Face recognition project

Team: Giancarlo Andriano, Joël Haubold, Natalia Markoborodova, Alan Barahona Ruiz

1 Project scope

The goal of this project is to train a Neural Network (NN) on being able to classify if a picture of fixed size represents a face or not. During the course of the project we were also able to use this NN to extract one or multiple faces from pictures of arbitrary size. Our work on this project can be represented by the following 5 steps:

1. Implement a basic learning loop including training and testing, to train the NN on classifying if a 36 by 36 picture is a face or not. (**Chapter 2-4**)
2. Build a coding framework to use the NN for face detection, including face detection, on pictures of variable size which might contain one or more faces. (**Chapter 5**)
3. Improving the NN as well as our post processing steps to achieve an improvement in accuracy of our detection and classification (**Chapter 6-8**)
4. Improving the speed of computation of the NN (**Chapter 9**)
5. Lastly defining further avenues for improvements of the NN (**Chapter 10**)

This report is structured alongside these steps which represent our workflow in this project.

2 Technologies

For developing a face recognition application we used such python libraries and NumPy and PyTorch. To evaluate the data we used pandas and to visualize the results we used OpenCV.

3 The Data

For this project we received training as well as testing datasets consisting of pictures the size of 36 by 36 pixels. These are labeled as containing either a face or something that is not a face. The split of the the datasets by these classes is as follows:

Dataset	**Class**	**Number of files**	**Share**
Train	Face	64,770	70.62%
Train	No Face	26,950	29,38%
Test	Face	797	10,45%
Test	No Face	6,831	89,55%

Table 1: Showing the Number of files as well as the share of each class in each dataset

While after step 1 of the project we switched off of the testing dataset to challenge the NN with more complex pictures we used the training training dataset to train the NN throughout the entirety of the project.

4 Learning loop

Our initial learning loop for, as well as the initial NN follow closely along the lines of the exemplary python files given to us at the beginning of the project. Our implementations can be seen in **test_parameters.py** as well as **net.py** file in the **JoelsChanges** branch.

Neural Network

For our Neural Networks we used three types of layers and operations:

Fully connected linear layers

These layers connect each input channel with each output channel with a variable weight, meaning every output channel can be dependent on every input channel.

2 dimensional convolutional layers:

These layers connect their 2 dimensional input and output through a Kernel of a defined height and width. Using for example a x by x Kernel, the top left most neuron of each output channel would be dependent on the neurons in the x by x square of each input channel, through a variable weight. As we do not use padding the dimensions of each output channel would be reduced by x-1.

2 dimensional max pooling operation

Analogously these operations connect their 2 dimensional input and output channels through a Kernel of a defined height and width. Each neuron in the output channel corresponds to the maximum value of the neurons in a possible kernel position over the neurons of the input channel. As such a x by x Kernel will divide the dimensions of the output channels by half compared to the input channel.

Our first Neural Network expects a 36 by 36 pixel input image and gives out an estimate of likelihood each for the image being a face and for the image not being a face. It is structured as follows :

Layer Number	Type	Kernel size	Number Inputs	Number outputs	Dimension at Output
1	Conv2D	5	1	6	32x32
2	MaxPool2D	2	6	6	16x16
3	Conv2D	5	6	16	12x12
4	MaxPool2D	2	16	16	6x6
5	Linear	-	576	32	1x1

6	Linear	-	32	16	1x1
7	Linear	-	16	2	1x1

Table 2: The structure of our first Neural Network. The columns show the order of the layers /operations from input to output, their type and Kernel size (if applicable) as well as the number of Input and Output Channels defined for each layer. The last column shows the dimension at the output channel of each layer.
Key: Conv2D - A 2 dimensional convolutional neural layer; MaxPool2D - A 2 dimensional max pooling operation; Linear - A fully connected linear neural layer.

We conceptually use convolutional layers to extract features from our input image and max pooling to extract the most relevant of these features. Then we use linear layers for classification, meaning we are reducing the features extracted so far by our Neural Network, down to the likelihood if the input represents a face or not.

To connect the two dimensional output of the 4th layer with the one dimensional input of the subsequent linear layers, we have to change the shape of the output tensor. Specifically we are changing it from containing 16 instances of 6 by 6 values, to one flat tensor containing 576 values.

