PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

Contributions in Face Detection with Deep Neural Networks

Thomas da Silva Paula

Dissertation submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Rodrigo Coelho Barros

Porto Alegre
2017

Ficha Catalográfica

P324c Paula, Thomas da Silva

Contributions in Face Detection with Deep Neural Networks / Thomas da Silva Paula . – 2017.
72 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Rodrigo Coelho Barros.

1. Deep Learning. 2. Face Detection. 3. Neural Networks. 4. Machine Learning. 5. Computer Vision. I. Barros, Rodrigo Coelho. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).

Thomas da Silva Paula

Contributions in Face Detection with Deep Neural Networks

This Dissertation has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Computer Science of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on March 28, 2017.

COMMITTEE MEMBERS:

Prof. Dr. Ricardo Matsumura de Araújo (UFPEL)

Prof. Dr. Felipe Rech Meneguzzi (PPGCC/PUCRS)

Prof. Dr. Rodrigo Coelho Barros (PPGCC/PUCRS - Advisor)

I dedicate this work to my parents.

“Wisdom is not a product of schooling but of the lifelong attempt to acquire it.”
(Albert Einstein)

ACKNOWLEDGMENTS

Most of all, I would like to thank everyone that helped me to arrive here. A huge thank for my parents Danilo and Karina, that did not measure any efforts for helping me in my life, for always encouraging me to take even larger and more challenging steps, and supporting all my personal decisions with love and wisdom.

To my grandparents Lourdes, Nelson (in memoriam), Jurema, and Ori. They taught me how to be a humble person, to be persistent, and to be a dreamer.

A very special thank for my girlfriend Amanda, who always stood by my side, helping me to overcome difficult moments, and making the joy moments even better. You make me really happy.

I would like to thank my advisor Rodrigo Barros for showing me the world of Machine Learning, for which I have fallen in love and turned into my professional and personal passion. I am grateful for all his support and guidance while working in my research.

I would also like to thank two professors that encouraged me to pursue a master's degree since the beginning of my graduation, Juliano Varella and Gabriel Simoes.

A special thank you for all my colleagues of our research group GPIN, who helped me along the way in many presentations, exams, and difficult moments along my master's degree. They are friends for life.

Another special thank you for some friends that worked with me and supported my choice of taking the challenge of a master's degree while working in a company. Rodrigo Wesz, Alberto Wondracek, Ricardo Angrisani, Rafael Cardoso, Roberto Reznicek, and Luciano Alves, a huge thank you for all your support while working with me at HP. You encouraged me to take this step.

Finally, I would like to thank two very special friends Christian Perone and Roberto Silveira. They helped me while I was writing this dissertation, giving advices and support. I am really grateful to have met you guys.

CONTRIBUTIONS IN FACE DETECTION WITH DEEP NEURAL NETWORKS

RESUMO

Reconhecimento facial é um dos assuntos mais estudos no campo de Visão Computacional. Dada uma imagem arbitrária ou um *frame* arbitrário, o objetivo do reconhecimento facial é determinar se existem faces na imagem e, se existirem, obter a localização e a extensão de cada face encontrada. Tal detecção é facilmente feita por seres humanos, porém continua sendo um desafio em Visão Computacional. O alto grau de variabilidade e a dinamicidade da face humana tornam-a difícil de detectar, principalmente em ambientes complexos. Recentementemente, abordagens de Aprendizado Profundo começaram a ser utilizadas em tarefas de Visão Computacional com bons resultados. Tais resultados abriram novas possibilidades de pesquisa em diferentes aplicações, incluindo Reconhecimento Facial. Embora abordagens de Aprendizado Profundo tenham sido aplicadas com sucesso para tal tarefa, a maior parte das implementações estado da arte utilizam detectores faciais *off-the-shelf* e não avaliam as diferenças entre eles. Em outros casos, os detectores faciais são treinados para múltiplas tarefas, como detecção de pontos fiduciais, detecção de idade, entre outros. Portanto, nós temos três principais objetivos. Primeiramente, nós resumimos e explicamos alguns avanços do Aprendizado Profundo, detalhando como cada arquitetura e implementação funcionam. Depois, focamos no problema de detecção facial em si, realizando uma rigorosa análise de alguns dos detectores existentes assim como algumas implementações nossas. Nós experimentamos e avaliamos variações de alguns hiper-parâmetros para cada um dos detectores e seu impacto em diferentes bases de dados. Nós exploramos tanto implementações tradicionais quanto mais recentes, além de implementarmos nosso próprio detector facial. Por fim, nós implementamos, testamos e comparamos uma abordagem de meta-aprendizado para detecção facial, que visa aprender qual o melhor detector facial para uma determinada imagem. Nossos experimentos contribuem para o entendimento do papel do Aprendizado Profundo em detecção facial, assim como os detalhes relacionados a mudança de hiper-parâmetros dos detectores faciais e seu impacto no resultado da detecção facial. Nós também mostramos o quão bem *features* obtidas com redes neurais profundas — treinadas em bases de dados de propósito geral – combinadas com uma abordagem de meta-aprendizado, se aplicam a detecção facial. Nossos experimentos e conclusões mostram que o aprendizado profundo possui de fato um papel notável em detecção facial.

Palavras-Chave: Aprendizado Profundo, Reconhecimento Facial, Redes Neurais, Aprendizado de Máquina, Visão Computacional.

CONTRIBUTIONS IN FACE DETECTION WITH DEEP NEURAL NETWORKS

ABSTRACT

Face Detection is one of the most studied subjects in the Computer Vision field. Given an arbitrary image or video frame, the goal of face detection is to determine whether there are any faces in the image and, if present, return the image location and the extent of each face. Such a detection is easily done by humans, but it is still a challenge within Computer Vision. The high degree of variability and the dynamicity of the human face makes it an object very difficult to detect, mainly in complex environments. Recently, Deep Learning approaches started to be applied for Computer Vision tasks with great results. They opened new research possibilities in different applications, including Face Detection. Even though Deep Learning has been successfully applied for such a task, most of the state-of-the-art implementations make use of off-the-shelf face detectors and do not evaluate differences among them. In other cases, the face detectors are trained in a multitask manner that includes face landmark detection, age detection, and so on. Hence, our goal is threefold. First, we summarize and explain many advances of deep learning, detailing how each different architecture and implementation work. Second, we focus on the face detection problem itself, performing a rigorous analysis of some of the existing face detectors as well as implementations of our own. We experiment and evaluate variations of hyper-parameters for each of the detectors and their impact in different datasets. We explore both traditional and more recent approaches, as well as implementing our own face detectors. Finally, we implement, test, and compare a meta learning approach for face detection, which aims to learn the best face detector for a given image. Our experiments contribute in understanding the role of deep learning in face detection as well as the subtleties of changing hyper-parameters of the face detectors and their impact in face detection. We also show how well features obtained with deep neural networks trained on a general-purpose dataset perform on a meta learning approach for face detection. Our experiments and conclusions show that deep learning has indeed a notable role in face detection.

Keywords: Deep Learning, Face Detection, Neural Networks, Machine Learning, Computer Vision.

List of Figures

2.1 Difference between traditional and deep learning approaches for the image classification problem. In deep learning, the feature extractor is learned. 40

2.2 Complex forms are a combination of simple forms. The top shows parts of a complex form (for instance, eyes and mouths), whereas the bottom shows complex forms themselves. (Image adapted from [68]). 41

2.3 A typical ConvNet architecture [67]. 42

2.4 Example of convolution applied on an image using a kernel to detect edges [2]. . . 42

2.5 Effect of different stride values in the output feature map. The red arrow indicates the kernel displacement caused by the stride. 44

2.6 Example of convolution with a 3×3 kernel over an input of 5×5 and a stride of 1. The same kernel is applied while it slides over the input. 45

2.7 Example of max pooling with a 2×2 sliding window, an input of 4×4, and stride of 2. The colors indicate the positions of the sliding window. 46

2.8 Translation invariance of the pooling operation. The red values are changes applied on the original input within each highlighted region. Although the values changed, the output values did not. 46

2.9 Example of fully-connected layers with one input layer, two hidden layers and an output layer. 47

2.10 AlexNet architecture in details. The two branches are used to illustrate the GPU parallelism, where each branch is processed by a different GPU [60]. 49

2.11 Example of simple Softmax classification output. 50

2.12 Example of log-likelihood loss computation for two different images. The first one is a correct classification while the second is an incorrect classification. 51

2.13 Comparison between image classification, object detection, and object recognition. 53

2.14 Values of IoU for different predictions. The green bounding box represents the grount truth. The red one is the prediction. 54
2.15 How R-CNN works (Image adapted from [35]). 56
2.16 A network with spatial pyramid pooling layer [42]. 58
2.17 How Fast R-CNN works (Image adapted from [34]). 59
2.18 VGG-19 architecture. 59
2.19 How Faster R-CNN works. 61
2.20 Region Proposal Network [93]. 62
2.21 Overview of MTCNN pipeline [132]. 63
2.22 MTCNN networks in details: P-Net, R-Net and O-Net [132]. 64
2.23 Simple example of Content-based Image Retrieval (Images from SIMPLIcity dataset [121]). 66
2.24 Classical siamese network architecture [16]. 67
2.25 Example of a siamese convolutional neural network. The output of both networks are joined through a layer that computes distance, which is a $L2$ (Euclidean) distance in this case. The output is a score that indicates how similar the two input instances are. 69
2.26 CNN architecture that maps the MNIST data to a low dimensional space (Image from [41]). 69
2.27 Experiments that demonstrate the dimensionality reduction results. A is an easy setup, where the digits are simpler. B is a hard setup, where some noisy images are used. (Images from [41]). 70
2.28 Results from the product search implementation based on siamese networks. (Image from [9]). 70
2.29 Intuition on the triplet loss objects. Through a learning procedure, it minimizes the distance between the anchor and the positive and maximizes the distance between the anchor and the negative objects.(Image adapted from [102]). 71
3.1 Examples of images of WIDER FACE dataset. 74
3.2 Example of Haar-like features used by Haar Feature-based Cascade Classifiers. *A* and *B* show two-rectangle features. *C* shows a three-rectangle feature whereas *D* shows a four-rectangle feature [120]. 75
3.3 To calculate the integral image, we can imagine that a blue line is drawn in a given region and all pixes above and to the left of that region are summed up. For instance, *A*, *B*, *C*, *D*, and *E* illustrate different portions being calculated. *F* has the resulting integral image. 76
3.4 Stages of cascade classifier. Each stage contains specific Haar-like feature extractors. The input is considered a face only if it passes through all stages. It is not considered a face if the verification fails in any of the stages. 77

3.5 LBP selects a given center pixel (10, in this case) and compares it to all the neighbor pixels. If their value is larger or equal to the center pixel, it is assigned the value 1. Otherwise, it is assigned to 0. The red arrow shows how the binary string representation is created. Afterwards, these binary representation is converted to a decimal number, which results in the feature extracted by LBP. Figure *A* is the original pixel matrix while Figure *B* is the thresholded version of it. 77
3.6 *A* and *B* show a possible representation of the left pixel matrix in a lighter and a darker environment, respectively. We can notice that both output the same binary representation in the end, which illustrates an interesting property of LBP. 78
3.7 9×9 MB-LBP.*A* illustrates how the image is divided into blocks and the mean of each block. *B* shows the thresholded values. The top left part shows how the average of pixels' values is calculated, by summing up all pixels and dividing by the sub-region size. 78
3.8 On the left, the gradients orientation of HOG face detection descriptor. It is important to notice that the gradients directions result in a face shape, which helps to understand the intuition behind HOG and why it works. On the right, the face identified by the given descriptor. 80
3.9 Images used for face detection tests. It is a subset of WIDER FACE validation set. 82
3.10 Some examples of images available in the validation set. 83
3.11 Top-5 results for each cascade file. *SF* is the Scale Factor, *MN* is the Minimum Neighbors parameter, and *MS* is the Minimum Size parameter. On the top of each plot, we can see the name of the cascade file. Colors do not represent any particular order. 86
3.12 Worst-5 results for each cascade file. *SF* is the Scale Factor, *MN* is the Minimum Neighbors parameter, and *MS* is the Minimum Size parameter. On the top of each plot, we can see the name of the cascade file. Colors do not represent any particular order. 87
3.13 Examples of changing parameters in OpenCV in an input image. Green boxes are ground truth whereas red are detections. 88
3.14 Top-10 and Worst-10. The first 10 labels in the legend are the 10 best parameters whereas the next 10 are the worst. 88
3.15 All 420 variations of OpenCV parameters that we used in our experiments. 89
3.16 Top-5 results for each cascade file based on per-image F1 score. *SF* is the Scale Factor, *MN* is the Minimum Neighbors parameter, and *MS* is the Minimum Size parameter. On the top of each plot, we can see the name of the cascade file. Colors do not represent any particular order. 90
3.17 The 4 variations of upsample parameter that we used in our experiments with Dlib. 92
3.18 Impact of changing upsample parameter in Dlib. 92

3.19 Results with Dlib face detector with different upsample values. 94
3.20 The 4 variations of upsample parameter that we used in our experiments with Dlib. In this case, results are based on F1 computed per image instead of the overall computation. 95
3.21 Top-10 results with Faster R-CNN. *M* is the Model, *CT* is the Confidence Threshold and *NMS* is the Nonmaximum Supression Threshold. 96
3.22 All 240 variations of Faster R-CNN models that we used in our experiments. 96
3.23 Examples of detection results with different Faster R-CNN models and parameters. *A* is result of a model from iteration $10,000$ with a low confidence threshold (0.5). *B* is of a model from the same iteration but with a larger confidence threshold (0.9). Finally, *C* is our best model, which is from iteration $60,000$ and has a confidence threshold of 0.8. 97
3.24 Examples of detection results with different Faster R-CNN models and parameters. 97
3.25 Results of all MTCNN executions. 98
3.26 Results of all MTCNN executions. In this case, evaluation of F1 score per image. . 99
3.27 Example of MTCNN execution with different parameters. 100
3.28 Another example of MTCNN execution with different parameters. 100
3.29 Best results of each face detector based on overall F1 score (10% of validation set).101
3.30 Best results of each face detector based on F1 score per image (10% of validation set). 101
3.31 Best results of each face detector based on the per-image F1 score (complete validation set). 102
3.32 Best results of each face detector based on the overall F1 score (complete validation set). 102
3.33 How our metadataset is created. 104
3.34 GoogleNet architecture (Image from [111]). 105
3.35 The three different Inception modules available in Inception v3. The first one is used in the original GoogLeNet implementation. 106
3.36 Three different architectures, comparing VGG-19 (left), a regular deep neural network (center), and a residual neural network with 34 layers (right). (Image from [43]).108
3.37 A residual block example. 109
3.38 Prediction with the meta classifier. 114
3.39 Different training and test losses during training. 115
3.40 Different training and test accuracies during training. 115
3.41 Prediction with the meta classifier. 117
3.42 Results on the IJB-A dataset with SVM meta classifiers. 118

3.43 Results on the IJB-A dataset with neural networks meta classifiers. 120
3.44 Results on FDDB dataset with SVM meta classifiers. 122
3.45 Results on FDDB dataset with neural networks meta classifiers. 124

List of Tables

2.1 One-hot encoding example 52

3.1 CNN architectures used followed by the respective layer that we used to extract features and their dimensionality. 107

3.2 The created meta training sets with their classes, number of detections per type of face detector, and total detections. The largest values of each column are bold. 110

3.3 The meta datasets chosen for our experiments (in bold). 111

3.4 Best trained SVM models. 112

3.5 Comparison of the trained SVMs results with different weight schemes (only for Inception v3). All results are using $C = 2$ and standardized data. 112

3.6 The twelve selected architectures. 114

3.7 Best trained neural network models. 116

3.8 Comparison of different models/detectors for given IoU thresholds for IJB-A dataset. The meta classifiers are based on SVMs. 119

3.9 Comparison of different models/detectors for given IoU thresholds for IJB-A dataset. The meta classifiers are based on neural networks. 121

3.10 Comparison of different models/detectors for given IoU thresholds in the FDDB dataset. The meta classifiers are based on SVMs. 123

3.11 Comparison of different models/detectors for given IoU thresholds for FDDB dataset. The meta classifiers are based on neural networks. 125

List of Acronyms

CNN – Convolutional Neural Networks

FCN – Fully Convolutional Network

GAN – Generative Adversarial Network

IoU – Intersection Over Union

MTCNN – Multi-task Cascaded Convolutional Networks

NMS – Non-Maximum Suppression

R-CNN – Regions with CNN features

ReLU – Rectified Linear Unit

RPN – Region Proposal Network

SSD – Single Shot MultiBox Detector

SGD – Stochastic Gradient Descent

SPP-Net – Spatial Pyramid Pooling Network

SVM – Support Vector Machine

YOLO – You Only Look Once

Notations

Numbers and Arrays

a – A scalar

$\boldsymbol{a}$ – A vector

$\boldsymbol{A}$ – A matrix

$\boldsymbol{a}^T$ – Transpose of vector $\boldsymbol{a}$

$\boldsymbol{A}^T$ – Transpose of matrix $\boldsymbol{A}$

$|a|$ – The absolute value of a

Indexing

$\boldsymbol{a}_i$ – The i-th element of vector $\boldsymbol{a}$

Sets

$[a, b]$ – The real interval including a and b

Functions

$log x$ – Natural logarithm of x

$argmin$ – $argmin$ of a given function (e.g. $f(x)$) is the value of x for which $f(x)$ attains its minimum

Probability

$P(A)$ – The probability of event A occur

$P(A|B)$ – The conditional probability that event A occurs, given that event B has occurred

Datasets

$\boldsymbol{X}$ – A set of training examples

$\boldsymbol{x}_i$ – The i-th training example, which belongs to $\boldsymbol{X}$

$\boldsymbol{y}$ – A set of training labels

y_i – The target (class) attribute, which is associated with $\boldsymbol{x}_i$

$\hat{y}_i$ – Predicted class attribute

N – Number of instances

Machine Learning

ψ – A neural network activation function

ϕ – The output of a convolutional neural network

ϕ_i – The output of the i-th layer of a convolutional neural network

λ – Regularization term of a given function

L – Loss (cost) function

L_i – Loss of the i-th instance

$\mathcal{L}$ – Partial loss function

$\boldsymbol{W}$ – The weights matrix

$\boldsymbol{w}$ – The weights vector

s – Scoring function

t_i – Prediction of a given regression

t_i^* – Ground truth (target) of a given regression

P – Region proposal

$\hat{P}$ – Predicted bounding box

$\hat{P}$ – Ground truth (target) of a given bounding box

$D_{\boldsymbol{W}}$ – A given distance function D parametrized by weights $\boldsymbol{W}$

$E_{\boldsymbol{W}}$ – A given energy function E parametrized by weights $\boldsymbol{W}$

$G_{\boldsymbol{W}}$ – A given complex function G parametrized by weights $\boldsymbol{W}$

Contents

1 INTRODUCTION **33**
1.1 Contributions 36
1.2 Outline 36

2 BACKGROUND **39**
2.1 Introduction 39
2.2 Architecture of Convolutional Neural Networks 41
2.2.1 Convolution Layer 42
2.2.2 Pooling Layer 44
2.2.3 Fully-Connected Layer 46
2.3 Convolutional Neural Networks for Classification 47
2.4 Convolutional Neural Networks for Localization 53
2.4.1 R-CNN 55
2.4.2 Fast R-CNN 58
2.4.3 Faster R-CNN 61
2.4.4 MTCNN 63
2.5 Convolutional Neural Networks for Metric Learning 66
2.6 Final Remarks 72

3 FACE DETECTION **73**
3.1 Introduction 73
3.2 Related Work Based on Hand-Crafted Features 75
3.2.1 Open CV (Viola-Jones) 75
3.2.1.1 Haar Feature-based Cascade Classifiers (Viola-Jones Face Detector) . . 75

3.2.1.2 Local Binary Patterns . . . 77
3.2.2 Dlib . . . 79
3.3 Related Work Based on Deep Learning . . . 80
3.3.1 Faster R-CNN . . . 80
3.3.2 MTCNN . . . 81
3.4 Individual Experiments . . . 81
3.4.1 Hand-Crafted Approaches . . . 84
3.4.1.1 OpenCV . . . 84
3.4.1.2 Dlib . . . 91
3.4.2 Deep Learning Approaches . . . 93
3.4.2.1 Faster R-CNN . . . 94
3.4.2.2 MTCNN . . . 98
3.4.3 Analysis . . . 99
3.5 Meta Learning Approach . . . 103
3.5.1 Meta Dataset Creation . . . 103
3.5.2 Training the Meta Classifiers . . . 110
3.5.2.1 SVM . . . 110
3.5.2.2 Neural Network . . . 112
3.5.3 Prediction . . . 116
3.5.4 Experiments . . . 117
3.5.4.1 IJB-A . . . 117
3.5.4.2 FDDB . . . 120
3.5.5 Conclusion and Discussion . . . 123
3.6 Final Remarks . . . 125

4 CONCLUSION . . . 127
4.1 Limitations . . . 128
4.2 Opportunities for Future Work . . . 129
4.2.1 Implement Face Detection based on Different Deep Learning Implementations . 129
4.2.2 Different Strategies for Feature Extraction . . . 129
4.2.3 Generate Meta Training Sets based on Alternative Metrics and Strategies . . . 130
4.2.4 Train Meta Classifiers in Different Datasets . . . 131
4.2.5 Evaluation with Larger IoU Thresholds . . . 131
4.2.6 Implement Light-Weight and Faster Face Detectors . . . 131
4.2.7 Evaluate the Impact of Improved Detectors in Face Recognition . . . 132

REFERENCES **133**

1 Introduction

We live in an age that is dominated by technology in which Computer Science plays a major role. In the beginning of Computer Science, *algorithm* was a common word for computer scientists and related jobs, who were used to deal with technology daily. In the last years, algorithms are everywhere, embedded not only in traditional computational devices such as notebooks and smartphones, but also in our houses, our work, our lives. We enter work with our badge that contains an RFID tag, which is used to access the building by reading the tag with an RFID reader placed at the entrance door. As soon as we enter our car, our smartphone pairs with the car's firmware through Bluetooth, and immediately starts to play our Spotify's playlist. Not limited to that, the car also enable us to perform phone calls through voice commands. All these are examples of algorithms present in our daily routines, and their existence is almost not perceived by us as the technology becomes more pervasive and ubiquitous.

Algorithms are, in a simple manner, a set of instructions that tells the computer to execute a given task to achieve a desired goal. They are the basis of Computer Science. Since the beginning, programmers always had to explicitly say what the algorithms should do, writing its instructions in a manually and maybe cumbersome way. In some cases, we do not even have an algorithm that is able to solve the task at hand. For instance, classify whether an email is spam requires the algorithm to decide if that email is legitimate or not based on the email message. As humans, when we face a challenge similar to this one we refer ourselves to previous emails that are considered legitimate and emails that are considered spams [26]. In other words, we look at *data* to learn what is the difference between the two types. With that, researchers started to wonder if we could make our computer learn from data in the same manner as we do. That is, algorithms look at available data as to learn to execute a task with a goal without being explicitly told to do so. What if an algorithm could read medical records, understand what causes certain diseases and automatically tell us the correct treatment? What if we could optimize the energy costs of our houses based on the usage patterns of people who live in it? [78].

Machine Learning is a field within Artificial Intelligence that aims to make algorithms automatically learn and, thus, make machines learn. The idea that machines could improve with experience was explored by Thomas Mitchell, back in the eighties [78]. Machine learning algorithms

are able to learn how to perform relevant tasks by learning and generalizing from examples [25]. Such algorithms can be roughly separated into four different classes: *supervised learning*, *unsupervised learning*, *semi-supervised learning*, and *reinforcement learning*. Supervised learning algorithms are those that learn from labeled data. They learn how to map a given input x into an output y. They are usually used for classification, where we want to assign a *label* (also know as *class*) to a given input, but also for regression, where the output is a continuous value — predicting stock prices, for example. In unsupervised learning, we do not have labeled data but only the data itself. Hence, algorithms of these paradigm aim to discover hidden structures of the data, learning relations between instances. Examples include clustering algorithms, which group instances based on a similarity criteria, and dimensionality reduction algorithms, which are able to reduce the representation size of the data. Semi-supervised learning can be considered a mid-term between the two mentioned paradigms since it uses labeled instances only in a part of the training process. This is motivated by the fact that there is more unlabeled than labelled data, and it works by combining both to build powerful classifiers, for example. Finally, reinforcement learning is derived from intelligent agents, whose goal is to perform a task as to have maximum *reward* [54]. It basically does the same but allowing the agents to learn a behavior based on feedback from the environment. One of its main uses is in the robotics field [20, 80].

Recently, machines have reached the same level and even surpassed humans abilities in many different tasks, such as Object Detection [93], Music Recommendation [117], and Image Classification [60]. The majority of these results have been possible due to Deep Learning. Deep learning can be understood as a machine learning approach that attempts to extract high-level and hierarchical features from raw data, such as images, audio, and videos [64, 101]. Deep learning approaches are the current state-of-the-art in a plethora of challenging areas. In the Natural Language Processing (NLP) field, deep learning is successfully applied for Text Translation [109], Question and Answering [62, 123], and Text Classification [53]. In the Computer Vision field, for Object Detection [76], Image Captioning [126], and Image Segmentation [18]. It is also successfully applied for audio tasks such as Speech Recognition [3, 39, 38, 99, 7, 124]. Very recently, they are yielding great results for generative models like Generative Adversarial Networks [37, 88], and for Reinforcement Learning [79, 122, 104] as well.

In the Computer Vision field, one of the most studied topics is Face Detection [130, 55]. Given an arbitrary image or video frame, the goal of face detection is to determine whether there are any faces in the image and, if present, return the image location and the extent of each face [127]. Face detection is easily done by humans, but it is still a challenge within Computer Vision. The high degree of variability and the dynamicity of the human face makes it an object very difficult to detect, mainly in complex environments. Most of the existing complexity is due to occlusion (i.e. face is partially visible), illumination conditions (e.g. darker and lighter environments), background clutter (e.g. background and object in scene are too similar in color intensities), and so on. Face detection is an important first step in different applications. It is used in, but not limited to, video

surveillance [32], face authentication [40], organization and presentation of digital photo-albums [57], and face recognition [102, 46, 71].

Face recognition is a topic that is studied for a long time, and it is one of the first Computer Vision areas with work that dates back to 50 years [11, 100, 31]. From 1990 to the beginning of 2000, a large amount of papers were published in the field [127, 45]. However, the implementations were not able to perform well under the *unconstrained setting* [130]. Unconstrained face detection — also known as face detection *in-the-wild* — refers to recognizing faces in images in real world conditions [47]. For instance, variations in pose, lighting, expression, race, ethnicity, age, gender, clothing, hairstyles are unconstrained conditions. The Viola-Jones face detector was one of the major breakthroughs because it allowed face detection to be feasible in real-world applications [120]. Most of the current digital cameras still use the Viola-Jones face detector. After that seminal implementation, all the subsequent attempts were focused on beating such results in both accuracy and speed. The main improvements can be roughly separated into four perspectives [130]: new robust feature extractors such as Scale Invariant Feature Transform (SIFT) features [77] and Histograms of Oriented Gradients (HOG) [19]; more powerful classifiers such as Support Vector Machines (SVM) [17], and more recently, Deep Learning approaches [60]; large-scale unconstrained face recognition datasets such as Labeled Faces in the Wild (LFW) [47] and FDDB [50]; the development and availability of high-quality code in the form of libraries and frameworks, such as OpenCV [12].

Along with the mentioned advances, several face detection and face recognition datasets emerged. LFW was held during much time as the face recognition benchmark, as it was built to study unconstrained face recognition. However, very recently the results became very high – around 99% – and the dataset does not represent a real challenge for the problem. Most of the current and recent approaches are competing by decimals in the accuracy. Implementations like HyperFace [89], DeepID3 [108], DeepFace [113], and FaceNet [102] are all based on deep learning and achieve great results on LFW and other datasets. Since the results of some of these implementations are due to the large private datasets they make use of (e.g. DeepFace from Facebook and FaceNet from Google), some researchers started to compare their work only with models trained in public datasets [84]. Face detection is fundamental in all these implementations as a first step in face recognition, which is usually comprised of face detection, face aligment, feature extraction, and classification [46]. Nevertheless, when looking at most of the state-of-the-art implementations, they make use of off-the-shelf face detectors and do not seem to evaluate differences among them. In other cases, the face detectors are trained along the network but in a multitask manner along with face landmark detection, age detection, and so on. We argue that very few work indeed focused on face detection, trying to both evaluate the existing algorithms as well as the impacts of their differences in available datasets.

Hence, our proposal for this dissertation is to focus on the face detection problem itself, aiming to contribute for the theme through a rigorous analysis of some of the existing face detectors as well as implementations of our own, exploring the results and drawing conclusions for future work.

1.1 Contributions

Our main goal is to evaluate some of the evolution in face detection algorithms and the role of deep learning on these improvements. In addition, we implemented some ideas as to compare with the existing ones. Our contributions can be summarized as follows:

- We detail many advances of deep learning in Computer Vision, explaining how different architectures and implementations work. We believe that it is a valuable resource for future researchers as a guideline for learning about deep learning or understanding some of its main contents in a simple but consistent manner.
- We execute experiments with both hand-crafted and deep learning approaches for face detection, focusing on some well-know face detectors such as Viola-Jones and comparing them against more recent implementations.
- We propose and implement our own deep learning face detector, which is based on Faster R-CNN implementation trained on face detection datasets. To the best of our knowledge, at the time we started the implementation it was the first work to use such implementation for face detection. However, at the time of publication of this dissertation, we found unpublished work regarding the same idea.
- For all the before-mentioned implementations, we execute a large number of experiments with different hyper-parameters as to have an idea of their impact in face detection. We compare the visual differences of the different parameters and draw conclusions of the best parameters for each implementation. To the best of our knowledge, it is the first work that performs such an analysis of the given algorithms.
- The idea that no face detector is the best for every situation led us to try a meta learning approach for this case. We propose, implement, and test a meta learning approach for face detection, which aims to learn the best face detector for a given image. It is based on different features extracted with different deep learning models in different datasets. We train two different classifiers on top of these features and test them against the traditional face detectors. To the best of our knowledge, it is the first time a meta learning approach based on features extracted from deep learning models is used for face detection.

1.2 Outline

This dissertation is organized as follows.

Chapter 2 reviews and explains deep learning, focusing on convolutional neural networks (CNNs). We detail how CNNs work, their different architectures and implementations. We also explain how they are applied in classification, detection, and metric learning.

Chapter 3 focuses on face detection itself, what it is, and an overview of both classic implementations and most recent ones. We detail our experiments with each of the selected approaches and draw conclusions from them. We also explain, detail, and test our approach with meta learning, evaluating the difference when compared with the other implementations.

Finally, Chapter 4 has our concluding remarks. We first explore the limitations of this dissertation. Then, we detail and explain our directions for future work.

2 Background

In this chapter, we present the background needed for the understanding deep learning approaches of this work. The chapter focuses mostly on convolutional neural networks, a successful approach for different machine learning tasks. It is organized as follows. First, we briefly introduce to deep learning in Section 2.1. Then, we explain convolutional neural networks and its main components in Section 2.2. Next, we present the application of convolutional neural networks for classification and object localization in Sections 2.3 and 2.4, respectively. Finally, we detail convolutional neural networks for metric learning in Section 2.5.

2.1 Introduction

Artificial Intelligence is an active area of research in Computer Science. Machine Learning is a sub-area that is growing and evolving quickly, as it enables algorithms to automatically learn how to perform a given task based on available data. Machine learning systems are used to, for instance, recognize objects in images, select relevant results of a search, and transcribe speech into text, just to name a few. Recently, these applications make use of a technique called deep learning. Deep learning methods allow the models to have multiple processing layers and learn complex representations from data, improving the state-of-the-art of different applications [64].

Traditional machine learning techniques are very limited to process raw data such as audio, text, and images. The feature extraction portion is based on hand-crafted features, which demand domain expertise to create and are domain specific. Domain experts need to carefully create algorithms that transform the raw data (for example, pixels) into a representation that a machine learning algorithm can detect patterns. Additionally, an algorithm that extracts features from faces cannot be used to extract features to identify cats. On the other hand, deep learning techniques can learn a feature extractor directly from raw data, which makes them powerful when compared with hand-crafted feature extractors.

Figure 2.1 shows the difference between traditional and deep learning approaches for the image classification problem. In traditional approaches, a hand-crafted feature extractor - the one

created by domain experts - is used to get features from the cat image. Then, such features are used as input to a trainable classifier (e.g. SVM). In this case, machine learning is applied only to the classification part, where a model is trained to classify the images based on the extracted hand-crafted features. On the contrary, deep learning approaches perform joint training, learning not only to classify the image but also to extract representative features.

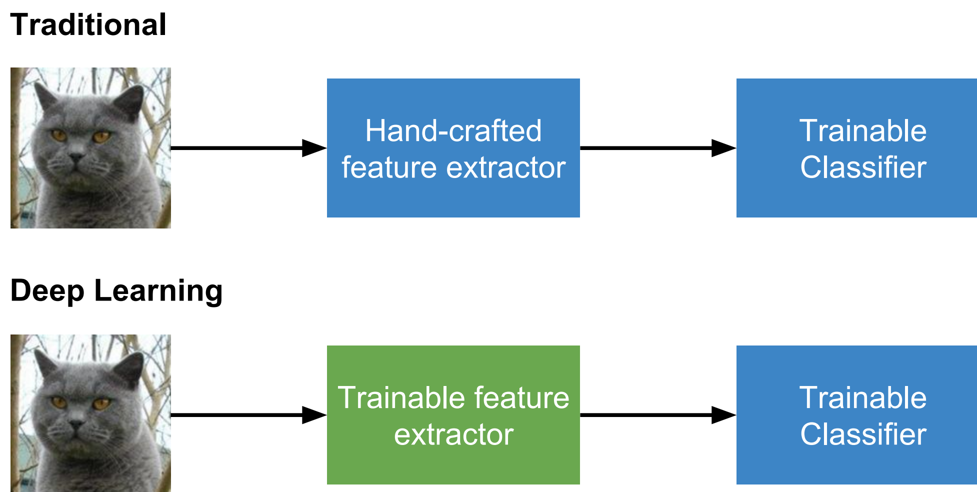


Figure 2.1: Difference between traditional and deep learning approaches for the image classification problem. In deep learning, the feature extractor is learned.

Deep learning does not have a single definition in the literature. However, all existing definitions share common descriptions: multiple layers of processing units; layers form a hierarchy of features, transforming the representation from low-level to a higher and more abstract level. Such high-level features increase aspects of the input that are important for discrimination and suppress irrelevant variations. If the input is an image, for example, the first layer may be able to detect the presence or the absence of edges and contours. The next layer might detect more complex forms by focusing on specific portions of the image. This process continues on the following layers, which learn even more complex forms of objects and combinations of them. Figure 2.2 shows an intuition behind the learning process enabled by the hierarchy of layers. Natural data is compositional, which means it is a combination of simple forms and complex forms (a face is composed of two eyes, a nose and a mouth). The key aspect is that these layers of features are learned from data, using a general-purpose learning procedure [64].

Such characteristics allowed deep learning to improve the state-of-the-art in different areas and applications. For instance, it outperformed previous implementations for image classification [60, 29], speech recognition [3, 98], and sentiment analysis [27], to name a few. Most of these results are based on implementations of convolutional neural networks (CNNs), which were firstly successfully implemented for image classification. Convolutional neural networks, how it works and its main applications will be explored in details in the following sections.

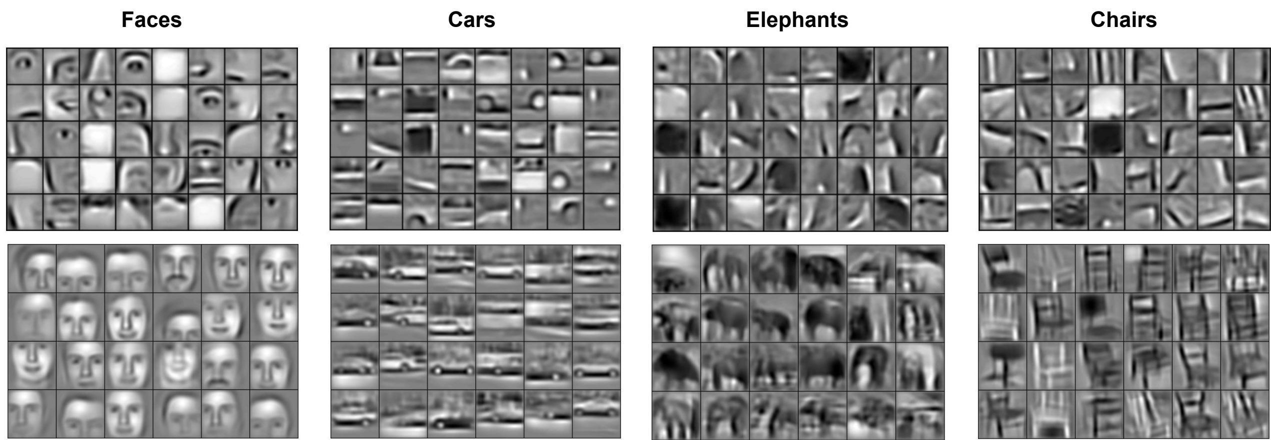


Figure 2.2: Complex forms are a combination of simple forms. The top shows parts of a complex form (for instance, eyes and mouths), whereas the bottom shows complex forms themselves. (Image adapted from [68]).

2.2 Architecture of Convolutional Neural Networks

Convolutional Neural Networks — or simply ConvNets or CNNs — are feed-forward artificial neural networks that can learn features invariant to translation, rotation, and shifting. They are neural networks that use convolutions instead of general matrix multiplication in at least one of their layers [36]. They are able to process data that come in the form of multiple arrays - 1D, 2D and even 3D [64]. Such flexibility enables one to create ConvNets to process signals and sequences [53], images [60], audio spectrograms [23], and videos [51].

CNNs are inspired by the human visual cortex, which contains regions of cells that are sensitive to sub-regions of the visual field [1]. Experiments with cats showed that individual neuronal cells in the brain responded only to the presence of edges of a given orientation [48]. Some cells only respond to horizontal edges whereas others to vertical edges and, hence, these cells always look for specific characteristics. Additionally, these experiments showed that the visual cortex contains simple, complex, and hyper complex cells, as well as combinations of them. The insights gathered from [48] inspired some properties of the deep neural networks architecture, which contains local connections, layering, and spatial invariance.

A typical CNN has three main layers: Convolution Layer, Pooling (or Subsampling) Layer, and Fully-Connected Layer. These layers are stacked on top of each other to form the Convolutional Neural Network. Figure 2.3 shows a network with two convolutional layers, two subsampling layers, and one fully-connected layer at the end. The number of convolutional layers and subsampling layers and the way they are stacked can vary. In the following sections, we will detail each of these different layers.

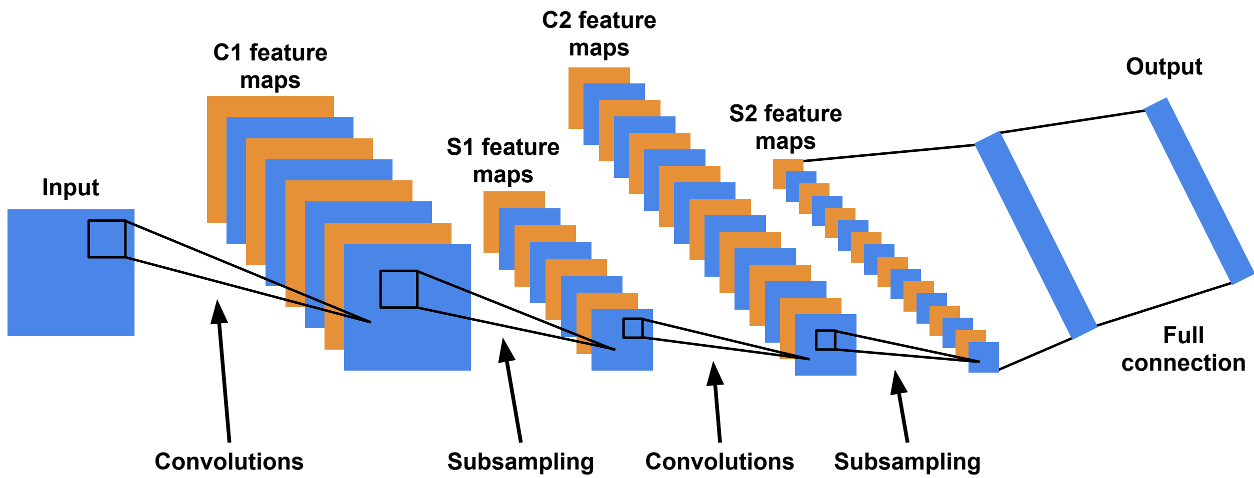


Figure 2.3: A typical ConvNet architecture [67].

