Informatics Department

2nd Cycle - 5th Year

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

OT2: Machine Learning and Data Analytics

Report on the Deep Learning Project

PHUNG Minh Duc - PHAM Quoc Viet

1. Introduction

The objective of the Deep Learning project is to discover and develop a Convolutional Neural Network (CNN), specifically for a classifier that is used to recognize faces. Then, the classifier can be augmented to improve in performance and accuracy, before being used for detecting multiple faces in a single image, using different techniques.

In this project, we first construct a face classifier with a reasonably high confidence using the CNN. Then, utilizing the model of the classifier, we discover two different techniques for face detection: Sliding Window and Selective Search. Finally, we have a look at a pre-trained Region Based Convolutional Neural Network (R-CNN) implemented by PyTorch.

2. Model training pipeline

The code that was given contains only the hyperparameters and the data loaders required for training the model.

First of all, we coded the algorithm to train the model based on the given hyperparameters. We set the batch_size to be 64 and go for 32 epochs and found out that it may seem unnecessary as the total running loss stabilizes at around 15 epochs. For the optimizer, we used the Stochastic Gradient Descent optimizer with a learning rate of 0.0001.

The model had a 92% accuracy, which is already very good, and shows the quality of the dataset that we got. However, the process was painfully slow, as it would take up to almost an hour for the model to go through 20 epochs. Furthermore, the running loss was quite high, which may indicate that the model was not learning very well.

Therefore, we made some modifications to improve the performance and accuracy:

- The GPU is used instead of the CPU (if available) to massively speed up the training process.

```
# Use CUDA if possible
device = torch.device("cpu")
if torch.cuda.is_available():
    device = torch.device("cuda")
```

- The number of workers in the data loaders is set to 4 to load the data in parallel. Adding even more workers improved the performance for more powerful computers, but for some, it used too much of the GPU's memory and caused the entire process to crash. So, we settle at 4 to be sure that it works for everyone in our group.

```
# Dataloaders (take care of loading the data from disk, batch by batch, during training)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=train_sampler, num_workers=4)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler, num_workers=4)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True, num_workers=4)
```

- The optimizer was changed to Adam with a learning rate of 0.001.

```
optimizer = optim.Adam(net.parameters(), lr=0.001, weight_decay=1e-4)
criterion = nn.CrossEntropyLoss()
```

These modifications cause the model to have an accuracy of around 94%, and the running loss stabilizes at a low value of 2 to 4 per epoch. Plus, we were satisfied with our learning speed. As the model hasn't reached the desired accuracy of 98% yet, we tried to improve the model in other pipelines.

3. Augmented training pipeline

In this part, we will try to use an augmented training dataset to train our model. Data augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data. It includes making minor changes to the dataset or generating new data points. The goal of data augmentation technique is to make the data rich and sufficient and thus makes the model perform better and accurately. We will take a look at the accuracy of the model after training with an augmented dataset.

The data augmentation technique in our case was fairly simple: it consists only of randomly flipping horizontally 50% of the images of the original dataset. We then proceed with the same process as the first part.

```
# Define the transformations for augmentation
transforms_augmented = transforms.Compose([
        transforms.Grayscale(),
        transforms.RandomResizedCrop(36),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.5,),std=(0.5,))
])
```

The accuracy of the prediction on the test images is fairly higher than the model trained on the original dataset. It shows that a simple augmentation can lead to better performance and higher accuracy.

We then tried to mess with the parameters of the convolutional neural network and its architecture. However, we did not find a better set of parameters as the accuracy would diminish to 80-90%. Therefore, we kept the architecture that was given in the beginning.

This model is also saved for the next parts.

```
epoch: 1, running_loss: 745.0629272
epoch: 2, running_loss: 432.7305298
epoch: 3, running_loss: 341.3478088
epoch: 4, running_loss: 292.0621338
epoch: 5, running_loss: 267.2956848
epoch: 6, running_loss: 251.9956818
epoch: 7, running_loss: 238.5132904
epoch: 8, running_loss: 232.7164001
epoch: 9, running_loss: 224.1664276
epoch: 10, running_loss: 222.7513428
epoch: 11, running_loss: 206.9162445
epoch: 12, running_loss: 213.8405457
epoch: 13, running_loss: 203.5930023
epoch: 14, running_loss: 203.8099213
epoch: 15, running_loss: 201.2661285
epoch: 16, running_loss: 196.9024353
```

```
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on test images: %5.6f %%' % (
    100 * correct / total))
```

```
Accuracy of the network on test images: 97.981122 %
```