Training

In preparation of the training of the NN we define the optimizer, the criterion used to determine the loss with each training iteration and the loader for the training data. To start out we chose a Stochastic Gradient Descent with a learning rate of 0.01 as our optimizer and a cross-entropy loss function as our criterion. The data loader took samples from 70% of the training data, with the other 30% being reserved for validation, which would be implemented later on.
To train the NN we run our training test loop for a defined number of epochs. In each epoch we iterate over the training loader and with each iteration step we apply a single step of the optimizer to the NN:

```
1 for data in train_loader:
2    optimizer.zero_grad()
3    images, labels = data
4    outputs = net(images)
5    loss = criterion(outputs, labels)
6    loss.backward()
7    optimizer.step()
```

Line 1 loads our training data in batches of 32 labeled images each, for each of these batches the following operations are performed.
In line 2 we reset the gradient of the optimizer from the values it had from the previous batch iteration step to zero, so that the gradients resulting from the previous steps don't directly affect the calculations in this iteration. In line 3 and 4 we split the images and their labels and get the output resulting from each image fed into the NN. Finally in line 5-7 we calculate the loss gradients from the outputs of the NN and the desired labels with our cross-entropy loss function.

Then we apply a single step of the optimizer, using these calculated gradients, to the weights of the NN.

Since the NN has two outputs each one regarding the probability for a class, the labels for our training data are always a 1 for one output and a 0 for another.

Testing

After every training epoch is completed, we test the accuracy of the NN with a single iteration over the provided training data.

```
1 for data in test_loader:
2   images, labels = data
3    outputs = net(images)
4    _, predicted = torch.max(outputs.data, 1)
5    total += labels.size(0)
6    correct += (predicted == labels).sum().item()
```

Line 1-3 are analogous to their equivalent in the previous training phase. Line 4 defines a classification from the output of the NN. For each input the class that the NN calculated with the highest likelihood gets chosen as the predicted class. Besides this comparison, the actual value of the NN outputs is not taken into account. Line 5 and 6 then calculate the amount of classified images, as well as the amount of correct classifications.

With these the accuracy of the NN can then be determined as:

$$accuracy = \frac{100 * correct}{total}$$

Results

We tested the NN with a different number of epochs to test the effects that the amount of training would have on the network.

Number of Epochs	Accuracy	Running TIme
5	90.96%	129.41s
10	92.21%	234.50s
15	92.22%	361.12s
20	92.81%	609.91s
25	93.67%	784.07s

Table 3: Accuracy and runtime depending on the number of epochs used to train the NN. The data is taken from the mean of 6 runs for each number of epochs

.

With a higher number of epochs comes a small increase in accuracy. That increase is of course offset by a linear increase in runtime.
Later in this project we will be testing the effect of other parameters, as well as a different NN model to seek for further improvements to classificational and runtime performance.

5 Sliding pyramid algorithm

The implementations for this chapter can be found in the **natasha** branch.
One problem that is present in the provided implementation of the NN is that it lacks a way to run against real images. That is to say that an image that is bigger than 36x36 cannot be given to the NN. To solve this, a sliding pyramid algorithm is used, which work as follows:

1. Create chunks of size 36x36 in the image, sliding through to get all the possible squares of 36x36 in the image. An example can be seen in Figure 1.

Figure 1: Visual representation of the sliding window on three iterations

2. Scale down the image by a specified factor, so that a chunk in the scaled down version covers a wider area in the previous image. The relationship between coordinates of the scaled and the original version is as follows, where f_s is the scaling factor, (x_0, y_0) the original position and (x_s, y_s) the scaled version:

$$f_s \cdot (x_0, y_0) = (x_s, y_s)$$

If the image has been scaled n times, then relation is as follows:

$$f_s^{\,n} \cdot (x_0, y_0) = (x_s, y_s)$$

A visual representation of the scaling can be seen in Figure 2..

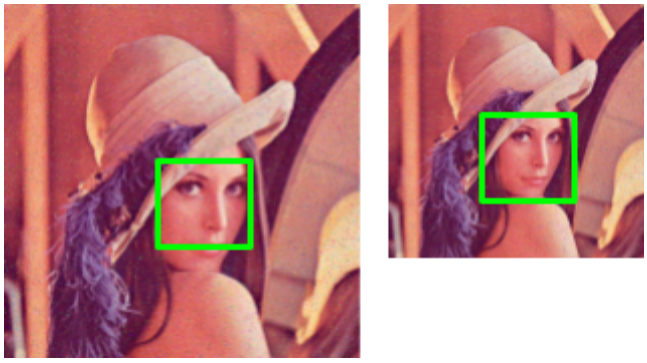

Figure 2: Visual representation of scaling down the image. The green squares are the same in both images.