2.2.1 Convolution Layer

Convolution is a mathematical operation that *convolves* a filter — also called kernel — over images. The filter slides over the image spatially, computing dot products between the input matrix region and the kernel matrix. The output of a convolution operation is called feature map, and feature maps can also be used as input to a convolution. Figure 2.4 shows an example of a convolution operation with a given kernel using an image as input. On the left part of the figure we can see the input image and on the right side the resulting feature map after applying the defined kernel (expressed by the matrix in the middle). The output feature map is a representation of the image, which extracts some characteristics (features) from it. In this scenario, the resulting feature map represents the edges of the input image. Hence, we can understand that the convolution operation with a given kernel is able to identify the presence of specific patterns in the image.

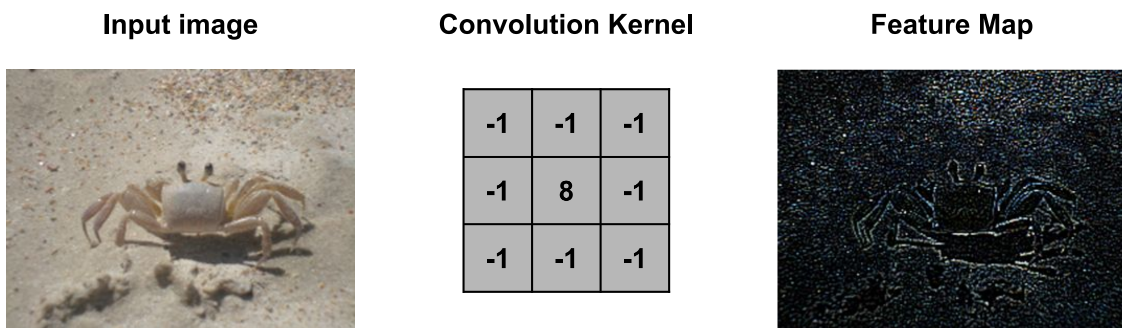


Figure 2.4: Example of convolution applied on an image using a kernel to detect edges [2].

Taking images as an example, the convolution is applied at convolutional layers to extract features from neighborhood pixels on feature maps. The pair (x, y) refers to the position of a given unit in feature map j of layer i, whose value is defined by v_{ij}^{xy}. The value v_{ij}^{xy} is given by Equation 2.1, where b_{ij} is the bias for this feature map, k indexes over the set of feature maps in

the $(i-1)th$ layer connected to the current feature map, w_{ijk}^{pq} is the value at the position (p, q) of the kernel connected to the kth feature map, P_i is the height, and Q_i is the width of the kernel. After the result of the three sums is computed and summed up with the bias term b_{ij}, an activation function ψ (or non-linearity) is applied.

$$v_{ij}^{xy} = \psi\left(b_{ij} + \sum_k \sum_{p=0}^{P_i-1} \sum_{q=0}^{Q_i-1} w_{ijk}^{pq} \; v_{(i-1)k}^{(x+p)(y+q)}\right) \tag{2.1}$$

An activation function takes a single number and performs a specific mathematical operation over it. Such activation function aims to introduce non-linearity in the neural network, which means that the output cannot be reproduced from a linear combination of the inputs. Traditional neural networks make use of the Sigmoid activation function, which is given by Equation 2.2. It maps the given number x into a range within $[0, 1]$. There are other activations functions such as the Hyperbolic Tangent (Tanh) and the Rectified Linear Unit (ReLU). We will discuss details of activation functions in Section 2.3.

$$\psi(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Convolutional layers have additional hyper-parameters other than the activation function: number of filters, filter size, stride, and zero padding. The values of those hyper-parameters can vary in different convolutional layers. Filter size determines the width and the height of the filters. Stride is the distance between two consecutive positions of the filter: if the stride is one, the filter jumps one position at a time. The larger the stride, the smaller the output volume (Figure 2.5 shows how different stride values affect the output). Zero padding is the number of zeros added around the input. For instance, if the input is 5×5 and zero padding is 1, the final input size is 7×7. Zero padding allows one to control the spatial size of the output volumes. For example, the input and the output can have the same width and height. It is important to mention that filters can be seen as feature extractors and that such filters are learned with Backpropagation [96] (details in Section 2.2.3).

There are three important aspects of convolutional layers: *sparse connectivity*, *weight sharing* and *equivariant representations* [36]. Traditional neural networks have full connections between layers, which means that every single input node is connected to every neuron in the following layer. For instance, a 250×250-pixels gray image has $62,500$ values as input. If the hidden layer contains 200 neurons, the total number of parameters would be $12,500,000$. On the other hand, convolutions are locally-connected, i.e. a given layer is connected to a particular region of the image. Hence, sparse connectivity means that fewer parameters need to be stored in memory and the number of computations is reduced.

The second important aspect is weight sharing. In traditional neural networks, a given value of the weight matrix is used only once to compute the output. This is usually referred to as *tied weights*, because the value of the weight applied to one input is tied to the value of a weight

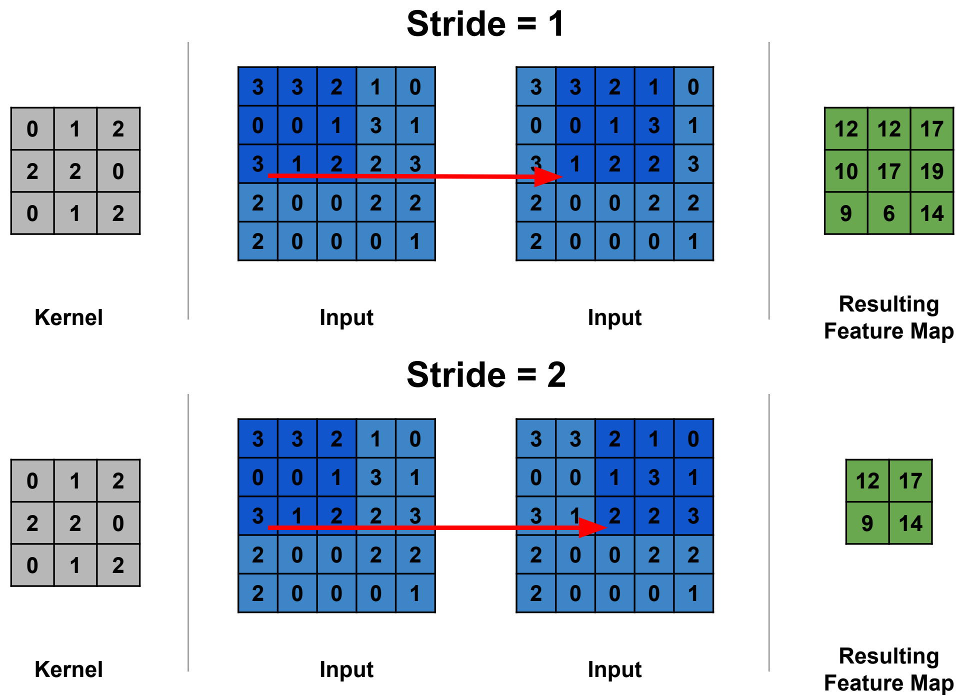


Figure 2.5: Effect of different stride values in the output feature map. The red arrow indicates the kernel displacement caused by the stride.

applied elsewhere [36]. In CNNs, the same kernel weights are applied to a given input. Figure 2.6 illustrates how the same kernel is applied to a given input. The example has a 3×3 kernel and an input of 5×5. The same kernel slides over the input, calculating a dot product between the kernel and the region of the input. In the second step, the kernel jumps one position — considering a stride of one. This process is repeated until the kernel slides over all image, generating the output feature map. The final step is in the rightmost part of the figure, with the resulting feature map.

Finally, the last aspect is equivariant representation. This particular form of parameter sharing enables the layer to be equivariant to translation. It means that if the input changes, the output will change in the same way. For example, we have an image that has an object in its leftmost region. A convolution applied on such image will yield a given feature map with specific values on its leftmost part. If the object is shifted some pixels to the right and a convolution is applied, the values that were in the leftmost part will be slightly shifted to the right in the output volume. Hence, we can understand that when the kernel slides over the image, it is actually searching for given features wherever they are. However, it is worth mentioning that convolution is not naturally equivariant to transformations like scaling or rotation of the image [36].

2.2.2 Pooling Layer

Pooling layers are also an important part of a CNN architecture. The pooling operation reduces the size of the feature map through a function that summarizes a given subregion [36]. This reduction is performed with functions that take, e.g. the maximum or the average value within a

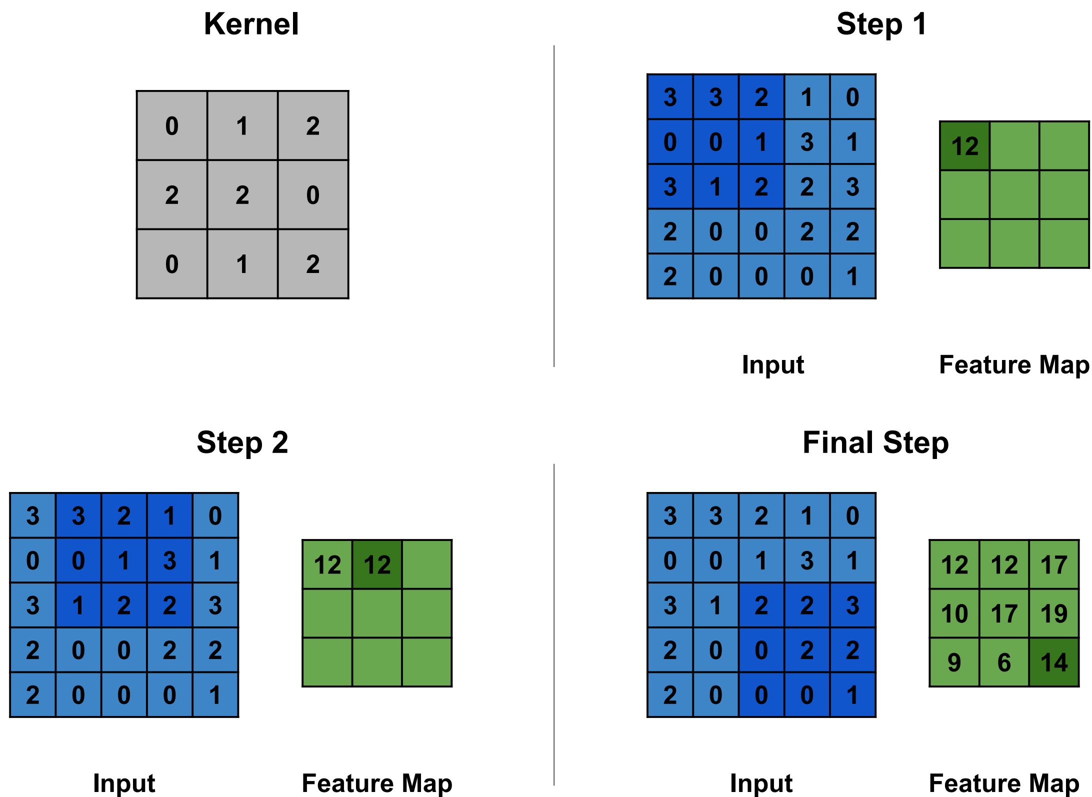


Figure 2.6: Example of convolution with a 3×3 kernel over an input of 5×5 and a stride of 1. The same kernel is applied while it slides over the input.

subregion, acting like a statistic summary of the neighbor values. Shrinking the size of the feature map results in reduced number of parameters and computation of the network, and therefore it helps to control overfitting. Pooling layers are usually used immediately after convolution layers.

Pooling works very similarly to a convolution: it slides a window over the input performing some computation. The main difference is pooling does not require parameters. Figure 2.7 shows an example of the max pooling operation. A window slides over the input and takes the maximum value within a region. Each region in the figure is represented by different colors. Even though max-pooling is the most common pooling function, average pooling (average within a region) and stochastic pooling (pooling regions are selected randomly according to a multinomial distribution) are also used.

Pooling has three important properties: it does not have additional parameters, it does not change the output depth and it is invariant to small translations in the input. First, as pooling operation uses a fixed function (e.g. max-pooling), no additional parameters are introduced. Hence, pooling layers do not require additional memory use. Second, pooling does not change the output depth. Recall that convolutions have three dimensions - *height*, *width* and *depth* (represented by the number of kernels). When applying pooling, only the width and the height of the volume are downsized, while the number of kernels are kept. Finally, pooling provides additional *translation invariance*. If the values of the input suffer a small translation, the values of most of the output do not change. It can be a very useful property since finding a given feature may be more interesting

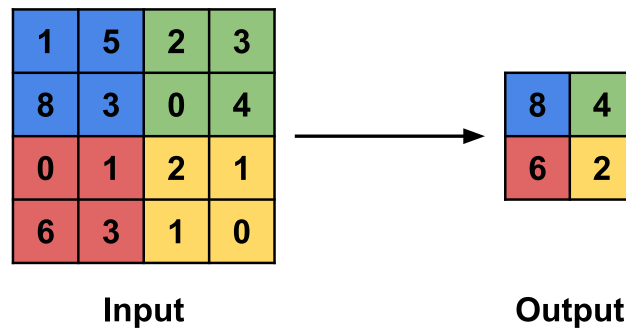


Figure 2.7: Example of max pooling with a 2×2 sliding window, an input of 4×4, and stride of 2. The colors indicate the positions of the sliding window.

than knowing exactly where it is [36]. Figure 2.8 illustrates the translation invariance. Even though the values highlighted in red changed their position, the output values are the same.

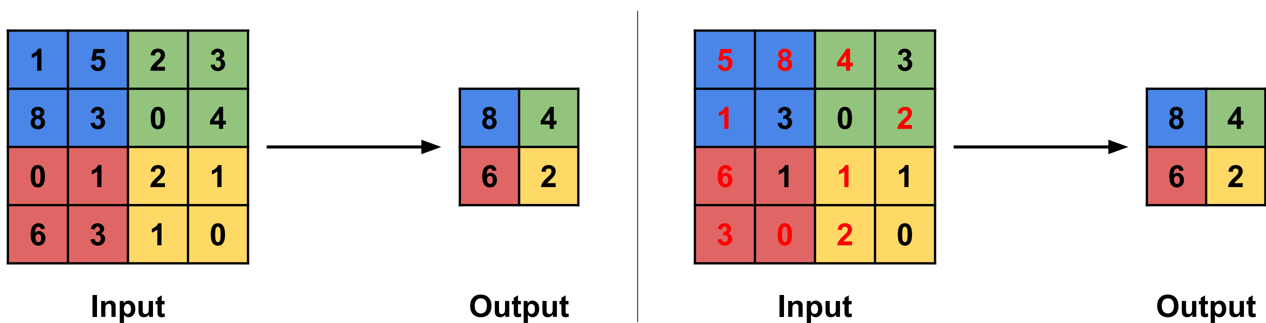


Figure 2.8: Translation invariance of the pooling operation. The red values are changes applied on the original input within each highlighted region. Although the values changed, the output values did not.

2.2.3 Fully-Connected Layer

The final layer of connections in a convolutional neural network is a fully-connected layer. It takes all neurons in the previous layer, which can be convolutional, pooling or even other fully-connected layer, and connects it to every single neuron it has. Typically, a CNN has more than one fully-connected layer, where the last one is the output of the network. For instance, for a classification problem of 10 classes, the last layer is fully-connected with 10 neurons, each of which representing one class. Figure 2.9 shows an example of fully-connected layers.

A stack of fully-connected layers form a neural network. The input layer is where data gets into the network. Hidden layers are responsible for learning representations of the data, learning how to map the input values to output values. Finally, the output layer contains neurons that are used for learning purposes, e.g. transforming the outputs in classes (labels). Each arrow in Figure 2.9 represents the weights (parameters) of the network. The weights (along with the biases) are learned during the training procedure, in which a neural network basically *forwards* the input

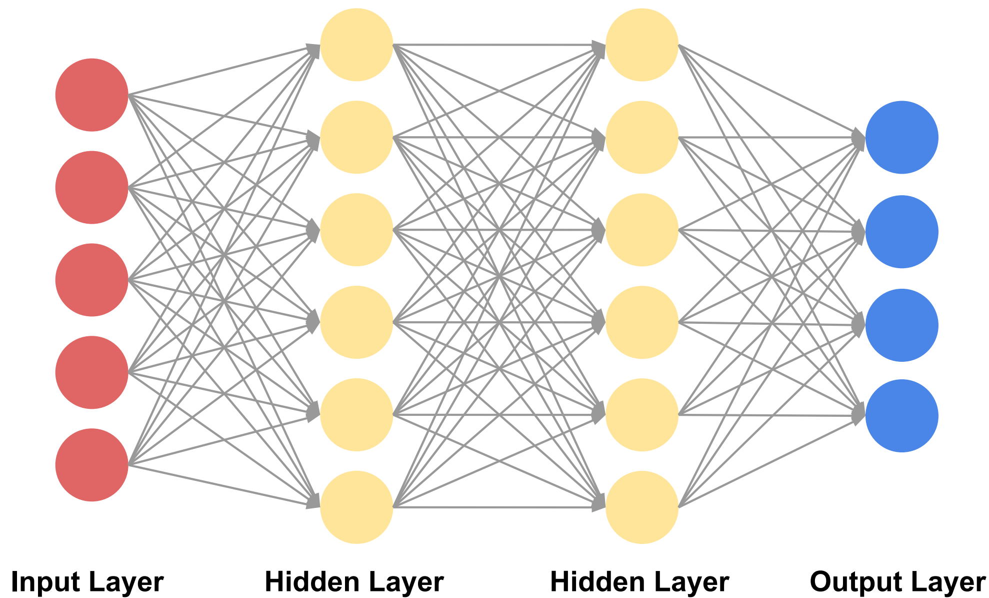


Figure 2.9: Example of fully-connected layers with one input layer, two hidden layers and an output layer.

through the network, calculates the errors (difference between predicted values and expected values), and propagates the errors backwards in order to update the weights. This procedure is called Backpropagation [96]. Since the idea is to minimize the classification error of the network, an algorithm is used to accelerate the convergence: *Stochastic Gradient Descent* (SGD). There are other algorithms that enable the training procedure of a neural network minimize the error. We will explore them along this dissertation.

Traditional neural networks, and hence fully-connected layers, accept only vectors as input (one-dimensional input). The output of the last layer before the fully-connected is a volume (output of a convolution or a pooling layer). Therefore the volume is *flattened*, resulting in a vector that is used as input.

2.3 Convolutional Neural Networks for Classification

Classification is the task of assigning objects to one of several predefined categories [114]. It maps a given input instance $\boldsymbol{x}$ to a class label y (also known as category or target attribute). For instance, detecting whether an email message is spam or not is a classification task. Another example is character recognition, where a classification algorithm is able to recognize which number is present in a given image. Classification algorithms are *trained* with examples and learn a *model*. The learned model is then used to classify unseen examples. In character recognition, the training set could be comprised of $(\boldsymbol{x}, y)$ pairs, where $\boldsymbol{x}$ are image pixel's intensities and y indicates the label of that image (0 if the image contains a 0, 1 if the image contains a 1, and so on). A machine learning classification algorithm trained over such data learns a model, which is then used to classify

new examples — data that has not been seen during training. In the training phase, a function is used to compute how much the model is classifying the data correctly or not and what is the classification error. This function is called the *loss function*..

Classification algorithms are among the most popular in Machine Learning. Examples include *Logistic Regression*, *Decision Trees*, *Support Vector Machines* (SVM), and *Neural Networks*, just to name a few. Each of these algorithms require different amounts of data in order to be able to train a model effectively. Many public and domain-specific large datasets emerged recently, each of which are used to solve problems that range from Speech Recognition to Object Classification.

A very notable dataset in the Computer Vision field is ImageNet [21]. ImageNet is a large image dataset comprised of $14,197,122$ annotated images organized in $21,841$ categories. Such a large dataset enabled new challenges and competitions, which resulted in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [97]. ILSVRC is an annual challenge that is the current benchmark for object category classification and object detection and it runs since 2010. In 2010 and 2011, the winners of the challenge were implementations based on classic image feature descriptors combined with classification algorithms. In 2010, the authors used *Histogram of Oriented Gradients* (HOG) and *Local Binary Patterns* (LBP) as feature descriptors (Chapter 3 explains them in more detail), and SVM for classification [75]. On the other hand, in 2011, the implementation was based on *Fisher Vectors* as feature descriptors, *Product Quantization* for dimensionality reduction and SVM for classification [86]. The top-5 classification error was 28.20% for the 2010 implementation and 25.80% in 2011. These approaches are considered *shallow* implementations, in the sense that they have few hidden layers. Even though SVM is not a neural network, the portion that learns the features can be considered a hidden layer. However, in 2012 a deep architecture was able to beat a shallow implementation for the first time (with an error rate of 16.40%). That implementation was based on convolutional neural networks and it is considered the debut of CNNs in the Computer Vision field. The winning implementation architecture is called AlexNet [60], which can be seen in Figure 2.10. AlexNet is a CNN architecture comprised of five convolutional layers, five max-pooling layers, and three fully-connected layers. The first ten layers are interleaved (convolution followed by pooling, forming a stack of convolutions), and there are three fully-connected layers at the end. ReLU is the activation function.

CNNs act as feature extractors, which means that the convolutional layers extract features from the data. The fully-connected layers at the end of the CNN can be seen as a regular neural network. In the AlexNet case, the neural network architecture is the following: 4096 features as input, two fully-connected layers of 4096 neurons each, and an output layer of 1000 neurons. The input of the fully-connected layer must match the output of the last convolutional layer, which is usually flattened into an array (explained in Section 2.2.3). The output layer classifies the data into one of the available classes (in this case 1000 classes). Usually, the Softmax classifier is used and it normalizes the output values, transforming them into probabilities. Softmax is a generalization of the binary Logistic Regression. It is able to handle problems that can have more than two classes.

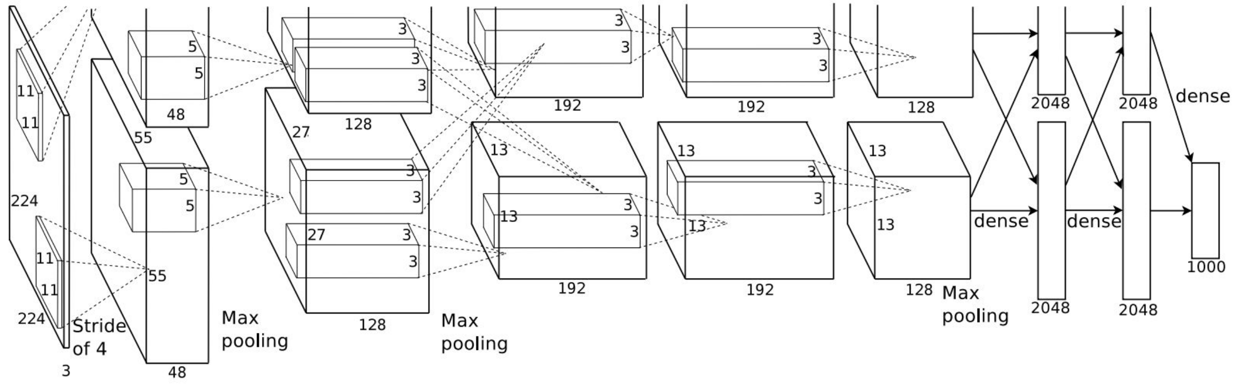


Figure 2.10: AlexNet architecture in details. The two branches are used to illustrate the GPU parallelism, where each branch is processed by a different GPU [60].

Let us take image classification problem as an example. A classifier receives an image as input (pixels) and it aims to map such pixels to an output class label. Let us denote the input pixels set as $\boldsymbol{x}$ and the weights learned by the neural network as $\boldsymbol{W}$. The *scoring function* performs a dot product between $\boldsymbol{x}$ and $\boldsymbol{W}$. Such function is presented in Equation 2.3. The scoring function outputs scores for the different available classes. In order to allow a neural network to learn the weights $\boldsymbol{W}$, one must choose among different loss functions and activation functions. The choice depends on the type of the problem at hand and the goal. For example, different problems require different activation functions and different loss functions. We will see now some examples.

$$s = f(\boldsymbol{x_i}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x_i} \tag{2.3}$$

Binary classification aims to classify an input among two classes. Usually, it indicates the presence (*true*) or the absence (*false*) of a given property. For instance, classify whether an image is a cat or not can be considered a binary classification problem. One could train a neural network with a single output neuron, whose activation function is the Sigmoid. The Sigmoid outputs a value between 0 and 1 and a threshold is used to determine if the given input belongs to a class or not. For the cat classification problem, we could say that if the output of the Sigmoid function < 0.5, the image is not of a cat. On the other hand, if the output is ≥ 0.5, the image is a cat. Different loss functions can be used for binary classification problems. One classic implementation is the *binary cross-entropy* — also known as *logistic loss*. The binary cross-entropy function is given by Equation 2.4.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\Big] \tag{2.4}$$

Such loss function is used for penalizing the classifier for being too confident. For instance, if the predicted value $\hat{y}_i$ is 0.7 (*true* for cat) but the correct value y_i is 0 (*false* for cat), all the left side of the loss function is equal to zero and the loss for the i-th instance is given by the rightmost

part — in this case, the resulting value is $(1-0)\log(1-0.7) = -0.52$. On the contrary, if $\hat{y}_i$ is equal to 0.2 (*false* for cat), the resulting value $(1-0)\log(1-0.2) = -0.10$. The larger the resulting value (0.52 in this case) the larger the loss.

Recently, the Sigmoid function has been replaced by Softmax in the last layer of the neural network (the classification one). Softmax is an activation function that squashes the output values between 0 and 1, and the resulting sum of the values is 1. It is presented in Equation 2.5, where k refers to a given class, $\boldsymbol{x_i}$ is the current instance being classified, $P(Y = k|X = \boldsymbol{x_i})$ is the probability of the correct class Y being k given the input $\boldsymbol{x_i}$, s_k is the result of the scoring function s for the k-th class, and $\sum_j$ sums over all results of the scoring function s applied to all k classes.

$$P(Y = k|X = \boldsymbol{x_i}) = \frac{e^{s_k}}{\sum_j e^{s_j}} \tag{2.5}$$

Figure 2.11 illustrates a simple example of Softmax. An image is used as input to a simple classifier, which outputs the predictions for the four classes - *cat*, *dog*, *car* and *frog*. The exponential operation is applied to such predictions, resulting in the *unnormalized log probabilities* of the classes. Then, such predictions are normalized, resulting in different probabilities that sum to 1. The resulting probabilities allow us to, for example, say that this picture is 10.62% dog and 2.61% frog. In this case, the classifier correctly identified that the image is a cat, with a probability of 86.75% (highlighted in the rightmost table).

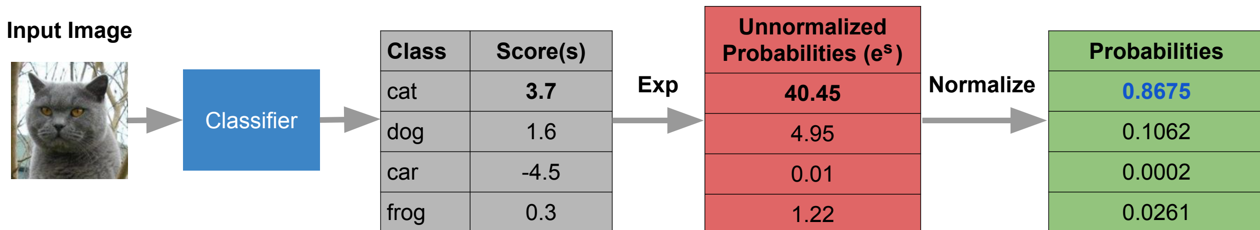


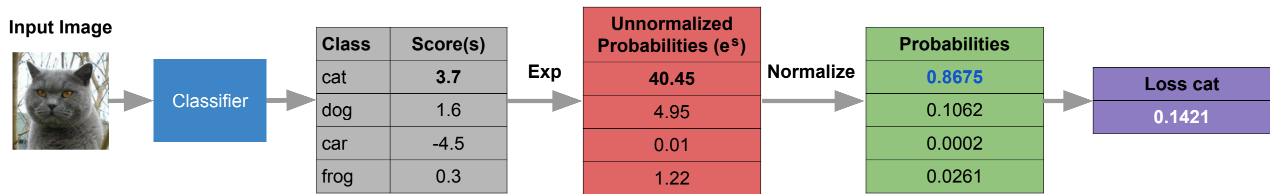
Figure 2.11: Example of simple Softmax classification output.

Many classification problems require the algorithm to classify an instance among more than two classes. Such problems are referred to as *multiclass* or *multinomial* classification. For example, we may want to classify whether the image is a cat, a dog or a frog. The classification algorithms make the assumption that each instance (in this case, an image) is assigned to one and only one label: an image can be either a dog or a frog but not both at the same time. Since now we have more than two classes, the binary cross-entropy (Equation 2.4) loss function is not suitable anymore and, hence, a loss function that can handle more classes is needed. Equation 2.6 illustrates the *multiclass cross-entropy*, also known as *log-likelihood*, which is used for that kind of problem.

$$L_i = -log_e\Big(P(Y = \hat{y}_i|X = \boldsymbol{x_i})\Big) = -log_e\left(\frac{e^{s_{\hat{y}_i}}}{\sum_j e^{s_j}}\right) \tag{2.6}$$

The loss of the i-th instance is the negative log of the Softmax function applied to the given instance, where $\hat{y}_i$ refers to the predicted class, $\boldsymbol{x_i}$ is the current instance being classified,

and $P(Y = \hat{y}_i|X = \boldsymbol{x}_i)$ is the probability that instance $\boldsymbol{x}_i$ belongs to class $\hat{y}_i$. Figure 2.12 shows an example of loss computation. The flow on the top shows the loss computation of a correct classification. In such a scenario, the input image is a cat and the output with highest probability is cat (86.75%). On the other hand, the flow on the bottom contains an image of a dog as input, which is classified as a frog (54.22%). We can see that the loss is closer to zero when the classifier is correct, whereas it is higher when the input is misclassified. It is important to mention that the loss is computed only with respect to the target (correct) class. For instance, in the incorrect classification example, even though the highest probability is 54.22% (frog class), the loss is computed over the dog probability 32.86%.



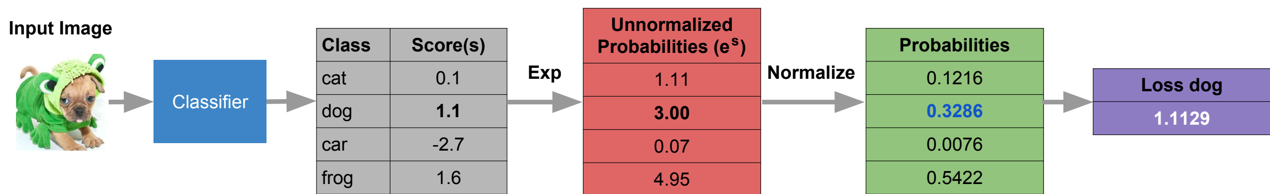


Figure 2.12: Example of log-likelihood loss computation for two different images. The first one is a correct classification while the second is an incorrect classification.

Another important aspect is that it is common to transform the target class values into a representation called *one-hot encoding*, since only the target class matters when computing the loss. Table 2.1 shows the example based on Figure 2.12. Each input is represented as a sparse vector, with the value 1 on the corresponding target class positions and 0 on all other positions. The loss function is then calculated as a sum over all classes K, which multiplies the negative log-likelihood. Therefore, we can rewrite the loss for the i-th instance L_i as Equation 2.7, where K is the number of classes, y_ik is the value of the one-hot encoded vector y for class k and instance i. Finally, Equation 2.8 computes the loss over all instances and divides the total sum by the number of instances N.

$$L_i = \sum_{k=1}^{K} -y_{ik} log_e\left(\frac{e^{s_{\hat{y}_{ik}}}}{\sum_j e^{s_j}}\right) \tag{2.7}$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i \tag{2.8}$$

Sigmoid and Softmax are very similar: they can be used for both binary and multi-class classification problems. Nevertheless, they model different probability distributions. The probabilities

in Sigmoid are not constrained to sum 1, which means that a class output could be 0.7 and another class be 0.8. On the other hand, Softmax output probabilities that sum to 1 — Figure 2.11 shows such example, where the outputs are $0.8675 + 0.1062 + 0.0002 + 0.0261$. In other words, Sigmoid models a *marginal distribution*, whereas Softmax models a *joint distribution*. A marginal distribution is that with unconditional probabilities. That is, the probability *P(A)* of a given event occur does not depend on any other event. Conversely, joint distributions are the ones whose probabilities are determined jointly by all variables involved. For instance, let us assume a classification problem of 4 classes. For a marginal distribution, all class probabilities are independent — *P(K=1)*, *P(K=2)*, *P(K=3)* and *P(K=4)*. For a joint distribution, however, the probability of each class depends on each other — *P(K=1, K=2, K=3, K=4)*. Such differences allow one to choose among the two depending on the problem at hand. For classification problems where classes are disjoint, Softmax is preferred.

Table 2.1: One-hot encoding example

	Classes			
Input	**Cat**	**Dog**	**Car**	**Frog**
Cat image	1	0	0	0
Dog image	0	1	0	0

Traditional neural networks with Sigmoid activation function have problems when the number of layers is increased. Intuitively, one might think that increasing the number of hidden layers would allow a neural network to learn more from data and hence, learn more complex functions. However, deep neural networks that make use of Sigmoid in their hidden layers suffer from a problem called *the vanishing gradient*. Recall that in Section 2.2.3 we mentioned Backpropagation, which is an algorithm that propagates the errors (also called *gradients*) in a neural network on a backward fashion. When the architecture has many hidden layers, the gradients tend to *vanish* (that their values are ≈ 0) and hence, the more distant to the output layer a hidden layer is, the slower it learns [81]. Since Sigmoid outputs values in the range $[0, 1]$, there are large regions of the input space which are mapped to an extremely small range. In such regions, even a large change in the input will produce a small change in the output and, hence, the gradient is small. This problem gets worse when we have multiple layers, where this consecutive mapping of larger inputs to smaller outputs affect the learning of the network. As a result, even a large change in the parameters of the first layer will not change the output by much. An alternative to overcome such problems is to use Rectified Linear Units (ReLU) instead of Sigmoid in the hidden layers of the network. ReLU (Equation 2.9), is an activation function that outputs 0 if $x \leq 0$, and x when $x > 0$. Therefore, all activations are thresholded to 0.

$$\psi(x) = max(0, x) \tag{2.9}$$

Unlike Sigmoid, ReLU does not squash the input space into small regions. Instead, it thresholds the value to 0, but it does not limit the maximum value of the activation. Besides

helping to solve the vanishing gradient, ReLU is preferred over other activation functions (for the hidden layers) as it accelerates the convergence of stochastic gradient descent, thus accelerating the training time [60]. Additionally, ReLU is also less computationally expensive when compared with other activation functions. For instance, Sigmoid is exponential whereas ReLU is a simple threshold.

Convolutional neural networks started to be applied to other tasks after the success in image classification, such as object detection and image verification. The next sections explore the application of ConvNets to such domains.

2.4 Convolutional Neural Networks for Localization

Object localization (or detection) is the task of detecting objects of known classes in images [6]. It searches for an object within an image and outputs its location, usually with bounding boxes or any other type of location identification. For instance, detecting faces in images is an object detection task. Another example is to detect people in surveillance videos. Along with object detection, there is also object recognition. It refers to the task of both detecting/localizing an object and assigning a label to it. A simple comparison between classification, detection, and recognition can be seen in Figure 2.13. The classification task assigns a label (*dog*) to the first image. In the middle image, a localization algorithm is able to detect two objects, without identifying them. In the rightmost image, the dog and the ball are both detected and correctly classified (recognition). Object detection is considered harder than image classification problems due to its complexity. The localization of objects depends on numerous candidate object locations that need to be refined in order to achieve a precise localization [34]. Sometimes, object detection is related to detecting and recognizing objects. However, throughout this dissertation we will refer to object detection as the task of only locating the object within an image, and object recognition as the task of detecting and classifying it. Two datasets for object localization and recognition are notable: PASCAL VOC [28] and MS COCO [74].

Figure 2.13: Comparison between image classification, object detection, and object recognition.

PASCAL Visual Object Classes (VOC) [28] is a dataset for building and evaluating algorithms for image classification, object detection, and segmentation. The PASCAL VOC challenge ran

from 2006 until 2012, and it has competitions for the three before-mentioned tasks: classification, detection, and segmentation. The creators increased the quantity of images throughout the years of the challenge. For instance, its 2010 version is comprised of around $10,103$ training images that contain $23,374$ objects and $9,637$ test images that contain $22,992$ objects. All images in the dataset were downloaded from Flickr [1] and they are organized into 20 different classes. PASCAL VOC is the current benchmark for object detection and recognition and it is a reference for evaluation methods as well.

Microsoft Common Objects in Context (MS COCO) [74] is a large-scale dataset for scene understanding. It is comprised of 91 common object categories with 82 of them having more than $5,000$ labeled instances. The dataset is significantly larger in number of instances per category than PASCAL VOC. When compared with ImageNet, MS COCO contains less classes but more images per class. MS COCO is also a benchmark for current object recognition algorithms and it is usually used for comparison along with PASCAL VOC. In 2015, MS COCO hold challenges for object detection and image captioning (describe what an image contains with natural language), whereas in 2016 the focus was on object detection and keypoint detection (simultaneously detecting people and localizing their keypoints).

Object detection differs from classification in the evaluation methodology as well. In classification, we compute whether the correct label is assigned to a given instance. In object detection we need to evaluate whether a predicted bounding box is correct with respect to the ground truth bounding box. A bounding box is an area defined by a quadruple (x, y, w, h), where x and y are the coordinates of the beginning of the bounding box, w is the width and h is the height of the box. A measure called *Intersection Over Union* (IoU) is used to calculate the amount of overlap between two bounding boxes. Equation 2.10 shows how it is calculated. The IoU between two bounding boxes A and B is given by the intersection between A and B divided by their union. The result is in range $[0, 1]$, where 1 represents 100% of overlap and 0 represents no overlap whatsoever. Figure 2.14 shows an example of different bounding box predictions and the impact on the IoU value. The larger the IoU value, the more similar are the two bounding boxes.

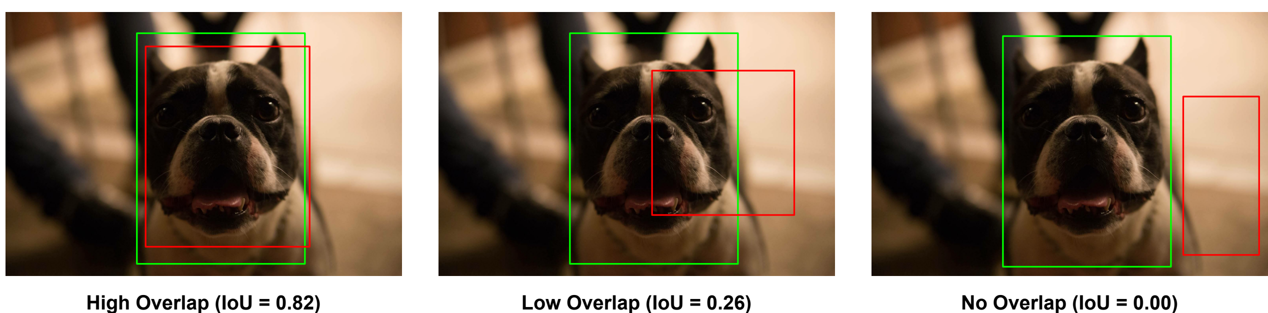


Figure 2.14: Values of IoU for different predictions. The green bounding box represents the grount truth. The red one is the prediction.