4. Imbalance training dataset

In this part, we will try to implement imbalanced data as the input. It refers to the situation where the distribution of classes in the target variable is not equal. In our case, we will try to train the model on a dataset that comprises much more non-face images than face images. Imbalanced data can cause classification algorithms to have a biased decision boundary. As such the algorithms may favor the majority class, leading to poor performance and low prediction accuracy for the minority class. We will see if the model cannot predict well the images showing a face - the minority class in our situation.

We load the data from the same folders and with the same transformation as in previous parts. However, we use the ImbalancedDatasetSampler class instead of the SubsetRandomSampler class, which will take more non-face images thana face images to our training dataset.

```
# Define two "samplers" that will pick examples from the training and validation set in an imbalanced way
train_sampler = SubsetRandomSampler(train_new_idx)
valid_sampler = SubsetRandomSampler(valid_idx)
```

The percentage of face images is only around 17%, which is very small compared to the number of non-face images (there's 1 face image for around 6 non-face images).

```
print("number of train samples: ", len(train_sampler))

nb_faces = 0
for data, target in train_loader:
    nb_faces += target.sum().item()

print("number of face images is: ", nb_faces)
```

```
number of train samples:  25904
number of face images is:  4349
```

We keep the same optimizer and criterion as the previous part to train our model, and it resulted in a 92% accuracy.

As the result seems to be a setback compared to the augmented model, we create a classification map that contains the number of true positive, false positive, true negative, and false negative of the predictions of our trained model.

```
classification_map = {"TP" : 0,
                      "FP" : 0,
                      "TN" : 0,
                      "FN" : 0}

correct = 0
total = 0
count = 0
with torch.no_grad():
    for data in test_loader:
        count+=1
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)

        for i in range(0,len(labels)):
            if predicted[i].item() == labels[i].item():
                if predicted[i].item() == 1:
                    classification_map["TP"] +=1
                else:
                    classification_map["TN"] +=1
            elif predicted[i].item() == 1:
                classification_map["FP"] +=1
            else:
                classification_map["FN"] +=1

        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %5.6f %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 91.858941 %
```

```
print ("TP: ", classification_map["TP"])
print ("TN: ", classification_map["TN"])
print ("FP: ", classification_map["FP"])
print ("FN: ", classification_map["FN"])

print("\n")

stats_map = {
            "Specificity" : float(classification_map["TN"]) / float(classification_map["TN"] + classification_map["FP"]),
            "Recall" : float(classification_map["TP"]) / float(classification_map["TP"] + classification_map["FN"]),
            "Precision" : float(classification_map["TP"]) / float(classification_map["TP"] + classification_map["FP"]),
            "Accuracy" : float(classification_map["TP"] + classification_map["TN"]) / float(classification_map["TP"] + clas
        }
stats_map["F-score"] = 2.0 / float((1.0 / float(stats_map["Precision"])) + (1.0 / float(stats_map["Recall"])))

for key, value in stats_map.items():
    print(key, ": ", value)
```

```
TP:  183
TN:  6824
FP:  7
FN:  614


Specificity :  0.998975259844825
Recall :  0.22961104140526975
Precision :  0.9631578947368421
Accuracy :  0.9185894074462506
F-score :  0.3708206686930091
```

Now, let's take a look at the performance metrics of the model. The scores are good enough, except for the Recall score and F score. The Recall score calculates the ratio of good predictions of face images over total number of face images. This score is very low (only 0.23 in this case) because the model predicted many false