3. If the image still cannot be contained in a chunk of 36x36 as a whole, then it is scaled down and the process is repeated. This holds true unless the resulting image is less than 36x36, then the process is finished.

The implementation of the algorithm has been provided. However, some adjustments and improvements needed to be made.

Representation of a detection

The original way that the implementation represents a detection is by storing two opposite corners of the square, starting with the upper left corner. However, the way it stores all the detections is convoluted. Using a json representation:

```
"detections_in_every_scale": [
    "detections_in_a_scale": [
        "scaling_factor": int,
        "detections": [
            "x0": int,
            "y0": int,
            "x1": int,
            "y1": int
        ]
    ]
]
```

This representation was changed in favor of:

```
"detections": [
      "detection": [
            "x0": int,
```

```
            "y0": int,
            "x1": int,
            "y1": int,
            "prob": float
        ]
]
```

This representation includes the probability of the detection of being a face. This value is used for the cleaning process.

6 Testing on real pictures

The implementations for this chapter can be found in the **natasha** branch.
After implementing the Pyramid Sliding Window algorithm we tried to test our model on a real picture. As a test picture we chose a JPG file of the size 1200x724 pixels with 5 faces: presumably 1 easily recognizable face, 2 slightly covered by hair, and 2 difficult to detect faces (partially covered by hair). The test picture is attached as Figure 3.

Figure 3: Test image of size 1200x724 pixels JPG. Corresponding to the band slowdive

First run of our model would detect more than 8000 faces and 83% network accuracy. The visualisation of the result is represented in Figure 4.

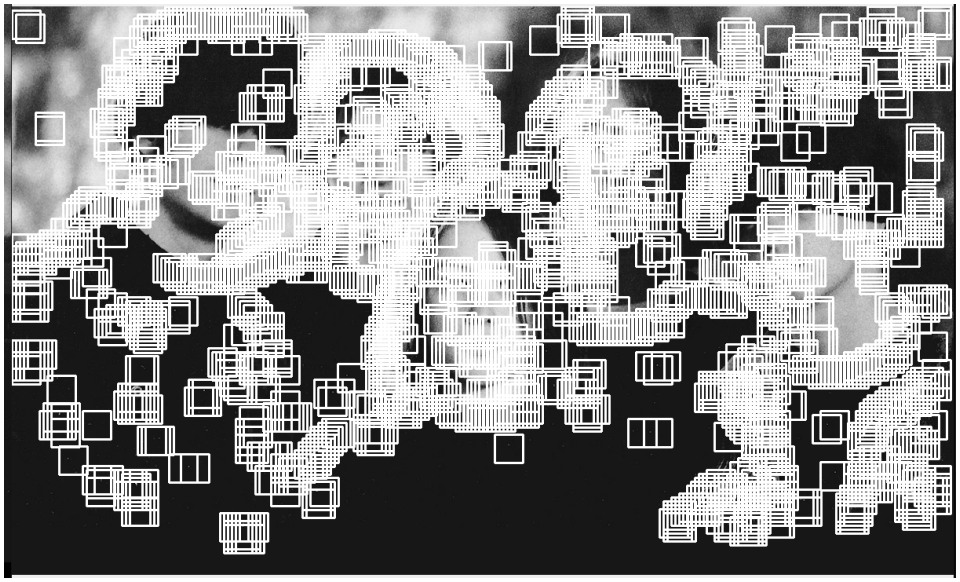

Figure 4: The model detected over 8000 in just one scale.

At first we tried to resolve the problem by adjusting the training parameters, such as number of training epochs and learning rate.
Setting a lower learning rate would only bring worse results. For example, decreasing it from 0,01 to 0,001 would cause deterioration of the network accuracy from 83% to 10% and the network would detect over 187000 faces, as one can see in Figure 5. Increasing the learning rate would also influence the results in a negative way. Therefore we decided to keep the learning rate parameter of 0,01.