[1] Flickr is a photo management and sharing application. Available at: https://www.flickr.com

$$IoU(A,B) = \frac{A \cap B}{A \cup B} \tag{2.10}$$

Similarly to classification, the first approaches that won the PASCAL VOC challenge were based on hand-crafted features. For instance, in 2010 the winner implementation was based on HOG and *Discriminatively Trained Part Based Models* [30]. After the success of ConvNets for ImageNet, the research community wondered to what extent the CNN classification performance on ImageNet could be generalized to the object detection task on the PASCAL VOC challenge. In 2014, R-CNN was the first implementation based on CNNs that was able to beat hand-crafted approaches [35]. However, R-CNN had limitation mainly regarding training and prediction time, as well as disk usage. Such problems were addressed first by Fast R-CNN [34] and then by Faster R-CNN[93]. Additionally, many other deep learning based approaches such as Single Shot Detection [76], Yolo [91] and MTCNN [132] emerged. We will explore some of these implementations in the following sections.

2.4.1 R-CNN

Regions with CNN features (R-CNN) is a combination of region proposals with convolutional neural networks [35]. Figure 2.15 shows how R-CNN works. Given an input image, a region proposal algorithm extracts regions of interest. Each extracted region is warped into a fixed size representation and used as input to a pre-trained convolutional neural network. The CNN extracts features of such regions, which are used as input to linear SVMs for classification. These steps are detailed as follows.

Given an input image, a region proposal algorithm extracts around $2,000$ regions of interest (RoI). Different region proposal algorithms exist in the literature, but R-CNN is agnostic to this choice. Nevertheless, the authors of R-CNN make use of Selective Search [115] in order to compare it with previous approaches mentioned in their paper. Selective Search is an algorithm that detects objects at any scale, considers multiple grouping criteria (e.g. color, brightness and texture) and is fast. It basically generates an initial segmentation of an image with candidate objects (bounding boxes of regions). From this set of regions, the most similar are selected and combined into larger ones. This is executed until only one region remains, which results in a hierarchy of regions. The hierarchy is then used to produce candidate object locations. The segmentation performed prior to the generation of candidate objects makes this algorithm both accurate and fast. More details are available in [115] and [116].

Each region proposal is warped to a fixed size, which is used as input to a convolutional neural network. The CNN is an AlexNet [60] and it extracts a 4096-dimensional feature vector that is used for classification. In R-CNN, the CNN is pre-trained on ImageNet so it learns how to classify images. The network is then fine-tuned on the PASCAL VOC dataset. The last fully-connected layer is replaced, changing from 1000 to 21 classes, but the rest of the architecture remains

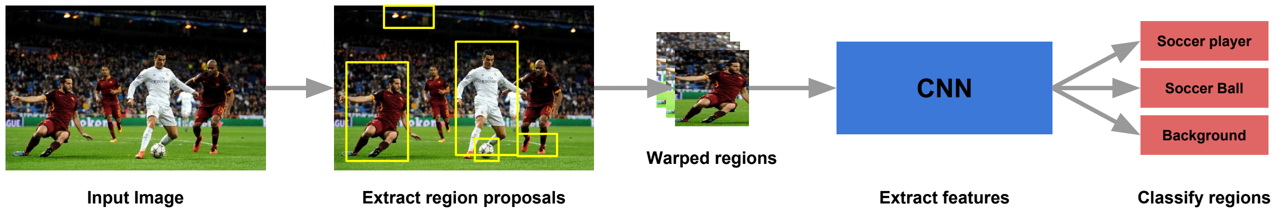


Figure 2.15: How R-CNN works (Image adapted from [35]).

unchanged [35]. Each region proposal is warped to 227×227, to be compliant with AlexNet input size.

The next step is to classify the features extracted from the fine-tuned CNN. In order to use a CNN as a feature extractor, one must remove the classification layer. Therefore, the previous layer — usually a fully-connected layer — is used as a feature vector that represents a given input image. For R-CNN, the features in the fifth pooling layer are used as input to a binary SVM that is optimized per class. Each binary SVM classifies a given region proposal as an object or background.

Object recognition algorithms have two outputs: a label for each identified object and the bounding boxes. Besides training SVMs for classification, R-CNN also trains a bounding box regressor with the goal of adjusting specific region proposals to output a final (and possibly more accurate) bounding box prediction. It works as follows: in the training phase, the features extracted from the CNN are used as input to a linear regression algorithm only if the IoU between the ground truth bounding box GT and the region proposal $P \geq 0.6$. The input for the training algorithm is a set of N training pairs (P, GT). A region proposal is represented by $P^i = (P^i_x, P^i_y, P^i_w, P^i_h)$, where P^i_x and P^i_y are the x and y coordinates of the bounding box and P^i_w and P^i_h are the height and the width, respectively. Each GT bounding box is represented in the same manner $GT^i = (GT^i_x, GT^i_y, GT^i_w, GT^i_h)$. The linear regression aims to transform the input region proposal P into a predicted bounding box $\hat{P}$. The transformation is parametrized by four functions that are learned: $d_x(P)$, $d_y(P)$, $d_w(P)$ and $d_h(P)$ [35]. With these learned functions, $\hat{P}$ is obtained with

$$\hat{P}_x = P_w d_x(P) + P_x \quad (2.11)$$
$$\hat{P}_y = P_h d_y(P) + P_y \quad (2.12)$$
$$\hat{P}_w = P_w e^{d_w(P)} \quad (2.13)$$
$$\hat{P}_h = P_h e^{d_h(P)}. \quad (2.14)$$

Each $d_*(P)$ is modeled as a linear function between the features ϕ extracted from a given region proposal P (Equation 2.15), where w_* is the vector of learned weights and ϕ_5 are the features from the fifth pooling layer extracted with the CNN. The weights are learned with the optimization of *regularized least squares* (also called *ridge regression*), given by Equation 2.16, where t_* are the regression targets for the training pair $\{P, GT\}$, which are defined as

$$d_* = \boldsymbol{w}_*^T \phi_5(P) \tag{2.15}$$

$$\boldsymbol{w}_* = argmin \sum_{i=1}^{N} (t_*^i - \hat{\boldsymbol{w}}_*^T \phi_5(P^i))^2 + \lambda ||\hat{\boldsymbol{w}}||^2 \tag{2.16}$$

In test time, the trained algorithm predicts a new region proposal, generating the new bounding boxes. All generated bounding boxes are passed to the Non-Maximum Suppression (NMS) algorithm before being evaluated. NMS eliminates duplicated detections by the prioritization of regions with higher classification scores according to the SVMs [69].

$$t_x = \frac{(GT_x - P_x)}{P_w} \tag{2.17}$$

$$t_y = \frac{(GT_y - P_y)}{P_h} \tag{2.18}$$

$$t_w = log(\frac{GT_w}{P_w}) \tag{2.19}$$

$$t_h = log(\frac{GT_h}{P_h}). \tag{2.20}$$

R-CNN optimizes three different object functions at the same time: Log Loss for Softmax classifier in the fine-tuning process, the Hinge Loss for SVM and least squares for bounding boxes regression. The last two are post-hoc, which means they are executed after the CNN training is completed and do not affect the weights learned. The training procedure takes almost 84 hours and the extracted features require hundreds of gigabytes of storage [34]. The inference is also slow because R-CNN applies the CNN for each of the $2,000$ region proposals in an image basis. The slow inference problem was afterwards solved by SPP-Net [42].

Spatial Pyramid Pooling Network (SPP-Net) [42] aims to solve the problem of the CNN accepting only fixed-size inputs while improving inference speed. The convolutional layers accept arbitrary input sizes, but they produce outputs of variable sizes. On the other hand, the classifiers (SVM or Softmax) or fully-connected layers require vectors of fixed size. Spatial pyramid pooling performs the pooling operation in the so-called local spatial bins. These spatial bins have sizes proportional to the image size, hence the number of bins is fixed regardless of the image size [63]. SPP-Net enables CNNs to have different input sizes by changing the last pooling layer with a spatial pyramid pooling layer. The output are $(k \times M)$-dimensional vectors, where M denotes the number of bins and k is the number of filters in the last convolutional layer. Figure 2.16 shows more details about SPP-Net.

R-CNN uses a CNN to extract features from $2,000$ different region proposals, which makes this process one of its main bottlenecks. SPP-Net solves this issue by extracting the feature maps of the image only once. It means that instead of using regions of the image as input, SPP-Net makes use of regions in the feature maps. The spatial pyramid pooling is applied on each region of interest

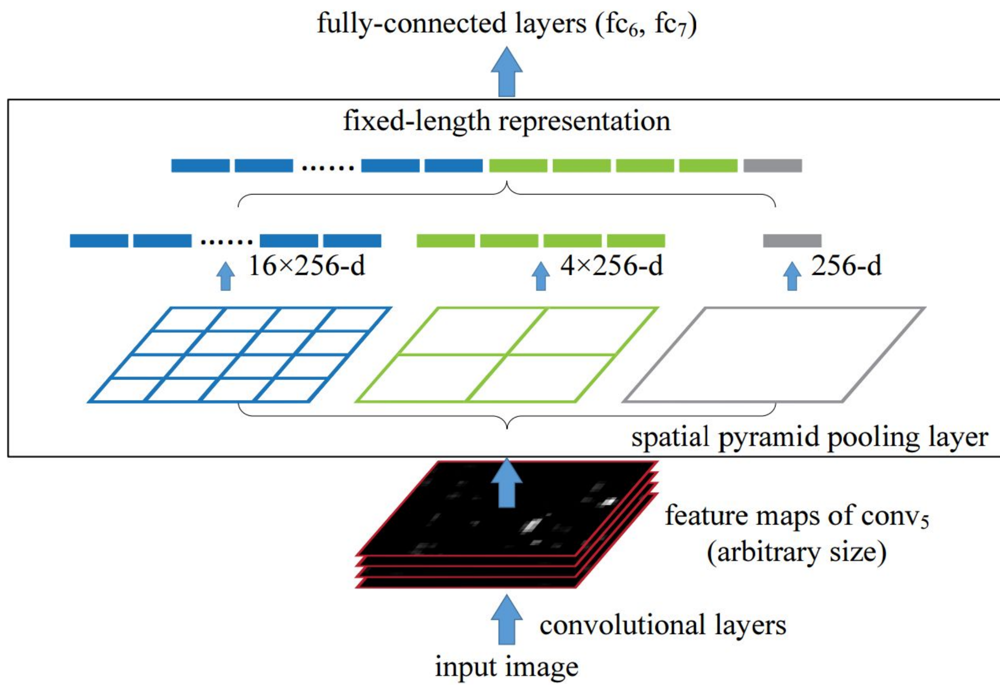


Figure 2.16: A network with spatial pyramid pooling layer [42].

within the feature maps to pool a fixed-length representation of this region. Since the convolution operation was a big issue for inference, reducing drastically the number of convolutions resulted in a faster implementation.

Even though SPP-Net solved the inference slowness of R-CNN, the other problems remain: slow training time and disk consumption. Additionally, SPP-Net introduces a new problem, which is related to the update of parameters that are below the spatial pyramid pooling layer during the training phase. Such problems are addressed by the improved implementations of R-CNN: Fast R-CNN and Faster R-CNN.

2.4.2 Fast R-CNN

Fast Region-based Convolutional Network (Fast R-CNN) [34] is an improved version of R-CNN. One of the major improvements is that Fast R-CNN is trained to jointly learn to classify object proposals and refine their spatial locations. Such change along with other choices reduce training time and disk space requirements, making Fast R-CNN run faster and require less storage. Additionally, it improves the main drawback of SPP-Net: the fine-tuning algorithm cannot update the convolutional layers that lie before the spatial pyramid pooling. Such problem limits the accuracy of very deep networks. Therefore, Fast R-CNN fixes the problems of both R-CNN and SPP-Net, as well as being faster to train and to test. Additionally, Fast R-CNN has higher detection quality, it is trained in a single stage pipeline with a multi-task loss (instead of three different loss functions), the training process is able to update all network layers, and no features are cached into disk.

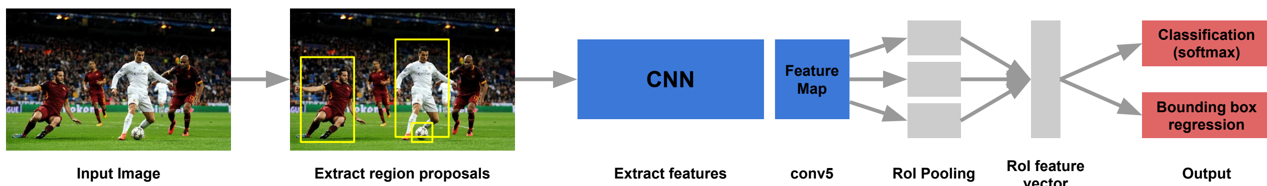


Figure 2.17: How Fast R-CNN works (Image adapted from [34]).

The architecture changed compared with the original R-CNN architecture. Fast R-CNN receives as input an image and a set of object proposals. The network processes the image and outputs a feature map. As with SPP-Net, Fast R-CNN also uses a region of interest (RoI) pooling layer, which extracts a fixed-length feature vector from the feature map for each object proposal. The RoI pooling layer uses max-pooling to convert any region of interest features into a small feature map with a fixed spatial size of $H \times W$, where H and W are layer hyper-parameters that are not tied to any specific RoI. These feature vectors are used as input to fully-connected layers. Finally, the fully-connected layers have two different output layers: a softmax layer to classify the object proposals, and a layer to estimate the bounding boxes of each proposal (four real-valued numbers representing bounding box positions). Figure 2.17 shows how Fast R-CNN works.

The authors tested three different convolutional neural network architectures with Fast R-CNN: AlexNet (the same used in R-CNN), referred to as small **S**; VGG [15], referred to as medium **M**; and very deep VGG [105], referred to as large **L**. VGG is a deep neural network developed by Oxford's Visual Geometry Group (VGG - hence the name), and it was the runner-up of ImageNet 2013 challenge. It has two different implementations, where one has 16 layers (VGG-16) and the other has 19 layers (VGG-19). Figure 2.18 shows VGG-19 architecture. When compared with AlexNet, it contains more layers and smaller kernels and it showed that increasing the depth of the network improves the performance. Nevertheless, VGG is more expensive to evaluate and uses a lot more memory and parameters.

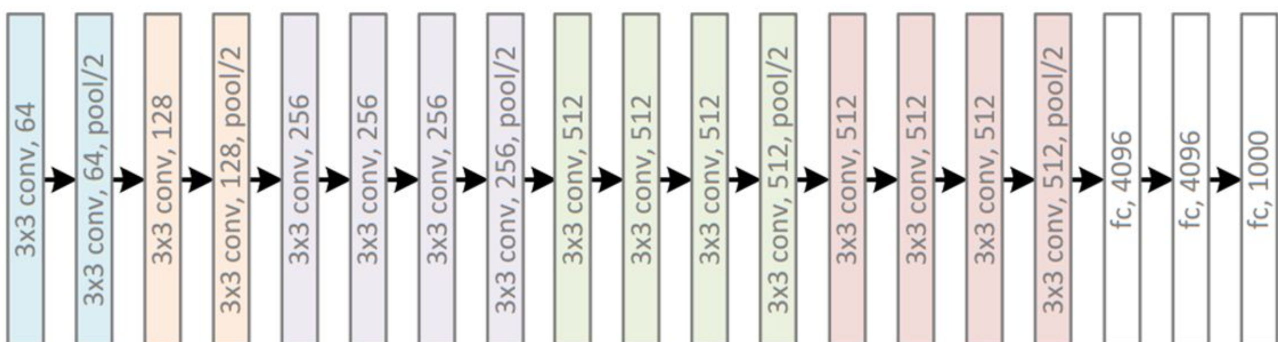


Figure 2.18: VGG-19 architecture.

For pre-training, all networks suffered three changes: max-pooling layer replaced by a RoI pooling layer; the last fully-connected layer and softmax are replaced with two sibling layers (a fully-connected layer and a softmax and bounding box regressors), and the input is changed to accept a list of images and a list of RoIs in such images. Besides these changes, Fast R-CNN also has

feature sharing during training phase and joint optimization in the training process [34]. During the CNN training, proposals from the same image share computation and memory in both forward and backward passes of the network. This is achieved by sampling mini-batches hierarchically, sampling by N images and by R/N proposals from each image. The authors used $N = 2$ and $R = 128$. The training phase jointly optimizes a softmax classifier and the bounding box regressors instead of having three separate stages like R-CNN. The joint optimization is possible due to a multi-task loss, given by Equation 2.21.

$$L(p, u, t^u, v) = \mathcal{L}_{cls}(p, u) + \lambda[u >= 1]\mathcal{L}_{loc}(t^u, v) \tag{2.21}$$

Given $K + 1$ classes (K classes plus background), p is a discrete probability distribution, which is the output of the softmax for all $K+1$ classes. t^k is the bounding box regression output for each of the K object categories (indexed by k). Each RoI training instance has a ground truth class label u and a ground truth bounding box target v. $\mathcal{L}_{cls}(p, u)$ is the log-likelihood for class u. $\mathcal{L}_{loc}$ is computed with respect to a ground truth bounding box v for class u and a target t. Both are a quadruple in the form (x, y, w, h). The expression $[u >= 1]$ is equal to 1 if class u is larger or equal to 1 and 0 otherwise. It means that such loss is computed for all classes except the background, which is ignored. The bounding box regression loss is defined in Equation 2.22.

$$\mathcal{L}_{loc}(t^u, v) = \sum_{i \in \{x,y,w,h\}} smooth_{L1}(t_i^u - v_i) \tag{2.22}$$

Instead of using $L2$ loss as in R-CNN, Fast R-CNN uses a $L1$ loss instead, given by Equation 2.23. $L1$ loss is less sensitive to outliers [34]. The ground truth regression targets v_i are normalized to have zero mean and unit variance, as to not allow one feature "dominate" another. The hyper-parameter λ in Equation 2.21 is used to control the balance between the two different losses. In the original implementation the value of λ is 1.

$$smooth_{L1}(x) = \begin{cases} 0.5x^2 & if|x| < 1 \\ |x| - 0.5 & otherwise \end{cases} \tag{2.23}$$

Fast R-CNN solved most of R-CNN gaps. The training and test time speed greatly increased ($9\times$ faster in training and $213\times$ faster in test), the features are no longer cached in disk, all network parameters can be updated, training is a single-stage that makes use of a multi-task loss, and it significantly improved detection results in Pascal VOC. However, Fast R-CNN still depends on an "external" algorithm for generating region proposals (e.g. Selective Search). Faster R-CNN is an upgrade of Fast R-CNN that implements the generation of region proposals within the network pipeline.

2.4.3 Faster R-CNN

Faster R-CNN [93] is the improved version of Fast R-CNN. The main limitation of Fast R-CNN is that the region proposals are generated outside the network [34]. Algorithms as Selective Search [115] or EdgeBoxes [135] are used to this end. Even though EdgeBoxes has a good proposal quality and speed, the execution time is simillar to the running time of the detection network [93]. Faster R-CNN uses *Region Proposal Network* (RPN), which is a fully-convolutional network that both predicts the bounding box position and the scores at each position. Such RPN is trained in an end-to-end manner to generate high-quality region proposals that are used by Fast R-CNN for detection. RPN and Fast R-CNN are merged into a single network and they share some convolutional layers, which makes Faster R-CNN (the resulting combination) faster and more accurate than Fast R-CNN. Figure 2.19 illustrates Faster R-CNN overall architecture. Faster R-CNN is comprised of two modules. The first one is a convolutional neural network that proposes regions (RPN) and the second is the Fast R-CNN detector, which uses the proposed regions.

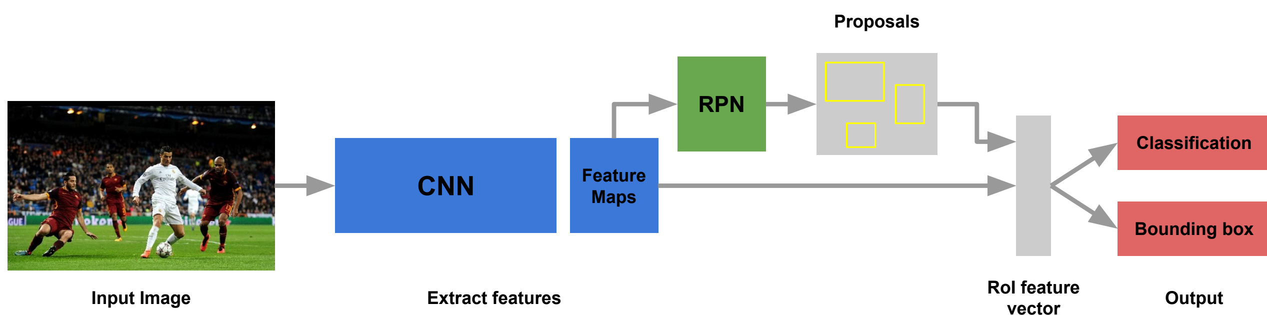


Figure 2.19: How Faster R-CNN works.

RPN replaces the previous hand-crafted approaches for proposals generation. Given an input image, it outputs a set of region proposals along with a score. Such score indicates whether the given region proposal is from a class or from background. The generation of region proposals work as follows: a window is slided over the convolutional feature map, generating the region proposals. The proposals are mapped to a feature vector whose dimensionality is tied to the CNN model being used. The feature vector is then used as input to two sibling fully-connected layers: a bounding box regressor and a bounding box classifier. Figure 2.20 shows how RPN works. Multiple proposals are generated at each sliding window location, where the maximum number of proposals is defined as k. In Figure 2.20, we can see that the classification layer (*cls*) outputs $2k$ scores, while the regression layer (*reg*) outputs $4k$ bounding box coordinates. $2k$ means that each proposal k has 2 different outputs - background or the given class. The k proposals are generated based on *anchors*. The anchor is centered at the given sliding window. Based on the center, 9 anchors are generated - 3 different scales (areas of 128^2, 256^2 and 512^2) with 3 different aspect ratios ($1:1$, $1:2$ and $2:1$) [93]. Hence, Faster R-CNN is able to classify and predict bounding boxes based on anchor boxes of multiple scales and aspect ratios.

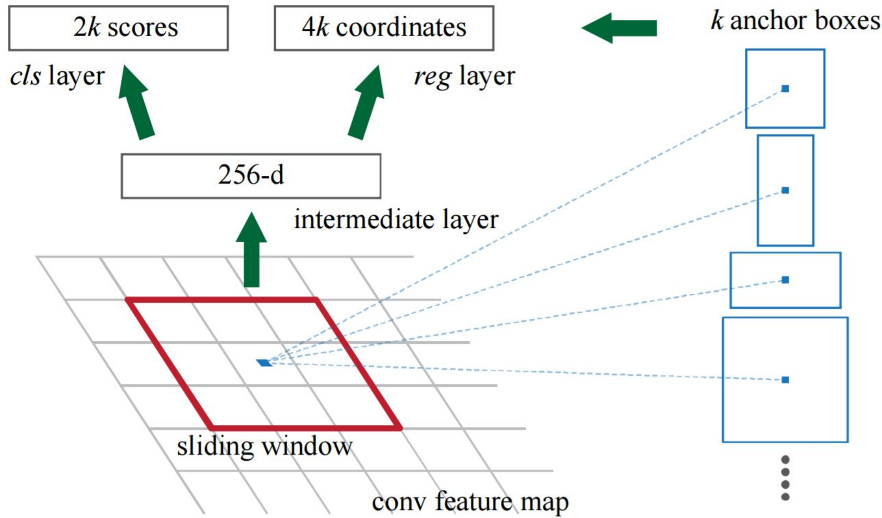


Figure 2.20: Region Proposal Network [93].

RPN uses a multi-task loss similar to the one used in Fast R-CNN, given by Equation 2.24. i is the index of a given anchor (proposal) and p_i is the probability of such anchor be an object. p_i^* is the ground truth label, which is 1 if the anchor is an object and 0 otherwise. t_i and t_i^* are the predicted and ground truth bounding boxes coordinates, respectively. In the RPN case, the classification loss L_{cls} is over two classes (object and not object). The regression loss L_{reg} is the same as in Fast R-CNN, which uses the smooth $L1$ function. It is ignored if the ground truth label p_i^* is 0, which means that it is not an object. N_{cls} and N_{reg} are normalizing both losses while λ is a balancing parameter. As the k anchors have different sizes, k bounding box regressors are learned. However, the regressors do not share weights.

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*) \tag{2.24}$$

Even though the authors mention that the network is trained in an end-to-end manner, there are four different steps in Faster R-CNN training. The first step is to train the RPN. Such network is pre-trained with ImageNet and fine-tuned on PASCAL VOC for the region proposal task. The second step is to train the Fast R-CNN with the proposals generated by the RPN fine-tuned in the first step. Fast R-CNN is also pre-trained with ImageNet. In the two first steps, no convolutional layers are shared between RPN and Fast R-CNN. However, in the third step the convolutional layers are shared but only RPN layers are fine-tuned. The final step is to fine-tune only the Fast R-CNN layers.

The training method just described is the one used for all experiments in the Faster R-CNN original paper [93]. However, the authors implemented another method that trains the whole network as a single model, jointly optimizing regression and classification losses. Such version is simpler to train and yields similar results in accuracy [2].

[2]Unpublished results are available at https://www.dropbox.com/s/xtr4yd4i5e0vw8g/iccv15_tutorial_training_rbg.pdf?dl=0.

2.4.4 MTCNN

Multi-task Cascaded Convolutional Networks (MTCNN) is an implementation that performs both face detection and face alignment [132]. The implementation does both tasks in a single pipeline using unified cascaded CNNs and a multi-task learning method. The architecture is basically comprised of three stages. The first stage generates candidate windows with a shallow CNN. The second refines the candidate windows with a more complex CNN, which is able to reject windows that do not contain faces. Finally, the last stage has an even more complex CNN that refines the windows again and outputs the bounding box along with face landmarks. The CNNs are referred to as Proposal Network (P-Net), Refinement Network (R-Net) and Output Network (O-Net), respectively. Figure 2.21 illustrates the pipeline.

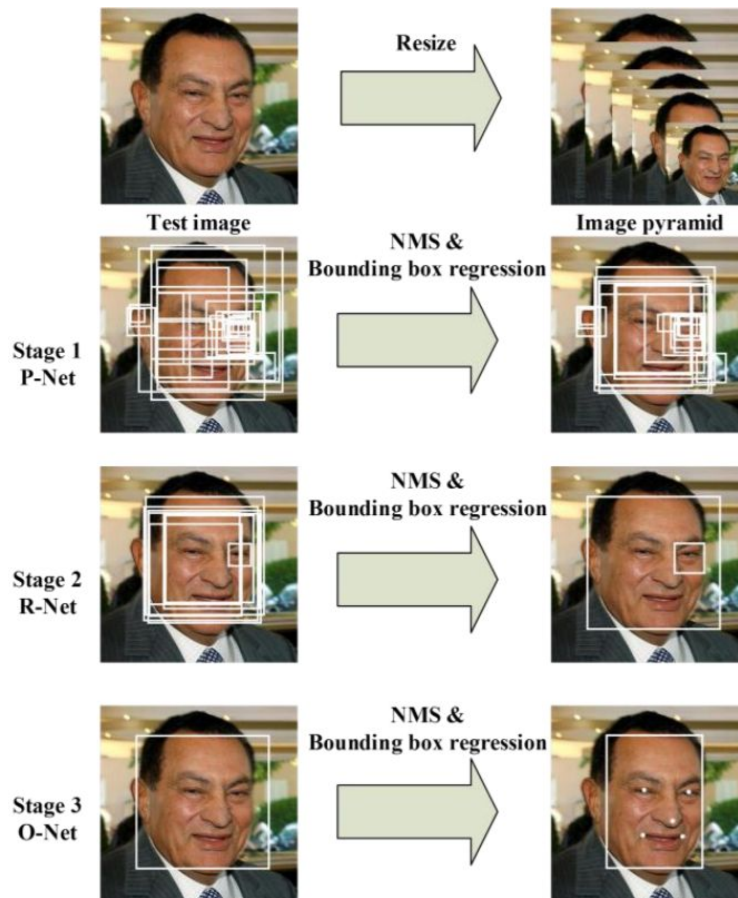


Figure 2.21: Overview of MTCNN pipeline [132].

Before the first stage, a given image is resized to different scales in order to build an image pyramid. The pyramid is used as input to the first stage, which contains the P-Net. P-Net is a fully convolutional network (FCN) that obtains the candidate windows and their bounding box regression vectors [132]. The windows are calibrated based on the bouding box regression vector. Finally, nonmaximum supression is applied to combine windows that have a high overlap in a way that is similar to Faster R-CNN. A fully convolutional network is very similar to a regular CNN, with

the difference that it does not contain fully-connected layers at the end [103]. When we remove fully-connected layers of a CNN we also remove the need of a fixed input size. Instead of a label or a feature vector, the output of an FCN is called *heatmap*. Each point of the heatmap is related to a region in the original input. Each activation is used as the proposal probability, which means that every point in the original input image has a probability of being a face. Such probability is then used to locate faces [129].

In the second stage, all proposals (candidate windows) are fed to another CNN: the R-Net. R-Net is responsible for rejecting a large number of false proposals. It is also responsible for performing calibration with bounding box regression. The final process of the second stage is to apply nonmaximum supression. The third and final stage is very similar to the second one, the only difference is that the CNN is deeper and the number of filters is increased. Figure 2.22 details the three different architectures.

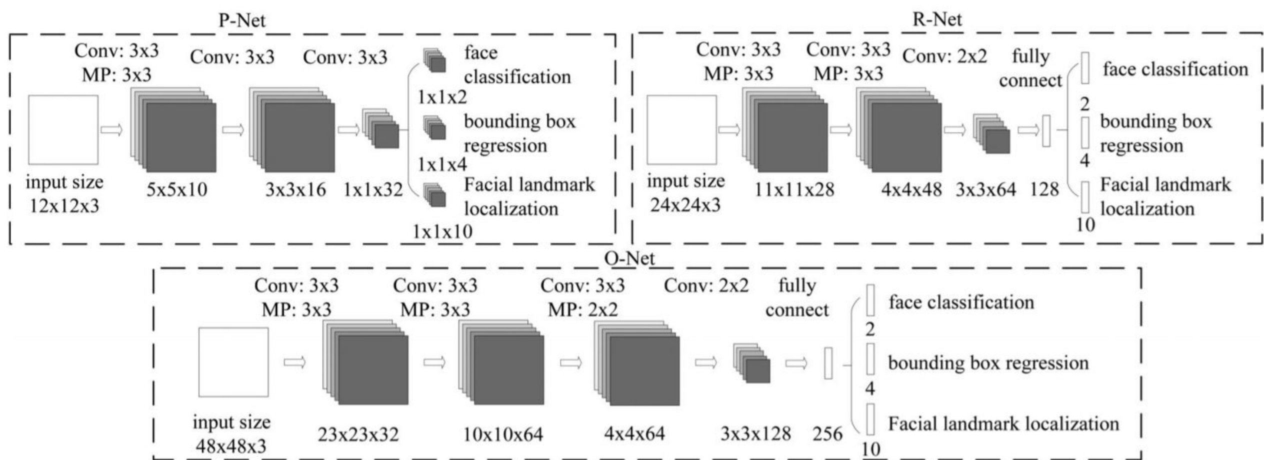


Figure 2.22: MTCNN networks in details: P-Net, R-Net and O-Net [132].

P-Net is a fully-convolutional network comprised of three convolutional layers, with 10, 16, and 32 filters, respectively. The kernel size for all convolutions is 3×3. A max-pooling of 3×3 is used after the first convolution. P-Net contains three outputs: the face classification, which is a softmax; a bounding box regression (similar to Faster R-CNN); and a facial landmark localization, which are 10 values representing 5 facial landmarks (five x, y pairs). The output of the bounding box regression is used to project the bounding box in the input. The input is then completed with zeros until reaching the size of 24×24, which is used as input to R-Net. R-Net is a regular CNN, comprised of three convolutional layers and a fully connected layer. The two first convolutions are followed by max-pooling layers. R-Net contains more filters and larger volumes when compared with P-Net. The output of the third convolution is flattened and connected to 128 fully-connected neurons. This fully-connected layer has again three outputs, which are the same as P-Net. Again, as it happens for P-Net, the bounding box regression output is projected in the input, completed with zeros and used as input to the O-Net. O-Net is a more complex network, which contains four convolutional layers and more neurons in the fully-connected layer at the end (256). The network also contains three outputs, which are the final predictions of MTCNN.

MTCNN uses a multi-task learning method that optimizes three different loss functions: one for face classification, one for bounding box regression and another for facial landmark localization. The first one makes use of the cross-entropy loss (Equation 2.4), which we explained in Section 2.3. In this case, there are only two classes (1 and 0) that represent whether the input is a face or not, respectively. We will denote this loss as L_i^{face}. The second uses the *Euclidean loss*, represented by Equation 2.25. L_i^{box} is the loss of input x_i. t_i^{*box} is the ground truth bounding box whereas t_i^{box} is the predicted bounding box. In this case, each bounding box is represented by four values - top, left, height, and width. Finally, the facial landmark loss is computed similarly to the bounding box regression. Equation 2.26 illustrates this loss, which is also an Euclidean loss, denoted by L_i^{land}. t_i^{*land} are the ground truth facial landmark's coordinates whereas t_i^{land} are the predicted facial landmark's coordinates. There are five facial landmarks, including left eye, right eye, nose, left mouth corner, and right mouth corner. Therefore, both t_i^{land} and t_i^{*land} contain ten values, which correspond to five (x, y) pairs (one for each coordinate).

$$L_i^{box} = ||t_i^{*box} - t_i^{box}||_2^2 \tag{2.25}$$

$$L_i^{land} = ||t_i^{*land} - t_i^{land}||_2^2 \tag{2.26}$$

The multi-task loss is given by Equation 2.27. N is the number of training instances. j represents the task - classification (*face*), bounding box regression (*box*), or landmark detection (*land*). α^j represents the task importance. It is used because each network gives a different importance for the different losses. P-Net and R-Net use $\alpha^{face} = 1$, $\alpha^{box} = 0.5$, $\alpha^{land} = 0.5$, as their main goal is to reject proposals without faces. On the other hand, O-Net uses $\alpha^{face} = 1$, $\alpha^{box} = 0.5$, $\alpha^{land} = 1.0$ aiming a better facial landmark localization. β_i^j denotes the instance type. If the given input instance is background (does not contain any face), only L_i^{face} loss is computed, while the other two losses are set to 0 (setting their β to 0).

$$min \sum_{i=1}^{N} \sum_{j \in \{face, box, land\}} \alpha^j \beta_i^j L_i^j \tag{2.27}$$

MTCNN uses the idea of classical face detectors, which are based on a cascade of classifiers. Each stage of the cascade aims to reject images that do not contain faces. The first stages are easier, in the sense that they are simpler and lightweight. The last stages are more complex and usually have a larger computational cost. We will detail the classic approaches of cascade classifiers in Chapter 3.

2.5 Convolutional Neural Networks for Metric Learning

Classification usually requires that all labels are known in advance and that each category (class) contains training examples. Additionally, traditional classification methods may be unsuitable for applications where the number of categories is very large or where the number of instances per category is small, and where only a subset of the categories is known at time [16]. For example, face recognition, where we may have a large amount of people but a few images per person. One approach to such kind of problem is distance-based methods, which compute a similarity metric (distance) between a given input and a dataset. Metric Learning — also known as Distance Metric Learning — is the task of learning a distance metric (or function) for a given task [61]. The idea is to learn a function that maps input patterns into a target space such that a distance measure in the target space approximates the "semantic" distance in the input space [16].

Unlike classification, where training examples are given class labels, the training examples of a metric learning algorithm are similar and dissimilar pairs. A similar pair is given by $S = (x_i, x_j)$, where x_i and x_j should be similar. A dissimilar pair is given by $D = (x_i, x_k)$ where x_i and x_k should be dissimilar. Therefore, x_i should be more similar to x_j than to x_k. Besides, a label indicates whether the given input pair is similar or dissimilar. Usually, the values 0 and 1 are used to indicate such similarity. In some scenarios, such values indicate the distance itself, where 0 means that the inputs are equal and 1 that they are completely different. Hence, based on a dataset comprised of similar pairs S and dissimilar pairs D, a metric learning algorithm learns a distance function that is able to measure how similar two objects are.

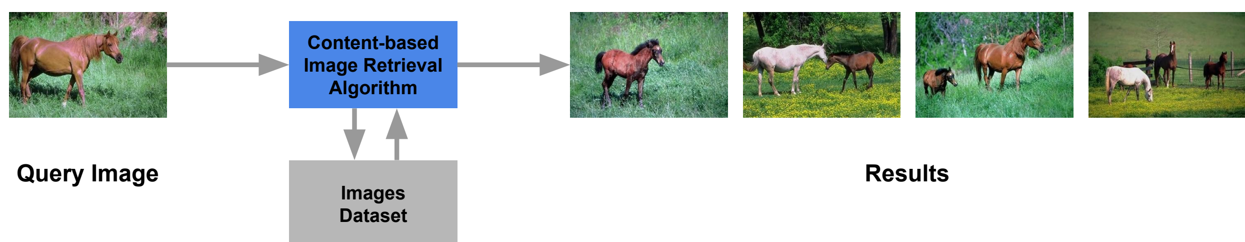


Figure 2.23: Simple example of Content-based Image Retrieval (Images from SIMPLIcity dataset [121]).

Any application that requires the comparison between two instances could benefit from metric learning. For instance, metric learning could learn a distance function that verifies whether two faces are from the same person or not. Content-based image retrieval is another example. Given an input image (also called query image), an algorithm must retrieve images that are most similar to the input [95]. Figure 2.23 illustrates an example of image retrieval. Classic implementations of content-based image retrieval are based on hand-crafted features and a distance-based algorithm that evaluates whether the images are similar or not. For instance, a hand-crafted feature extractor such as Local Binary Patterns (explained in details in Section 3.2.1.2) is used to extract a feature

vector that represents the images. Then, a distance-based algorithm like *K-means* is used to group most the similar images [118].

Digital signature verification was a relevant problem back in the nineties. The idea is to evaluate whether two signatures are the same or not. One successful implementation for such problem is *siamese neural networks*. Siamese networks are an implementation of neural networks that aim to learn a distance metric. They are a special kind of neural networks comprised of two identical sub-networks that are joined at the output [13]. A siamese network has two inputs and it outputs a similarity score that specifies whether the two inputs are similar. Figure 2.24 shows an example of a siamese network architecture. x_1 and x_2 denote the inputs. $G_{\boldsymbol{W}}$ is a complex function (e.g. a regular neural network, a convolutional neural network). This complex function is the same in both branches (one receives x_1 whereas the other x_2) and it is parametrized by $\boldsymbol{W}$, which stands for the weights of the neural network (shared between both branches).

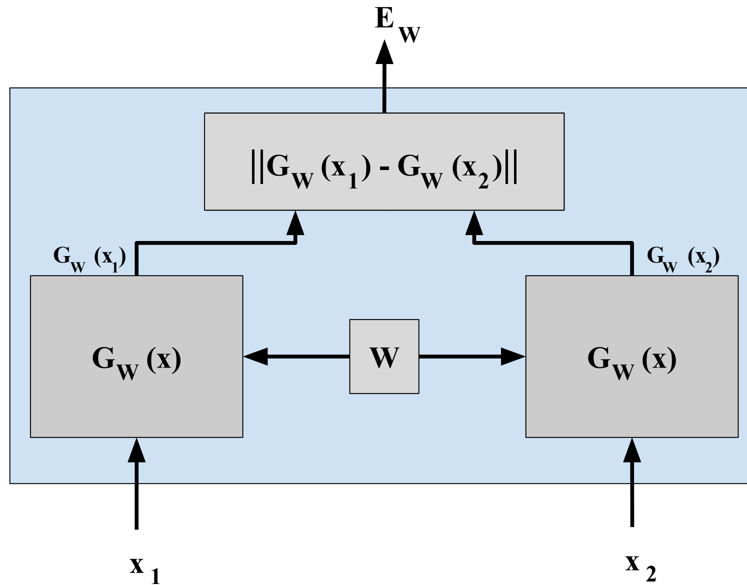


Figure 2.24: Classical siamese network architecture [16].

