Deep Learning for Face Recognition - Report

Mohamed Abderrhamne Taleb Mohamed, Stefania Curila, Alfredo Mahns,
Omar Ormachea Hermoza, Anna Wachtel

5IF - OT2 Project

Contents

Introduction

The number of databases composed of images and videos is increasing at a very fast pace each year. Manual analysis of such databases is no longer an efficient method, which creates the necessity of automatic and intelligent systems capable of exanimating and extracting information from these resources.

The face of a human being plays a crucial role in communicating the identity and the feelings of an individual. Therefore, automatic face detection has an important role in face recognition, facial expression recognition, head-pose estimation, human–computer interaction, emotion understanding and other fields. Face detection is a computer vision-based technique that determines the position and the size of a face in a given image.

The main goal of our project is using an efficient method for detecting faces in an image and understanding how variations of parameters and changes in the architecture of the model can influence in the performance of the algorithm.

The main structure of the model is composed of a convolutional neural network (CNN). The initial architecture of the model is modified to understand how these changes can influence the performance of the algorithm. To test the model, a sliding window method with a bigger input and number of targets is used. Then, a non maximum suppression algorithm is applied to improve the face selection. Finally, a bootstrapping method is applied to improve the accuracy of the model and to avoid overfitting.

Architecture

Convolutional Neural Networks

In Deep Learning, a Convolutional Neural Network, also known as CNN or ConvNet, is a class of artificial neural network (ANN) that can uncover key information in both time series and image data. CNNs are mostly used for classification and computer vision tasks. For identifying patterns within an image, a CNN uses principles from linear algebra, such as matrix multiplication.

The CNN architecture is similar to the connectivity pattern of the human brain. Just as the brain is made up of billions of neurons, CNNs also have neurons arranged in a specific way. In fact, the CNN's neurons are arranged like the brain's frontal lobe, the area responsible for processing visual stimuli. This arrangement ensures that the entire visual field is covered.

Model description

For our model we used 3 main layers: the convolutional, pooling and fully connected layers.

The main part of the CNN are the convolution layers. The convolution operation extracts high-level features such as edges, from the input image. The very first convolution layer usually extracts low-level features such as color or gradient orientation. When adding some more layers, the architecture can capture high-level features as well, giving us a network that has a more global understanding of the images in the dataset, similar to how humans would.

The Pooling layer reduces the spatial size of the convolved feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training the model. We chose Max Pooling because it also performs as noise suppressant.

The Fully connected (FC) layer is where image classification happens in the CNN based on the features extracted in the previous layers. Fully connected means that each node in the output layer connects directly to all nodes in the previous layer. All the layers in the CNN are not fully connected because it would be computationally expensive. Also, having only FC layers would increase losses and affect the output quality.

Our selected model uses the ReLU (Rectified Linear Unit) activation function. ReLU is a linear function that outputs the input directly if is positive, or zero otherwise. There are many other activation functions, but ReLU is the most used activation function for many types of neural networks today, because of its linear behavior. It is also easier to train, and it often achieves better performance.

The chosen architecture for our model uses 2 convolutional layers, 2 max pooling layers and 3 linear (fully connected) layers. After each convolution we apply the ReLU activation function and an additional max pooling layer. The feature maps resulting after the convolutions and the max poolings are then passed to the fully connected layers. For the first 2 linear layers we use ReLU as an activation function and for the last linear, which gives us the classifier's final result, we apply the SoftMax function in order to have a value between 0 and 1 for the face/non-face probability.