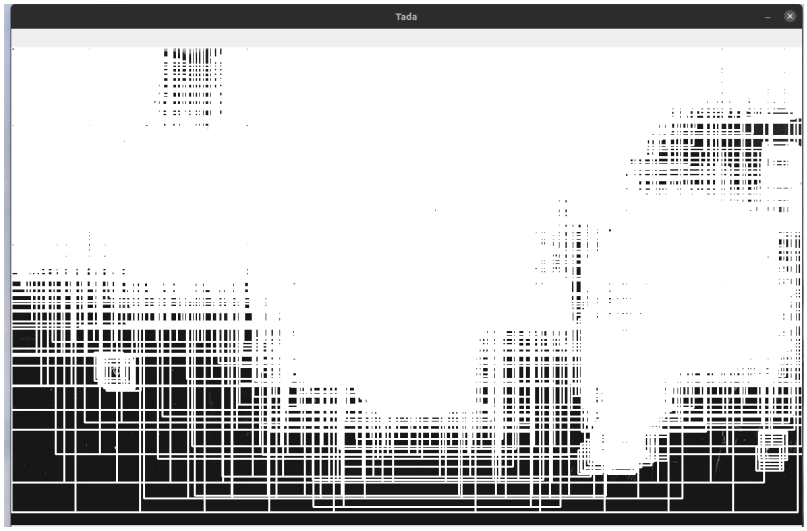

Figure 5: Lower learning rate causing higher number of detected faces.

Adjusting the number of epochs brought us slightly forward: with 10 epochs we managed to reach the network accuracy of 93% and to decrease the number of recognized faces to 2574, though the running time increased significantly.

After experiments with the above mentioned parameters the model was still not satisfactory at detecting faces on unprocessed images. Therefore, our next step was to add validation to the training process. We used the evaluation mode for this purpose and each time were saving the state of the model that had a lower loss than the previous one:

```
net.eval() # validation
for data in valid_loader:
   optimizer.zero_grad()
   images, labels = data
   outputs = net(images)
   loss = criterion(outputs, labels)
   valid_loss = loss.item() * len(data)

if min_valid_loss > valid_loss: # if loss smaller than last one
   min_valid_loss = valid_loss # update the best loss
   torch.save(net.state_dict(), 'saved_model.pth') # save the model
```

The results of the experiments are represented in Table 3.

Learning rate	Number of epochs	Validation enabled	Running time	Number of detected faces	Network accuracy
0,01	1	no	90,6s	3643	83%
0,001	1	no	89,6s	187376	10%
0,005	1	no	89,6s	40839	17%
0,02	1	no	91,2s	6705	81%
0,05	1	no	89,6s	9981	79%
0,01	10	no	394,2s	2574	92%
0,01	15	no	603,6s	3435	92%
0,01	20	no	759,7s	3684	93%
0,01	10	yes	914,2s	2856	93%
0,01	20	yes	1205,3s	1403	93%

Table 4: Performance of the network depending on different parameters.

Adding validation to the model improved the results but we were still having the problem of detecting too many faces. After proper analysis of the code we realised that the values for our

test picture were not normalized. So we normalized it by converting it to grayscale and by mean and standard deviation:

```
image_grayscale = cv2.imread(file, cv2.IMREAD_GRAYSCALE) # convert to grayscale
norm_image = cv2.normalize(image_grayscale, None, alpha=0, beta=1,
norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F) # mean = 0, standard deviation = 1
```

This change remarkably improved our model. The result of the change is represented by Figure 6. As we can see, at that point the model could recognize faces and there were not as many false positives as before. The next improvement to perform was to eliminate overlappings that would occur recognizing the same face but slightly shifted.