A siamese architecture works as follows: given a complex function $G_W(x)$, it aims to find the value of the weights $\boldsymbol{W}$ in a way that a similarity measure, given by Equation 2.28, yields a small value if x_1 and x_2 are from the same class and a larger value otherwise. Equation 2.28 is an *Energy Function*, which is based on *Energy-based Models*. The idea behind Energy-based Models is to capture the dependencies between variables, which are obtained by a measure of compatibility among the variables (e.g. the energy function). These dependencies allow the learned model to evaluate the values of unknown variables given the values of known variables [66]. Hence, Equation 2.28 can be seen as an Energy Model that aims to measure the compatibility of x_1 and x_2.

$$E_w(x_1, x_2) = ||G_W(x_1) - G_W(x_2)|| \tag{2.28}$$

The input of the siamese network may be a pair of images (x_1, x_2) and a label y, which represents whether the pair of images is genuine, from the same category or class, or impostor, different categories or classes. Usually, the values used are 0 for genuine pairs and 1 for impostor pairs. The *Contrastive Loss Function* is used to calculate the loss and it is responsible for allowing the training of such a network to happen [16]. This loss function is presented in Equation 2.29. P is the number of training samples (pairs) and $(y, x_1, x_2)^i$ is the i-th sample (pair of instances and label). $\mathcal{L}_G$ is the partial loss function for a genuine pair whereas $\mathcal{L}_I$ is the partial loss function for an impostor pair. L_G and L_I aim to decrease the energy of genuine pairs and increase the energy of the impostor pairs through the minimization of $\mathcal{L}$. $\mathcal{L}$ can be different loss functions. One common approach is given by Equation 2.31. $D_{\boldsymbol{W}}$ is the distance between the pair (x_1, x_2), m is the margin (larger than zero). The effect of such a margin is to allow dissimilar pairs to contribute to the loss function only if their distance is within such margin [41].

$$L(\boldsymbol{W}) = \sum_{i=1}^{P} \mathcal{L}(\boldsymbol{W}, (y, x_1, x_2)^i) \tag{2.29}$$

$$\mathcal{L}(\boldsymbol{W}, (y, x_1, x_2)^i) = (1-y)\mathcal{L}_G(E_{\boldsymbol{W}}(x_1, x_2)^i) + (y)\mathcal{L}_I(E_{\boldsymbol{W}}(x_1, x_2)^i) \tag{2.30}$$

$$L(\boldsymbol{W}, (y, x_1, x_2)) = (1-y)\frac{1}{2}(D_{\boldsymbol{W}})^2 + (y)\frac{1}{2}max(0, m - D_{\boldsymbol{W}})^2 \tag{2.31}$$

$G_{\boldsymbol{W}}$ is any given complex function. It can be a shallow neural network or even a convolutional neural network. In the case of siamese convolutional neural networks, the architecture has some differences when compared with classification CNNs, for example. A siamese CNN contains two identical branches with the weights shared between the branches. However, instead of a classification layer (e.g. softmax) at the end of the network, the output layer of both networks are joined in a layer that computes the distance between both outputs. Figure 2.25 shows an example.

There are implementations of siamese convolutional neural networks for different tasks. We will talk about two implementations. The first one is related to dimensionality reduction with siamese convolutional neural networks [41]. The problem is to learn a function that maps high dimensional input patterns to lower dimensional outputs, given similarities between instances in a given input space. The implementation uses a dataset of handwritten characters called MNIST [65]. MNIST is comprised of a training set of $60,000$ examples, and a test set of $10,000$ examples. All examples are 32×32 images with handwritten digits. The data is organized in genuine and impostor pairs, as to be compatible with the siamese network input. Figure 2.26 illustrates the architecture used as $G_{\boldsymbol{W}}$. It is a simple convolutional neural network comprised of two convolutional layers, one pooling layer, and a fully-connected layer. The output of the network is a vector of two positions, which represents a (x, y) point in the Euclidean space. It is a reduced representation of the input

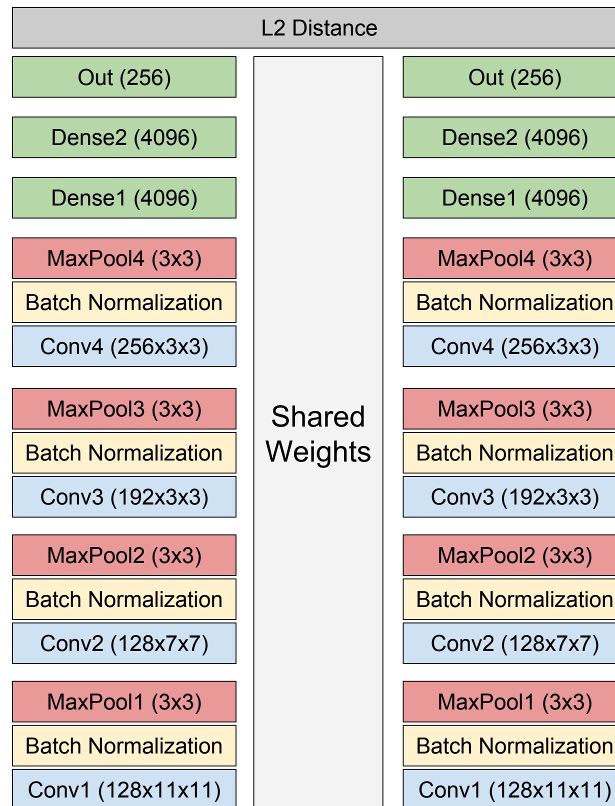


Figure 2.25: Example of a siamese convolutional neural network. The output of both networks are joined through a layer that computes distance, which is a $L2$ (Euclidean) distance in this case. The output is a score that indicates how similar the two input instances are.

32×32. Figure 2.27 shows some results. We can see that the resulting reduced vectors that represent each digit are easily separated.

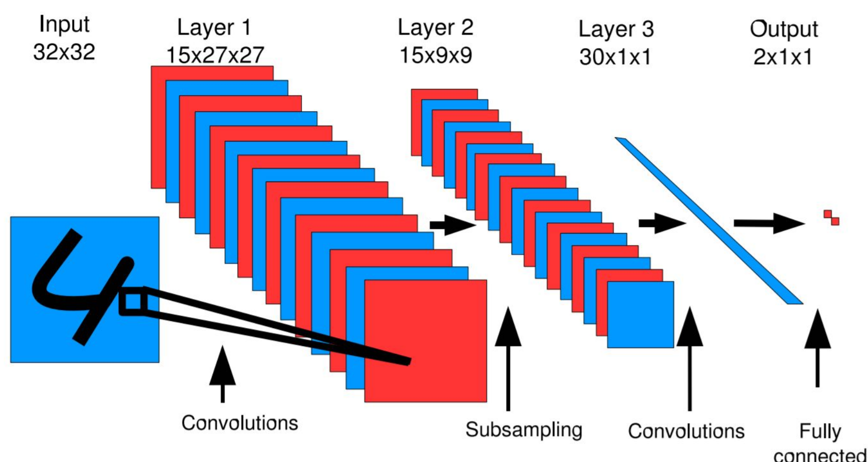


Figure 2.26: CNN architecture that maps the MNIST data to a low dimensional space (Image from [41]).

The second work is an implementation that learns an embedding for visual search in interior design [9]. It makes use of a siamese CNN that is able to learn an embedding from images. The embedding is used for different applications, such as finding products in images, finding designer

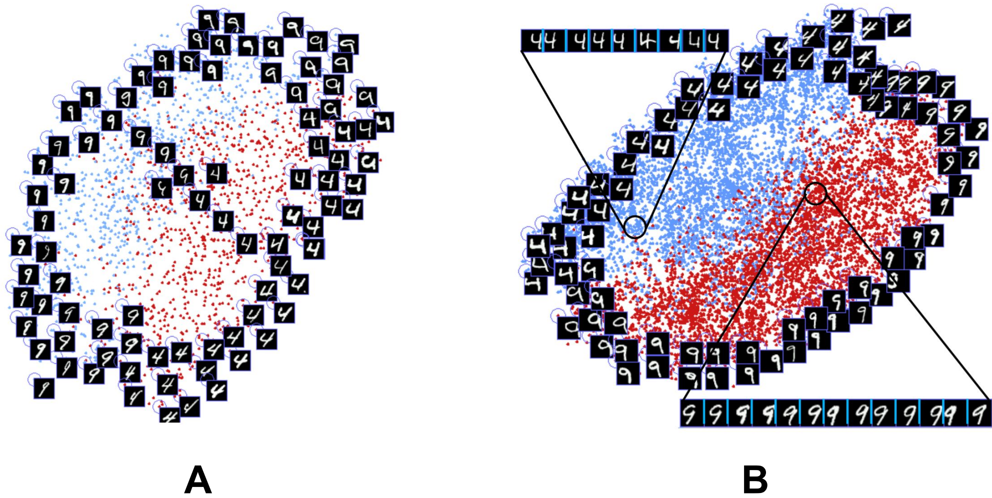


Figure 2.27: Experiments that demonstrate the dimensionality reduction results. A is an easy setup, where the digits are simpler. B is a hard setup, where some noisy images are used. (Images from [41]).

scenes that show products, and finding visually-similar products among different categories. They downloaded $7,249,913$ product photos and $6,515,869$ room photos from Houzz [3]. After removing duplicate and incorrect images (those whose metadata is not compatible with the image), they kept $3,387,555$ product and $6,093,452$ room photos. From such images, they generated their genuine and impostor pairs, which are used for training the siamese CNN.

Figure 2.28: Results from the product search implementation based on siamese networks. (Image from [9]).

In Figure 2.28, the genuine pairs (I_q, I_p) are two views of the same object. I_q (image query) is an image of the object in a given room. I_p is the object image itself. On the other hand, impostor pairs are represented by the (I_q, I_n), where I_q is again the query image and I_n is a negative example (an object of another class). Figure 2.28 shows some results of the embeddings learned by the siamese network. For a given query image I_q, the table shows the three most similar images (shortest distance). Just for reference, the table shows a tag object I_p, which was used along with I_q

[3]Houzz is a platform for home remodeling and design. Available at: http://www.houzz.com/

for training. We can notice the top three images are visually similar to I_pe. Hence, the embedding learned by the siamese network is very representative.

Most siamese networks make use of Contrastive Loss as an activation function. However, other loss functions such as *Triplet Loss* [102] started to be used with siamese networks recently. Instead of pairs as input, triplet loss receives a triplet as input. Therefore, instead of two sibling branches, the siamese network is comprised of three network branches and that is why it is known as *triplet network*. The triplet used as input is composed by two positive (genuine) instances and one negative (impostor) instance. Using triplets as inputs, triplet loss minimizes the distance between an *anchor* and a *positive* (similar to a genuine pair), while maximizing the distance between the same *anchor* and a *negative* (similar to a impostor pair). Figure 2.29 illustrates this idea. Since the data representation is two-dimensional, the intuition is that similar instances are closer in the Euclidean space and different instances are farther.

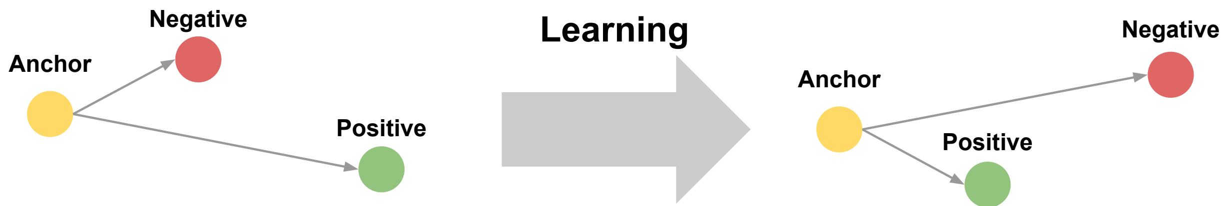


Figure 2.29: Intuition on the triplet loss objects. Through a learning procedure, it minimizes the distance between the anchor and the positive and maximizes the distance between the anchor and the negative objects.(Image adapted from [102]).

A convolutional neural network implementation for face recognition called *FaceNet* is optimized with triplet loss. Therefore, the following explanation will use faces and images as examples. Triplet loss wants to ensure that x_i^a (anchor) of a specific person is closer to all other images x_i^p (positive) of the same person than it is to any image x_i^n (negative) of any other person. The triplet loss function is represented by Equation 2.32. $f(x)$ denotes the embedding (representation vector of face x) and α is a margin that is enforced between negative and positive pairs. Equation 2.33 defines the rule for margin α used in the loss function. It defines that the Euclidean distance between positive pairs embeddings $\|f(x_i^a) - f(x_i^p)\|_2^2$ summed with a given margin m must be smaller than $\|f(x_i^a) - f(x_i^n)\|_2^2$. It is the same rule as in the contrastive loss function in Equation 2.31. Equation 2.33 defines the triplets. τ is the set of all possible triplets in the training set.

$$L = \sum_{i}^{N} \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + m \right] \tag{2.32}$$

$$\|f(x_i^a) - f(x_i^p)\|_2^2 + m < \|f(x_i^a) - f(x_i^n)\|_2^2 \quad , \tag{2.33}$$

$$\forall (f(x_i^a), f(x_i^p), f(x_i^n)) \in \tau \tag{2.34}$$

2.6 Final Remarks

In this chapter we presented the theoretical basis needed for understanding the deep learning approaches used in this dissertation. First, we introduced deep learning, the current state-of-the-art for many machine learning applications. Then, we introduced convolutional neural networks, which are feed-forward artificial neural networks that are able to extract high-level features from raw data. We also presented the application of convolutional neural networks for classification, localization, and for metric learning, explaining in details each of them.

In Chapter 3 we will present an overview of face detection, which comprises some important face detectors. Additionally, we will detail our proposed approach.

3 Face detection

This chapter explains what is face detection and gives an overview of both classic implementations and most recent strategies. It is organized as follows. First, we give a brief introduction of face detection in Section 3.1. Next, in Section 3.2 we present some face detection classic approaches, which are based on hand-crafted features. Then, in Section 3.3 we talk about deep learning approaches for face detection. Finally, in Section 3.5 we explain our meta learning approach and show results of our experiments.

3.1 Introduction

Face detection refers to the task of identifying human faces in images. Given an arbitrary image, the goal of face detection is to determine whether there are any faces in the image and, if present, return the image location and the extent of each face [127]. Face detection is easily performed by humans, but it is still a challenge within Computer Vision. The high degree of variability and the dynamicity of the human face makes it very difficult to detect, mainly in complex environments. Most of the existing complexity is due to occlusion (i.e. face is partially visible), illumination conditions (e.g. darker and lighter environments), background clutter (e.g. background and object are too similar in color intensities), and so on.

Currently, face detection is one of the most studied topics in the Computer Vision literature [130]. The countless applications that make use of face detection contribute to such popularity. Face detection is used in several applications in areas such as content-based image retrieval, video coding, video conferencing, and crowd surveillance [45]. Many of the applications use face detection algorithms as a first step. For instance, facial emotion recognition evaluates the emotion of a person among a set of available emotions [8]. Before recognizing the emotion, the algorithm needs to detect the face of the person. Another example is Facebook, which uses face detection as a prior step to auto-tag friends on photos.

During the nineties, several implementations of face detection emerged. However, the majority of them were not able to yield good results in unconstrained conditions [130]. Unconstrained face detection — also known as face detection *in-the-wild* — refers to recognizing faces in images

in real-world conditions [47]. For instance, variations in pose, lighting, expression, race, ethnicity, age, gender, clothing, hairstyles are unconstrained conditions. The Viola-Jones face detection implementation was one of the major breakthroughs because it allowed face detection to be feasible in real-world applications [120]. Most of the current digital cameras still use The Viola-Jones face detector [130]. We will explain how it works in Section 3.2.1.

Large datasets have become available for face detection. A recent and very challenging dataset is WIDER FACE [128]. It is a face detection benchmark dataset, composed of $32,203$ images and $393,703$ labeled faces. The images are organized into 61 different classes (60 classes plus 1 for background) that represent events, such as *Ceremony*, *Dancing*, and *Football*. For each event class, the authors randomly selected 40% of data as training, 10% as validation and 50% as testing. The dataset is challenging mainly due to the high degree of variability in scale, pose, and occlusion that the faces present. Figure 3.1 illustrates some scenarios. All images of WIDER FACE are a subset of the well-known Web Image Dataset for Event Recognition (WIDER) dataset [125]. Other notable datasets are Face Detection Data Set and Benchmark (FDDB) [50] and IARPA Janus Benchmark A dataset (IJB-A) [59]. FDDB contains $2,845$ images with $5,171$ labeled faces and all images are from the Faces in the Wild Dataset [10]. On the other hand, IJB-A is a dataset that contains both images and videos from 500 different subjects. In total, the dataset has $5,712$ images and $2,085$ videos in a variety of positions. These datasets introduced not only a great quantity of faces for training the algorithms, but also standard benchmarks for the evaluation of face detection algorithms. Back in 2000, the lack of standard evaluation methods did not allow a fair comparison between existing face detectors [45], which is now possible due to such datasets.

Figure 3.1: Examples of images of WIDER FACE dataset.

The large amount of images in face detection datasets allowed deep learning algorithms to learn representative features for detecting faces and outperform Viola-Jones and other previous implementations. In the following sections, we will explore some implementation of face detectors based on hand-crafted and deep learning approaches. We will explain how they work and execute experiments to evaluate them in different datasets. Finally, we will detail our meta learning approach for recommending the best face detector per image.

3.2 Related Work Based on Hand-Crafted Features

3.2.1 Open CV (Viola-Jones)

Open Source Computer Vision (OpenCV)[1] is an open source computer vision and machine learning library that contains several functions for computer vision tasks. As it is mostly written in C and C++, it enables fast real-time computer vision and is used in many different applications (it also contains Python adapters). Among its functionalities, OpenCV contains two face detection methods: the Haar Feature-based Cascade Classifiers [120], and Local Binary Patterns (LBP) [71]. OpenCV contains methods for training classifiers from scratch as well as methods that allow one to use trained classifiers. In the next sections, we will explain how each of these classifiers work.

3.2.1.1 Haar Feature-based Cascade Classifiers (Viola-Jones Face Detector)

Haar Feature-based Cascade Classifiers is an effective method for object recognition, which is based on machine learning [120]. It allows one to create an object recognition system (i.e. recognizing cars, planes, people) using simply a set of positive and a set of negative examples. For instance, in order to train a classifier to detect faces, one must provide a set of training images, which is comprised of positive images and negative images. Positive images are the ones that contain a face while the negative images are the ones without faces.

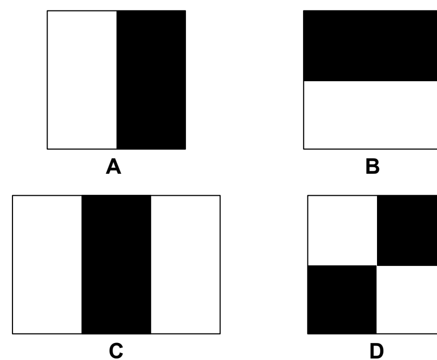


Figure 3.2: Example of Haar-like features used by Haar Feature-based Cascade Classifiers. *A* and *B* show two-rectangle features. *C* shows a three-rectangle feature whereas *D* shows a four-rectangle feature [120].

During the training phase, the algorithm extracts Haar-like features from the images rather than using the pixels intensities. Haar-like features are based on Haar wavelets, which is a set of functions that encode differences in average intensities between regions [83]. These features are very similar to features extracted by convolutional kernels (refer to Chapter 2, for more details on convolutions), which are used to detect whether the specific feature is present on a given image.

[1] Available at http://opencv.org/

Figure 3.2 contains examples of such features. Besides the ones shown in Figure 3.2, there are other Haar-like features, which are generated through the rotation of the existing ones [72].

In order to calculate the features, the sum of the pixels within the white rectangles is subtracted from the sum of pixels in the black rectangles, which results in a single value. Nevertheless, to calculate such features in a whole image is computationally expensive, since there are about $180,000$ different features in a 24×24 detector, for example. Therefore, instead of using a regular image the Haar Feature-based Cascade Classifiers use an intermediate representation of the image called integral image (Figure 3.3). Given an integral image at (x, y), it contains the sum of all the pixels above and to the left of (x, y) inclusive, which can be computed in a single pass over the original image [120]. It allows the algorithm to execute only once the most computationally expensive portion and re-use the calculated values (integral image) to extract different Haar-like features.

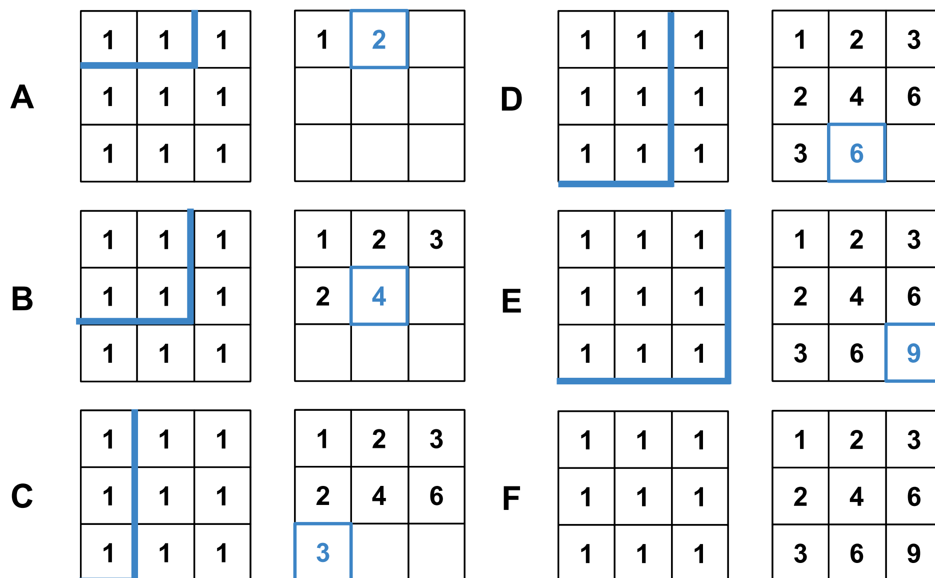


Figure 3.3: To calculate the integral image, we can imagine that a blue line is drawn in a given region and all pixes above and to the left of that region are summed up. For instance, *A*, *B*, *C*, *D*, and *E* illustrate different portions being calculated. *F* has the resulting integral image.

The next step of the algorithm is to select the best features to be used in the classifier. For such a task, it is used a slightly modified version of AdaBoost [33]. AdaBoost is an algorithm that boosts the classification of a simple (also called weak) learning algorithm. The modification makes AdaBoost return a weak classifier that depends only on a single feature, acting like a feature selector. Therefore, each of the resulting weak classifiers are combined to build a stronger one (ensemble learning).

In training and in test time, each image is sampled in sub-windows. For each of these sub-windows, all the Haar-like filters are applied, which leads to a considerable computational cost. In order to solve this issue, the algorithm uses cascade classifiers. The cascade classifiers are comprised of different stages, each of which have specific Haar-like features being extracted (Figure 3.4). As

each image is sampled in sub-windows, each stage works in a way that allows it to discard sub-windows that do not have the face before going to the next stage, avoiding unneeded computation.

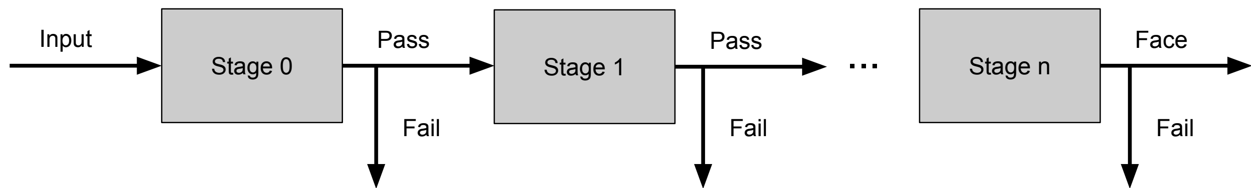


Figure 3.4: Stages of cascade classifier. Each stage contains specific Haar-like feature extractors. The input is considered a face only if it passes through all stages. It is not considered a face if the verification fails in any of the stages.

3.2.1.2 Local Binary Patterns

Local Binary Patterns (LBP) is a simple but efficient feature extractor [82]. It is one of the best performing texture descriptors and has been used for different applications such as face detection, face recognition, facial expression analysis, and etc. [4]. LBP works in a 3×3 neighborhood, using the center pixel value as a threshold [87]. Therefore, it basically extracts features of local structures through the comparison of a given pixel with its neighboring pixels, thresholding the values of each pixel based on the given pixel (Figure 3.5). If the value of the pixel being compared is larger or equal to the selected pixel, it is thresholded to 1, whereas if the value is smaller it is thresholded to 0. As there are 8 neighbor pixels, a total of $2^8 = 256$ texture units is possible.

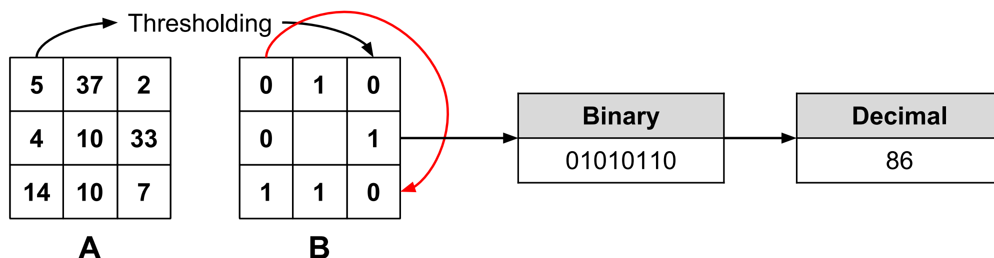


Figure 3.5: LBP selects a given center pixel (10, in this case) and compares it to all the neighbor pixels. If their value is larger or equal to the center pixel, it is assigned the value 1. Otherwise, it is assigned to 0. The red arrow shows how the binary string representation is created. Afterwards, these binary representation is converted to a decimal number, which results in the feature extracted by LBP. Figure *A* is the original pixel matrix while Figure *B* is the thresholded version of it.

In the beginning, LBP was introduced as a texture descriptor. However, it started to be used for feature extraction and classification due to two main advantages: the tolerance related to monotonic illumination changes, and the computational efficiency [4]. Being robust to monotonic illumination changes means that if all pixels in a given area suffer a small change due to illumination conditions (for instance, all pixels intensities increase or decrease), LBP would still be able to extract

the same features (see Figure 3.6). Therefore, a small invariance would not result in a different feature extracted. However, it is important to mention that if a different illumination is present on part of the image, for instance a shadow over a specific portion of an object, only a region of pixels will be darkened, and hence it will have a different binary representation.

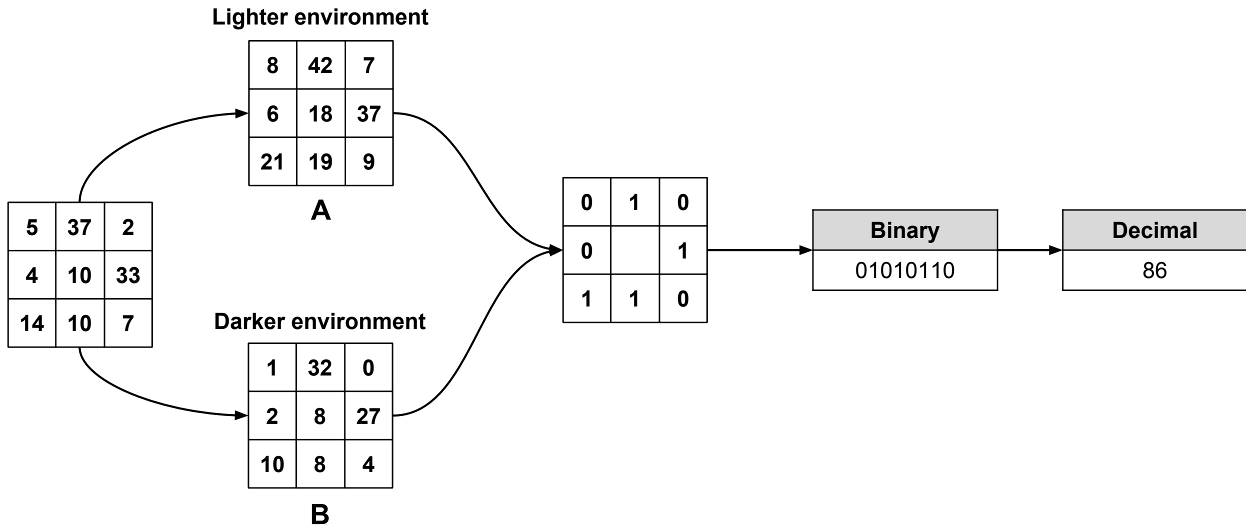


Figure 3.6: *A* and *B* show a possible representation of the left pixel matrix in a lighter and a darker environment, respectively. We can notice that both output the same binary representation in the end, which illustrates an interesting property of LBP.

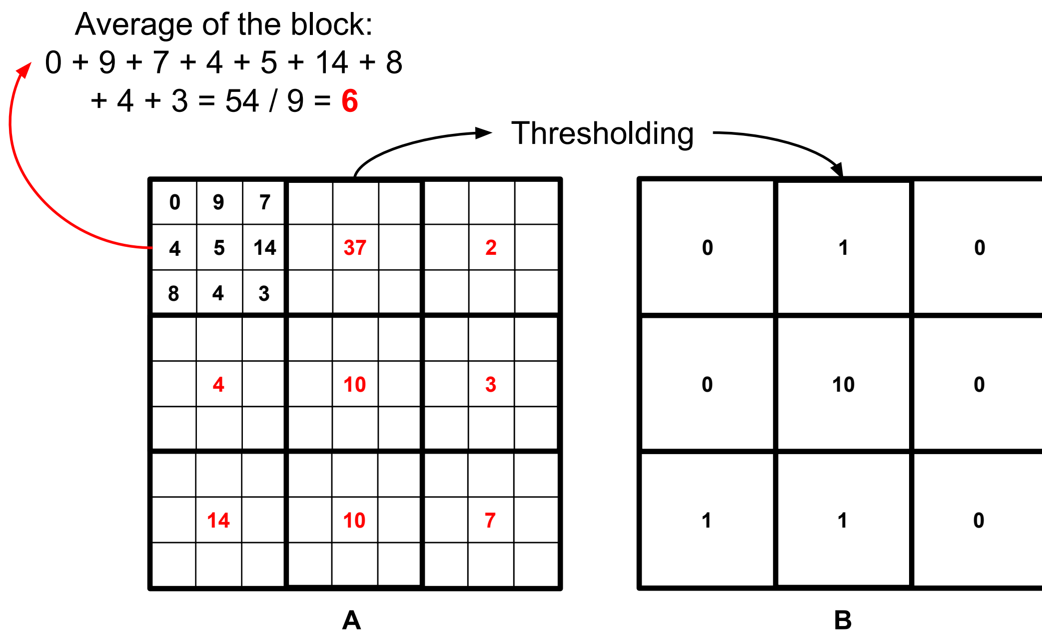


Figure 3.7: 9×9 MB-LBP.*A* illustrates how the image is divided into blocks and the mean of each block. *B* shows the thresholded values. The top left part shows how the average of pixels' values is calculated, by summing up all pixels and dividing by the sub-region size.

The potential of LBP led to the development of a face descriptor based on it. Nevertheless, the original LBP is not good enough to capture local information needed for face description, so a

method able to capture such information was implemented [5]. In this method, the face is divided into local regions and the texture descriptors are captured from each location. The descriptors (in form of histograms) are then combined to form a single face descriptor.

Even though methods using LBP for face recognition evolved, they were still not able to capture larger scale structure. To overcome such limitation, Multi-scale Block Local Binary Pattern (MB-LBP) was proposed [71], which is the one used in OpenCV. Instead of computing in a pixel level, MB-LBP is calculated based on the average values of sub-regions. Figure 3.7 shows an example of a 9×9 MB-LBP. It basically divides the region into 9 blocks, and in which the average of pixels is calculated (represented by the red value). Then, the average of each sub-region is used to threshold the values between 0 and 1 in the same manner as the regular LBP. AdaBoost is also used to select the best region histogram features and construct a face classifier [131]. Therefore, as it happens for Haar classification, OpenCV's implementation of LBP also uses a cascade classifier for face classification. It means the pipeline followed by Haar is the same for LBP: each stage of the cascade classifier works in a way that allows it to discard sub-windows that do not have the face before going to the next stage, avoiding unneeded computation.

3.2.2 Dlib

Dlib[2] is an open source C++ library that contains machine learning algorithms and tools to solve complex problems. It comprises a collection of different software components and it is intended to both research and commercial use [58]. Among the different functions it offers, we focused our attention on the tools for detecting objects in images, which includes a frontal face detection algorithm. The main face detection algorithm in Dlib is based on Histograms of Oriented Gradients (HOG).

Histograms of Oriented Gradients (HOG) is a dense feature descriptor that is used to detect objects in images. Dense means that it extracts features for all locations over an image or a region of interest instead of computing over local neighborhood pixels. It provides great performance with respect to other object feature descriptors and it was first developed with the goal of identifying pedestrians in images, which was not well achieved by the existing methods. The Haar feature-based method, explained in Section 3.2.1.1, was one among such methods [19]. HOG is strongly based on the intuition that the distribution of intensity gradients and edge directions can describe an object's shape and appearance in an image. It means the feature descriptor should be able to identify people, for example, capturing the main edges and forms that distinguish people from other objects. An image gradient is calculated by the computation of how much the intensity values change to a given direction within a group of pixels. It denotes a directional change in the intensity of an image. The magnitude of the gradient tells how quickly the image is changing whereas the direction of

[2]Available at: http://dlib.net/

the gradient tells the direction in which the image is changing most rapidly. Figure 3.8 shows an example of the gradients of a HOG face detection descriptor and the detection result.

Figure 3.8: On the left, the gradients orientation of HOG face detection descriptor. It is important to notice that the gradients directions result in a face shape, which helps to understand the intuition behind HOG and why it works. On the right, the face identified by the given descriptor.

The use of HOG has many precursors, but it only started to be more used along with Scale Invariant Feature Transformation (SIFT) [77]. SIFT is an image feature descriptor that is invariant to translation, scaling, and rotation. It basically transforms the image into a set of local feature vectors, each of which with such invariance properties. The classic HOG implementation is done as follows. After dividing the image into sub-regions called cells (or blocks), the first step is to compute the gradients before-mentioned. They are calculated by convolving an image with a given kernel, such as the Sobel Filter [106]. The second step relies on the creation of cell histograms, which is achieved by accumulating the calculated gradients into a histogram. Finally, the cell histograms are combined to form a single histogram, which represents the image. The main idea is that the cell histogram will not change much if the cell moves slightly.

A more detailed explanation about each of these steps is available in [19, 14]. The good results obtained with object recognition resulted into the use of HOG for face detection [22].

3.3 Related Work Based on Deep Learning

3.3.1 Faster R-CNN

Faster R-CNN [93] is a deep neural network that was created to address the object detection problem. Nevertheless, the way it works can be adapted to handle face detection as well. Our proposal is therefore to train Faster R-CNN with images that contain faces and use it as a face detector. To the best of our knowledge, it is the first time Faster R-CNN is used as a face detector. However, while developing this work, we found an unpublished work that presents a similar idea [52]. In order to use Faster R-CNN as a face detector, one must choose a face detection dataset and train the network with it. For our case, we choose WIDER FACE [128] for training. Faster R-CNN has

two different implementations available: one that is based on Matlab [3] and another in Python [4]. As our implementations are Python-based, we chose to use the Python one.

Faster R-CNN implementation contains scripts that enable one to use pre-trained models and fine-tune the network on an object detection dataset. It contains scripts for fine-tuning on PASCAL VOC and MS COCO. We changed the library to make it possible to train the network on the WIDER FACE dataset. Usually, Faster R-CNN for object detection trained on a dataset with K classes contains $K + 1$ classes, where this extra class is used as *background*. In our case, we trained our model with only 2 different classes, where one is the *background* and the other is *face*. We consider that our target is to optimize the bounding box loss instead of the classification loss and hence, we used only these 2 classes instead of all WIDER FACE classes. We used WIDER FACE default training and validation sets. We detail our experiments in Section 3.4, as well as the different models and hyper-parameters we tested.

3.3.2 MTCNN

MTCNN is a deep learning approach that aims to solve the face detection and alignment problem. Differently from Faster R-CNN, MTCNN is already trained on face detection datasets. The authors did not specify the training process, but data is carefully selected as to allow the three networks to learn. The whole network is trained with four different types of data. First, *negative* instances, which contain less than 0.3 IoU to any ground truth face. Second, *positive* instances, which contain more than 0.65 to a ground truth face. Third, *part face* instances, which contain an IoU between 0.4 and 0.65 to a ground truth face. Finally, the fourth are *landmark faces* instances, which are faces labeled with five landmark positions. The IoU gap between negative and part faces (0.3 to 0.4) is used to avoid confusion between the two types, mainly because they are very similar [132]. *Negative* and *positive* instances are used for face classification tasks, whereas bounding box regression uses *positive* and *part faces* instances. *Landmark faces* are used for facial landmark localization. MTCNN has different implementations available. The original one is in Matlab [5]. There are other implementations in C++ and Python. We chose the Python implementation [6].

3.4 Individual Experiments

In order to evaluate each face detector individually, in the following sections we execute the face detectors on a given set of images. For all experiments, we used the validation set from the WIDER FACE dataset. The validation set is comprised of $3,226$ images divided into 61 classes. However, for our evaluation we picked up 10% of each, which resulted in 326 images. Figure 3.9

[3] Available at: https://github.com/ShaoqingRen/faster_rcnn
[4] Available at: https://github.com/rbgirshick/py-faster-rcnn
[5] Available at: https://github.com/kpzhang93/MTCNN_face_detection_alignment
[6] Available at: https://github.com/DuinoDu/mtcnn

shows the distribution of images per class. We can notice that the classes are distributed almost equally. Our idea is to use such a dataset to evaluate face detectors individually due to the its diversity between the classes. For instance, the dataset contains images from crowded places as well as images that contain very few people. Figure 3.10 shows some examples. We can notice in the images that there is a considerable variety of poses and environments. This is why such dataset is very challenging.

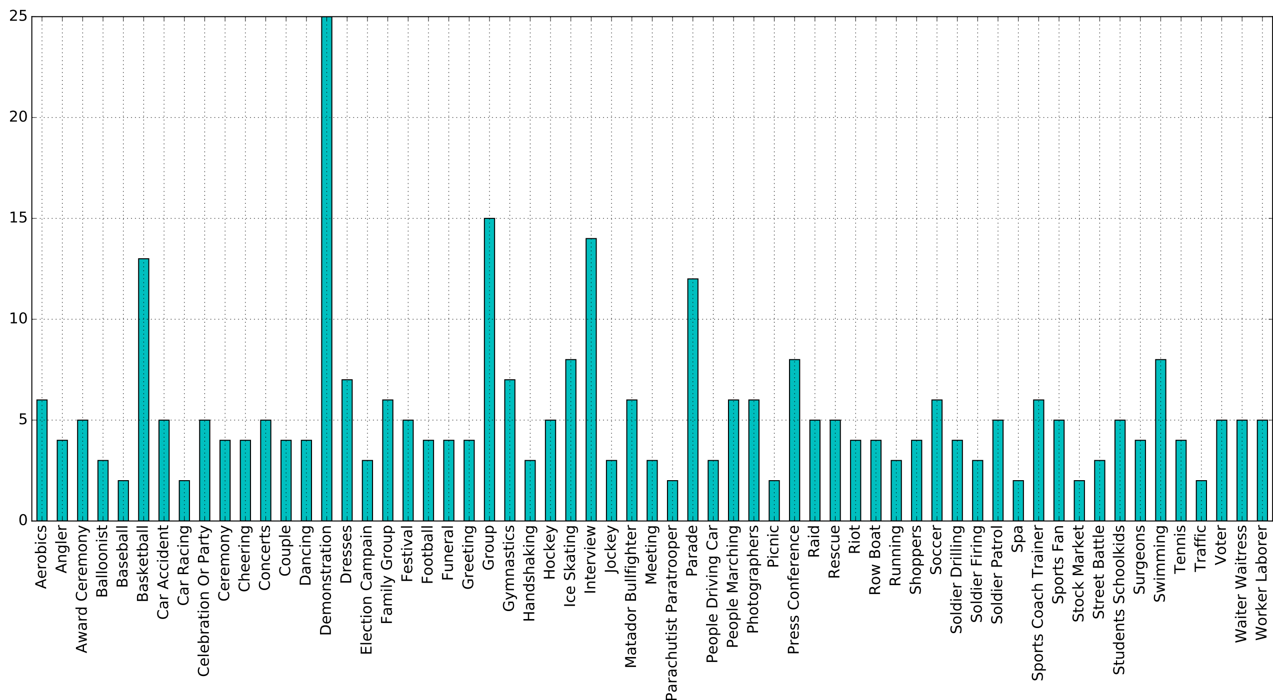


Figure 3.9: Images used for face detection tests. It is a subset of WIDER FACE validation set.