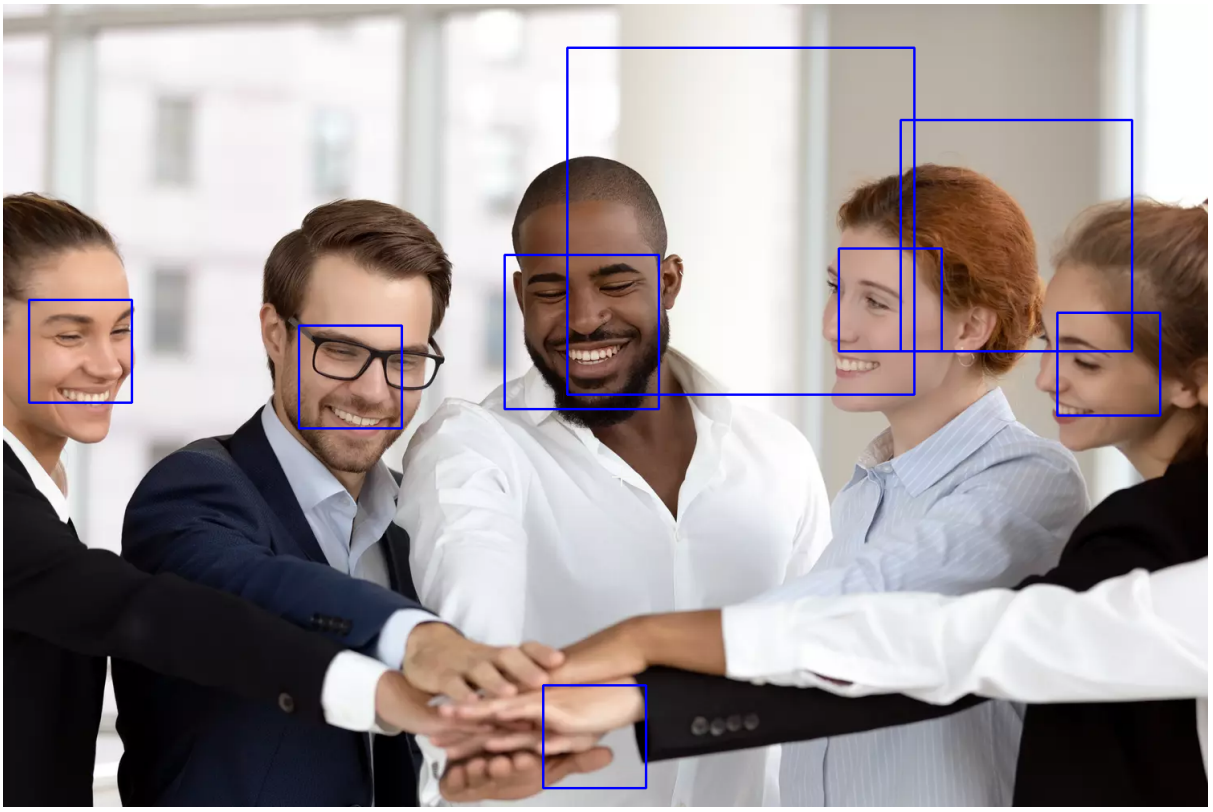
negatives. It is consistent with the known issue that with an imbalanced training dataset there will be low prediction accuracy for the minority class (in our case is the face image). Since the Recall score is low, then the F score is also low.

With the issue of having an imbalanced training dataset, it is necessary to avoid this problem. There are several techniques that can be implemented to reduce the effect of imbalanced data such as oversampling (adding more samples to the minority class), undersampling (removing samples from the majority class), or by using class weights.