A schema of the architecture can be seen in the following image:

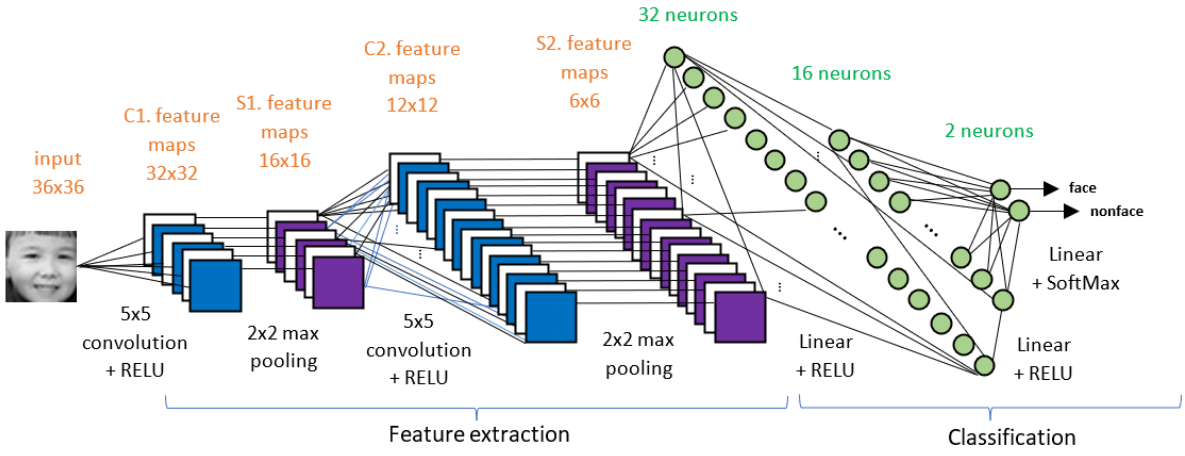

Fig 1. Schema of the selected CNN architecture

As input the CNN takes a normalized 36x36 grayscale image.

At first, we apply a 5x5 convolutional layer with 1 input channel (corresponding to the one input image), and 6 output channels. Therefore, we have 6 feature maps of size (36-5+1) x (36-5+1) = 32x32. Then we apply the ReLU activation function and a 2x2 max pooling layer which converts the new feature maps size to 16x16.

We add a second 5x5 convolutional layer which has 16 feature maps as output. Each one of the 6 input channels of the second convolutional layer is connected to all the resulting output feature maps. After applying the second convolution, the 16 feature maps being yielded as output have a size of (16-5+1) x (16-5+1) = 12x12. In the next step the ReLU activation function and another 2x2 max pooling layer are applied, which leads to 16 feature maps of size 6x6.

All the previous steps allowed us to do the feature extraction. Now we pass on to the classification part.

The first Linear will flatten our data (to 1D) and will connect the 16*6*6 input features to a hidden layer of 32 neurons. We use a second FC (linear) layer that will link the 32 input neurons to another hidden layer of 16 neurons. Finally, we use a third FC layer that will result in 2 neurons. As mentioned previously, after the first two linears we applied ReLU as activation function and for the last linear we used SoftMax.

Some tested models

We tried different architectures for our CNN to aim at improving the global and per class accuracy of the model and finally kept the one giving us the best accuracies.

At first, we used 1 epoch and set the batch size to 32. For the loss function we selected the cross-entropy. The cross-entropy loss is used when adjusting model weights during training and is suited for classification because it can measure the performance of a model whose output is a probability value between 0 and 1. Finally, for our first tries we used the Stochastic Gradient Descent (SGD) optimizer with a 0.9 momentum a 0.001 learning rate.

In the following subparagraphs you can find the different versions we worked through listed and explained.

1st try

To start off we implemented the network as described in the section '*Model Description*'. We also used the hyperparameters and functions presented before this subparagraph.

This resulted in the following network:

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=32, bias=True)
  (fc2): Linear(in_features=32, out_features=16, bias=True)
  (fc3): Linear(in_features=16, out_features=2, bias=True)
)
```

With this network we achieved 88.0% as overall accuracy, non-faced images were detected with 96.9% accuracy and faced images with 16.3% accuracy.

This architecture is the starting point for the all the following ones. The only elements that we changed compared to the 1st try are presented in each subparagraph of the next tries.

In a second step we tried to improve this accuracy by taking a closer look into the training dataset.

2nd try

After investigating the training dataset and how the faces and non-faces images are distributed, we recognized that there are significantly more images of faces than non-faces, being as there are 64,770 images of faces and only 26,950 images of non-faces. As a result of this imbalance, the model would be more biased towards detecting faces in images rather than non-faces.

To counteract this, we used the ImbalancedDatasetSampler as a sampler that would achieve a balanced training dataset. This sampler rebalances the dataset by estimating the sampling weights automatically.