For the evaluation of each face detector, we followed an approach similar to [24]. It is a modified version of the scheme used in PASCAL object detection challenges. For a given image, each face detector should return N bounding boxes along with their confidence scores, which indicate the confidence of the detector in the detected faces. For each detected bounding box, we calculate the Intersection over Union (IoU) of the given bounding box with respect to the ground truth. The IoU score is then used for assigning a detection to a given bounding box and it works as follows. After executing a face detector on a given image, the detections are ordered in a decreasing fashion, from highest to lowest confidence. The second part is to assign the detections to ground truths. For a given ground truth and a given detection, the IoU is computed. If the IoU is larger than a threshold, the detection is considered correct (a *true positive*), the detected face is removed from the detection list and the routine picks up the next ground truth. Usually, the threshold used is 0.5, which means that the ground truth and the detection should have an overlap that is larger than 50% [24]. This process repeats until all ground truths were processed. The remaining ground truths — the ones that were not assigned to any detection — are considered *false negatives*. The detected faces that were not assigned to any ground truth are considered *false positives*. The evaluation process just described is formally defined by Algorithm 3.1. Such algorithm has three input parameters for a

given image: a list with the detected faces (ordered by confidence scores), a list with the ground truths, and the IoU threshold. It yields *true positives*, *false positives*, and *false negatives*.

Figure 3.10: Some examples of images available in the validation set.

Algorithm 3.1: Algorithm for evaluating detected faces versus ground truth faces.

1: **function** $evaluate_detection\,(detected_faces, gt_faces, iou_threshold)$
2: $true_positives \leftarrow 0$
3: $false_positives \leftarrow 0$
4: $false_negatives \leftarrow 0$
5: **for** each ground truth face gt_f in gt_faces **do**
6:     **for** each detected face f in $detected_faces$ **do**
7:         $iou \leftarrow compute_iou\,(f, gt_f)$
8:         **if** $iou > iou_threshold$ **then**
9:             $true_positives \leftarrow true_positives + 1$
10:             $detected_faces.remove\,(f)$
11:             **break**
12:         **end if**
13:     **end for**
14: **end for**
15: $false_negatives \leftarrow length\,(gt_faces) - true_positives$
16: $false_positives \leftarrow length\,(detected_faces)$
17: **return** $true_positives, false_positives, false_negatives$

Next, after we obtained the number of true positives and true negatives, we can finally compute metrics that enable us to evaluate how good our detection system is. For our experiments, we compute three different measures: *Precision*, *Recall* and *F1 measure*. Precision, given by Equation 3.1, determines the fraction of the detected faces that are actually faces, penalizing the detector for incorrect detections. Recall, given by Equation 3.2, is the proportion of positive cases that are properly identified. In our case, it is the fraction of ground truth faces that were successfully detected. It means that if the detector does not find a large portion of the existing faces, recall value will decrease. With precision and recall we can finally compute the F1 measure (Equation 3.3). F1 represents the harmonic mean between recall and precision. Therefore, a high value of F1 ensures that precision and recall are reasonably high [114]. In our experiments, we use the F1 measure to compare each face detector and evalute which one is better.

$$Precision = \frac{tp}{tp + fp} \tag{3.1}$$

$$Recall = \frac{tp}{tp + fn} \tag{3.2}$$

$$F1 = 2 * \left(\frac{Precision * Recall}{Precision + Recall} \right) \tag{3.3}$$

As in previous sections, we divide the experiments in detection using hand-crafted and deep learning approaches.

3.4.1 Hand-Crafted Approaches

Hand-crafted approaches are those based on hand-crafted feature extraction. Hence, the following sections contain experiments with OpenCV and Dlib.

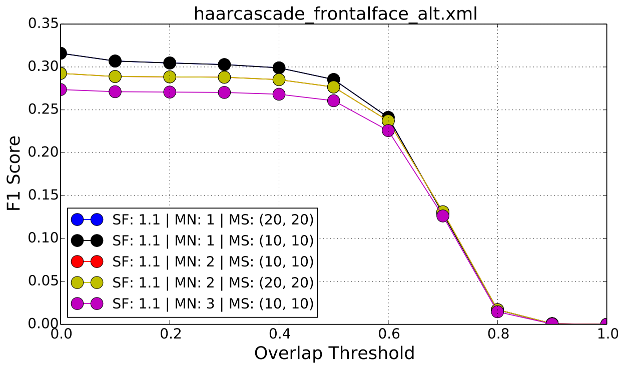
3.4.1.1 OpenCV

Haar-based and LBP-based feature classifiers work similarly, even though LBP is faster in the cascade classifier training time. The face detector in OpenCV contains four main parameters: *Cascade File*, *Scale Factor*, *Minimum Neighbors*, and *Minimum size*. The first parameter is related to the trained cascade file, which is used for classification. For instance, there are files based on Haar features and others based on LBP features. There are also files that are specific to detect eyes, nose, or the face itself. *Scale Factor* is related to how much the image size is reduced at each image scale. It means that one can find smaller faces by scaling larger ones, making the face detectable by the algorithm. *Minimum Neighbors* refers to how many neighbors each candidate rectangle should have to retain. If this parameter has a small value, the algorithm will be susceptible to false positives. It means that larger values detect less but high-quality faces whereas small values do the opposite. Finally, *Minimum Size* is related to the minimum object size. Any object (face) found that is smaller than this parameter will be ignored.

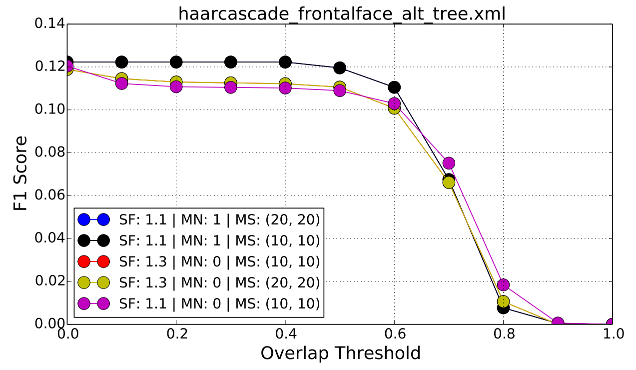
We executed a considerable amount of possible combinations of parameters in OpenCV. The values and parameters used are the following: 1.1, 1.3, 1.5, 1.7, 2.0 for Scale Factor; 0, 1, 2, 3 as Minimum Neighbors; $(10, 10)$, $(20, 20)$, $(30, 30)$ as Minimum Size; and all 7 existing pre-trained cascade files (Haar-based and LBP-based cascade classifiers). Therefore, we executed 420 different executions, each of which containing variations of the before-mentioned parameters. Since face detection executions are deterministic, we executed all experiments once and selected some of the most interesting cases to show. We computed the F1 measure over all selected images (10% of the WIDER FACE validation set), with all 420 different parameters, and with different IoU thresholds. We generated plots as to give a better idea of the impact of changing parameters in OpenCV's face detectors. The results were organized in the following manner: for each of the cascade classifiers (7 in total), we selected the Top-5 and Worst-5 results. All plots have IoU thresholds (overlap threshold) and F1 measure values in the x and y axes, respectively.

Figure 3.11 shows best OpenCV parameters for each different pre-trained cascade classifier. Some results are straightforward: for instance, both profile face detectors in Figure 3.11(e) and Figure 3.11(g) have the worst results overall. The nature of the images on the dataset could be responsible for such results, as the majority of them do not contain profile faces. Another interesting point is that Haar classifiers outperform LBP overall. We can also see that the best value for scale factor is 1.1. In all plots, the best results (first and second, at least) use such value. The two best results overall are in Figure 3.11(a) and Figure 3.11(c). Both are Haar classifiers, and the top-5 are configured with scale factor of 1.1, minimum size of $(10, 10)$ or $(20, 20)$, and minimum neighbors of 1, 2, or 3. All F1 scores are superiorly limited by 0.35. On the other hand, Figure 3.12 shows worst OpenCV parameters for each different pre-trained cascade classifier. All F1 measure values are bounded by 0.12. Comparing both Top-5 and Worst-5, we can see that some parameters make the detections decrease drastically. For instance, when the scale factor is 2.0 or 1.7 and the minimum neighbors is 2 or 3, the detectors yield the worst results. We can see that the best results are still based on Haar classifiers and frontal face detectors. A comparison with the Top-10 and the Worst-10 results in a single plot are presented in Figure 3.14. It contains all the information regarding parameters like the other plots, along with a new value *C*, which represents the cascade file. For instance, *C: (a)* represents the pre-treined cascade classifier filename in Figure 3.11(a), which is *haarcascade_frontalface_alt.xml*.

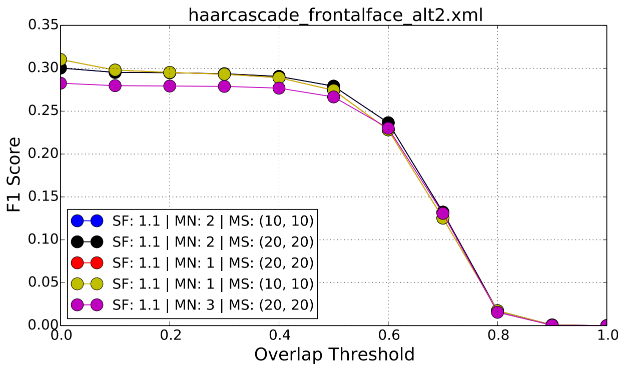
We can conclude that the best performance of OpenCV face detectors is achieved with the following parameters: scale factor 1.1; minimum neighbors of 1 or 2; and minimum size of $(10, 10)$, $(20, 20)$, or $(30, 30)$. Such values corroborate with the results of the individual plots analysis. On the contrary, we can conclude that any face detector based on *lpbcascade_profileface_alt.xml* and with a scale factor different than 1.1 yields the worst results. Figure 3.14 shows all the 420 variations of OpenCV parameters. We can see that it is crucial to evaluate the parameters in order to get the best result from the detectors. Figure 3.13 shows an example of different parameters on the same image. In *A*, scale factor is 2.0, minimum neighbors is 2, and minimum size is $(20, 20)$. In *B*, scale factor is 1.1, minimum neighbors is 2, and minimum size is $(20, 20)$. We can see that by only changing the scale factor, the impact is large. Green bounding boxes represent the ground truth whereas red are the predicted ones.



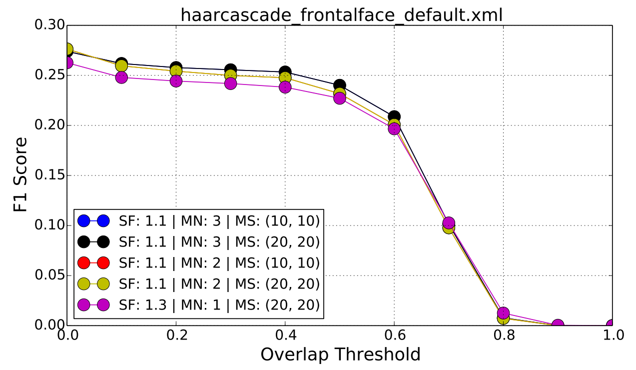
(a) Haar frontal face alt



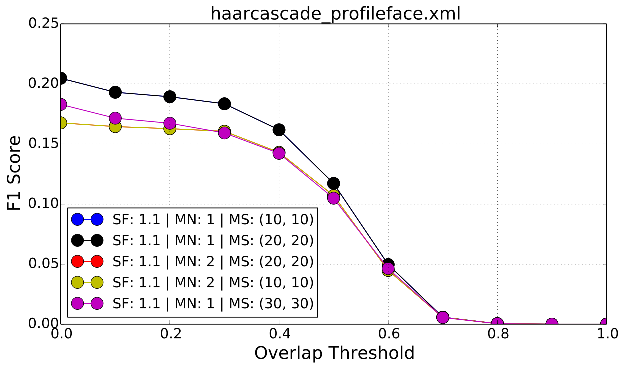
(b) Haar fontal face alt tree



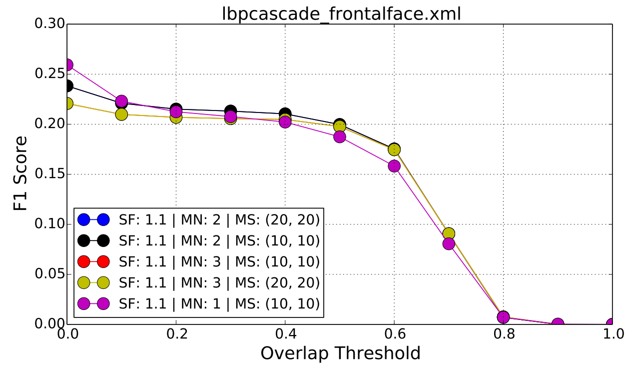
(c) Haar frontal face alt 2



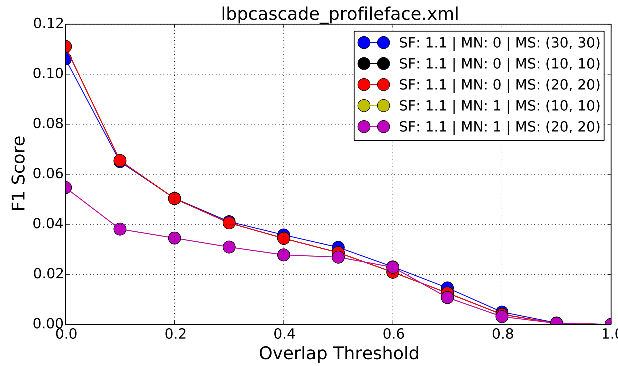
(d) Haar frontal face default



(e) Haar profile face

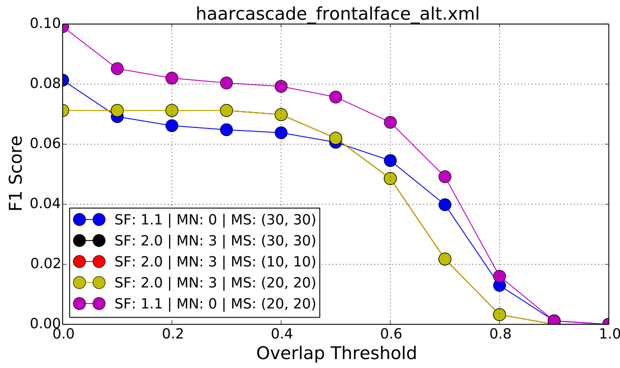


(f) LBP frontal face

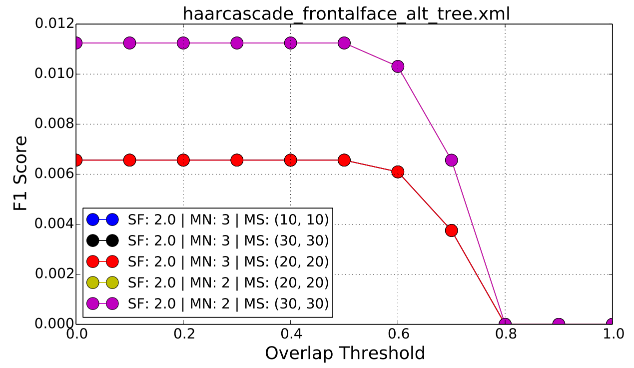


(g) LBP profile face

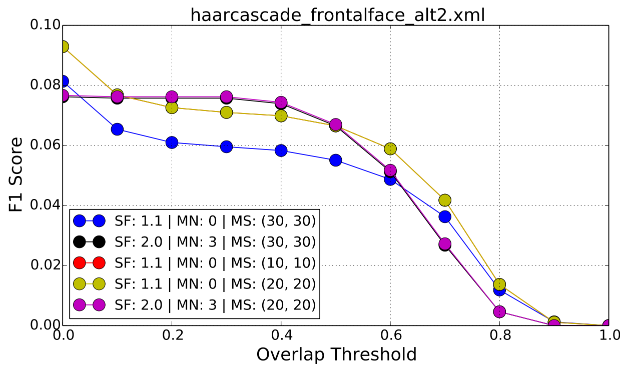
Figure 3.11: Top-5 results for each cascade file. *SF* is the Scale Factor, *MN* is the Minimum Neighbors parameter, and *MS* is the Minimum Size parameter. On the top of each plot, we can see the name of the cascade file. Colors do not represent any particular order.



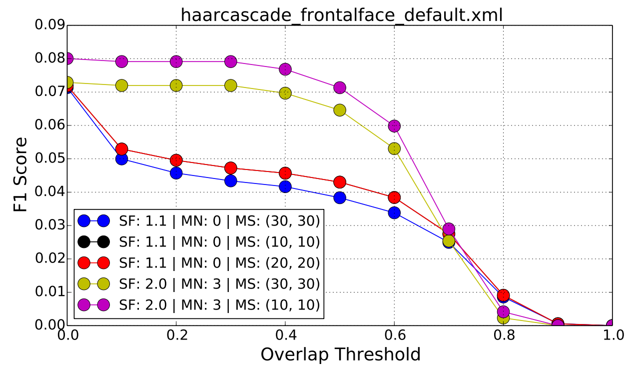
(a) Haar frontal face alt



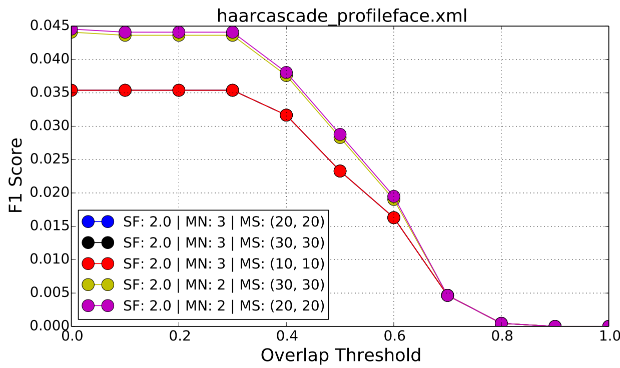
(b) Haar fontal face alt tree



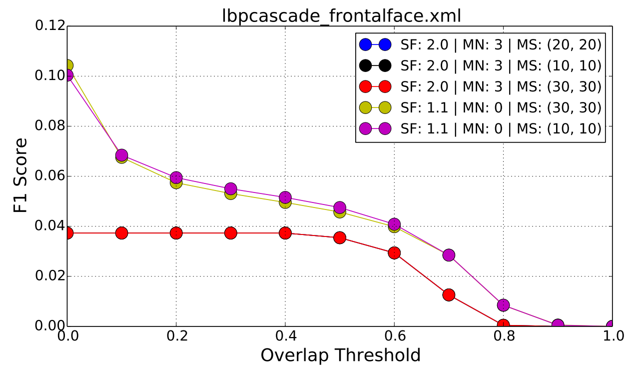
(c) Haar frontal face alt 2



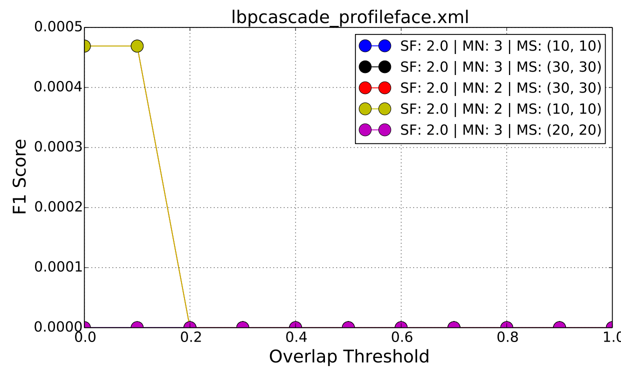
(d) Haar frontal face default



(e) Haar profile face



(f) LBP frontal face



(g) LBP profile face

Figure 3.12: Worst-5 results for each cascade file. *SF* is the Scale Factor, *MN* is the Minimum Neighbors parameter, and *MS* is the Minimum Size parameter. On the top of each plot, we can see the name of the cascade file. Colors do not represent any particular order.

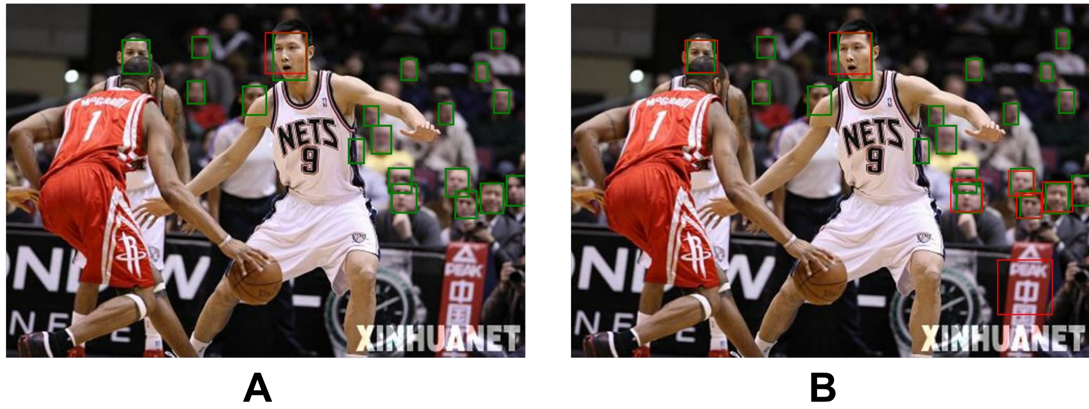


Figure 3.13: Examples of changing parameters in OpenCV in an input image. Green boxes are ground truth whereas red are detections.

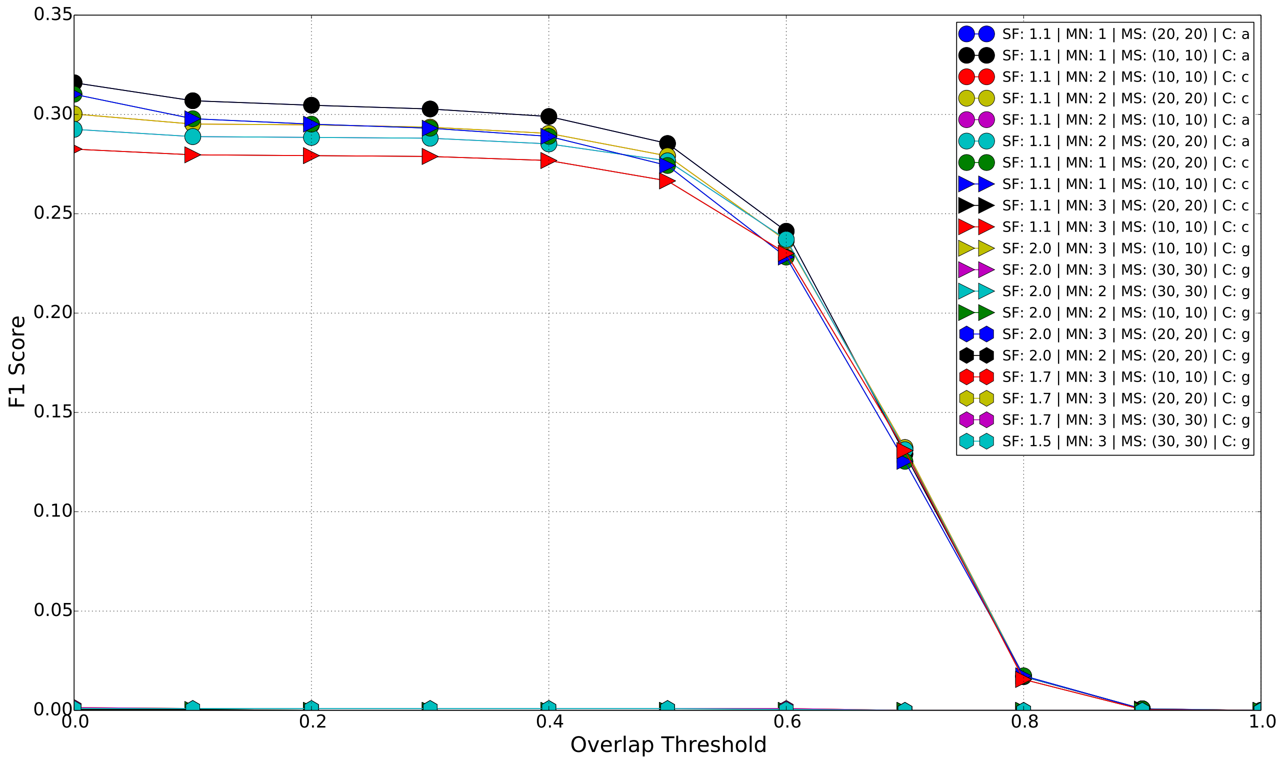


Figure 3.14: Top-10 and Worst-10. The first 10 labels in the legend are the 10 best parameters whereas the next 10 are the worst.

OpenCV is a well known library for Computer Vision, however we did not find any specification with the best parameters for its face detectors. For our experiments with WIDER FACE images, the best parameters are Scale Factor 1.1, Minimum Neighbors of 1, and Minimum Size of $(10, 10)$. Nevertheless, we calculated the F1 score based on the whole dataset. It means that we computed all true positives, false negatives, and false positives for each, summed all of them and calculate the F1 score based on such values. One might argue that we should calculate the F1 score on an image basis, sum all F1 scores and divide by the total number of images. In order to evaluate the difference, we did exactly that: instead of calculating the F1 score only once based on all true

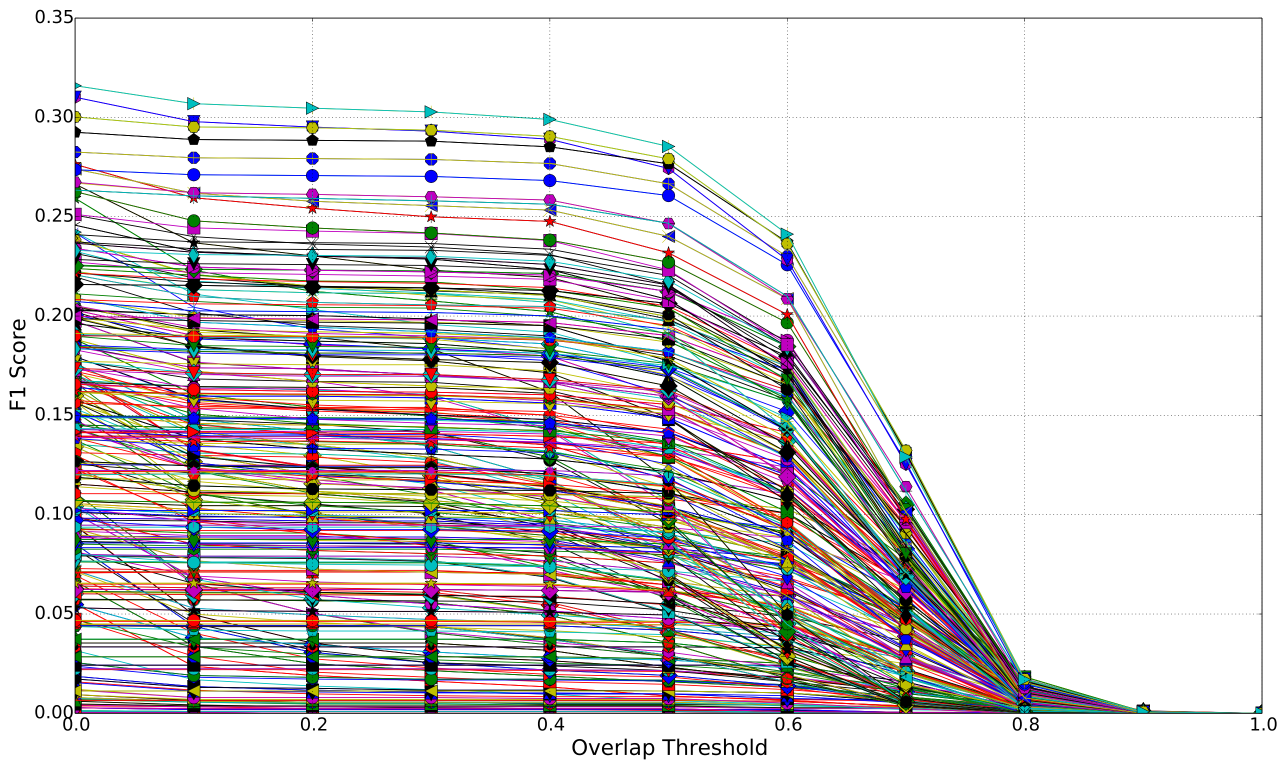
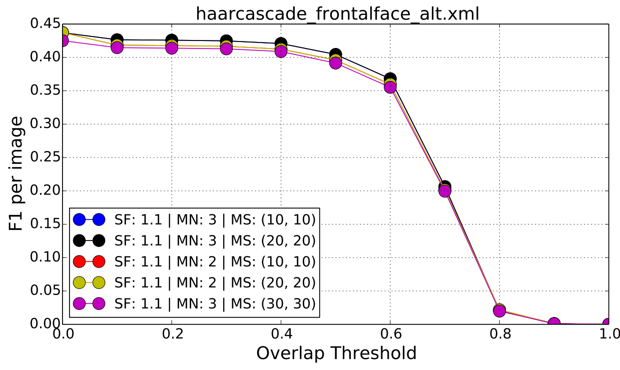


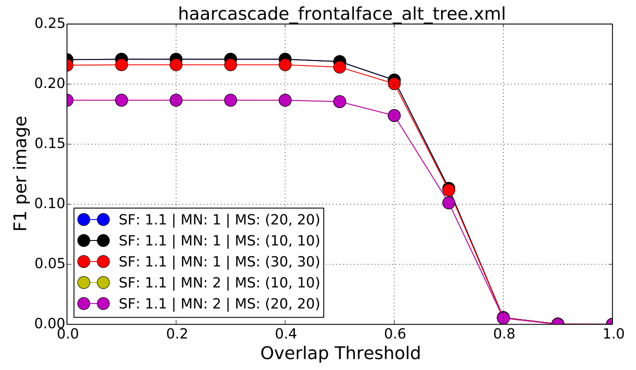
Figure 3.15: All 420 variations of OpenCV parameters that we used in our experiments.

positives, false negatives, and false positives, we calculated the F1 score for each image and divided by the total of images. The results can be seen in Figure 3.16.

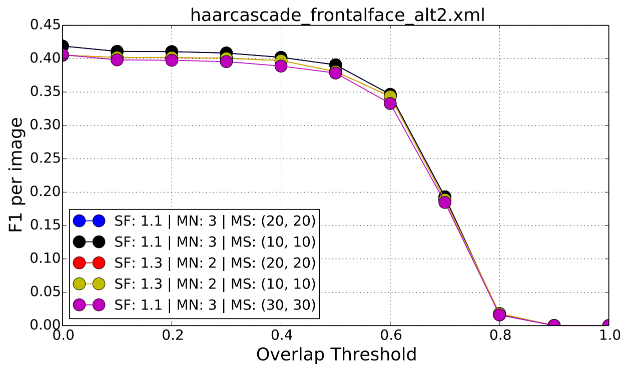
Initially, the results seem straightforward. All the pre-trained classifiers that were better when computing the overall F1 score continue to be the best when computing the same F1 score on an image basis. However, when we look closely to the results, we can see that the best parameters for each cascade classifier change slightly. For instance, let us compare the best cascade classifier *haarcascade_frontalface_alt.xml*. In Figure 3.11(a), we can see that the best parameters are Scale Factor 1.1, Minimum Neighbors of 1, and Minimum Size of $(10, 10)$. It achieves an score of almost 0.3 when the overlap threshold is 0.5. In Figure 3.16(a), the best parameters are also Scale Factor 1.1 and Minimum Size as $(10, 10)$, however the Minimum Neighbors parameter is 3. It results in an F1 score of about 0.43 for the same IoU threshold of 0.5. Therefore, results indicate that when comparing image by image, it is better to use a higher value for Minimum Neighbors.



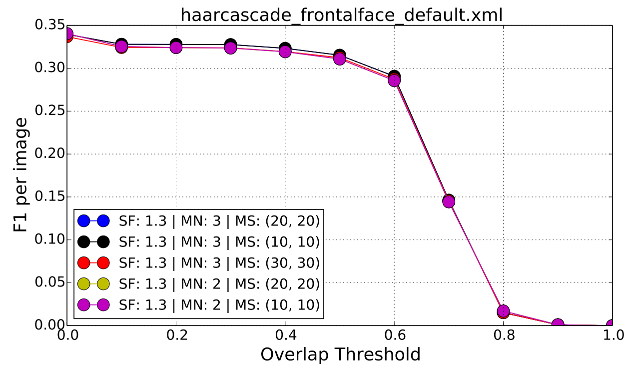
(a) Haar frontal face alt



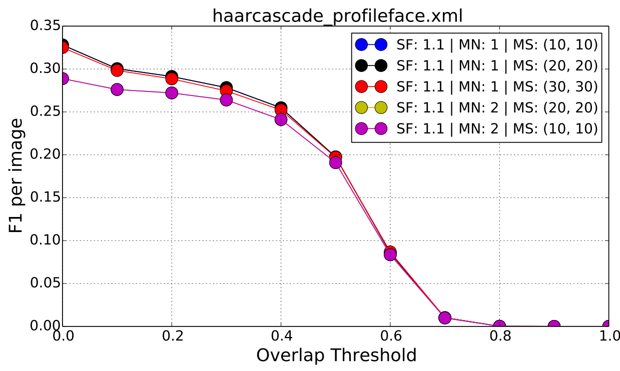
(b) Haar fontal face alt tree



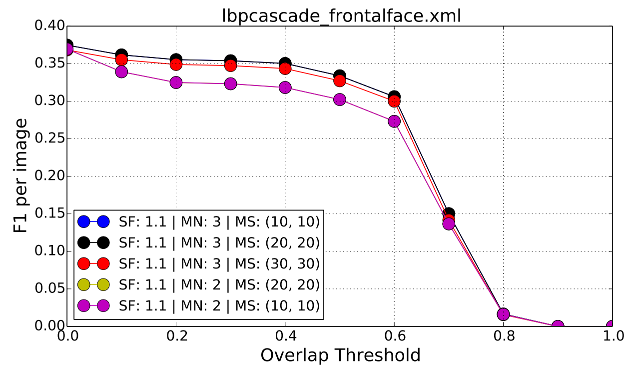
(c) Haar frontal face alt 2



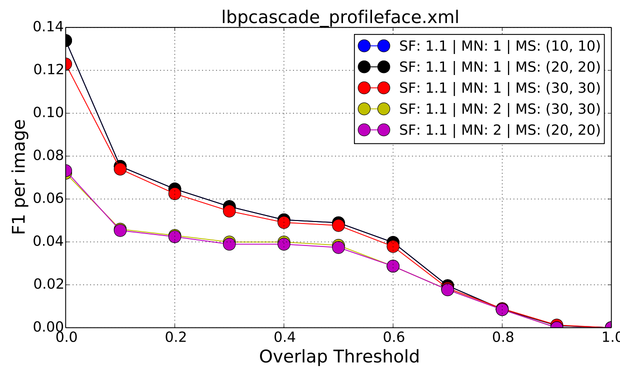
(d) Haar frontal face default



(e) Haar profile face



(f) LBP frontal face



(g) LBP profile face

Figure 3.16: Top-5 results for each cascade file based on per-image F1 score. *SF* is the Scale Factor, *MN* is the Minimum Neighbors parameter, and *MS* is the Minimum Size parameter. On the top of each plot, we can see the name of the cascade file. Colors do not represent any particular order.

We can conclude that the F1 score per image seems to be a reasonable choice when compared with the overall F1 score. We believe that the overall score tends to penalize the best results. For example, let us suppose that an image contains 100 faces and another one contains 10. Now, let us suppose that the detector found 5 faces and got only 3 correct in the first image. However, in the second, it correctly detected 9 out of 10 faces. The results would be: 5 true positives, 3 false positives and 95 false negatives for the first image; 9 true positives, 1 false positives and 1 false negative for the second one. Computing the overall F1 score results in 0.22. However, when computing the per image score, we would get 0.093 and 0.9 for the first and the second images, respectively. Summing up such values and dividing by two results in around 0.5. Hence, we can see that when computing the overall score the best detections are heavily penalized and the result tends to be lower. Therefore, for the experiments with the next face detectors, we are going to show plots based on both the overall F1 measure value and the per-image F1 measure.

3.4.1.2 Dlib

Differently from OpenCV, Dlib's methods for face detection provide only one tunable parameter. The parameter corresponds to the numbers of times an image will be upsampled. For instance, let us suppose that we have an image of size 240×240. The method's default configuration is to look for faces that are at least 80×80 (regardless of the size of the input image). If the upsample parameter is changed to 1 the minimum face size changes to 40×40. If it is 2 the minimum size is 20×20, and so forth. Even though the detection rates raise when the parameter value is large, the drawback is performance, which is degraded when running detection even over a single image. Therefore, we executed experiments with four different upsample values: 1, 2, 3, and 4.

As we did for the experiments with OpenCV, we computed the F1 measure over all selected images with 4 different parameters and with different IoU thresholds. Additionally, we also computed the F1 measure value in an image basis. Figure 3.17 shows results of our experiments. We can see that the larger the value of upsample used in Dlib the better. Additionally, we can notice that the values 2, 3, and 4 are very similar in the results. Such results are understandable and smaller because when we increase the upsample value, the face detector searches for smaller faces in the image, being able to find more faces than usual. However, increasing upsample value results in larger execution time to find a face. Also, we can see that when the IoU threshold is large — larger than 0.6, for example — the difference between different upsample values decreases until 0.8 where they yield almost the same F1 measure value. Increasing the threshold makes it more difficult for a detector to detect faces correctly. One possible conclusion is that when we need a precise detection of the faces, the upsample parameter does not make much difference when using Dlib.

In order to evaluate the impact of the upsample parameter in execution time in a more concise manner, we randomly sampled 10 images from our subset of the WIDER FACE validation set. We then executed face detection in these 10 images with different values of upsample. Figure 3.18 contains the results from the execution [7]. To measure the execution time, we executed 10 times the

[7] All executions were performed in a server machine with 32GB of RAM memory and 2 Xeon processors.

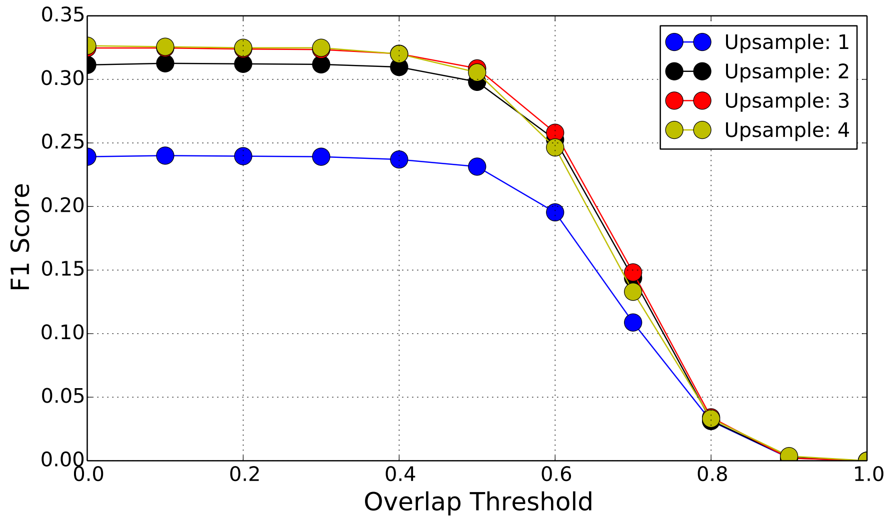
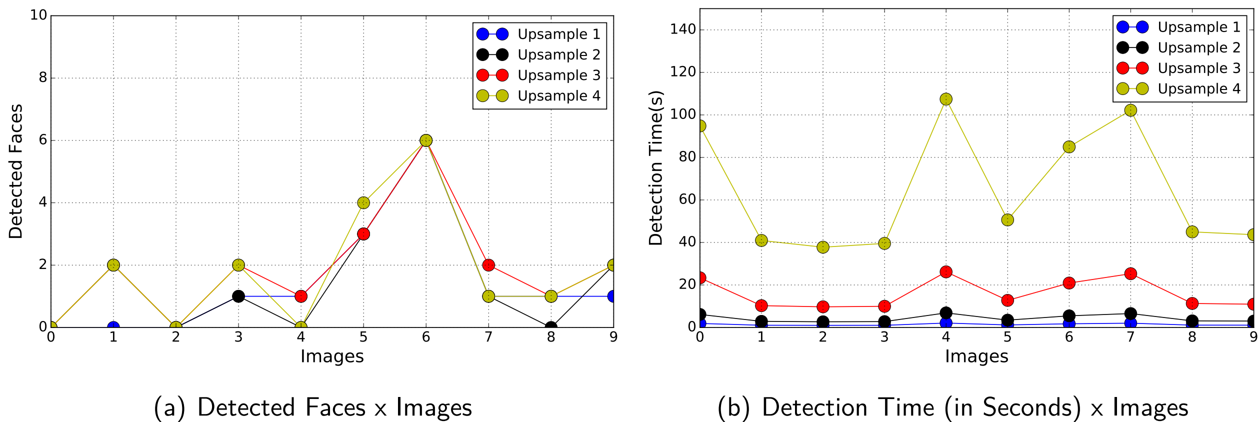


Figure 3.17: The 4 variations of upsample parameter that we used in our experiments with Dlib.

same experiment and calculated a simple average over the values. In Figure 3.18(a), we can see the impact of the upsample parameter in the number of detected faces in a given image. As we could see in Figure 3.17, the larger the value of upsample the better the results. Figure 3.18(b) shows the impact of upsample in execution time. We can see that the time increases for values of upsample larger than 2. Values larger than 4 showed to be impractical [8] in our experiments and hence we did not perform experiments with larger values than that.