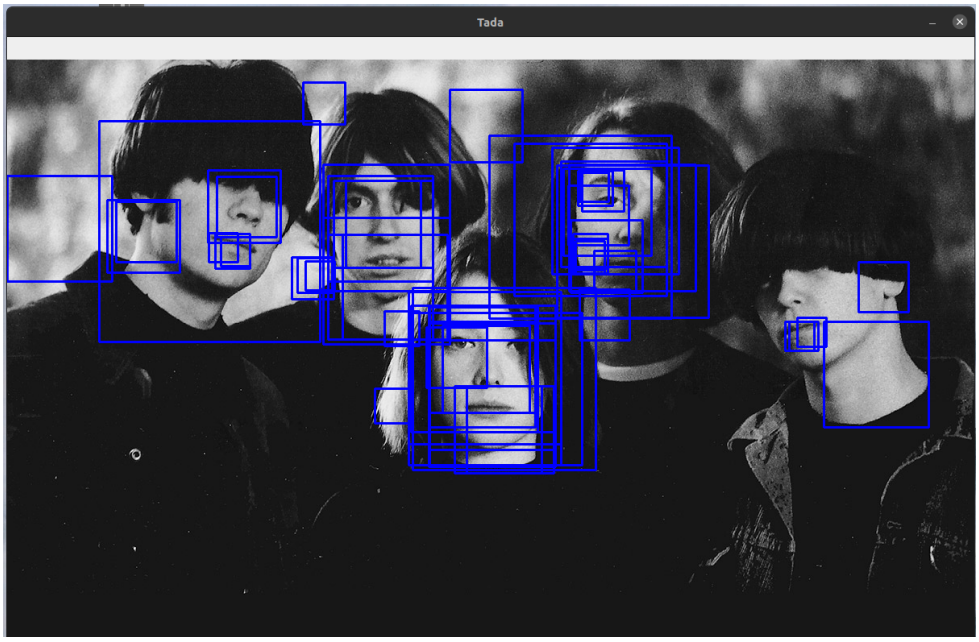

Figure 6: Model recognizing faces after normalization of the input data.

7 Non-maximum suppression

The implementations for this chapter can be found in the **natasha** branch.
Another problem was to prevent our network from recognizing the same face too many times. Considering a given scale, if one face is detected in (x_0, y_0), then the immediate surroundings have a high chance of being recognized as a face as well (given that the step at which the algorithm moves through the image is small enough). When including the fact that several scales of the image are analysed, then the problem arises again but in another scale. A visualization of this can be seen in Figure 7.

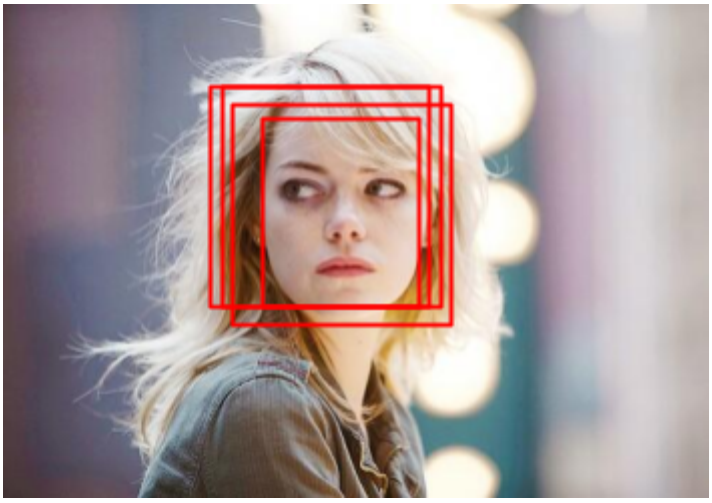

Figure 7: Visual representation of the proportion calculated by the intersection over union algorithm. It corresponds to the yellow area over the sum of the blue and red areas.

To prevent this, an algorithm called non-maximum suppression is used. The algorithm calculates the proportion of the area of the intersection over two areas in relation to the sum of both areas minus the intersection. In case that the proportion is greater than a certain threshold, then it is said that both squares are detecting the same face. A visual representation of this can be seen in Figure 8.

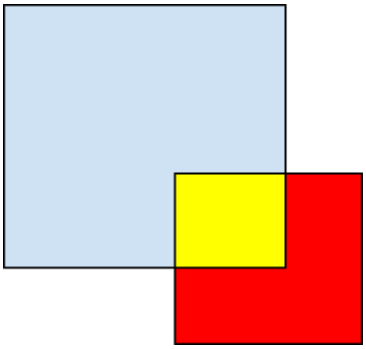

Figure 8: Visual representation of the proportion calculated by the intersection over union algorithm. It corresponds to the yellow area over the sum of the blue and red areas.

A special consideration is taken in the case that the intersection corresponds to approximately the same area of one of the squares. If said square is small enough, then the proportion of the intersection is minuscule, even though one of the squares is inside the other one. This situation can be seen in Figure 9.

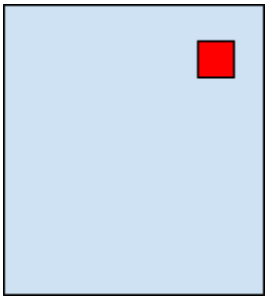

Figure 9: Visual representation of the visual representation of the special case.