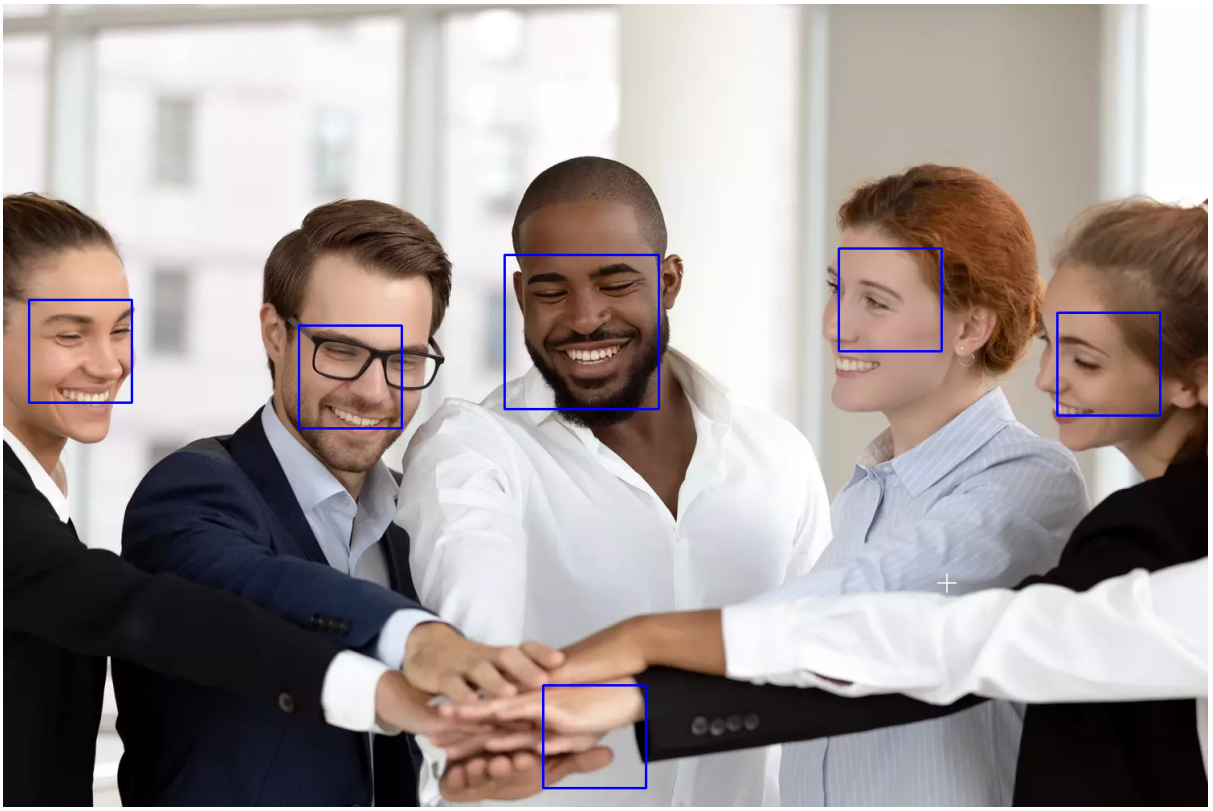
5. Face detection using the Sliding Window technique

After the classifier was trained, the next step was to use it for face detection. We used the sliding window to create an image pyramid, which was used to extract the region of interest (ROI) from the image. The ROI was then resized to 36x36 pixels and fed into the classifier. The classifier then outputs the probability of the ROI being a face. The ROI was then classified as a face if the probability was greater than 0.95, determined using PyTorch's softmax function. Non-maximum suppression (NMS) was then used to remove overlapping bounding boxes, using the implemented function in the Torchvision library.

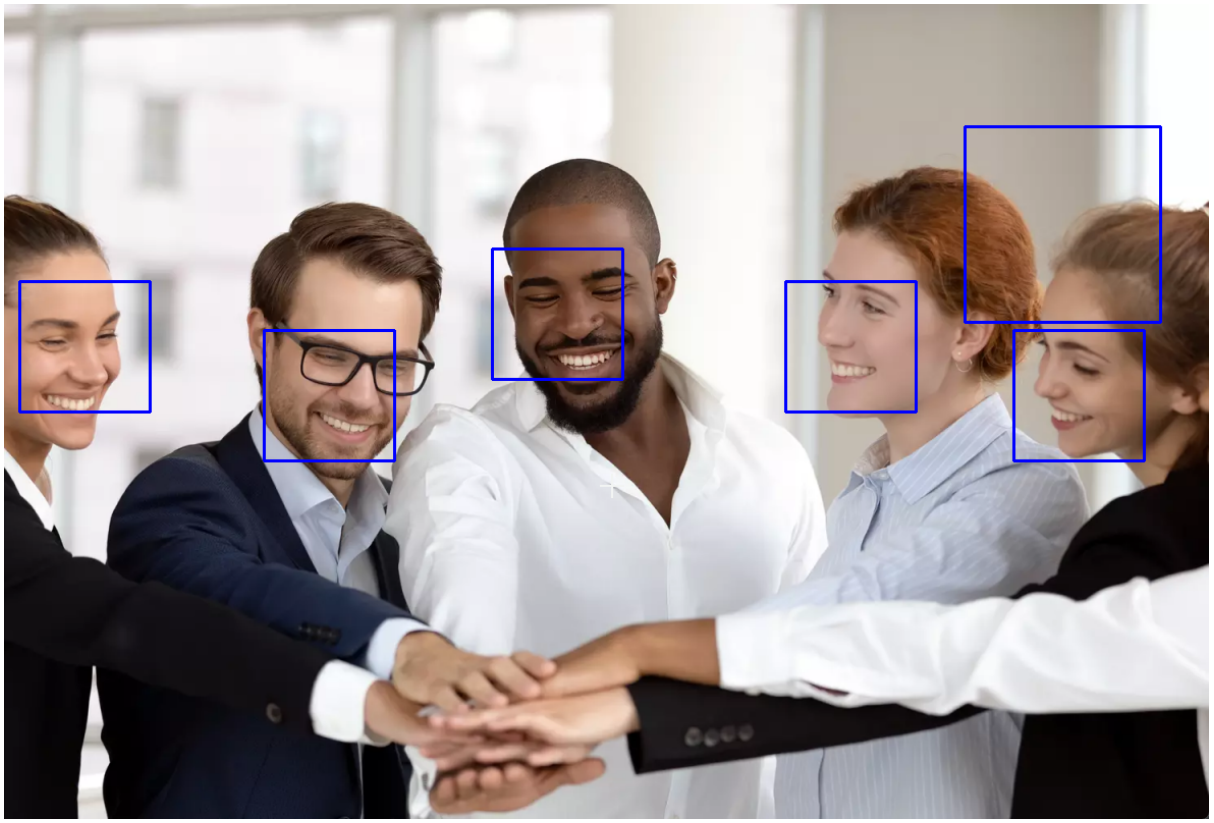
As for the results, The bounding boxes are not a perfect fit as its size is limited to multiples of a 36x36 square, but they were right on the faces. There were many false positive detections, which we were able to remove by reducing the threshold of the non-maximum suppression function, or by reducing the size of the original image. Having the NMS threshold at 5% with the image width of 1500 pixels gave us the same result as having the NMS threshold at 10% with the image width of 1200 pixels.