(a) Detected Faces x Images

(b) Detection Time (in Seconds) x Images

Figure 3.18: Impact of changing upsample parameter in Dlib.

In Figure 3.18(a) larger upsample values sometimes find less faces than smaller values, and the opposite also happens. We can see that images 3 and 7 respectively represent such cases. We believe that larger upsample values are more susceptible to incorrect detections. To check if our assumption is correct, we selected some cases among the 10 images used for performance evaluation. They are depicted in Figure 3.19. All detection results show how changing upsample affects the overall number of detected faces but not how good the detection is. Figure 3.19(a) shows an interesting case. Recall that for image 5, the best result was with 4 as the upsample value.

[8] The detector took more than 1 hour to detect the faces in a single image.

All different configurations found the three faces in the image. However, we can notice that the leftmost person's tie contain a tiny yellow square, which represents the second "face" detected by Dlib with upsample equals to 4. Figure 3.19(b) shows a scenario where the four different setups yielded the same results, finding all the six faces in the image. Figure 3.19(c) illustrates a case similar to the one in Figure 3.19(a). All the setups found the face, however upsample equals to 3 found two faces. The tiny face found is on the right part of the image, inside the camera lens on the back. Again, increasing the upsample value may find more faces but with a risk of increasing the number of incorrect detections. Finally, the image in Figure 3.19(d) shows a case where upsample of 2 was not able to find the given face. We are not sure of what exactly affected the detection with such value and hence, we think it is an outlier case.

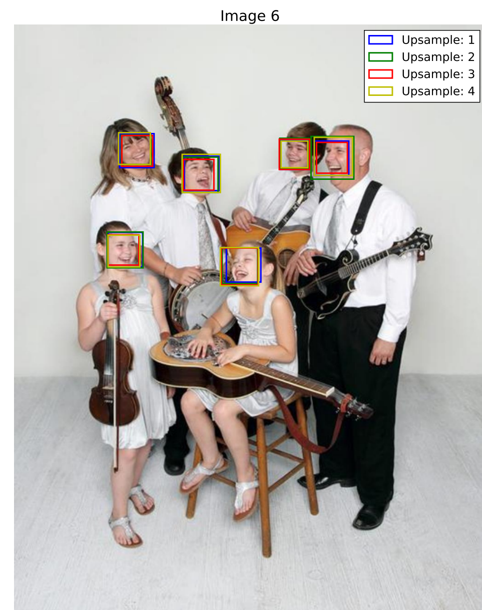
As we did for OpenCV, we computed the F1 score on an image basis. Figure 3.20 illustrates the results. We can see that the results in Dlib improve considerably when computing the F1 score individually, which is similar to OpenCV. In Figure 3.17, the largest F1 score was 0.33, while in Figure 3.20 it is almost 0.5. Additionally, we can notice that now the best results are when the upsample value is 2. These results corroborate with the detection results in Figure 3.19. When we increase the upsample value, Dlib face detector is more susceptible to find false positives. Hence, when computing the measure per image, upsample equals to 2 seems to be more robust than the others. We believe that the number of false positives increase for larger upsample values because the detector tries to find smaller faces (for instance 5×5). As far as we could see, the smaller the face the more susceptible to false positives Dlib face detector is.

With our experiments with different Dlib parameters we can conclude that upsample of 4 is the best overall when computing the F1 measure over all detections. However, when we consider a per-image basis, upsample vof 2 outperforms the other values. We can also notice that when the overlap threshold (IoU) increases, values 2, 3, and 4 seem to have a very similar performance, as we can see in Figure 3.17 and Figure 3.20. Since the default IoU value is equal or larger than 0.5, all our experiments with Dlib as a face detector will use upsample of 1 and 2. The choice of 2 instead of 3 and 4 is due to the fact that, when we compare image per image, 2 is better for almost all thresholds. We will also use another Dlib face detector with upsample as 1 because when the overlap is larger than 0.6, it becomes very similar to 2. Hence, in our future experiments, we will refer to Dlib-1 and Dlib-2 to indicate a Dlib face detector with upsample of 1 and 2, respectively.

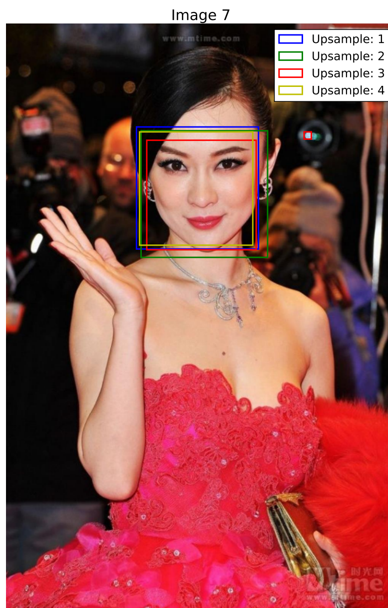
3.4.2 Deep Learning Approaches

Deep learning approaches are those based on learned feature extractors. In the following sections, we will detail the experiments we performed with these approaches.

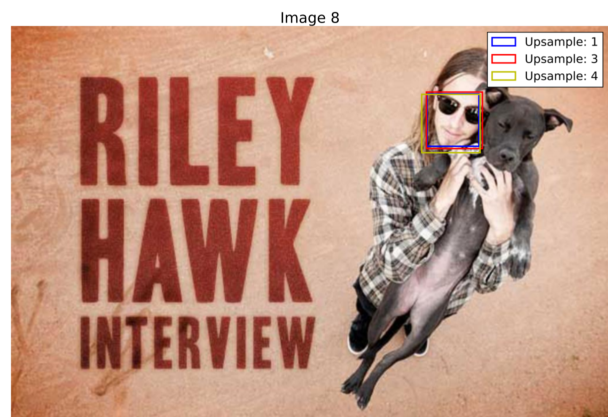
(a)



(b)



(c)



(d)

Figure 3.19: Results with Dlib face detector with different upsample values.

3.4.2.1 Faster R-CNN

Our main hypothesis is that deep learning based approaches are capable of outperforming classic implementations because they learn how to extract the best features in order to classify or execute a detection correctly. The two losses available in Faster R-CNN make it a network very difficult to train. One cannot easily assume that a small loss is equal to a well-trained network. Therefore, for our experiments with Faster R-CNN, we tested different models, each of which trained for a different number of iterations. We followed an approach similar to [52] for training. We used a VGG-16 model pre-trained on ImageNet and fine-tuned our model for $70,000$ iterations in the

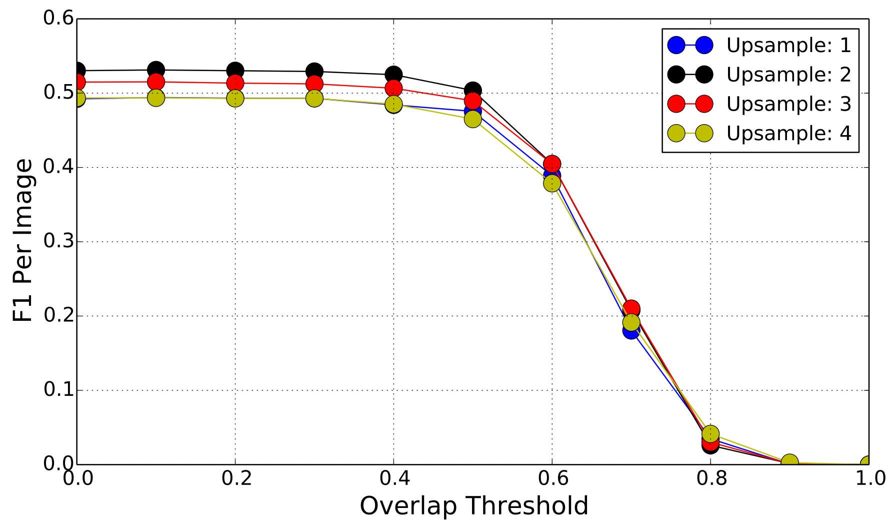


Figure 3.20: The 4 variations of upsample parameter that we used in our experiments with Dlib. In this case, results are based on F1 computed per image instead of the overall computation.

WIDER FACE training set. For the first $50,000$ iterations, we used a learning rate of 0.04. For the next $20,000$, we decreased the learning rate to 0.004. Every $10,000$ iterations we saved a snapshot of the model in order to have more models to test. Therefore, we ended up with 7 different models. Faster R-CNN also has parameters for the confidence threshold and the nonmaximum suppression. We used the values 0.5, 0.6, 0.7, 0.8, and 0.9 for the confidence threshold, whereas for NMS we used 0.5, 0.4, 0.3, 0.2, and 0.1. Such variations along with the 7 trained models resulted in 175 different combinations.

We followed the same idea for evaluation: computed the F1 measure over all selected images varying the parameters of the detector and the IoU thresholds. We also computed the per-image F1 score as it showed improved results with both OpenCV and Dlib (refer to Section 3.4.1.1 and Section 3.4.1.2). Figure 3.21 presents some results of our experiments. It shows the Top-10 results of our 175 different executions. We can see the impact of the fine-tuning for the last $20,000$ iterations. All the best results are from either iteration $60,000$ or $70,000$ and hence, we can conclude that the fine-tuning is effective for our scenario. Additionally, the confidence threshold is 0.7 or 0.8, which indicates that the higher the better. When we increase the NMS threshold, it seems to negatively affect the results. Only one of the Top-10 results has a NMS threshold larger than 0.3, and it is the worst result. However, when we increase the IoU value, those differences vanish. All F1 measure values in Figure 3.21 are larger than those presented by the hand-crafted approaches, which confirms our hypothesis — deep learning approaches should yield similar or better results than hand-crafted ones.

Figure 3.22 shows all 175 variations of Faster R-CNN that were executed. As it happens for OpenCV, we can see that it is crucial to evaluate the parameters in order to get the best result from Faster R-CNN. Even though the last model snapshots (the ones trained for more iterations) should yield the best results, we should carefully evaluate the confidence threshold and the NMS threshold to get even better results. NMS ignores bounding boxes that significantly overlap each other and,

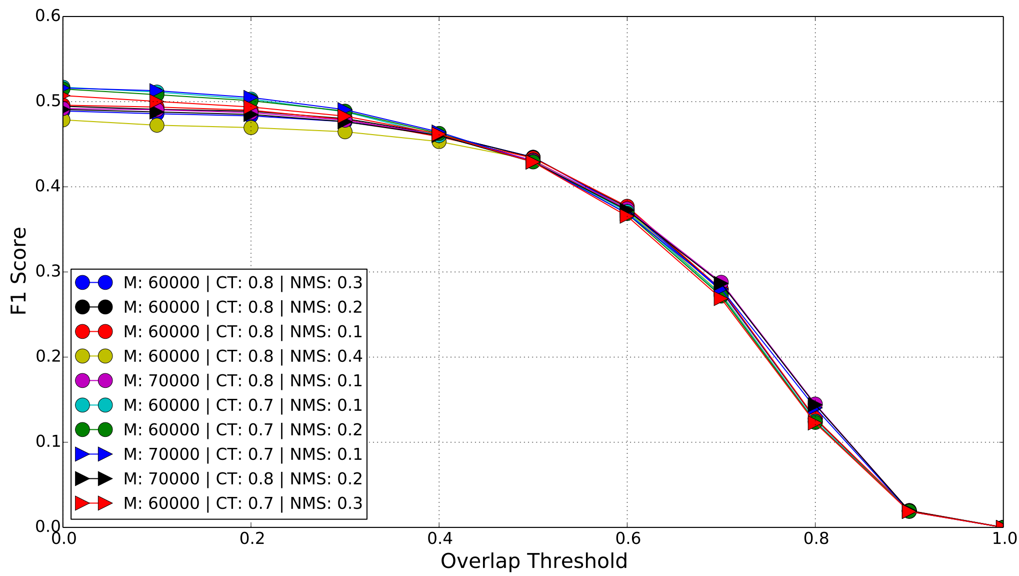


Figure 3.21: Top-10 results with Faster R-CNN. *M* is the Model, *CT* is the Confidence Threshold and *NMS* is the Nonmaximum Supression Threshold.

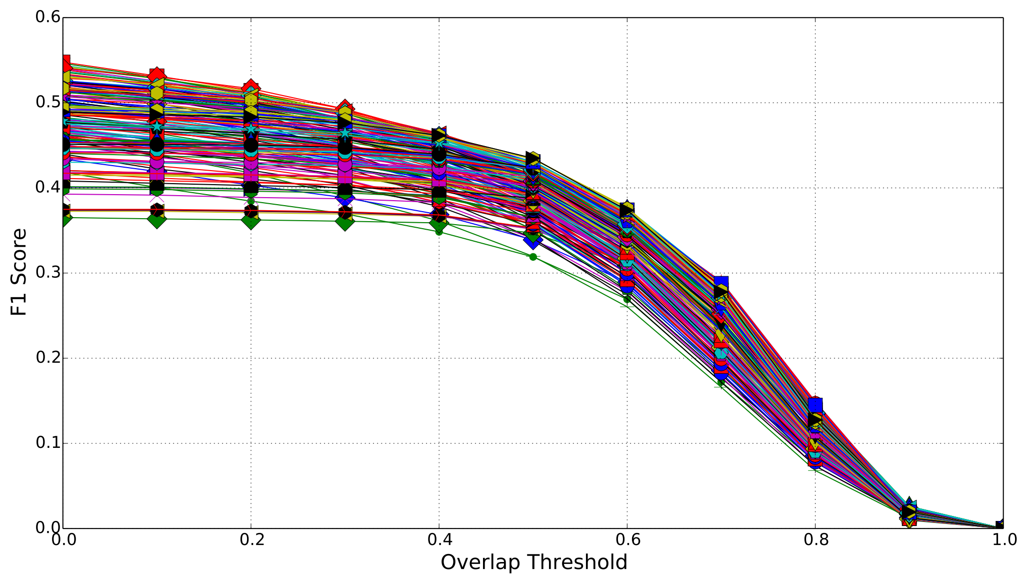


Figure 3.22: All 240 variations of Faster R-CNN models that we used in our experiments.

hence, results in a better and more accurate detection. The confidence threshold indicates how confident the model is with the detection — the higher the better. However, many good detections (the ones between 0.7 and 0.9, for example) may be ignored with higher thresholds. Figure 3.22 illustrates an example of different parameters with the same image. In *A*, the model is from iteration $10,000$, with confidence threshold of 0.5 and NMS threshold of 0.5. All faces are detected, however there are many false positives (e.g. the ball on the right) and detections that overlap. *B* shows an improved version that is from iteration $10,000$ as well, but the confidence threshold is now 0.9. We can see that the number of false positives reduced, however some of them continue on the image. Additionally, the first athlete of the second row is not identified anymore. *C* makes use of the best parameters for Faster R-CNN. It uses the model from iteration $60,000$, confidence threshold of 0.8

and NMS threshold of 0.3. The results are according to our evaluation with all parameters based on the F1 score, whose parameters were the same used for *C*.

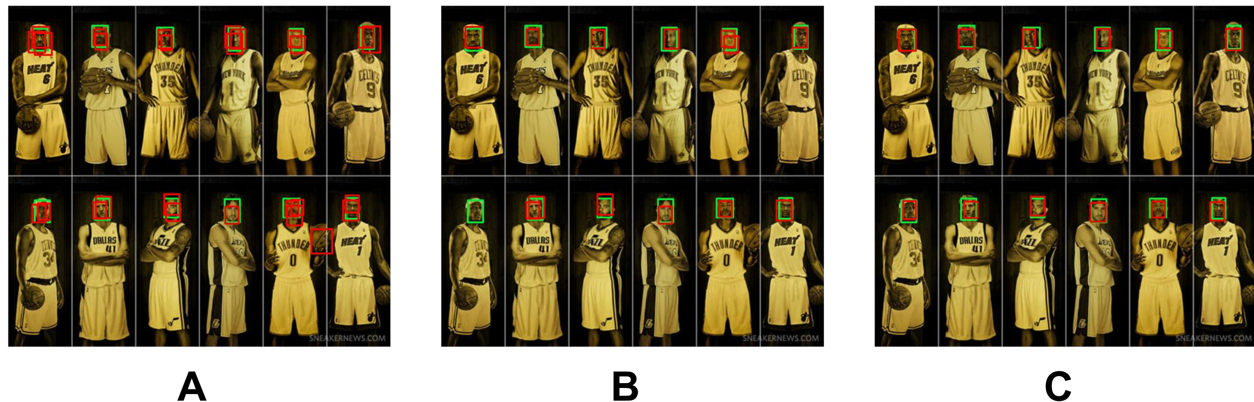


Figure 3.23: Examples of detection results with different Faster R-CNN models and parameters. *A* is result of a model from iteration $10,000$ with a low confidence threshold (0.5). *B* is of a model from the same iteration but with a larger confidence threshold (0.9). Finally, *C* is our best model, which is from iteration $60,000$ and has a confidence threshold of 0.8.

As we did for OpenCV and Dlib, we computed the F1 score on an image basis as well. The results are shown in Figure 3.24. The F1 score increased and reached almost 0.9. All the Top-10 models are from iterations $50,000$, $60,000$, and $70,000$, as it happens with the Top-10 for the F1 computed for all images. However, the confidence threshold is now larger: all the models have a confidence threshold of 0.9. The NMS threshold seems quite similar as all of them are between 0.1 and 0.4. Additionally, the models yield very similar F1 score per image and their difference is almost not perceivable in the plot.

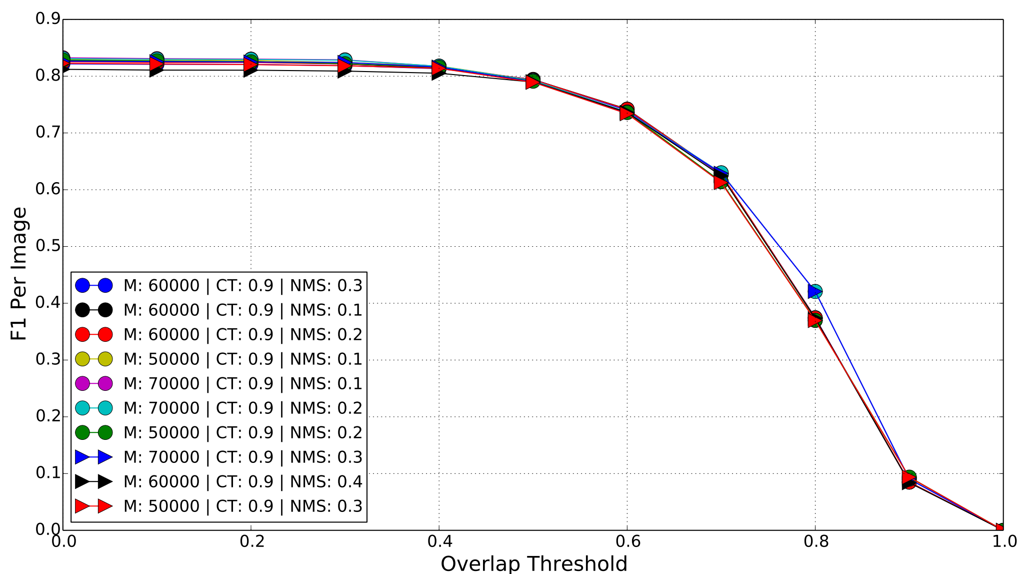


Figure 3.24: Examples of detection results with different Faster R-CNN models and parameters.

With our experiments with different Faster R-CNN trained models and parameters we can conclude that models from iteration $60,000$ outperform the other models. If we consider the Top-

10 results with F1 score based on all images instead of those calculated individually, the confidence threshold would be 0.8 and the NMS threshold 0.3 or 0.2. Nevertheless, our experiments with Faster R-CNN will use the model from iteration $60,000$, a NMS threshold of 0.3 and a confidence threshold of 0.9.

3.4.2.2 MTCNN

MTCNN experiments followed the same setup of the other experiments: execute the detector on the subset of validation images with different parameters. The implementation is based on three models (one per network) and it has three parameters: Minimum Size, Scale Factor, and Thresholds. Differently from Faster R-CNN, MTCNN is a deep learning approach that is trained for face detection and landmark localization. Therefore, we believe that it should yield better, or at least similar results to Faster R-CNN. The original MTCNN implementation and the other adaptations that we mentioned in Section 2.4.4 use different parameters. Some of them use 10 as the minimum size whereas others use 20. The scale factor strangely is 0.79 or 0.709. The thresholds are a triple that contains a threshold for the confidence of each of the three networks. The implementations also have varying values. Therefore, we used all the variations in our tests: $(0.6, 0.7, 0.7)$, $(0.5, 0.3, 0.3)$, and $(0.6, 0.7, 0.8)$. We ended up with 12 different executions with MTCNN.

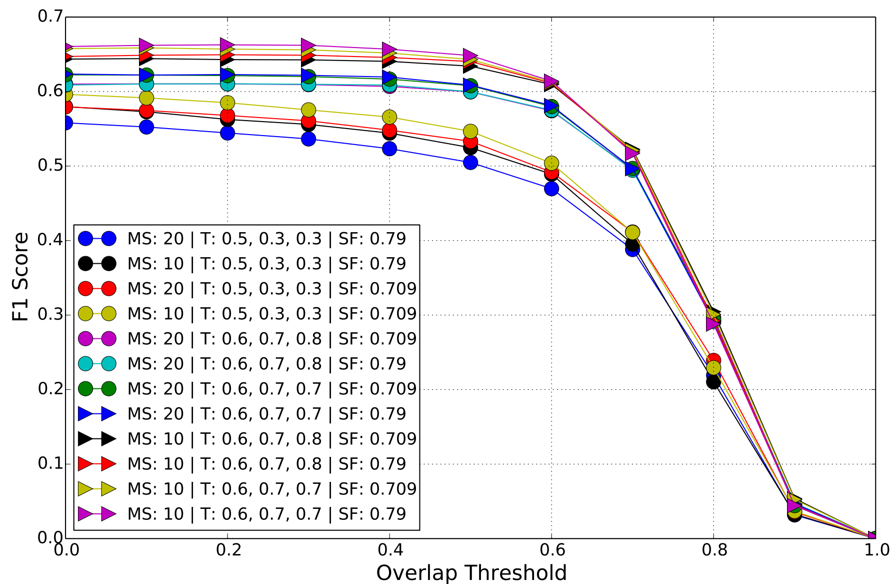


Figure 3.25: Results of all MTCNN executions.

Figure 3.25 shows the results of all 12 MTCNN executions. The best result is the one with 10 as minimum size, 0.79 as scale factor, and thresholds of $(0.6, 0.7, 0.7)$. On the other hand, the worst results have a larger minimum size of 20 and thresholds of $(0.5, 0.3, 0.3)$. They indicate the minimum size 10 yields better results than using 20. However, when we evaluate the F1 score on an image basis, we can see in Figure 3.26 that the scenario changes. The best execution results contain the same scale factor and thresholds but the minimum size is now 20 instead of 10. Our hypothesis is that it is similar to what happens with upsample parameter in Dlib. For smaller values,

the detector is able to find even smaller faces, however it is more susceptible to false positives (incorrect detections). In order to evaluate the visual impacts of the different values, we selected some images and executed the MTCNN detector.

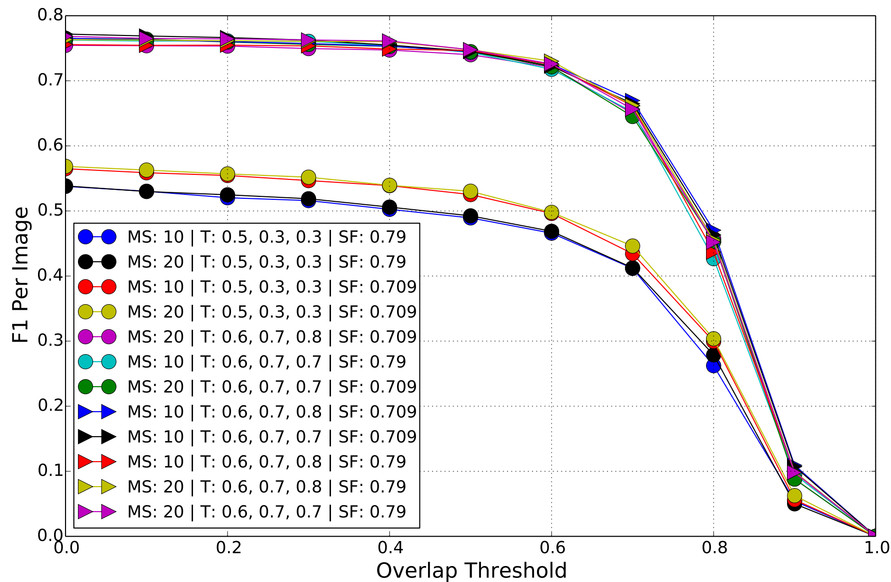


Figure 3.26: Results of all MTCNN executions. In this case, evaluation of F1 score per image.

The first example is illustrated in Figure 3.27. Both results *A* and *B* have 0.79 as the scale factor and thresholds of $(0.6, 0.7, 0.7)$. The only difference is that *A* uses a minimum size of 10 whereas *B* uses 20. We can see that both detect almost all faces in the image except the one on the leftmost part (which is very difficult due to occlusion). However, in *A* there are two false positives: one is in the bottom right portion of the image and it is not a face; the other is almost in the middle and it is a detection very close to other detection that is correct. Figure 3.28 shows another example of the impact of the parameters. We used the same parameters, but now we evaluate the results on images with fewer faces. *A* has two cases where the detector identified the existing faces correctly, however there were some false positives as well. On the other hand, *B* is the result of a detector that correctly identified the faces. Again, we can see the subtle differences between the same detector with different parameters.

The experiments we executed with MTCNN showed results compatible with Faster R-CNN. Analyzing the different executions we performed, we can conclude that using a minimum size of 20, scale factor of 0.79 and thresholds as $(0.6, 0.7, 0.7)$ are the best choice. Therefore, for our next experiments we will use such parameters.

3.4.3 Analysis

In the previous sections, we evaluated each face detector individually. Now we will evaluate their behavior together. We selected the best result for each classifier based on the higher F1 score for an IoU that is larger than 0.5. The choice of 0.5 is not arbitrary: this is the default threshold

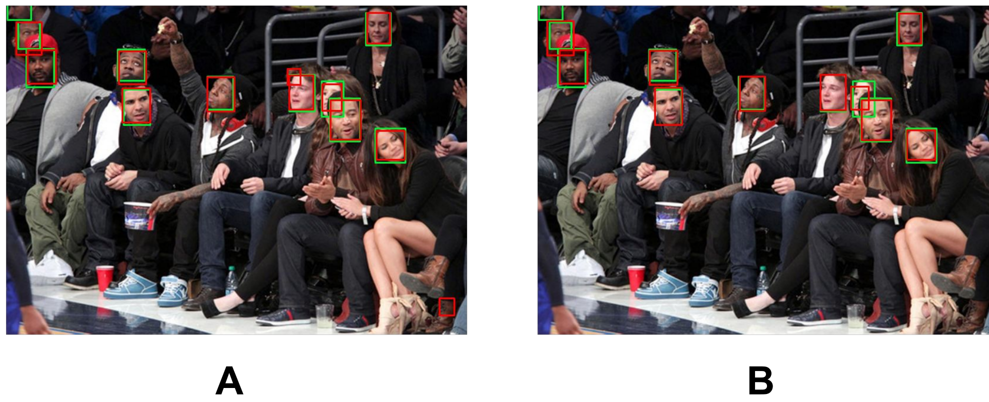


Figure 3.27: Example of MTCNN execution with different parameters.

Figure 3.28: Another example of MTCNN execution with different parameters.

for considering a detection as true positive or false positive. Therefore, we believe that choosing the best result for each detector based on such score is a suitable approach. We selected both the highest overall F1 score as well as the F1 score calculated in an image basis. Figure 3.29 shows the results based on the overall F1 score. We can see that the best result in all different thresholds is from the MTCNN implementation. It is significantly larger than OpenCV, Dlib-1, and Dlib-2, and consistently larger than Faster R-CNN. OpenCV is better than Dlib-1 and very similar to Dlib-2. However, we can see that Dlib-2 is slightly better than OpenCV overall.

We also plotted results with respect to F1 score per image. Figure 3.30 shows the results. Now, Faster R-CNN is consistently better than MTCNN. The inflection point is when the IoU threshold is 0.65, which is when MTCNN is slightly better than Faster R-CNN. If we analyze the results based on 0.5 of IoU, which is the value we will use in our experiments, Faster R-CNN is the best face detector. The difference between OpenCV and Dlib-2 increased: now Dlib is almost 0.1 better than OpenCV in F1 score. Besides, Dlib-1 is now better than OpenCV as well. This result is confirms the unpublished results we found [9], which found Dlib-1 – equivalent to Dlib with its default parameters – to be better than OpenCV for face detection. When evaluating both figures, it is clear that we should not evaluate the detectors based on F1 score of only one threshold, as

[9]We did not find any research papers with these experiments. However, a video at Youtube shows some differences regarding robustness – https://www.youtube.com/watch?v=LsK0hzcEyHI

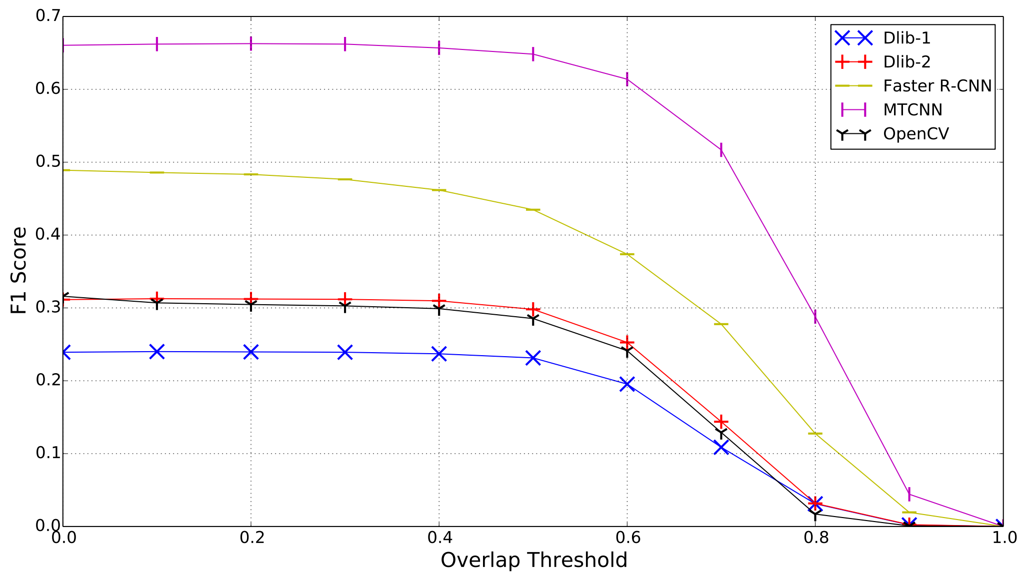


Figure 3.29: Best results of each face detector based on overall F1 score (10% of validation set).

the difference between them changes along the different values. In addition, the F1 score per image evaluation continues to give great insights about the face detectors and hence, we will keep using it to compare with our meta classifier implementation.

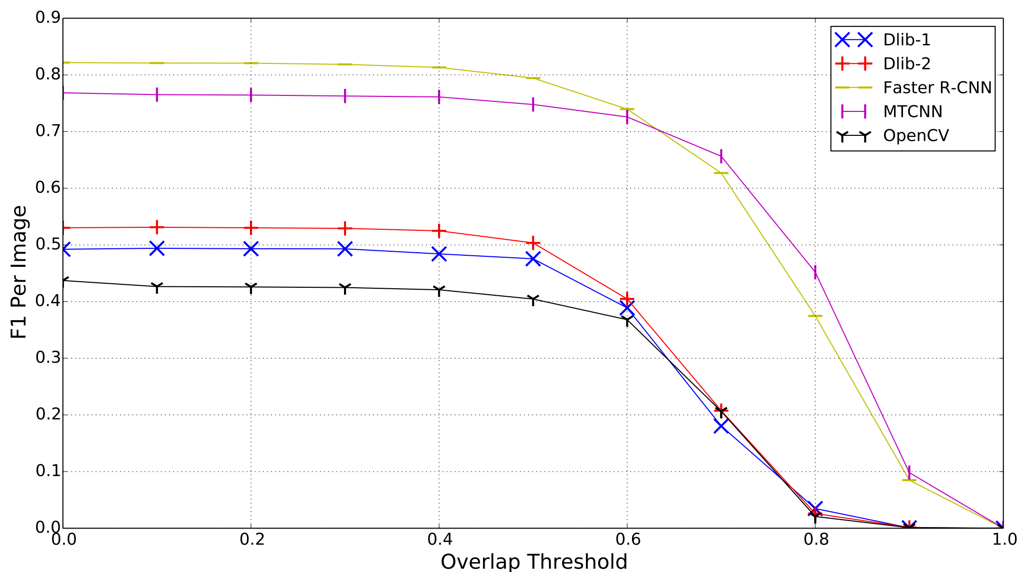


Figure 3.30: Best results of each face detector based on F1 score per image (10% of validation set).

The results indicate that Faster R-CNN is the best face detector based on F1 score per image. However, one might argue that the experiments are based only on 10% of the validation dataset of WIDER FACE and hence, such results may not represent the true results for the whole dataset. Besides, the results do not indicate that for other images and other datasets the same detectors will be the best ones. To solve such issues we: $1)$ executed the evaluation on all WIDER FACE validation set; $2)$ executed experiments with other datasets along with our meta learning approach.

Figure 3.31 illustrates the results of executing all face detectors in all images of the WIDER FACE validation set. We can see that the behavior is consistent with what happens for 10% of the dataset. Faster R-CNN continues to be the best face detector overall, followed by MTCNN. The differences between OpenCV, Dlib-1, and Dlib-2 are still the same. The main change is with respect to the top limit F1 measure score for each face detector. All of them decreased proportionally — about 0.1 or 0.2 in the overall score. Figure 3.32 shows the results based on the overall F1 score. Recall that the values here are worst than in the previous figure. Again, the order of the detectors is the same, following the behavior of Figure 3.29 – MTCNN is better than Faster R-CNN and OpenCV is better than Dlib-1 but worst than Dlib-2.

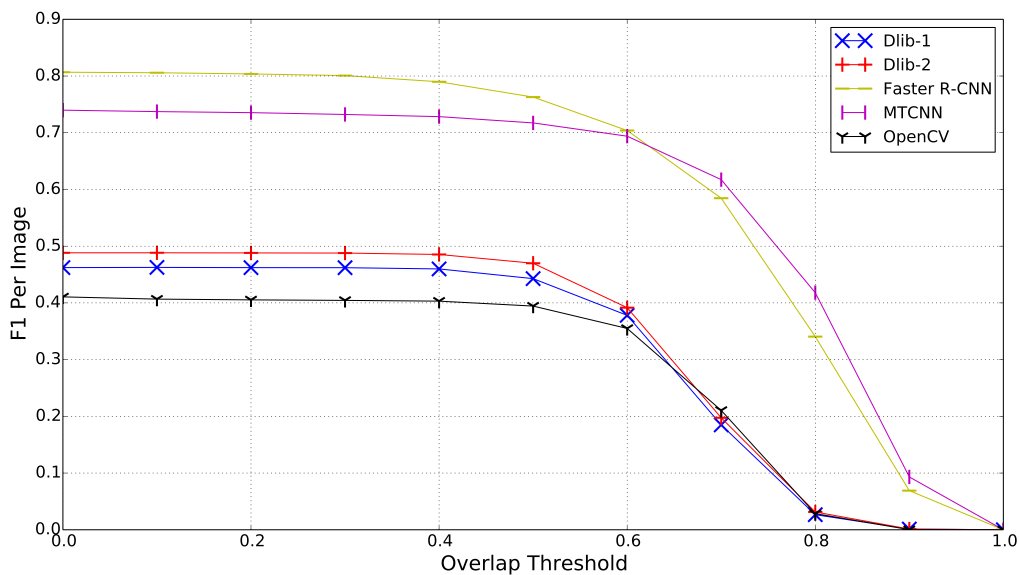


Figure 3.31: Best results of each face detector based on the per-image F1 score (complete validation set).

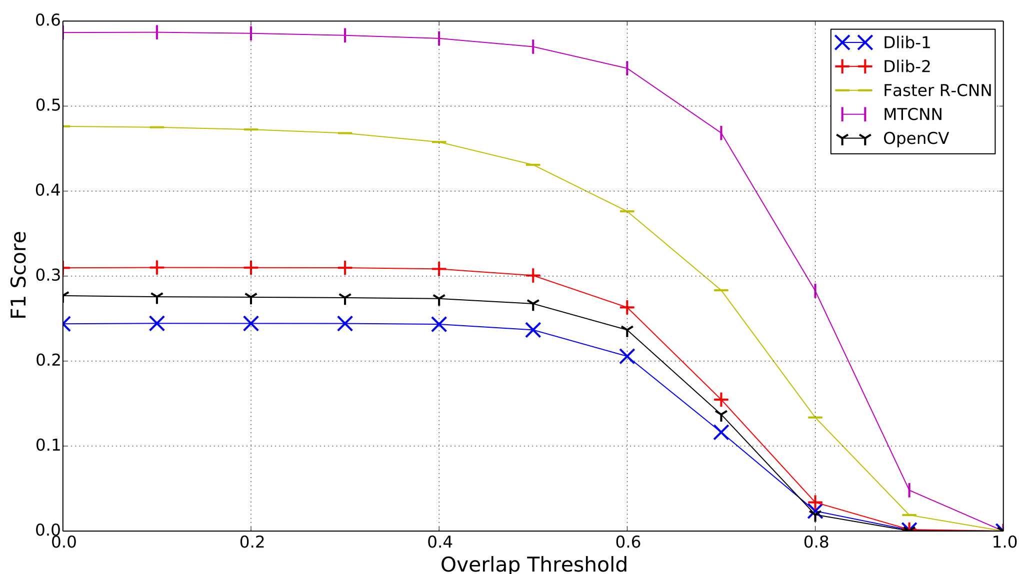


Figure 3.32: Best results of each face detector based on the overall F1 score (complete validation set).

3.5 Meta Learning Approach

In the previous section, we evaluated each face detector and compared them against each other. We believe that each face detector is suitable to specific images. For instance, for images that contain many small faces in a crowd, the best face detector could be Faster R-CNN. However, for images that contain faces that are larger, the best face detector could be OpenCV. Towards this end, we propose an approach to evaluate each face detector individually as well as combinations of them. For a combination of all different detectors, we implemented a *Meta Learning* approach. *Meta Learning* aims to choose the best classifiers dynamically, as opposed to *Base Learning* where a single trained classifier is used [119].

Our approach consists in training a classifier whose features are extracted from a given image using a deep learning network trained on a general purpose dataset – for instance, ImageNet or PASCAL VOC. We believe that specific features of an image will give the trained meta classifier information to decide which is the best detector to be used. Additionally to all individual face detectors and the meta learning approach, we also implemented a random classifier as to have a baseline. Such classifier randomly chooses one face detector among the implemented ones for a given image. We can roughly explain our approach in three different steps: meta training set creation, training, and prediction. We will explain each of this steps in the following section. Section 3.5.1 details the meta training set creation, whereas Section 3.5.2 explains how we trained each of the classifiers. Section 3.5.3 details how prediction works, while Section 3.5.4 details our experiments. Finally, Section 3.5.5 concludes our experiments.