```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
sampler=train_sampler, num_workers=1)
```

With this second try we achieved an overall accuracy of 88%, 98.0% accuracy for non-face images and 11.3% for faces.

Still not satisfied with the results, we continued to change our model to improve its accuracies further.

3rd try

In this version we used the Softmax as an activation function for the last FC layer in our model. The Softmax activation function is mostly used for classification tasks, as our task is too. When presented with an input vector this function computes a probability distribution for all the classes in the model. The different probabilities will together add up to equal one.

Generally, Softmax is used at the very end of the network where it receives the output of the last layer as its input, therefore we applied the Softmax function right after the last fully connected layer in our implementation:

```
m = nn.Softmax(dim=1)
x = m(self.fc3(x))
```

With this model we achieved an overall accuracy of 79% and 80% for non-face images and 70.9% for face images.

4th try

To continue our improvement, we changed the architecture in our fourth try. We changed the parameters of the fully connected layers to end up with more features for the hidden layers:

```
self.fc1 = nn.Linear(16 * 6 * 6, 64)
self.fc2 = nn.Linear(64, 32)
self.fc3 = nn.Linear(32, 2)
```

As a result, for this try we ended up with an 84% overall accuracy, 92.5% for non-faces and 17.2% for face images.

5th try

In a fifth try we again tried some different parameters for the fully connected layers, similar to the try before. We choose the following parameters for the number of neurons per layer:

```
self.fc1 = nn.Linear(16 * 6 * 6, 128)
self.fc2 = nn.Linear(128, 64)
self.fc3 = nn.Linear(64, 2)
```

In this try we achieved an overall accuracy of 88%, 96.4% accuracy for non-faces and 21.3% for face images.

6th try

As a next step towards improving our model we changed the optimizer from an SGD optimizer to an Adam optimizer. Adam's method is a stochastic optimization algorithm that implements an adaptive learning rate, whereas in normal SGD the learning rate has an equivalent type of effect for all the weights/parameters of the model.

This is especially benefiting in deep networks, where the optimal learning rate varies across parameters. For example, gradients should be small at early layers and the learning rates for other parameters can then be adjusted correspondingly.

Another advantage to Adam is, that because the learning rates are adjusted automatically, manual tailoring is not needed, which would be necessary in SGD. It is only required to select the hyperparameters of the model, but its performance is less sensitive to these parameters when using Adam rather than SGD.

In our implementation we applied Adam as the optimizer as following.

```
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

For our model this resulted in an overall accuracy of 89%. Non-faced images are detected with 94.5% accuracy and faces with 50.3%.

7th try

For our seventh try we adjusted the learning rate in the SGD optimizer and decreased it from 0.01 to 0.1 to see how this affects our model and its accuracies.

```
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

As a result, we achieved an overall accuracy of 92%, 98.3% for non-face images and 47.1% for faces.

8th try

As a next step for improvement, we tried a similar approach as before, but changed the learning rate of the Adam optimizer to 0.01.

```
optimizer = optim.Adam(net.parameters(), lr=0.01)
```

This ended up with an overall accuracy of 87%, 95.9% for non-faces and 14.4% for images with faces.

9th try

In another try we increased the number of epochs from one to two and went back to the SGD optimizer with its learning rate of 0.01. Additionally, we chose to use the sigmoid activation function after the first two fully connected layers, instead of the ReLU function.

```
n_epochs = 2
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
x = torch.sigmoid(self.fc1(x))
x = torch.sigmoid(self.fc2(x))
```

This gave us an accuracy of 91% and 99.1% for non-faced images and 26.3% for faced images.

10th try

For this 10th try we chose the same architecture and model as in the previous 9th try but exchanged the sigmoid function back to a ReLU function.