8 Adding layers

Our implementations can be seen in **test_parameters.py** as well as **net.py** file in the **JoelsChanges** branch.
With the problem of overlapping faces being eliminated in post processing, our next big problem was the performance of the NN itself. As can be seen in the results given above, the current NN is prone to detect and classify faces in the background of pictures as well as in the clothing of people. To eliminate this problem we added two more fully connected linear layers to the NN to conceptually improve its ability to classify faces and non faces. This resulted in the following structure:

Layer Number	Type	Kernel size	Input Channels	Output Channels	Dimension at Output
1	Conv2D	5	1	6	32x32
2	MaxPool2D	2	6	6	16x16
3	Conv2D	5	6	16	12x12
4	MaxPool2D	2	16	16	6x6
5	Linear	-	576	32	1x1
6	Linear	-	32	20	1x1
7	Linear	-	20	16	1x1
8	Linear	-	16	8	1x1
9	Linear	-	8	2	1x1

Table 5: In black the changed parts of the structure of our second Neural Network.
The columns show the order of the layers /operations from input to output, their type and Kernel size (if applicable) as well as the number of Input and Output Channels defined for each layer. The last column shows the dimension at the output channel of each layer.

Key: Conv2D - A 2 dimensional convolutional neural layer; MaxPool2D - A 2 dimensional max pooling operation; Linear - A fully connected linear neural layer. The data was taken from the mean of 6 runs for each number of epochs.

To not increase the amount of weights of the NN by an amount, that could prolong the time its training would take noticeably, we kept the number of outputs of the fifth layer of the NN at 32, instead of increasing it.

Results with the test dataset

We first tested our new NN model with the testing dataset provided to us for this project:

Number of Epochs	Accuracy [old model]	Accuracy [new model]	Running TIme [old model]	Running TIme [new model]
5	90.96%	49.91%	129.41s	124.69s
10	92.21%	78.22%	234.50s	239.43s
15	92.22%	92.66%	361.12s	363.28s
20	92.81%	92.91%	609.91s	603.06s
25	93.67%	92.94%	784.07s	772.70s

Table 6: Difference in performance between the two NN models. The columns represent the number of epochs used in each run, as well as the accuracy and runtime for each model for those epoch numbers.

The new model did not take substantially longer to train and, at a higher number of epochs, could reach the same classificational performance as the old one. However at a lower number of epochs it performs notably worse than the old model. This is explained with the following table:

Number of Epochs	Number of runs	Runs with Accuracy < 11% [Old model]	Runs with Accuracy < 11% [New model]
5	6	0%	50%
10	6	0%	16.6%
15	6	0%	0%
20	6	0%	0%
25	6	0%	0%

Table 7: The potential for insufficient training that comes with more layers on a low number of epochs.
The columns represent the number of epochs used in each run, the number of runs performed with each

combination, and the percentage of runs under a defined threshold. The data was taken from the mean of 6 runs for each number of epochs.

When being trained only for a small amount of epochs the new NN model might not be trained to classify a face at all, and will perform substantially worse than even a random choice would.
To note is that the new model, when not falling below the 11% threshold, would perform just as well as the old model. This is the case even when using a low number of epochs.

While these first tests did not prove any improvements that the new model could provide to the classification process, its power would show itself in tests with real pictures, where face detection was a necessary step to the classification:

Results with real pictures

The implementations for this subchapter can be found in the **natasha** branch.
The results of this change when using the NN on real pictures is represented by the Figures 10, 11 and 12.

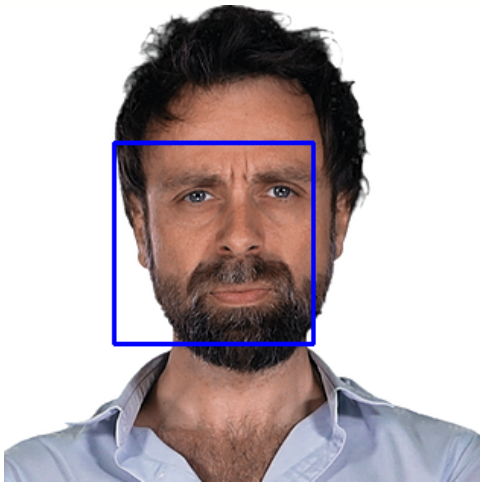

Figure 10: The model recognizes a single face with a beard.