3.5.1 Meta Dataset Creation

In order to create our meta dataset we have four main steps: execute all face detectors on a given image set; compare detections with ground truths and compute each detector F1 score for each image; get a feature vector for each of the images by forwarding them to different trained ConvNets; create the datasets themselves.

First, we use all face detectors we have explained in this work and the training set of the IJB-A dataset. IJB-A contains 10 different folds for training (and testing). We executed experiments with the first training set. The training set is comprised of $16,010$ images with faces. Figure 3.33 shows how the meta training set creation works. The first two steps are $1)$ computing F1 measure over the resulting bounding boxes of each face detector and $2)$ extracting features from all images. Hence, for each image we execute all face detectors, extract the bounding boxes, and compute the F1 score. It results in a list with each image followed by the F1 score for each detector. With such score, we are able to decide whether one face detector is better over another. In cases where two or more face detectors have the same F1 score, we reject such instance. We argue that these instances would confuse the meta classifier since a face detector would not be better than the other.

In the second step, we extract the feature vector that represents the image. To do so, we forward each image to a trained Convolutional Neural Network and get the feature vector from a given layer. We executed tests with four different CNN architectures, namely *VGG*-16, *VGG*-19, *GoogLeNet Inception v*3, and *ResNet* 50. The network architectures are the original implementation or an evolution of an architecture that won — or was a runner-up, in the case of VGG — the ImageNet Challenge, mentioned in Chapter 2. We have already detailed VGG-16 and VGG-19 in the previous chapter. They are two deep architectures comprised of 16 and 19 layers, respectively. The VGG-16 was the runner-up of the 2013 ImageNet Image Classification challenge, whereas VGG-19 is an improved implementation that contains more convolutional layers.

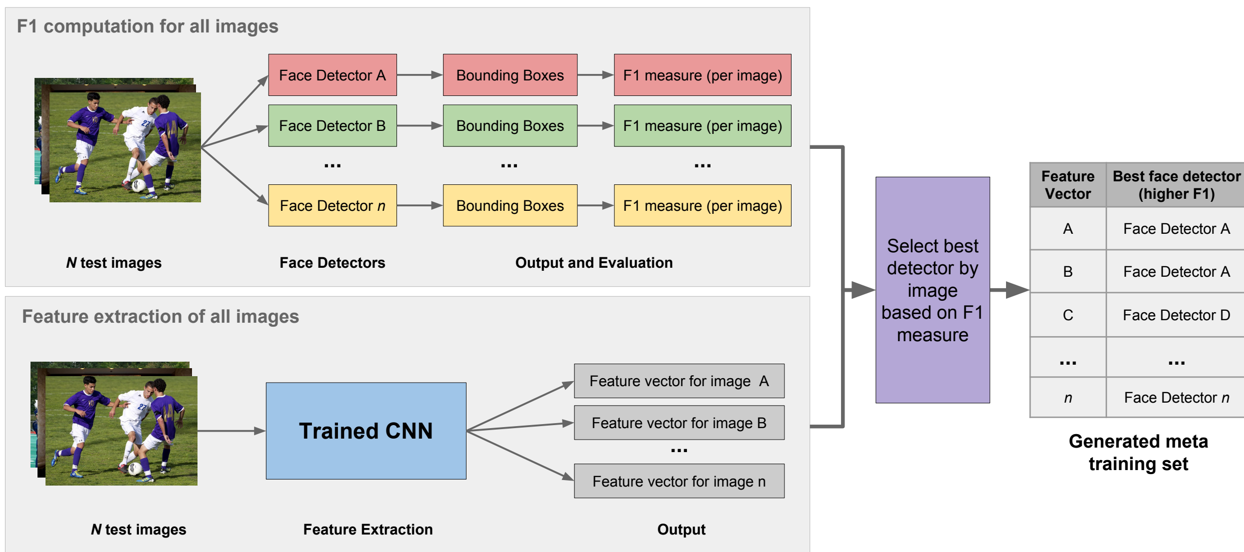


Figure 3.33: How our metadataset is created.

Inception v3 [112] is an evolution of GoogleNet [111], which won the ImageNet challenge of 2013. GoogLeNet is a very deep neural network that is more complex than VGG but contains fewer parameters (GoogLeNet has 5 million parameters while VGG has about 140 million parameters). Its architecture is depicted in Figure 3.34. In summary, the main difference of GoogLeNet is the so-called *Inception* module, which is illustrated in Figure 3.35(a). Instead of applying a convolution of, let us say 3×3, the Inception module applies different convolutions and combines their results at the end. Each Inception module contains four different branches, each of which have different convolutions with different filter sizes applied. From left to right, the first one is a convolution with filter size of 1×1. The second is a convolution with filter 1×1 followed by a 3×3 convolution. The next one is a convolution with filter 1×1 followed by a 5×5 convolution. Finally, the last is a 3×3 max-pooling followed by a 1×1 convolution. The idea is that each convolution is able to extract different aspects from the image, where the smaller filters look for more local information while the larger ones look for more contextual information. Additionally, another contribution of such an architecture is to remove the fully-connected layers at end, replacing them by an average pooling [73]. Average pooling is a simple strategy that concatenates the output features maps of

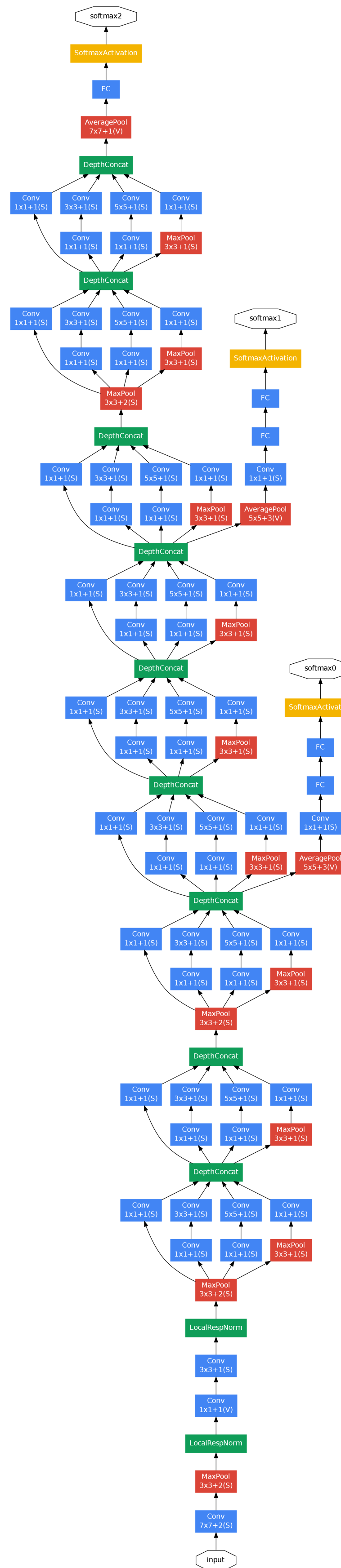


Figure 3.34: GoogleNet architecture (Image from [111]).

the four different branches into a single vector. The vector is then followed by a Softmax layer, which performs the classification. Inception v3 contains three different inception modules. The first one is the same as in Figure 3.35(a). The second contains two 3×3 convolutions instead of using 5×5 convolutions, which is illustrated in Figure 3.35(b). Figure 3.35(c) shows the last one, where one of the 3×3 and the 5×5 convolutions are replaced by two convolutions of size 3×1 and 1×3 each. The concept behind both new inception modules draws inspiration from convolution factorization, which aims to increase the computational efficiency of the convolutions, changing larger filters for smaller ones [112]. It results in a faster training and reduction of the number of parameters (weights) of the network. Since Inception v-3 is a newer, faster, and, "lighter" version of GoogLeNet, we will use its implementation as one of our CNN architectures.

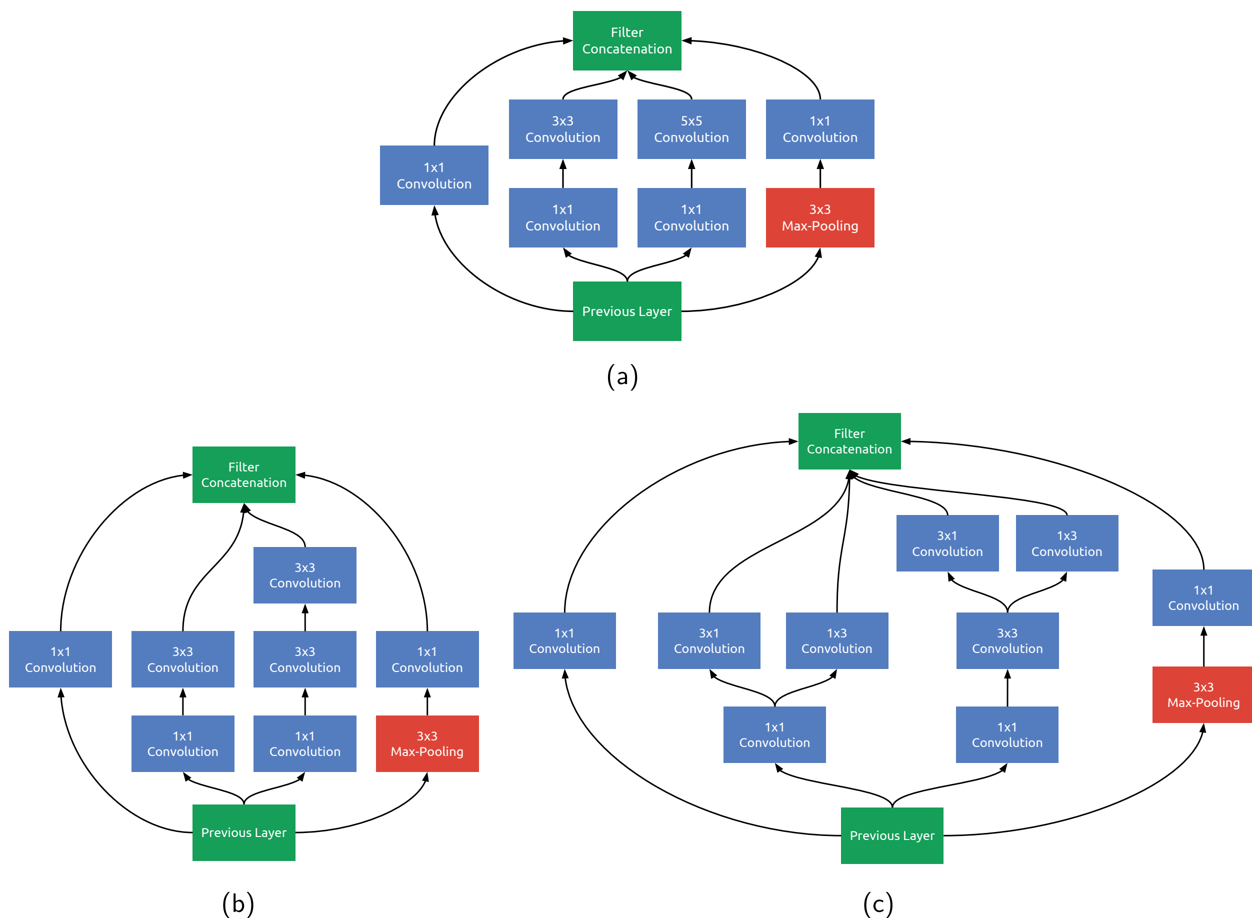


Figure 3.35: The three different Inception modules available in Inception v3. The first one is used in the original GoogLeNet implementation.

ResNet 50 is a particular implementation of ResNet [43]. GoogLeNet advanced the state-of-the-art in accuracy but also with respect to the depth of deep learning models – 22 layers when compared with 19 layers from the larger VGG, and only 8 layers from AlexNet. ResNet increased depth, reaching 50, 101, and 152 layers implementations. Figure 3.36 shows a simple comparison between VGG-19, a regular neural network, and a residual neural network. ResNet is a major improvement over previous architectures and was the winner of several competitions regarding Computer Vision. In summary, ResNet was able to overcome the *vanishing gradient* problem by

using the so-called *residual connections*. Figure 3.37 depicts a residual connection. It is a simple but effective concept, which enables the network to send the *gradient* in back-propagation directly to upper levels, skipping some layers. Hence, it helps during training and does not vanish the gradients of very deep architectures [44]. ResNet, similarly to GoogLeNet, has no fully-connected layers at the end and it makes use of an average pooling layer. For our experiments, we chose ResNet 50, which is the representation comprised of 50 layers.

For each of those architectures, we extract features from different layers. Table 3.1 shows the different layers we used for each of them. We chose the layers based on the use of them in the literature. Usually, for feature extraction purposes, the last layer before a Softmax layer is used as the representation of a given instance. FC2 is commonly used for any VGG architecture, even though FC1 could be used as well. In the case of both Inception v3 and ResNet-50, the average pooling layers are most commonly used. We selected different ConvNet architectures as to have diversity on the features extracted from the images. As we will explain in the experiments section (Section 3.5.4), features extracted from different architectures indeed yield different results.

Table 3.1: CNN architectures used followed by the respective layer that we used to extract features and their dimensionality.

Architecture	**Layer**	**Dimensionality**
VGG-16	Second Fully-Connected - FC2	4096
VGG-19	Second Fully-Connected - FC2	4096
Inception v3	Average Pooling	2048
ResNet-50	Average Pooling	2048

The last step is related to the meta dataset creation itself. To do so, we simply chose the detector whose F1 score is the largest. Each instance of our datasets is comprised of the feature vector as the features and the detector as the label. With such a dataset, we can train a classifier that is able to given a set of features from an image, choose which face detector to use. We created four different meta training sets, each focusing on a different scenario/setup. The first one comprises all five classes — namely *OpenCV*, *Dlib*-1, *Dlib*-2, *MTCNN*, and *Faster R-CNN*. The second one is comprised of four classes, namely *OpenCV*, *Dlib*-1, *MTCNN*, and *Faster R-CNN*. The third is comprised of two classes, where we chose one hand-crafted face detector — *OpenCV* — and one based on deep learning — *Faster R-CNN*. The last one comprises the two deep learning approaches, namely *MTCNN* and *Faster R-CNN*. Since we ignored cases where the F1 score is the same for multiple detectors (and different than 0), the datasets ended up with different number of instances. Table 3.2 shows all meta training sets. For each of them, the table shows the number of images with faces detected per face detector, the number of images where there was a tie between two or more face detectors, and the number of images where no face detector was able to find any face. The last column *Detected* is a sum of all detections of the detectors used in the given meta dataset without considering the ones with same F1 Score.

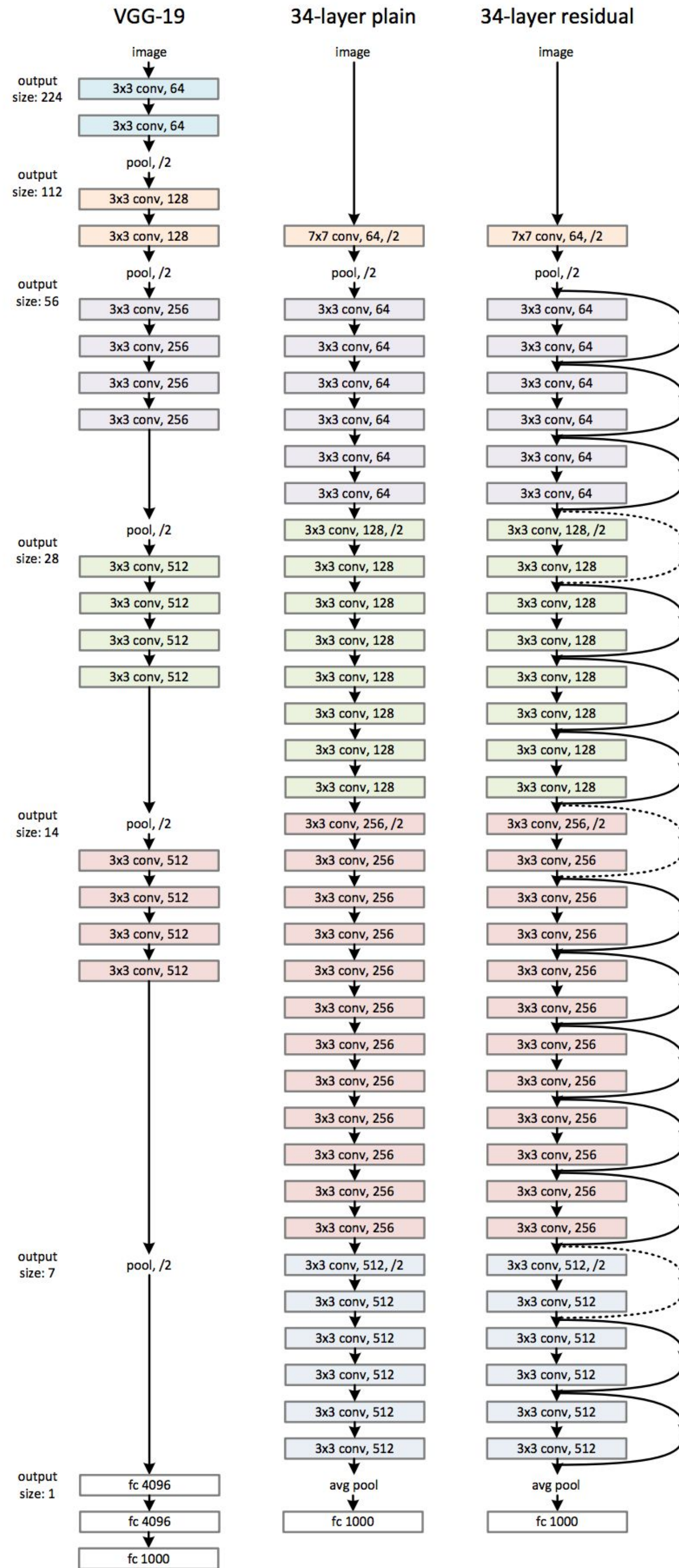


Figure 3.36: Three different architectures, comparing VGG-19 (left), a regular deep neural network (center), and a residual neural network with 34 layers (right). (Image from [43]).

When we observe Table 3.2, we can see interesting scenarios. For instance, when using many face detectors (last row), the best face detectors are penalized — Faster R-CNN and MTCNN

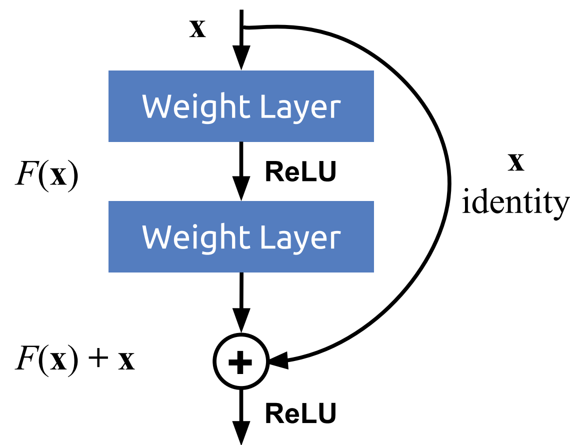


Figure 3.37: A residual block example.

— since they yield the same F1 score as some of the hand-crafted approaches. On the contrary, when Faster R-CNN achieves its best detection is with the face detector that finds less faces, which is Dlib-2. We can also see that deep learning approaches (MTCNN and Faster R-CNN) are better than hand-crafted ones (OpenCV, Dlib-1, and Dlib-2) overall. If we compare the best number of detections of hand-crafted features — 3854 for OpenCV, Dlib-1, and Dlib-2 — with deep learning — 4935 for MTCNN and Faster R-CNN — the difference is relatively small. If we take into account the ones with same F1 Score, the numbers change to 5948 and 12204, which can be now seen as a large difference. For our experiments, we choose only 4 out of the 10 available datasets. Table 3.3 shows additional information regarding the meta training sets along with the chosen ones, highlighted in red. The reasoning behind such choices is based on the number of detections summed up with the number of detections with the same F1. We argue that detections whose F1 score are the same can be considered successful detections because there was a tie between two detectors, meaning that at least something was detected. It is important to mention that we compute *Same F1* based only on F1 scores larger than 0 — zero for F1 score is counted as *Miss detections*.

The first two chosen meta training sets are the two highest values for column *All detected*: the one with all five classes — OpenCV, Dlib-1, Dlib-2,MTCNN, Faster R-CNN — and the one with only four (without Dlib-2). We consider the one that has four classes without Dlib-1 too similar to the one with it, and hence, we did not use it. The next one is a choice according to the results in the individual experiments, which favors detectors based on deep learning – Faster R-CNN and MTCNN. Finally, we chose the best one among the meta training sets with at least one hand-crafted feature extractor — OpenCV and Faster R-CNN. We believe that the chosen meta datasets are sufficient to cover our goals regarding experiments, since they are based on different combinations of hand-crafted and deep learning approaches. These meta training sets were then used for training our meta classifiers.

Table 3.2: The created meta training sets with their classes, number of detections per type of face detector, and total detections. The largest values of each column are bold.

Classes	OpenCV	Dlib-1	Dlib-2	MTCNN	Faster R-CNN	Same F1	Miss detections	Detected
OpenCV, Dlib-1, Dlib-2	**2527**	682	645	-	-	4260	7896	3854
Dlib-1, Dlib-2	-	**1712**	**1501**	-	-	2735	**10062**	3213
OpenCV, Faster R-CNN	1081	-	-	-	6899	4126	3904	7980
OpenCV, MTCNN	1633	-	-	**5315**	-	3911	5151	6948
Dlib-1, Faster R-CNN	-	536	-	-	7955	3292	4227	8491
Dlib-2, Faster R-CNN	-	-	497	-	**8147**	3141	4225	**8644**
Faster R-CNN, MTCNN	-	-	-	1282	3653	7269	3806	4935
OpenCV, Dlib-1, MTCNN, Faster R-CNN	596	142	-	889	2772	8230	3381	4399
OpenCV, Dlib-2, MTCNN, Faster R-CNN	610	-	126	892	2787	8213	3382	4415
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	527	88	73	858	2706	**8412**	3346	4252

3.5.2 Training the Meta Classifiers

The second step is to train the meta classifier that predicts the best face detector to be used given an input image. Towards this end, we tested different classifiers and compared the performance of each of them. In our experiments, we trained two different types of classifiers, namely SVMs and Neural Networks. We choose SVM because it is robust for high-dimensional data and it has great generalization capability. Additionally, it is suitable for problems where few instances are available, yielding results that are better than Neural Networks in some scenarios. On the other hand, we consider Neural Networks a straightforward choice, mainly because most of our work is based on them. For instance, the CNN models we used for feature extraction usually have a Neural Network at the end of the architecture. We separated the data in 80% for training and 20% for test, which is a common training/test split, in all setups. We will detail SVM and Neural Networks training in the following sections.

3.5.2.1 SVM

Support Vector Machine (SVM) is a classifier often used for classification and regression tasks before deep learning. It aims to find the best hyperplane that represents the largest margin between classes. As we mentioned in Chapter 2, most of the traditional approaches make use of a hand-crafted feature extractor and an SVM for classification. It is also a classifier that works well for scenarios with a few instances, which is our case. We performed experiments with different

Table 3.3: The meta datasets chosen for our experiments (in bold).

Classes	Same F1	Miss detections	Detected	All detected
OpenCV, Dlib-1, Dlib-2	4260	7896	3854	8114
Dlib-1, Dlib-2	2735	10062	3213	5948
OpenCV, Faster R-CNN	4126	3904	7980	**12106**
OpenCV, MTCNN	3911	5151	6948	10859
Dlib-1, Faster R-CNN	3292	4227	8491	11783
Dlib-2, Faster R-CNN	3141	4225	8644	11785
Faster R-CNN, MTCNN	7269	3806	4935	**12204**
OpenCV, Dlib-1, MTCNN, Faster R-CNN	8230	3381	4399	**12629**
OpenCV, Dlib-2, MTCNN, Faster R-CNN	8213	3382	4415	12628
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	8412	3346	4252	**12664**

hyper-parameters, class weights, and data standardization. The SVM implementation from *scikit-learn* [85] (the library we used) has as its default values $C = 1$ and Gaussian kernel with $\gamma = auto$ (1 divided by the number of features). We tested SVM with C values $[0.1, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0]$ and standardizing the data with *standard score* (also called *z-score*), which makes data have approximately zero mean and unit standard deviation. We tested SVM with features extracted from all different networks specified in Table 3.1 and with the four specified meta training sets in Table 3.3. Therefore, we ended up with about 160 different SVM executions. Since the datasets are unbalanced, we also performed experiments with different weights for the classes.

The main results are illustrated in Table 3.4. We choose the model based on the largest F1 score presented in validation data. We started our experiments with the meta dataset comprised of four classes - OpenCV, Dlib-1, MTCNN, and Faster R-CNN. We executed all the variations abovementioned along with different weights for the classes. The weights control the importance of the classes, since they are very unbalanced. For instance, in this four-classes meta training set, Faster R-CNN has larger F1 score in 2706 detections, which is larger than the sum of the other four, as we can see in Table 3.2. Using weights in detection is a strategy that can make the loss function of the classifier penalize correct classifications of a given class and reward correct classifications of another class. Our results in Table 3.4 indicate that almost all the best cases of the SVM executions make use of weights, except for the OpenCV and Faster R-CNN cases. For all meta training sets we tested two variations of weights: i) we penalize only Faster R-CNN, considering weight of 0.5 and 1.0 for the others; ii) we penalize each method proportionally to its number of instances. The impact of changing the weights can be seen in Table 3.5 (examples for Inception v3). We can see that the difference in F1 score is considerable when compared with the results in Table 3.4. For

instance, for the case of the meta training sets with four classes, the best case with Inception v3 features has F1 score of 0.61 in the validation set whereas with other weights it has a F1 score of 0.48. Hence, it is important to evaluate the weights used in classifiers to enhance its performance with unbalanced data.

Table 3.4: Best trained SVM models.

Meta training set	C	Features	Standardized	Weights	Train F1	Validation F1
OpenCV, Dlib-1, MTCNN, Faster R-CNN	2.0	VGG-16	Yes	$1.0, 1.0, 1.0, 0.5$	0.84	0.61
OpenCV, Dlib-1, MTCNN, Faster R-CNN	2.0	VGG-19	Yes	$1.0, 1.0, 1.0, 0.5$	0.84	0.62
OpenCV, Dlib-1, MTCNN, Faster R-CNN	2.0	Inception v3	Yes	$1.0, 1.0, 1.0, 0.5$	0.85	0.61
OpenCV, Dlib-1, MTCNN, Faster R-CNN	2.0	ResNet-50	Yes	$1.0, 1.0, 1.0, 0.5$	0.83	0.62
OpenCV, Faster R-CNN	2.0	VGG-16	Yes	-	0.93	0.85
OpenCV, Faster R-CNN	2.0	VGG-19	Yes	-	0.93	0.85
OpenCV, Faster R-CNN	2.0	Inception v3	Yes	-	0.93	0.85
OpenCV, Faster R-CNN	2.0	ResNet-50	Yes	-	0.92	0.85
MTCNN, Faster R-CNN	1.0	VGG-16	Yes	$1.0, 0.4$	0.83	0.73
MTCNN, Faster R-CNN	2.0	VGG-19	Yes	$1.0, 0.4$	0.83	0.74
MTCNN, Faster R-CNN	2.0	Inception v3	Yes	$1.0, 0.4$	0.89	0.74
MTCNN, Faster R-CNN	2.0	ResNet-50	Yes	$1.0, 0.4$	0.88	0.74
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	2.0	VGG-16	Yes	$1.0, 1.0, 1.0, 1.0, 0.5$	0.84	0.60
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	2.0	VGG-19	Yes	$1.0, 1.0, 1.0, 1.0, 0.5$	0.84	0.61
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	2.0	Inception v3	Yes	$1.0, 1.0, 1.0, 1.0, 0.5$	0.84	0.62
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	2.0	ResNet-50	Yes	$1.0, 1.0, 1.0, 1.0, 0.5$	0.83	0.62

Table 3.5: Comparison of the trained SVMs results with different weight schemes (only for Inception v3). All results are using $C = 2$ and standardized data.

Meta training set	Features	Weights	Train F1	Validation F1
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Inception v3	$0.24, 1.0, 0.05, 0.16$	0.56	0.48
OpenCV, Faster R-CNN	Inception v3	$0.15, 1.0$	0.86	0.82
MTCNN, Faster R-CNN	Inception v3	$1.0, 0.3$	0.84	0.71
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	Inception v3	$0.2, 1.0, 1.0, 0.1, 0.04$	0.60	0.55

Finally, after evaluating the training results, we selected all 16 models from Table 3.4 to our experiments. We choose all variations since each of them can perform differently in other datasets. Additionally, such variations could provide insight regarding their generalization abilities. The results are evaluated in Section 3.5.4.

3.5.2.2 Neural Network

Neural Networks are more complicated to train than SVMs due to variability of both hyper-parameters and architecture. Neural networks can vary with respect to number of hidden layers, number of neurons, optimizer, loss function, learning rate, regularization, and so on. No guideline

is available on which is the best way to start training a neural network and what are the best parameters, however some common practices are used. For instance, we usually start with a small network (one or two hidden layers) and a few neurons on each of them. We start with an optimizer that converges faster than the majority and a simple learning rate. We monitor the training and try to overfit the data before generalizing in order to evaluate if the network is able to learn the training data *too well*. Afterwards, we tweak its parameters to reduce overfitting until we reach an interesting result, which may vary according to the application. Some of these common practices are available at Stanford's CS231 course notes [70], which we used as a guide for our training steps.

Based on our SVM experiments results we decided to follow some specific approaches in our neural networks experiments. First, all data is normalized with *z-score* as it resulted in better results and faster convergence – less iterations to train the model. Second, we learned that in almost all best cases the same parameters were used for different features. For instance, if $C = 2$ and standardized data yielded best results for features extracted with VGG-16, it afterwards yielded the best results for Inception v3 as well. Finally, in all datasets the parameters that yielded the worst results are very similar. With all these in mind, we started our experiments with neural networks with different network architectures and hyper-parameters, with VGG-16 features for the meta training sets of five classes, and used the best results on it for the other features and datasets.

We firstly created simple neural networks, comprised of one hidden layer, 64 neurons, ReLU as the activation function, no dropout, and no regularization. In an incremental fashion, we added more hidden layers, more neurons, dropout, changed learning rate, and performed more changes like using other activation functions. Our approach was to increase neurons from 64 to 128, 256, and 512. In the meantime, before increasing the number of neurons, we added a new hidden layer with the same number of neurons. We tested 65 variations of the mentioned parameters in our experiments, starting with the meta training set of five classes. From these variations, we selected the ones that yielded the largest F1 measure score in validation. Such selection resulted in 12 variations, which were then used for the experiments with the other datasets. These 12 variations can be summarized in 4 different neural network architectures, which are depicted in Figure 3.38. The difference between the architectures is a summary of what guided our experiments. In *A*, it is a very simple neural network comprised of a single hidden layer with dropout. In *B*, we have the same concept of *A* but with two hidden layers instead of one. *C* has a batch normalization layer after each hidden layer. Finally, *D* has the same architecture as *C*, but the hidden layers' weights have a regularization term. We will now detail the rationale behind our different architectures.

We started with a very simple neural network *A*. In our first execution with only 64 neurons in the hidden layer, we started to suffer from overfitting in one of the tested meta training sets. The training accuracy was almost 99% but the validation accuracy was $\approx 60\%$ in the meta training set with five classes. A recent and well-known technique for tackling this kind of problem in a very efficient way is *dropout* [107]. It basically randomly drops neurons from the neural network during training. After this, we also wanted to test a model with at least two hidden layers, which is the main goal behind neural network *B*. In model *C* we added batch normalization, which helps to reduce

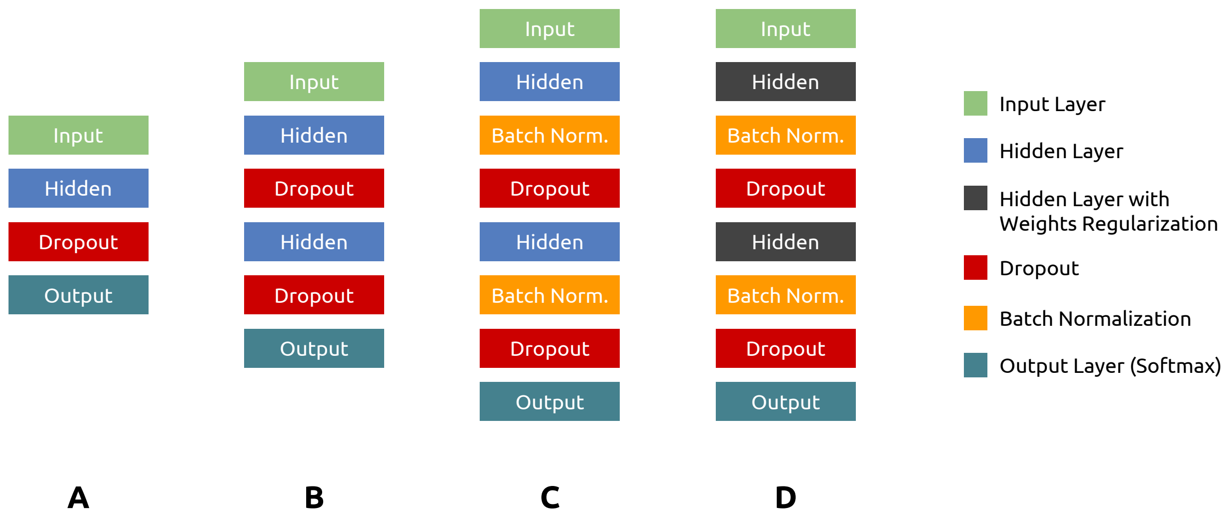


Figure 3.38: Prediction with the meta classifier.

problems of bad initialization of the weights and make convergence faster since the given batch is normalized [49]. Finally, we tried another common technique to prevent overfitting and increase the generalization capabilities of the model: *L2 weight regularization*, also known as *Ridge*, or *Weight Decay*, or even *Tikhonov regularization* [36]. In all our executions we used a learning rate 0.0001, trained for 100 epochs, and used Adam as our optimizer. The choice of all of them is based on the results of our executions with the 65 variations. It is important to mention that if we consider the four different features with the four different datasets, our 12 selected different architectures result in 192 different models. The twelve selected architectures are shown in Table 3.6.

Table 3.6: The twelve selected architectures.

	First Layer		Second Layer	
Reference Model	**Neurons**	**Dropout**	**Neurons**	**Dropout**
Model A	64	50%	-	-
Model A	128	50%	-	-
Model A	256	50%	-	-
Model A	512	70%	-	-
Model B	64	50%	64	50%
Model B	128	50%	128	50%
Model B	256	50%	256	50%
Model B	512	50%	512	50%
Model C	512	70%	512	50%
Model D	1024	50%	1024	50%
Model D	512	50%	512	50%
Model D	256	70%	256	70%

In order to choose among these 12 models, we adopted a different criteria. Instead of looking for the largest F1 score, we plotted the model's training and test accuracy and loss along the epochs. Such plot help us to understand if the large results are due to overfitting, for example.

Figure 3.39 shows one example of different results for loss whereas Figure 3.40 shows one example of different results for accuracy. We can see that the results are very different for different models that are similar in F1 score and accuracy. Regarding accuracy, the lower the distance between the training and validation accuracy curves, the better. A large distance means that the model is suffering from overfitting. In the case of loss, both curves need to drop in a very similar manner while keeping a small distance. If one drops but the other does not, it is also a sign of overfitting. The selected images show the training of different architectures with the same data, showing the impact in both accuracy and loss. They do not represent our best model neither what the best model should look like.

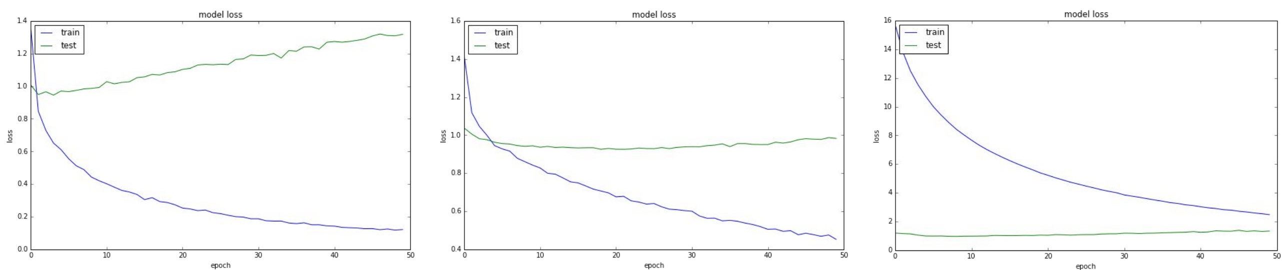


Figure 3.39: Different training and test losses during training.

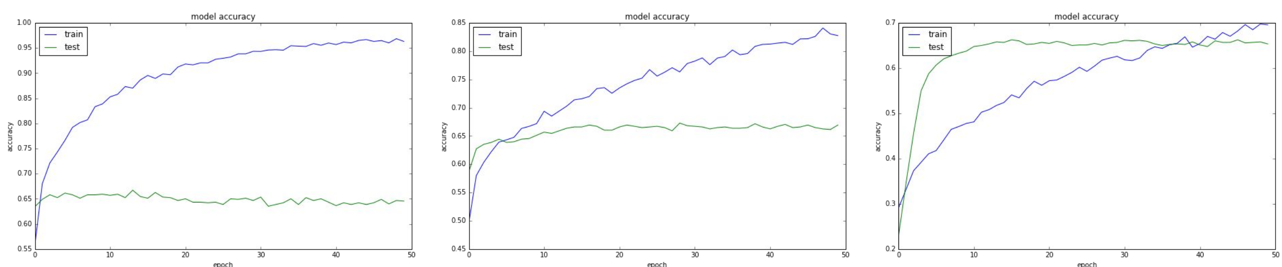


Figure 3.40: Different training and test accuracies during training.

Finally, Table 3.7 shows our selected models for prediction. We can see that the F1 score difference between the neural networks models and SVMs is narrow — the majority of cases is about 0.01 or 0.02. However, we can see that the difference is larger in training F1 values for some cases. For instance, for the meta training set comprised of MTCNN and Faster R-CNN, the difference is more than 0.1 (0.89 for the best SVM model against 0.99 for the neural network model). It shows that neural networks are able to learn training data well when compared with SVMs. Nevertheless, the resulting F1 score for validation is almost the same for both models, which means that even though the neural network is able to learn well the training data, to generalize for unseen instances is very hard for the problem at hand. The results of our executions with the selected models are evaluated in Section 3.5.4.

Table 3.7: Best trained neural network models.

Meta dataset	Features	Reference Model	Train F1	Test F1
OpenCV, Dlib-1, MTCNN, Faster R-CNN	VGG-16	*Model D*	0.90	0.62
OpenCV, Dlib-1, MTCNN, Faster R-CNN	VGG-19	*Model D*	0.90	0.62
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Inception v3	*Model D*	0.92	0.61
OpenCV, Dlib-1, MTCNN, Faster R-CNN	ResNet-50	*Model D*	0.89	0.61
OpenCV, Faster R-CNN	VGG-16	*Model D*	0.95	0.85
OpenCV, Faster R-CNN	VGG-19	*Model D*	0.95	0.85
OpenCV, Faster R-CNN	Inception v3	*Model D*	0.99	0.86
OpenCV, Faster R-CNN	ResNet-50	*Model D*	0.99	0.86
MTCNN, Faster R-CNN	VGG-16	*Model D*	0.96	0.73
MTCNN, Faster R-CNN	VGG-19	*Model D*	0.96	0.73
MTCNN, Faster R-CNN	Inception v3	*Model D*	0.98	0.73
MTCNN, Faster R-CNN	ResNet-50	*Model D*	0.97	0.76
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	VGG-16	*Model B*	0.95	0.61
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	VGG-19	*Model D*	0.97	0.62
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	Inception v3	*Model C*	0.97	0.61
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	ResNet-50	*Model C*	0.94	0.60

3.5.3 Prediction

After a classifier is trained, we are able to predict the best face detector for a given image. Figure 3.41 details the process. Given an input image, we forward such image on the same CNN that was used to extract features. To extract the features we get features from a specific layer, which depends on the architecture being used (as we saw in Table 3.1). The features are used as input to one of the trained classifiers, which outputs the possible best face detector for that particular image. Notice that after the classifier outputs the best face detector, we need to use such face detector over the image to detect the faces. Hence, the prediction can be summarized as follows. For a given trained meta classifier, for instance a SVM trained in a meta training set with VGG-16 features, we: 1) forward each image to VGG-16 model; 2) extract its features from the second fully-connected layer (FC2); 3) use the features as input to the trained SVM; 4) get resulting SVM prediction (class that represents face detectors to be used) and execute the predicted face detector onto the image; 5) compute the F1 score comparing each detection with its ground truth. The same applies for neural networks, which receives the feature vectors as input and outputs the best face detector for it. For the other five face detectors — OpenCV, Dlib-1, Dlib-2, MTCNN and Faster R-CNN — we execute each of them over all images and compute the F1 score to compare with the meta classifiers. The results are presented in details Section 3.5.4.