```
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
```

We ended up with 93% for the overall accuracy, 97.5% for non-faced images and 56.5% for faced images.

11th try

In this 11th try we chose the same model as before in the 10th try but increased the number of epochs to three.

```
n_epochs = 3
```

This resulted in an overall accuracy of 92% and 99.9% for non-faces and 32.7% for faces.

As we were satisfied with these results, we decided to keep this model and continue to work on it in our next steps.

Below we plotted the evolution of the value of loss of every 200 batches of images.

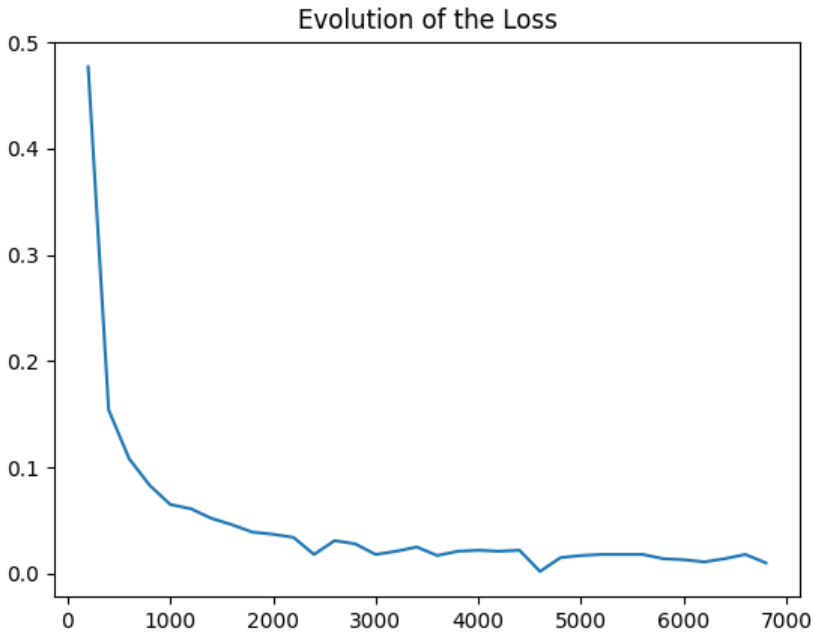

Fig2. Evolution of the loss as function of number batches

12th try

In order to experiment a little further, we decided to do some more changes to our model and observe their effects.

In this step we changed the architecture by adding an additional fully connected layer, as follows:

```
self.fc1 = nn.Linear(16 * 6 * 6, 32)
self.fc2 = nn.Linear(32, 16)
self.fc3 = nn.Linear(16, 8)
self.fc4 = nn.Linear(8, 2)
```

We observed an overall accuracy of 93%, 99.4% for non-faces and 39.1% for detecting faces in images.

Observations

We kept the 11th model because it was the most accurate one when testing it with the sliding window algorithm which is presented in the next section. Considering that we don't want to overlearn, and we also want to apply the bootstrapping technique which will improve our model, we believe that 92% (accuracy of the 11th model) is a good overall accuracy to have at this point.

The training set is highly imbalanced, with 26950 nonfaces and 64770 faces. We tried to balance it in the second try using the ImbalancedDatasetSampler function, but it didn't improve our results significantly. We consider that the bootstrapping would be a better technique for dealing with this imbalance.

To obtain a probability as an output of our neural network, we should have used the SoftMax activation function for the last layer. In most of our architecture tries we didn't apply it directly in the neural network and therefore we don't have good results for the per class accuracy. When running the sliding window in the next step, we applied the SoftMax manually after receiving the output of the neural network, so we get the expected results.

After we selected the best model, the next steps are applying the sliding window and the bootstrapping.

Sliding window

Once the model is trained, we apply the sliding window technique to test it on a bigger image containing multiple faces. The sliding window is a rectangular region of fixed width and height (36x36 pixels in our case) that "slides" across an image, such as in the following figures indicate:

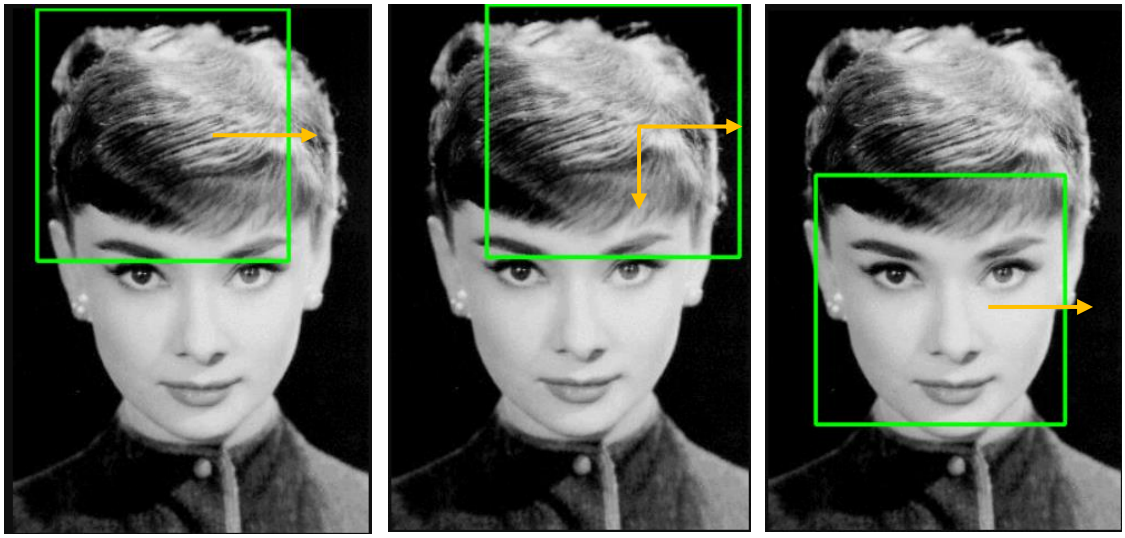

Fig3. Illustration of the siding window

The window loops over the whole image, it slides to the right with an exact step size given as a parameter. We used a step size of 6 pixels. The content of each window is passed as an input to our classifier which then determines if the window contains a face or not.

Because we do not know the exact size of a possible face in an image, we combined this algorithm with the image pyramids that allows us to detect varying scales and locations of the faces in the image. So instead of having one loop over the image, we do it many times while rescaling the image with -20% each time.

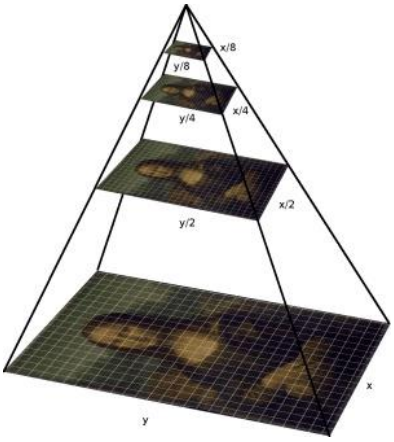

Fig4. Example of image pyramids

We saved all the faces which have been detected with a probability higher than a detected threshold. In our case, after trying different values, we decided to take *0.994* as threshold.

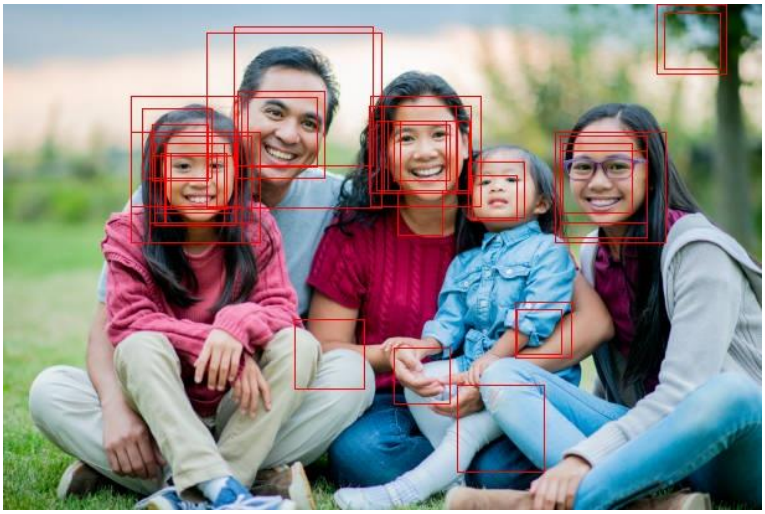

Fig5. Results of the sliding window using 11th try (net_11.pth)

All the detected faces are saved in the *cropped* folder and are then merged in the final image where the faces are marked with a red rectangle.

We implemented these algorithms in separate functions. The `pyramid()` function implements the image pyramids algorithm, in `sliding_window()` we implemented the sliding window algorithm and `apply_sliding_window_image_piramid()` applies the two previous functions on the input image and then feeds the windows content into the

neural network. Lastly, `save_final_image` shows all detected faces with a red rectangle in a final image.

The next step is applying the Non-Maximum Suppression algorithm in order to group the overlapping detected faces.

Non-Maximum Suppression

Because the detected faces were overlapping, we decided to apply the Non-Maximum suppression algorithm which selects the best entities over a set of overlapping entities. To meet this objective, many criteria can be used, such as probability thresholds or overlapping measures.