10% threshold - 1500x1000 size



5% threshold - 1500x1000 size



10% threshold - 1200x800 size

6. Face detection using Selective Search

Selective Search is a region proposal algorithm for object detection. It over-segments the image based on intensity, color, texture, and size. Then, it groups the similar regions together using a greedy algorithm. Finally, it outputs the bounding boxes of the grouped regions as region proposals.

Pipeline

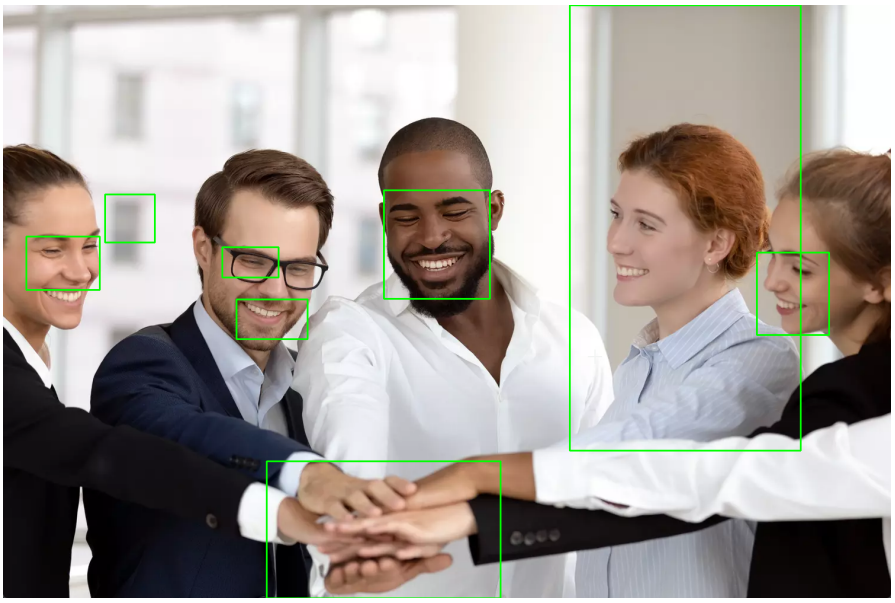
We made a face detection pipeline using the algorithm, which is implemented in OpenCV. It is as follows:

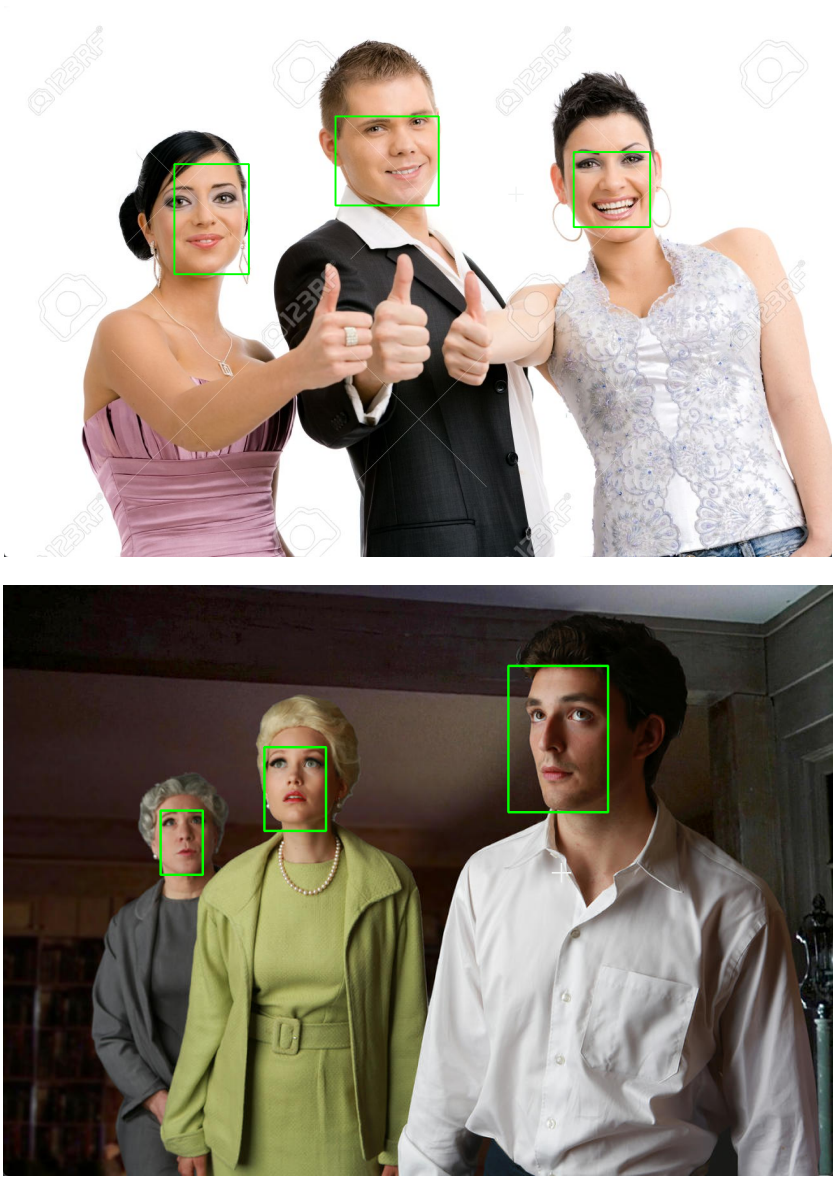
- Load the image
  The image is loaded in without any preprocessing.
- Run Selective Search
  The algorithm is run on the image to generate region proposals.
- Region filtering
  The regional proposals are filtered based on their size. If the regional proposal is too small, it is ignored. We used a threshold of 10% of the image size at first. However, since there was no preprocessing, for a bigger image, we got nothing out of the Selective Search algorithm. So, we reduced the threshold to 5% of the image size.
- Regional proposal preprocessing
  The regional proposals are converted to grayscale, and they are resized to 36x36 to fit the dataset we used.
- Extract features and classify
  The regional proposals are fed into the CNN to extract features and classify them as face/non-face.
- Non-maximum suppression
  The regional proposals are filtered based on the probability of being a face. We used a threshold of 90%.

Finally, the bounding boxes are highlighted in the original image.

Result

Even though the model has a 98% accuracy, the results were mixed. Sometimes the pipeline manages to detect most if not all the faces in an image, but for others it struggles. When it comes to speed, the pipeline took around 40 seconds for an image of size 1500x1000 even with a GPU (personally used Nvidia GeForce RTX 3050 Ti). This is because the Selective Search algorithm is very slow (or/and because there are other programs using the GPU in parallel). However, the CNN is very fast and takes only a few seconds to classify all the regional proposals.