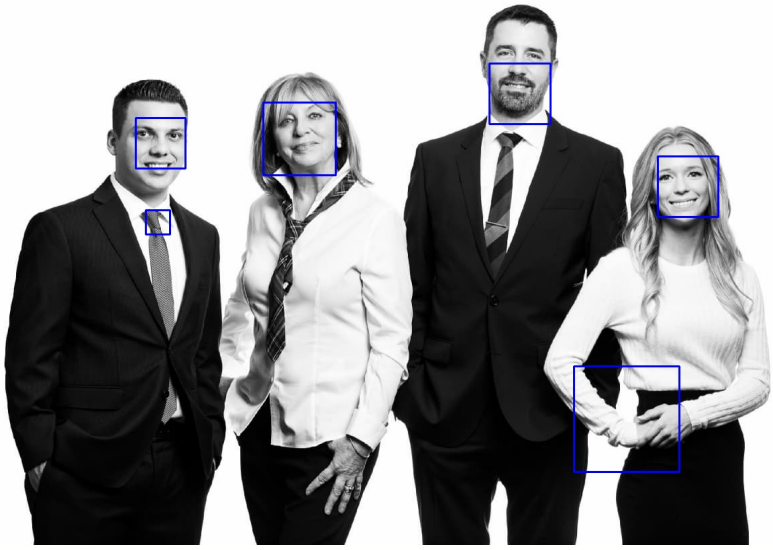

Figure 11: The model recognizes all the faces, two false positives are present.

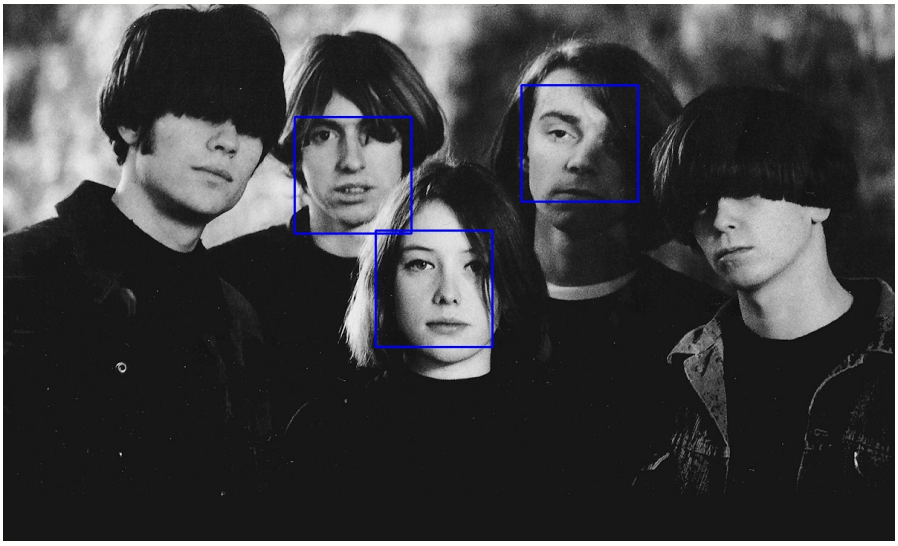

Figure 12: The model recognizes multiple faces. Two faces that we marked as difficult to recognize in the beginning were not recognized.

9 Cuda Processing and Parameters Evaluation

The changes in this chapter correspond to the **gianc** branch in the github.

Cuda Processing

For this purpose, we first needed to detect the device to use for computations since the code had to run in both cuda and non-cuda supporting laptops:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

We then used the notation ***kwargs* for passing different number of parameters to the *Test and Train Loaders* for managing the difference in settings between GPU and CPU computation:

```
kwargs = {'num_workers': 1, 'pin_memory': True} if device.type == 'cuda' else {}
```

At this point we ensured to load the network and test/training data into the right device, thus permitting the GPU (if available) to work instead of the CPU:

```
net = Net().to(device)
```

and we called the *to(device)* function also each time images and labels were taken from the two loaders. Furthermore, we also did similar modifications to the *Pyramid Sliding Window* algorithm in order to run it on the GPU.