The criteria we used is called Intersection over Union (IoU), also referred to as the Jaccard index. The Jaccard coefficient (1) measures similarity between finite sample sets, and is defined as the division between the size of the intersection and the size of the union of the sets:

$$J(S_1, S_2, \ldots, S_n) = \frac{|S_1 \cap S_2 \cap \ldots \cap S_n|}{|S_1 \cup S_2 \cup \ldots \cup S_n|} \quad (1)$$

Previously, the sliding window method was presented. The input received by our selected criteria needs to be an area represented by various bounding boxes, which are equivalent to the sample sets in this case. The necessity of applying this algorithm comes from the fact that many of the bounding boxes are overlapping, in other words, many of the detected faces are overlapping. Therefore, the Jaccard index has to be adapted to meet this need (2):

$$IoU(Box_1, Box_2, \ldots, Box_n) = \frac{|Box_1 \cap Box_2 \cap \ldots \cap Box_n|}{|Box_1 \cup Box_2 \cup \ldots \cup Box_n|} \quad (2)$$

Next, a mask is defined to improve the results of the NMS algorithm. To create this mask, a threshold is defined, so that every element with an IoU index score higher than the defined threshold will not be considered in the resulting sets. The threshold needs to be selected with an empiric observation to determine the best values.

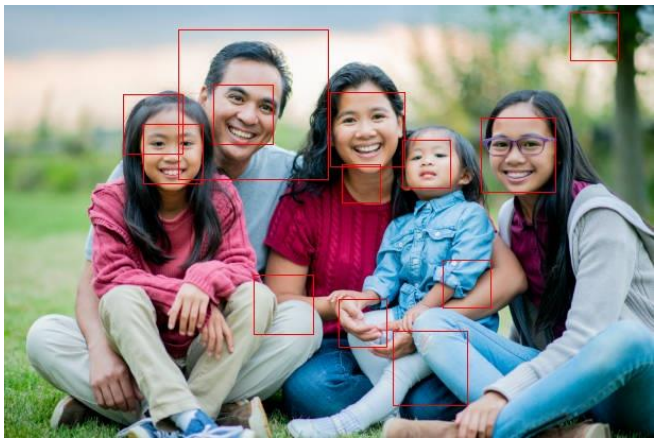

Fig 6. Results after applying NMS algorithm (using net_11.pth)

The NMS function that we implemented takes as input the selected faces from the sliding window along with the prediction probability and the overlap threshold that we set at 0.2. At first it selects the prediction with the highest confidence, and it calculates the IoU with the remaining predictions. If the IoU is greater than the threshold, the compared predicted face is removed from the list. In the end, the function outputs the remaining filtered faces that normally won't overlap.

Bootstrapping

Bootstrapping is a method used in Machine Learning to improve the stability of an algorithm and avoid overfitting the model. As an important resampling technique, bootstrapping presents the learning algorithm repeatedly with new randomly selected samples from the original data set. We followed a slightly different approach. Instead of re-sampling from the original dataset, we iteratively extended the dataset with an increasing number of "false alarms", taken by applying the trained model on a set of images that contain no faces. This would serve the purpose of rebalancing the classes as well, since we have many more samples of faces than non-faces.

Our main purpose is to achieve more accurate results on the sliding window procedure, removing the areas where the algorithm detected non existing faces like for example in the background or on the ground.

We started by selecting the additional set containing images of non-faces on which our model would later test on, we will call it the "texture set". It consists of a set of images of different textures, created by Mircea Cimpoi, Subhransu Maji, Iasonas Kokkinos, Sammy Mohamed, and Andrea Vedaldi in their paper "Describing textures in the wild". This data set consists of 5.640 images, each being between 300x300 and 640x640 pixels in size. For our purpose, we took 1780 images and rescaled them by a factor of 1/3.

As a first step in our bootstrapping algorithm, we started by selecting a validation set out of the training data with the same amount of faced images and non-faced images. We chose to select a validation set of approximately 1.000 images per class, where both classes have the same number of faces as non-faces. This resulted in a validation set of size 2.072 images. We removed these images from the training data set.

After that we could start the actual bootstrapping process.