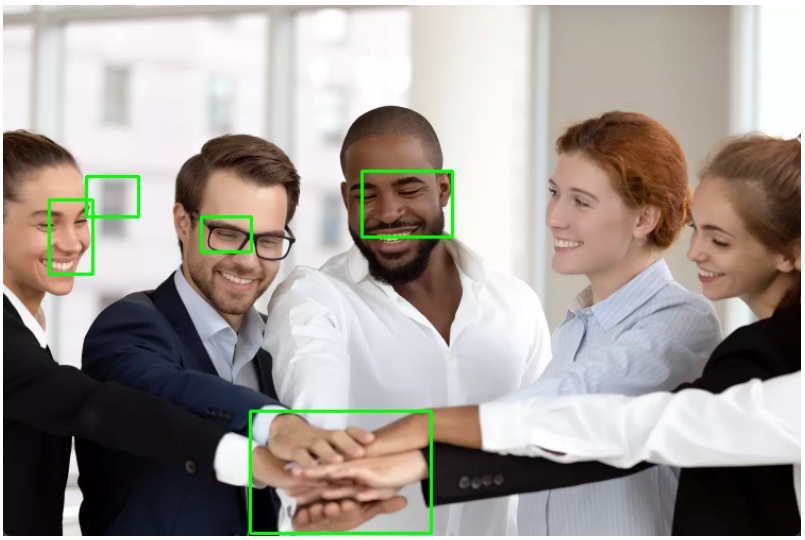
Some of the results from the face detection pipeline

To explain why there are differences in the results, there are many reasons:

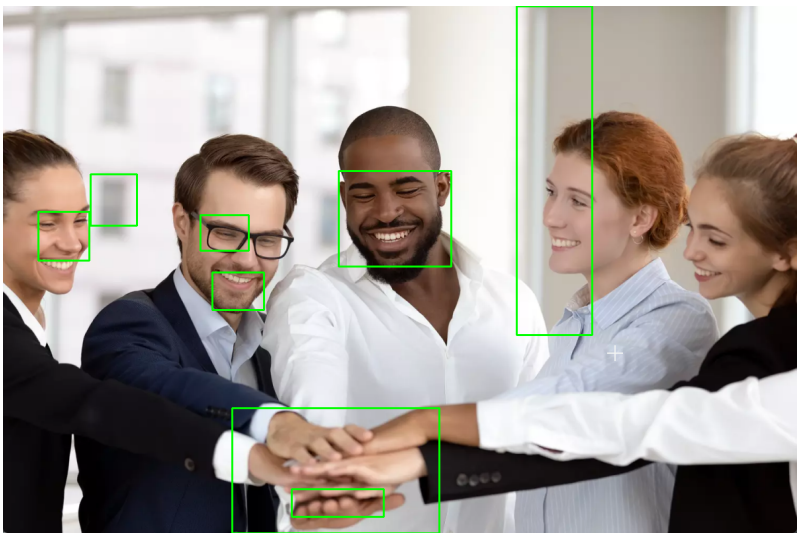
- The model we used is still not good enough.
- Image quality is inferior (lighting, people facing different directions,...)
- The Selective Search algorithm has issues due to the lack of preprocessing, which makes us have to fine-tune the hyperparameters manually for each case.

Improvements

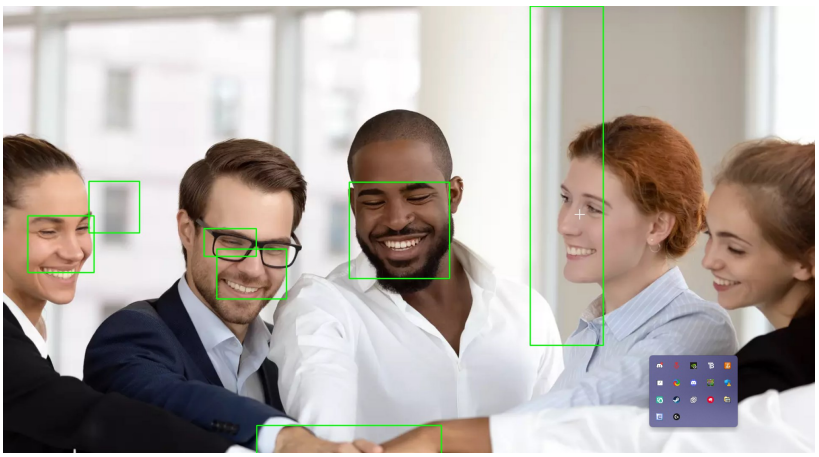
The third problem can be fixed by resizing the image to a fixed width (for us, we only make it so that the image gets smaller if it is bigger than a certain width, small images are not resized). This way, we can use the same hyperparameters for all the images. The results we got did not improve.



