
We compared the results and found the model that provided the best results on the test set.
In addition, we tested 4 common augmentations to improve the generalization of the algorithm.
In conclusion, the best model we found in this project was obtained using the ResNet18 architecture when we performed Transfer Learning using the Fine-Tuning method. We improved the model results by a few percent by using random rotation augmentation in the range of [-20,20] degrees. **The final model yielded an accuracy of 96.4706% on the test set.**

9 Future Works

- In the first stage of face recognition in the image, it is possible to examine the use of deep learning in order to improve the recognition percentage of the algorithm.

- Additional architectures can be selected for Transfer Learning using both methods - Feature extraction and Fine Tuning.
- Creating a GUI for loading images by the user and automatically build and select the best model.
- Use of a camera module OV7670 and Arduino for hardware implementation and examination of the model.

References

[1] Implementing Face Recognition Using Deep Learning and Support Vector Machines: https://www.codemag.com/Article/2205081/Implementing-Face-Recognition-Using-Deep-Learning-and-Support-Vector-Machines

[2] Face Recognition with Python, in Under 25 Lines of Code: https://realpython.com/face-recognition-with-python/

[3] Attribute error while using opencv for face recognition: https://stackoverflow.com/questions/36242860/attribute-error-while-using-opencv-for-face-recognition

[4] Basic Operations on Images: https://docs.opencv.org/4.5.3/d3/df2/tutorial_py_basic_ops.html#:%7E:text=making%20borders%20for%20images%20(padding)

[5] ways to rotate an image by an angle in Python: https://www.askpython.com/python/examples/rotate-an-image-by-an-angle-in-python

[6] Loading Dating without a separate TRAIN/TEST Directory - Pytorch (ImageFolder): https://discuss.pytorch.org/t/loading-dating-without-a-separate-train-test-directory-pytorch-imagefolder/130656

[7] Intro-to-PyTorch- Loading Image Data: https://www.kaggle.com/code/leifuer/intro-to-pytorch-loading-image-data

[8] Loss function — CrossEntropyLoss vs BCELoss in Pytorch: https://medium.com/dejunhuang/learning-day-57-practical-5-loss-function-crossentropyloss-vs-bceloss-in-pytorch-softmax-vs-bd866c8a0d23

[9] SAVING AND LOADING MODELS: https://pytorch.org/tutorials/beginner/saving_loading_models.html

[10] Error loading saved model: https://discuss.pytorch.org/t/error-loading-saved-model/8371/5

[11] SAVE AND LOAD THE MODEL: https://pytorch.org/tutorials/beginner/basics/saveloadrun_tutorial.html

[12] How to resize an image with OpenCV2.0 and Python2.6: https://stackoverflow.com/questions/4195453/how-to-resize-an-image-with-opencv2-0-and-python2-6

[13] How to get pixel location in after rotating the image: https://datascience.stackexchange.com/questions/86402/how-to-get-pixel-location-in-after-rotating-the-image