The algorithm works on a balanced subset of both faces and no-faces, meaning that at any point, the training data set has the same number of faces as no-faces images. As we started with the entire training data set for the iteration (minus the validation set), we started off with around 25.000 images. We trained our neural network on these images with the same parameters as for our candidate model (11th try - `epochs=3`; cross entropy loss with `lr=0.01`, `momentum=0.9`; `SGD` optimizer).

In a next step we continued by gathering false alarms that the model detected. In this step we used the texture set.

After adjusting the images in the texture set to have the same size and format as the images from the training data set, we ran our texture images through the sliding window algorithm and counted each detected face for which its confidence was above a previously defined threshold. This operation keeps going until either all the texture set has been tested or once we counted enough false alarms. As threshold for the confidence, we initially chose 80% and as the number of false alarms at which our algorithm should stop, we chose 12.845, which is equal to one fifth of the total images containing faces.

Lastly, we added the detected false alarms to our updated training data set and decreased the confidence threshold by 20%, until reaching 0%. This was done in order to focus on the strongest false alarms at the first iterations. At the first iterations, we expect the classifier to perform poorly so we ignored the more redundant false alarms below the given threshold. Later on, the constraint is relaxed, since we expect the classifier to perform better so that the weaker false alarms would not necessarily be redundant / irrelevant.

The process was reapplied on the updated training data set for 6 iterations.

We tested the performance of the bootstrap model by the general accuracy test, without any surprising result, and on the sliding_window.py test. This test takes an image of a group of people, and outputs the same image with a square box in the detected faces. Here, the NMS algorithm was utilized to improve the final result by merging the boxes that overlapped a lot. The overlapping could be caused by small displacements in the window that allow to capture the same face, or when the same face is captured after a resizing of the image.

Fig 7. Results after applying bootstrapping

The results for this test were positive. On iteration 0 (using only the original training data set), the model (3eps-iter-0.pth) already performs better than the candidate model (net_11.pth).

The only difference is that 3eps-iter-0.pth trains with much less face samples in order to keep the balance of classes. On iteration 1, the first appending of false alarms occurs, and performance seems to improve a lot, despite the number of false alarms not being very high at first. At iteration 2 (model 3ep-iter2.pth), the model improves even more, and it is actually the best model in terms of performance on the sliding_window test.

From iteration 3 on, the model performs worse. This may be due to overfitting since the texture images are always the same even if they produce different false alarms.

Possible improvements

There are multiple changes we could make to improve our model.

Once we finished implementing our model and the improvement methods, we should test it with a validation set. The validation set was created with 20% of the images, but we didn't have enough time to implement the validation of the model. For the selection of the best candidate model, we computed the accuracy using the test dataset instead of creating a validation set from the training data. On the bootstrap procedure, instead, a validation set was extracted and excluded from the training set. To keep it balanced, we tried to make it small (around 1.000 samples per class), since the original number of non-faces was quite small compared to the face class. This led the validation set to not be very informative. The tests done on it were giving very high overall and per-class accuracy even before the first bootstrapping iteration (> 97 %).