Size of 750x500



Size of 1200x800



Size of 1920x1280 (it was too big for our screen)

Therefore, we thought about the only simple reason: the image was not fitting for this model. The model probably has an issue recognizing faces that do not look straight. That is the reason why the middle person is detected without any issue, but for other people, it is basically impossible.

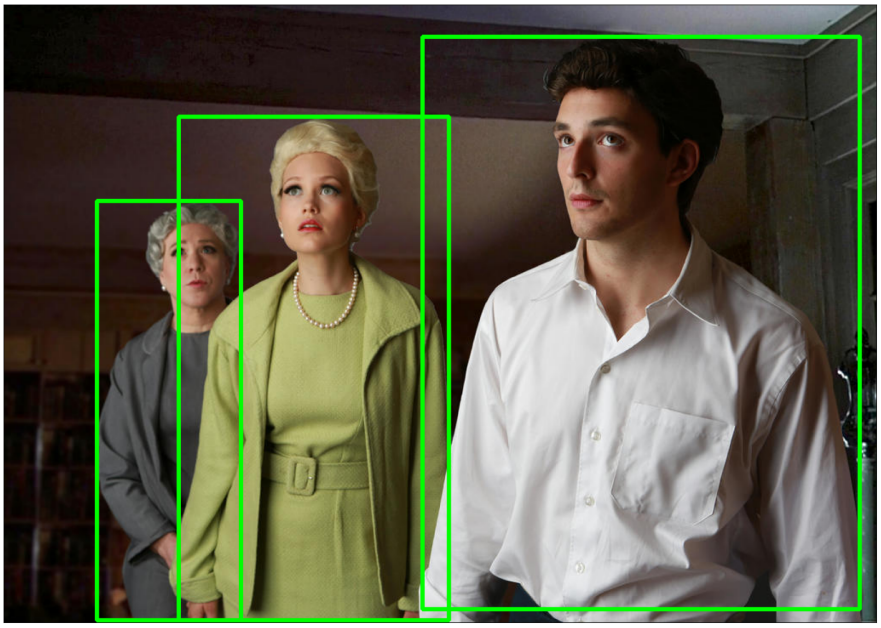
7. PyTorch's pretrained ResNet50 model for Object detection

Finally, for the R-CNN model, we decided to explore the ResNet50 Model pre-trained and implemented in Torch, as it is way too difficult to implement this from scratch. We did not manage to find another model that is trained to detect faces, so we decided to use this model instead. R-CNN is a visual object detection system that combines bottom-up region proposals with rich features computed by a CNN. It proposes a set of boxes in the image and see if any of them actually correspond to an object. It computes these proposal regions with a selective search algorithm. The model is trained on the COCO dataset, which can detect up to 80 different types of objects, but we are only interested in the "person" class. The model is trained to detect objects in images, and it returns the bounding box coordinates and the class of the object.

As it is a pre-trained model, implementing it was relatively easy. We just had to import the model, and then pass the image through the model to get the predictions. The model returns the bounding box coordinates and the class of the object. We then filter the predictions to only return the bounding box coordinates and class of the object if the prediction score is greater than the threshold. The threshold is set to

0.5, which means that the model is 50% confident that the object is a person. We then draw the bounding box on the image and display the image.

The result is not surprisingly, extremely accurate, as the person fits perfectly inside the bounding boxes. It is also extremely fast, as it only takes a few seconds to run the model and get the predictions.



ResNet50 R-CNN result

8. Conclusion

Through this project, we have gained necessary understanding of how a Convolutional Neural Network works by building a model to determine whether the content of an image is a human face as well as a face detection model. We have also pointed out some problems when processing training datasets and ways to overcome them. For the face detection part, there are some errors in our test images. This is due to the fact that the model used for this part is trained on an incomplete dataset. We can try to improve its performance by training the model with more samples but we should always be careful with overfitting the model.