Parameters Evaluation

After having set all improvements and changes to our model, it was time to find the best parameters for the training process. The approach was simple: we calculated the loss value for every different couples:

$$(\,learning\ rate\,,\ epoch\,)$$

using a logarithmic scale for the first one:

learning rate

	0.01	***0.025***	***0.05***	***0.075***	***0.1***	***0.25***	***0.5***	***0.75***	***1***
1	60.73	54.95	41.66	31.10	20.44	18.48	60.76	60.88	60.94
2	54.39	12.10	8.65	3.26	4.30	5.52	60.73	60.87	60.96
3	15.56	6.60	4.43	3.22	1.97	3.77	60.73	60.83	60.96
4	5.96	3.58	4.07	2.39	1.27	3.06	60.75	60.85	60.95
5	3.67	2.91	2.83	1.67	0.89	2.74	60.73	60.84	60.96
6	2.49	2.06	2.40	1.02	0.68	2.12	60.73	60.86	60.92
7	2.04	1.64	1.78	0.87	0.58	1.92	60.75	60.86	60.96
8	1.65	1.38	1.43	1.01	0.45	1.79	60.74	60.82	60.95

9	1.52	1.08	1.23	0.62	0.44	3.34	60.75	60.87	60.98
10	1.14	1.04	1.00	0.52	0.28	60.95	60.75	60.84	60.95
11	1.03	0.75	0.89	0.55	0.25	60.64	60.72	60.87	60.98
12	0.96	0.60	0.72	0.51	0.29	60.64	60.75	60.84	61.01
13	0.88	0.48	0.62	0.37	0.15	60.95	60.72	60.82	60.96
14	0.71	1.80	0.53	0.34	0.23	60.95	60.73	60.83	60.98
15	0.71	1.15	0.46	0.31	0.17	31.55	60.73	60.84	60.98

epochs

We made 2 observations:

- *Big values of learning rate are not appropriate for the training of a CNN*
- *Incrementing the number of epochs doesn't improve model efficacy*

It was clear then that *our model had many local minima*. So we could choose our final parameters between those who resulted in the smallest values of loss: we opted for the couple (green in the table):

learning rate = 0.1
n° epochs = 13

CPU vs GPU Performance Comparison

With these parameters we tested our NN model to evaluate performance improvements and timing difference between cpu and gpu executions. All tests were made using:

- GPU: Nvidea GTX 1050 Max-Q (mobile)
- CPU: i7-8565U CPU
- RAM: DDR4 2666 Mhz

$$GPU_{avg\ time} = 340.122 \text{ s}$$
$$CPU_{avg\ time} = 504.176 \text{ s}$$

thus resulting in *32.54% performance increase.*

10 Conclusion

Working on the project we reached a satisfactory level of facial recognition. Still the results are not perfect and we see a vector of further development. Due to the time restriction we were not able to perform all the improvements that we had in mind. In this chapter we are going to describe some possible steps to reach a better implementation.

1. Improve the performance of non-maximum suppression algorithms.
   There are certain situations when the implemented algorithm has complications in selecting which detection to remove. For example, this specific implementation works by taking a detection and treating it as a pivot with which to compare all the other detections. By doing this, the implementation is biased towards the order in which the detections are found. This means that the first detection will be prioritized even though it may not be the one that fits the recognized face the best. To solve this, we order the detections in terms of probability of it being a face. However, this is not enough. An example of this problem can be seen in Figure 11. The leftmost detection is not the best fit, but as is the one that got picked first, take prevalence over the other detections.

2. Make additional improvements to the network.
   One more possibility would be to extend our model by adding additional fully connected layers. We suppose that this could be impactful as we achieved significant improvements by adding two linear layers before.
   Another option would be to add more convolutional layers to the beginning of the network, to improve its data extraction capabilities. With the current methodology of using a max pooling operation after each convolutional layer, simply adding a convolutional layer would however result in less data beeing feed into the subsequent fully connected layers, due to the halving of the output dimensions with each max pooling operation, as well as the dimensional reduction through the kernel of the convolutional layers.
   This could be avoided by either using max pooling operations more sparsely, using some form of padding with these layers and operations or using a smaller kernel size.

3. Account for the imbalance between the two classes of face and no face in the training dataset.
   During this project we ignored the status of the training dataset as an unbalanced dataset on account of the imbalance not being overwhelming, with a ratio of 7 faces to 3 non faces. However taking it into account may still provide an improvement in either the classificational performance of the NN or the amount of epochs needed to sufficiently train it.
   This could be achieved either by implementation of a data-loader for the training dataset which would take the difference in numbers between the two into account when sampling, or by supplying the cross-entropy loss function with information about the imbalance.