As mentioned in the *Observations* section, we should apply the SoftMax to the output of the last fully connected layer directly in the net.py file, and not do it manually in the sliding window function.

Another possible improvement could be increasing the number of epochs.

Also, we could increase the dataset either by adding more images or augmenting the data by applying some minor transformations (rotation, enlighten/darken, add noise, blur, shift a bit). The parameters of these operations can be defined either before training or during the training with some initial random parameters that can be improved during the process.

Conclusion

In summary, this project aimed at building a Convolutional Neural Network, that is able to detect faces in images.

We progressively improved our network by changing its architecture and parameters. Once the model was trained, the sliding window method was applied to test images of different sizes and various faces. Next, we used the Non-Maximum Suppression algorithm and Bootstrapping on the model to achieve better accuracies.

In final we are satisfied with the accuracies we ended up with and are confident, that our model can be used in countless real-life problems and situations to detect faces correctly. A downside to our model is that we did not use a validation set. Nevertheless, we are pleased with our results.

Sources

1. Training a Classifier — PyTorch Tutorials 1.13.0+cu117 documentation [Internet]. [cited 2022 Dec 2]. Available from: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

2. Rosebrock A. Sliding windows for object detection with python and opencv [Internet]. PyImageSearch. 2015 [cited 2022 Dec 2]. Available from: https://pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/

3. Non maximum suppression: theory and implementation in pytorch [Internet]. 2021 [cited 2022 Dec 2]. Available from: https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/

4. A comprehensive guide to convolutional neural networks — the eli5 way [Internet]. Medium. 2022 [cited 2022 Dec 2]. Available from: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

5. What are convolutional neural networks? [Internet]. Enterprise AI. [cited 2022 Dec 2]. Available from: https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network

6. How to Implement the Softmax Function in Python [Internet]. W&B. [cited 2022 Dec 2]. Available from: https://wandb.ai/krishamehta/softmax/reports/How-to-Implement-the-Softmax-Function-in-Python--VmlldzoxOTUwNTc

7. What are convolutional neural networks? [Internet]. [cited 2022 Dec 2]. Available from: https://www.ibm.com/cloud/learn/convolutional-neural-networks

8. optim.Adam vs optim.SGD [Internet]. Medium [cited 2022 Dec 2]. Available from: https://medium.com/@Biboswan98/optim-adam-vs-optim-sgd-lets-dive-in-8dbf1890fbdc

9. Imbalanced Dataset Sampler [Internet]. GitHub [cited 2022 Dec 2]. Available from: https://github.com/ufoym/imbalanced-dataset-sampler

10. Garcia C, Delakis M. Convolutional face finder: a neural architecture for fast and robust face detection. IEEE Transactions on Pattern Analysis and Machine Intelligence [Internet]. 2004 Nov;26(11):1408–23. Available from: https://liris.cnrs.fr/Documents/Liris-6132.pdf

11. Example of image pyramids https://929687.smushcdn.com/2633864/wp-content/uploads/2015/03/pyramid_example.png?lossy=1&strip=1&webp=1

12. Describable textures dataset [Internet]. [cited 2022 Dec 2]. Available from: https://www.robots.ox.ac.uk/~vgg/data/dtd/