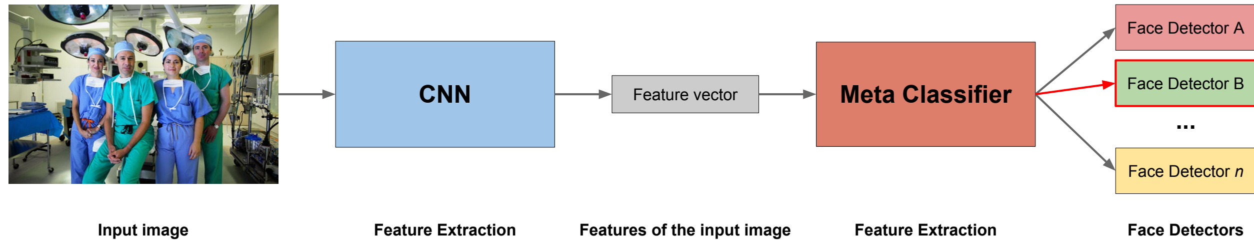


Figure 3.41: Prediction with the meta classifier.

3.5.4 Experiments

The previous sections described how we implemented our meta classifier approach, how we trained the meta classifiers, and some decisions we made regarding our implementations. This section focuses on showing results with the experiments we made, aggregating the individual face detectors and the meta classifiers results. First, we will separate our evaluation among the two datasets IJB-A and FDDB. Second, we will separate such evaluation in the different meta datasets, comparing each meta classifier trained on top of them with individual approaches. Finally, we will split these evaluations of each meta classifier in the ones based on SVMs and the ones based on neural networks.

3.5.4.1 IJB-A

As we mentioned in the previous sections, our meta classifiers are trained with IJB-A training data. Then, we test all meta classifiers and individual face detectors in the test set. In all cases, we follow the same protocol of our experiments throughout the dissertation, which is based on plots with F1 scores versus different IoU thresholds. We will start our analysis with the SVM results. Figure 3.42 illustrates them. The five plots represent four meta classifiers results plus one for random recommendation. The same applies for Figure 3.43, but now for Neural Networks instead.

The first thing we can notice is that all the approaches with meta classifiers yielded results very similar to Faster R-CNN. The first intuition behind such result is that the combination of the face detectors is superiorly limited by Faster R-CNN's performance. Even though the models we trained have weights that help to balance the unbalanced-classes scenario, it seems that Faster R-CNN performance dictates the meta classifier performance. We can also see that little difference is perceivable when comparing models trained with features extracted from different networks. One of our hypothesis was that more recent architectures would yield at least a small improvement over older ones, but it is not the case. Another interesting scenario is depicted in Figure 3.42(e), which is related to random recommendation. We can see that the random classifier that is based on OpenCV and Faster R-CNN yields results that are only smaller than the ones from Faster R-CNN itself. Hence, we can conclude that such random classifier chose Faster R-CNN over OpenCV detector in many

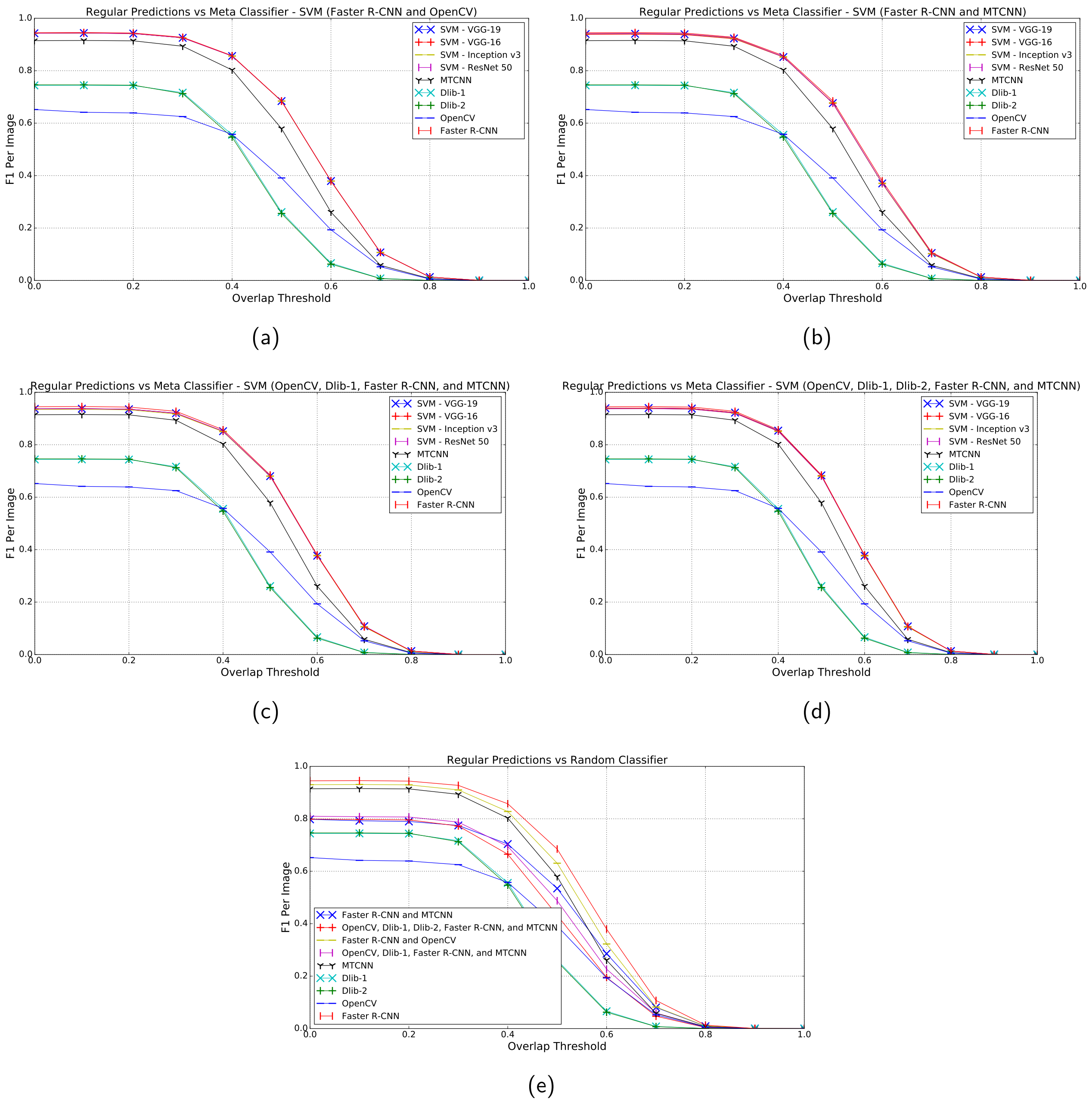


Figure 3.42: Results on the IJB-A dataset with SVM meta classifiers.

scenarios. As the plots are good to show an overview of the methods, we cannot see the subtle differences between Faster R-CNN and the meta classifiers. Therefore, we added some of the results in Table 3.8. We used the F1 score for three different overlap thresholds – 0.4, 0.5, 0.6. As we mentioned throughout this dissertation, the current threshold for face detectors is 0.5, so the choice of this threshold is straightforward. We also chose the other two to help get more intuition of the curve's behavior as the threshold augments. Looking at the results in the table, we can see that the meta classifiers yield results that are really close to the ones yielded by the Faster R-CNN itself. Even though we are now using more than one detector, in this scenario using Faster R-CNN alone seems to be the best choice. We can see that the only scenario where Faster R-CNN has a smaller F1 score is when the threshold is 0.6, but it is only by a small margin. For comparison purposes,

the last line of the table shows results for random classifiers, which use different face detectors, following the different meta training sets. We can notice that the two that have only two classes and one of them is Faster R-CNN (MTCNN and Faster R-CNN, and OpenCV and Faster R-CNN0 yield the largest results.

Table 3.8: Comparison of different models/detectors for given IoU thresholds for IJB-A dataset. The meta classifiers are based on SVMs.

Meta dataset	Detector	F1 with 0.4	F1 with 0.5	F1 with 0.6
-	Faster R-CNN	**0.8574**	**0.6846**	0.3794
OpenCV, Faster R-CNN	SVM VGG-19	0.8561	0.6842	0.3793
OpenCV, Faster R-CNN	SVM VGG-16	0.8561	0.6842	0.3793
OpenCV, Faster R-CNN	SVM Inception v3	0.8559	0.6838	0.3789
OpenCV, Faster R-CNN	SVM ResNet 50	0.8559	0.6843	**0.3798**
MTCNN, Faster R-CNN	SVM VGG-19	0.8521	0.6765	0.3702
MTCNN, Faster R-CNN	SVM VGG-16	0.8523	0.6762	0.3705
MTCNN, Faster R-CNN	SVM Inception v3	0.8531	0.6779	0.3699
MTCNN, Faster R-CNN	SVM ResNet 50	0.8535	0.6773	0.3717
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM VGG-19	0.8517	0.6801	0.3768
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM VGG-16	0.8508	0.6804	0.3769
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM Inception v3	0.8499	0.6798	0.3781
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM ResNet 50	0.8519	0.6810	0.3783
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-19	0.8539	0.6833	0.3768
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-16	0.8528	0.6821	0.3779
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM Inception v3	0.8511	0.6802	0.3759
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM ResNet 50	0.8515	0.6804	0.3777
MTCNN, Faster R-CNN	Random	0.7037	0.5344	0.2856
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	Random	0.6649	0.4321	0.1953
OpenCV, Faster R-CNN	Random	0.8278	0.6306	0.3226
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Random	0.6949	0.4875	0.2268

On the other hand, Figure 3.43 shows the results with the meta classifiers based on neural networks. In this case, we can see that the meta classifiers curves are still above MTCNN. However, they now have a small distance when compared with Faster R-CNN. Since we evaluated the results of the plot itself for the SVM case, we will now focus on the subtle changes in the F1 scores in the same manner we did for SVM. Table 3.9 shows the results. We can see now that Faster R-CNN is consistently better than all other meta classifiers in all the three given thresholds. Even though some methods are very close in the score, for instance the model Neural Network ResNet 50 trained on MTCNN and Faster R-CNN meta dataset, they are still not better than always using Faster R-CNN. In other words, there is no reason why we should execute a meta classifier to output the best face detector, and then execute such face detector, instead of executing Faster R-CNN directly. Regardless of the results, the trained models are still better than the random ones, which indicates that they in fact learned something from the images. As it happened for SVM, the best meta

classifiers are those that only contain two classes and one of them is Faster R-CNN. Hence, we can again conclude that Faster R-CNN results are pushing the classifier results higher.

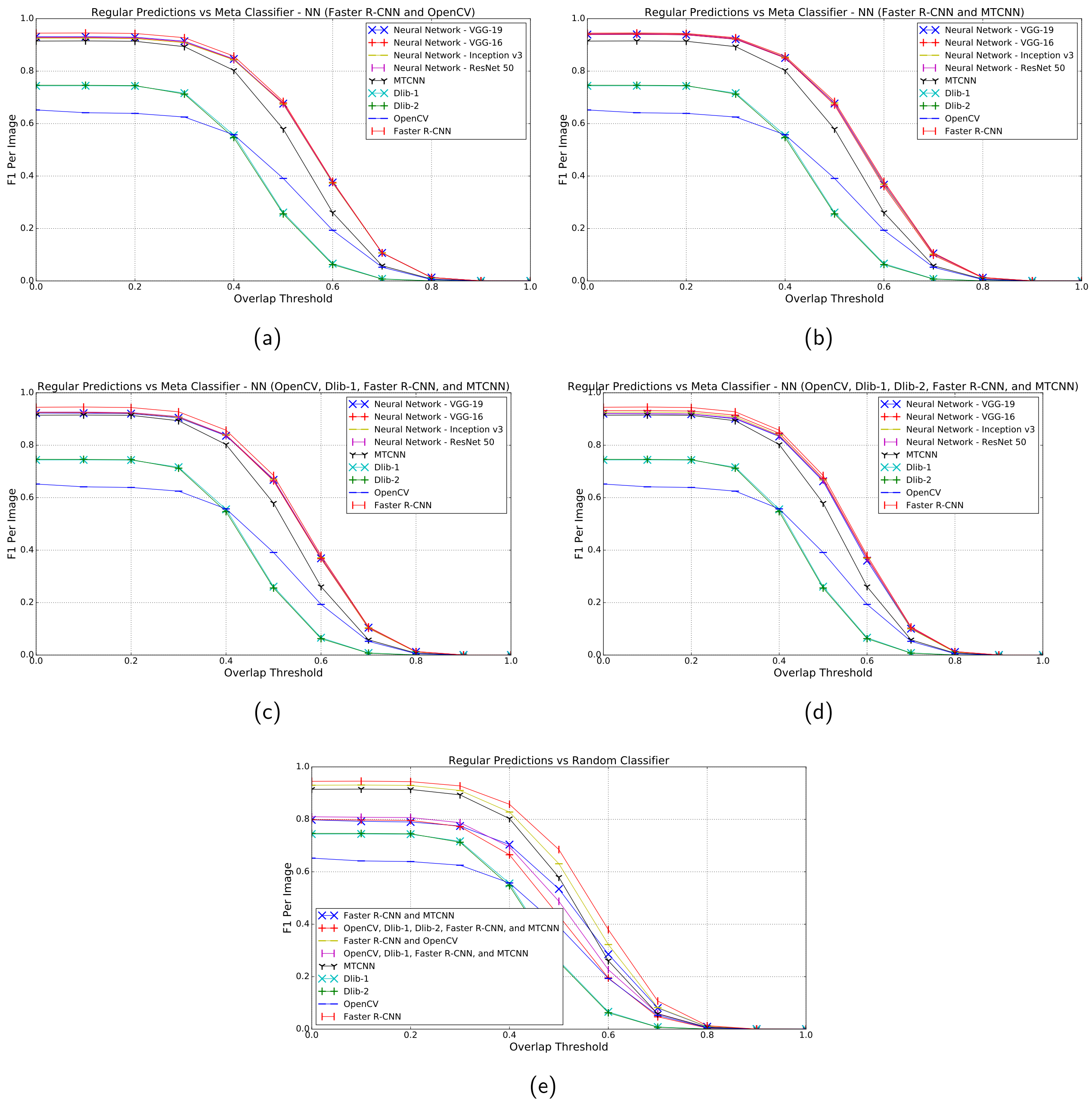


Figure 3.43: Results on the IJB-A dataset with neural networks meta classifiers.

3.5.4.2 FDDB

The same methodology defined for IJB-A dataset experiments and explained throughout this dissertation was used for FDDB as well. We also created two different plots for each of the meta classifier types – neural nets and SVMs – and a table to show the subtle differences. Figure 3.44 shows the results that are based on SVMs. The first aspect we can notice is that MTCNN and Faster R-CNN yield identical results for the first IoU thresholds. However, for thresholds that are larger than 0.5, Faster R-CNN has better F1 scores. The meta classifiers results are very similar to

Table 3.9: Comparison of different models/detectors for given IoU thresholds for IJB-A dataset. The meta classifiers are based on neural networks.

Meta dataset	**Detector**	**F1 with 0.4**	**F1 with 0.5**	**F1 with 0.6**
-	Faster R-CNN	**0.8574**	**0.6846**	**0.3794**
OpenCV, Faster R-CNN	Neural Network VGG-19	0.8464	0.6762	0.3755
OpenCV, Faster R-CNN	Neural Network VGG-16	0.8440	0.6787	0.3757
OpenCV, Faster R-CNN	Neural Network Inception v3	0.8427	0.6740	0.3729
OpenCV, Faster R-CNN	Neural Network ResNet 50	0.8460	0.6756	0.3757
MTCNN, Faster R-CNN	Neural Network VGG-19	0.8503	0.6766	0.3670
MTCNN, Faster R-CNN	Neural Network VGG-16	0.8504	0.6710	0.3602
MTCNN, Faster R-CNN	Neural Network Inception v3	0.8535	0.6785	0.3691
MTCNN, Faster R-CNN	Neural Network ResNet 50	0.8523	0.6775	0.3737
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network VGG-19	0.8362	0.6680	0.3687
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network VGG-16	0.8383	0.6641	0.3674
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network Inception v3	0.8397	0.6707	0.3713
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network ResNet 50	0.8392	0.6704	0.3733
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-19	0.8333	0.6626	0.3598
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-16	0.8458	0.6733	0.3717
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM Inception v3	0.8385	0.6678	0.3690
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM ResNet 50	0.8361	0.6667	0.3682
MTCNN, Faster R-CNN	Random	0.7037	0.5344	0.2856
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	Random	0.6649	0.4321	0.1953
OpenCV, Faster R-CNN	Random	0.8278	0.6306	0.3226
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Random	0.6949	0.4875	0.2268

the ones from Faster R-CNN, and once again are overall better than MTCNN (for IoU thresholds larger than 0.5). The last plot, which is from the random classifiers, shows that randomly choosing between Faster R-CNN and MTCNN yields results that are better than MTCNN alone, contributing to our conclusion that Faster R-CNN pushes the F1 measure to larger values. Table 3.10 contains results from all meta classifiers for thresholds 0.4, 0.5, and 0.6, as we did for IJB-A. The main difference for FDDB is that the best results are not from Faster R-CNN, but for different approaches depending on the specific IoU threshold. For 0.4, we can see that the best result is the random classifier based on OpenCV and Faster R-CNN. We can conclude that almost all images that Faster R-CNN got correct by itself are summed up with a few cases where OpenCV provide correct results but Faster R-CNN did not. This result corroborates with our intuition that the combination of face detectors should be better than a single face detector. However, when we look at the trained meta classifier that is based on such detectors, it yields the exact same result as Faster R-CNN alone. It seems that even though the meta classifier was able to learn how to choose the best face detector, it has a bias towards Faster R-CNN. For the case where IoU threshold is 0.5, the best approach is the meta classifier based on Faster R-CNN and MTCNN that uses Inception v3 features. Even though it is better than Faster R-CNN, the difference is very small — 0.9433 to 0.9407. Finally, for the 0.6 threshold the best result is the meta classifier with all detectors with VGG-16 features. The

difference in this case is even smaller than the previous one — 0.8953 for the meta classifier against 0.8945 for Faster R-CNN. We can argue that Faster R-CNN is the best approach when compared with the SVM meta classifiers.

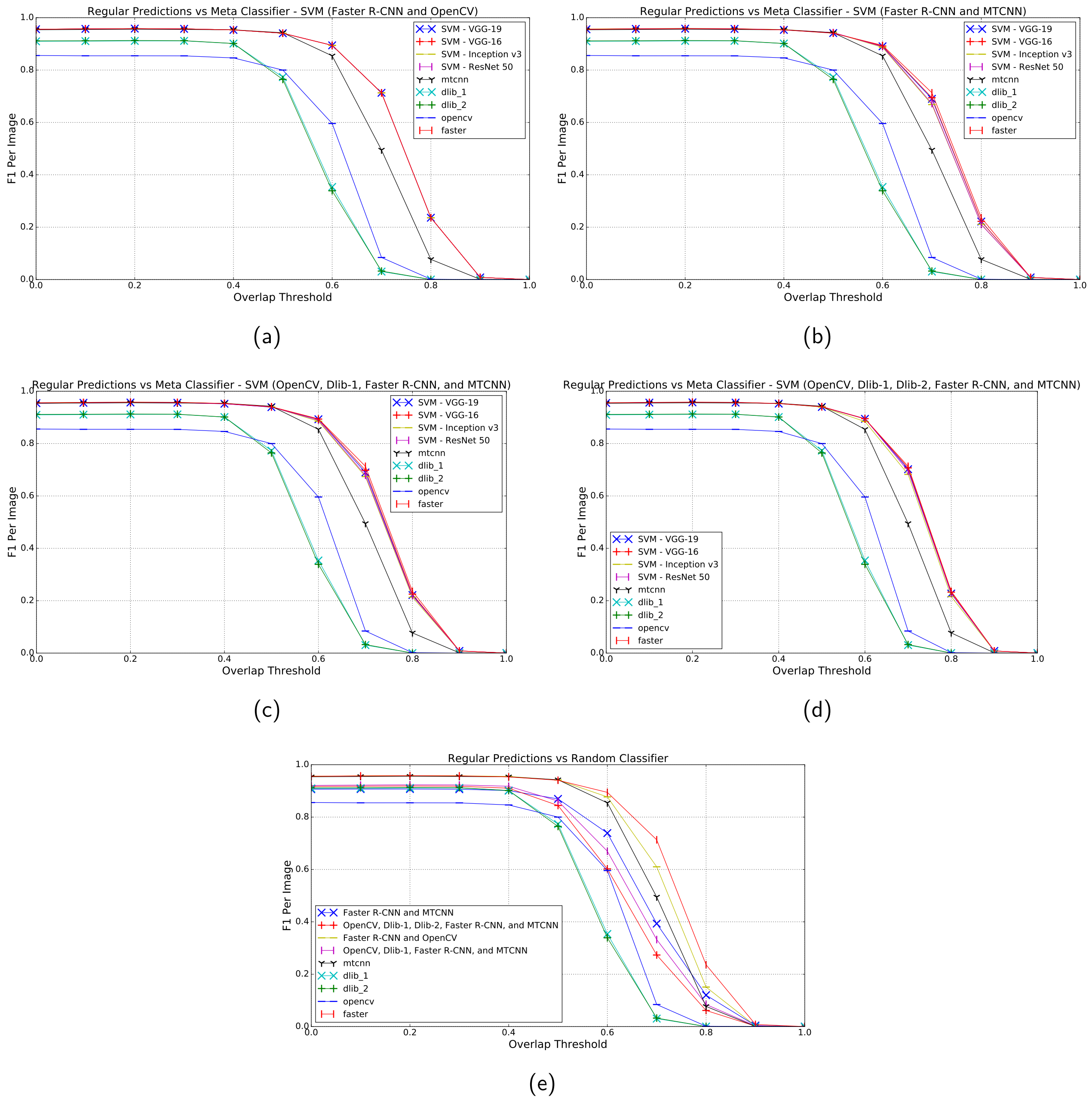


Figure 3.44: Results on FDDB dataset with SVM meta classifiers.

For the neural network meta classifiers, the results for each features are different. Figure 3.45 shows them. As it happened for SVM, the meta classifier approaches are better than MTCNN overall, but only after 0.5 of IoU threshold. In this case, we can notice that there are small differences in the F1 score among the meta classifiers. The F1 score results are detailed in Table 3.11. When the threshold is 0.4, the best result is again a random detector that uses OpenCV and Faster R-CNN as classes. We argue that is due to the same reason as before: it is slightly better than Faster R-CNN because of the combination of both detectors, where OpenCV is better in cases

Table 3.10: Comparison of different models/detectors for given IoU thresholds in the FDDB dataset. The meta classifiers are based on SVMs.

Meta dataset	Detector	F1 with 0.4	F1 with 0.5	F1 with 0.6
-	Faster R-CNN	0.9533	0.9407	0.8945
OpenCV, Faster R-CNN	SVM VGG-19	0.9533	0.9407	0.8945
OpenCV, Faster R-CNN	SVM VGG-16	0.9533	0.9407	0.8941
OpenCV, Faster R-CNN	SVM Inception v3	0.9533	0.9407	0.8945
OpenCV, Faster R-CNN	SVM ResNet 50	0.9533	0.9407	0.8945
MTCNN, Faster R-CNN	SVM VGG-19	0.9534	0.9415	0.8908
MTCNN, Faster R-CNN	SVM VGG-16	0.9540	0.9423	0.8919
MTCNN, Faster R-CNN	SVM Inception v3	0.9547	**0.9433**	0.8867
MTCNN, Faster R-CNN	SVM ResNet 50	0.9540	0.9432	0.8892
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM VGG-19	0.9522	0.9391	0.8929
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM VGG-16	0.9529	0.9402	0.8935
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM Inception v3	0.9527	0.9402	0.8853
OpenCV, Dlib-1, MTCNN, Faster R-CNN	SVM ResNet 50	0.9518	0.9395	0.8891
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-19	0.9526	0.9398	0.8945
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-16	0.9529	0.9412	**0.8953**
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM Inception v3	0.9526	0.9396	0.8843
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM ResNet 50	0.9530	0.9410	0.8952
MTCNN, Faster R-CNN	Random	0.9013	0.8695	0.7389
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	Random	0.9104	0.8443	0.6021
OpenCV, Faster R-CNN	Random	**0.9552**	0.9418	0.8772
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Random	0.9178	0.8614	0.6696

that Faster R-CNN alone is not. Interestingly, the best result for 0.5 is also based on MTCNN and Faster R-CNN, which is similar to the SVM. Nevertheless, it is based on VGG-16 features instead of Inception v3. Finally, the best result for 0.6 threshold is Faster R-CNN, which is almost 0.004 better than the second best result.

3.5.5 Conclusion and Discussion

We executed a large amount of experiments with face detectors, evaluating them individually. In our individual evaluation, the best face detectors were MTCNN and Faster R-CNN, which are both based on deep learning implementations. This result is according to all the literature related to deep learning approaches overcoming traditional ones in different tasks. Our implementation of Faster R-CNN yielded better results than MTCCN in our experiments. However, MTCNN is composed of three light-weight networks and, hence, it requires less computational resources.

We proposed and implemented a meta learning approach, whose goal was to combine different face detectors by choosing the most suitable given an input image. Our idea was that different face detectors were better to specific situations. Our results show that our implementations

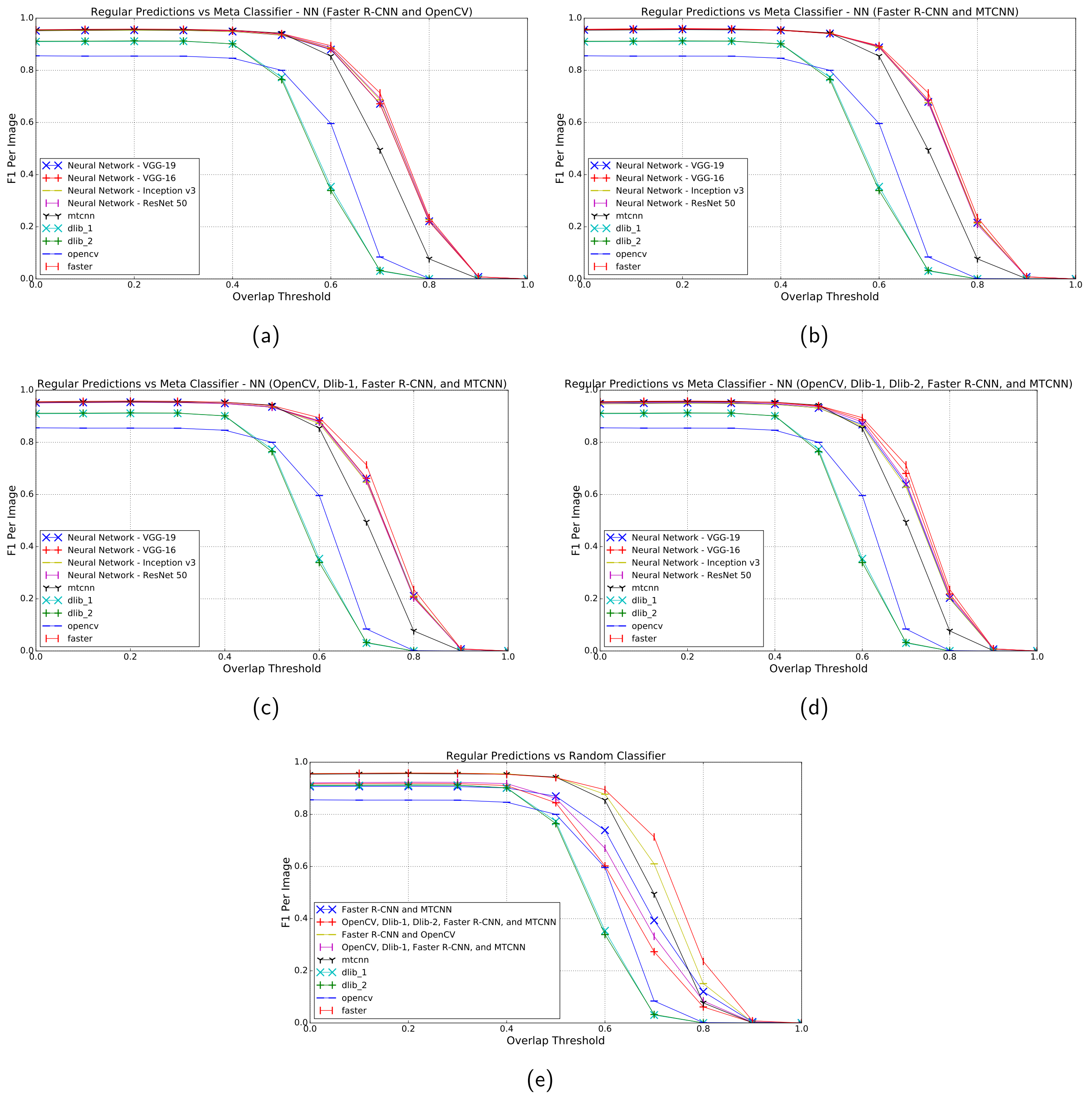


Figure 3.45: Results on FDDB dataset with neural networks meta classifiers.

of meta learning classifiers were not able to learn the best face detector for each situation as well as generalize to unseen examples. Such conclusion is supported by the experiments executed in the IJB-A and FDDB datasets. Additionally, Faster R-CNN yielded results better or equal to all meta learning approaches. Therefore, it is preferable to use Faster R-CNN than the meta classifiers, since the meta classifier approaches we used contain extra steps (e.g. extract features with a given deep learning architecture and classifying with a meta classifier prior to using the face detector directly).

With respect to the meta classifiers themselves, we noticed that our implementations with SVM yielded better results than the ones with Neural Networks in the compared datasets. These results corroborate with SVM's suitability for cases with few instances as our meta datasets. We also saw that features extracted with different deep learning architectures trained on the same dataset

Table 3.11: Comparison of different models/detectors for given IoU thresholds for FDDB dataset. The meta classifiers are based on neural networks.

Meta dataset	Detector	F1 with 0.4	F1 with 0.5	F1 with 0.6
-	Faster R-CNN	0.9533	0.9407	**0.8945**
OpenCV, Faster R-CNN	Neural Network VGG-19	0.9487	0.9350	0.8794
OpenCV, Faster R-CNN	Neural Network VGG-16	0.9498	0.9356	0.8815
OpenCV, Faster R-CNN	Neural Network Inception v3	0.9487	0.9361	0.8863
OpenCV, Faster R-CNN	Neural Network ResNet 50	0.9514	0.9391	0.8875
MTCNN, Faster R-CNN	Neural Network VGG-19	0.9531	0.9410	0.8881
MTCNN, Faster R-CNN	Neural Network VGG-16	0.9549	**0.9432**	0.8914
MTCNN, Faster R-CNN	Neural Network Inception v3	0.9531	0.9413	0.8878
MTCNN, Faster R-CNN	Neural Network ResNet 50	0.9532	0.9416	0.8897
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network VGG-19	0.9495	0.9360	0.8818
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network VGG-16	0.9489	0.9348	0.8825
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network Inception v3	0.9486	0.9356	0.8724
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Neural Network ResNet 50	0.9485	0.9347	0.8780
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-19	0.9464	0.9318	0.8682
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM VGG-16	0.9525	0.9394	0.8858
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM Inception v3	0.9450	0.9320	0.8623
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	SVM ResNet 50	0.9485	0.9366	0.8779
MTCNN, Faster R-CNN	Random	0.9013	0.8695	0.7389
OpenCV, Dlib-1, Dlib-2, MTCNN, Faster R-CNN	Random	0.9104	0.8443	0.6021
OpenCV, Faster R-CNN	Random	**0.9552**	0.9418	0.8772
OpenCV, Dlib-1, MTCNN, Faster R-CNN	Random	0.9178	0.8614	0.6696

(ImageNet) did not yield much difference in the final evaluation. It means that using VGG-16 or Inception v3 for our goal and with the data we used did not matter as much as it matters for other scenarios.

Our experiments show that it is still difficult to extract features trained on a general purpose scenario – image classification on ImageNet – to a complete different scenario – features to determine the best face detector for a given case. We thought that some face detectors would be better than others for, say, crowded images or profile faces. Our experiments showed that features do not suffice for such goal and hence, other type of features could be used instead. As a result of our study and experiments, we were able to enumerate a considerable number of possible improvements that one could make use to explore even more the scenario here presented.

3.6 Final Remarks

In this chapter we presented face detection, explained what it is, and gave an overview of both classic implementations and recent ones (based on deep learning). We introduced hand-crafted approaches of face detection, namely OpenCV and Dlib face detectors. Next, we explained how we

could make use of state-of-the-art deep learning approaches and apply them to face detection. Then, we explained our meta learning approach, where we trained SVMs and neural networks with features extracted with deep learning architectures. We executed experiments on some well-known face detection datasets and evaluated the results of both existing face detection methods – OpenCV, Dlib and MTCNN – and our implementations – Faster R-CNN and meta classifiers. We shown that Faster R-CNN is the best implementation for both datasets, even when compared with meta classifiers.

4
Conclusion

This dissertation presented contributions to face detection with deep neural networks. The face detection field is broadly studied in Computer Vision and the state-of-the-art is currently changing. Hence, we evaluated the evolution of face detection algorithms and the role of deep learning on such improvements.

We started explaining deep learning and one of tis main approaches — Convolutional Neural Networks. We detailed how they work and what is the intuition behind their architecture. We showed how they solve different problems, explaining in details their application for different scenarios, namely classification, detection, and metric learning.

Then, we presented face detection itself, explaining why it is a subject that is important to study as well as some challenges that it offers. We explained how traditional approaches for face detection – as the ones based in Haar-like features and HOG features – work. We also detailed how more recent techniques that are based on deep learning work, using MTCNN as an example. Additionally, we implemented our own deep learning face detector, which is based on Faster R-CNN. Our implementation shows promising results for further investigation in more approaches that are based on deep learning. Our hyper-parameter evaluation contributes to researchers that want to use face detection for any application and wonder which are the most suitable hyper-parameters to use. To the best of knowledge, this is the first work to do such a detailed analysis with the proposed face detectors.

We detailed and implemented a meta learning approach. Such kind of implementation aims to use a combination of algorithms to obtain results better than the algorithms individually. Our main assumption was that each face detector was better at a given set of images. For instance, that Faster R-CNN could be better for crowded images while OpenCV could be better for frontal face images. We created 10 different meta datasets, each of which comprised of different face detectors. We extracted features with different deep neural networks that were all trained in ImageNet. We then trained two types of meta classifiers – SVMs and neural networks. We performed a great amount of experiments, where we varied the meta training sets, the features, and the classifier and its hyper-parameters. To the best of our knowledge, this is the first time deep neural network's features are used to train a meta classifier. Our results suggest that the trained meta classifiers with

the given setup do not improve the results of the best individual face detector. Our experiments show that Faster R-CNN is overall the best face detector when compared with both the individual face detectors and the meta classifier approaches.

4.1 Limitations

One of the first limitations of our work is regarding the criteria we used for selecting the best face detectors to build the meta training sets. Such criteria led to a small number of instances for training and, consequently, problems to train the models. We believe that such problem deeply affected neural networks, as they became more prone to overfitting. We also believe that it affected the generalization capabilities of the trained models, which were trained in a small and limited number of examples.

A second limitation we see is related to the training of the meta classifiers itself. We did not explore all the possibilities related to SVMs and, mainly, neural networks, with respect to hyper-parameters. We believe that a more rigorous analysis could have been made as to have better trained models. However, we argue that such limitation came from the trade-off between large quantity of models and small hyper-parameters exploration versus small quantity of models vs large hyper-parameter exploration.

Another limitation is regarding the execution time of the different approaches we explored and implemented. Even though MTCNN and Faster R-CNN are much better than OpenCV and Dlib implementations, the execution time for running face detection in CPU is very different. The approaches based on deep neural networks have a larger execution time. In addition, the meta learning approaches suffer from a similar problem, as they add an extra layer in the detection flow.

The face detectors have different execution time. We did not performed an analysis with respect to the execution time of the face detectors, since some of them demand more computational resources than others. For instance, Faster R-CNN inference is very fast in GPU but slower on CPU. Hence, we believe that experiments assessing execution time should be performed.

Finally, our evaluation criteria is not the best at hand. We used datasets that are benchmarks for face detection but we did not followed their standard evaluations, which are based on the test sets. Such evaluation would allow us to compare our experiments with the current results on such datasets, which would help us to publish our research. These two datasets have standard submission formats that can be generated and submitted to their evaluation services. Additionally, we based our evaluations on a single type of plot, which is based on F1 score for different IoU thresholds. Even though it is intuitive and helps us understand the behavior of the detectors, there are other options in the literature that could be explored.

4.2 Opportunities for Future Work

In this section, we describe some opportunities that we foresee for future work. They all represent what we discovered while working on this dissertation, as well as ideas that we had based on recent improvements of the state-of-the-art of deep learning. We enumerate opportunities for the face detectors themselves, for the meta learning approach we explored, and for the evaluation methods.

4.2.1 Implement Face Detection based on Different Deep Learning Implementations

We used two different deep learning approaches for face detection: an off-the-shelf implementation called MTCNN, and we trained a Faster R-CNN on the WIDERFACE dataset. Both deep learning implementations yielded the best results overall, which confirms once more their results in other applications within Computer Vision.

Faster R-CNN is only one of the available deep learning object detectors. Hence, we believe that one could explore the application of other models for face detection in the same manner we did for Faster R-CNN. For instance, we can cite YOLO [91] and SSD [76]. The trade-off between accuracy and speed of algorithms is always present regardless of the type of application we are developing. You Only Look Once (YOLO) is an implementation that focuses on speed rather than accuracy, but provides a reasonable accuracy on its detections. There is also an improved version called YOLO9000 [92], which is even faster and more accurate than the previous one. Single Shot MultiBox Detector (SSD) is another deep learning implementation for face detection that uses a single neural network. It contains two versions: SSD300, which is able to process inputs of 300×300; SSD512, which is able to process inputs of 512×512. Both approaches are quite promising as they are very similar to Faster R-CNN in terms of results in object detection. Moreover, they provide open-source implementations [1] [2].

4.2.2 Different Strategies for Feature Extraction

For extracting features from images that are used to train the meta classifiers, we basically used four different convolutional neural network architectures: VGG-16, VGG-19, Inception v3, and ResNet 50. Besides, all were trained on ImageNet dataset.

We believe that such setup is limited, in the sense that we are using architectures trained for a general-purpose task (image classification), and expecting that the networks can generalize well to extract features that are relevant to choose one among known face detectors. Moreover,

[1] YOLO and YOLO9000: http://pjreddie.com/darknet/yolo/
[2] SSD300 and SSD512: https://github.com/weiliu89/caffe/tree/ssd

other deep learning and machine learning approaches could be used for such end. Hence, for each of these situations we envision the following:

- Datasets: Datasets that may capture global information instead of object information are an option, such as Places dataset [133]. Places is a dataset with around 10 million scene photographs, labeled with scene semantic categories and attributes. A network trained on top of such dataset might be able to capture global features of the image and thus help meta classifiers to learn better what represents an image that contains a face easier or harder to detect.
- Deep Learning Algorithms: One option is to use different CNN architectures instead of the four used in this dissertation. For example, ResNet 101 and ResNet 152 [43], and Inception $\mathrm{v}4$ and Inception-ResNet [110]. Another possibility is to explore features that are learned by unsupervised learning algorithms. An example is Generative Adversarial Networks (GANs), which are able to learn useful representations in an unsupervised manner. Attempts to use features learned by GANs for classification problems were proposed and they yielded promising results [88].

One could say convolutional neural networks trained for face recognition could be used for extracting features in this scenario. We argue that features relevant for face recognition might be tied to specific subtleties in the face used to discriminate between different identities. Moreover, the input of such networks is usually aligned and cropped, resulting in a distorted and small image. We believe such transformations may lead to information loss that ends up not helping in the scenario at hand. However, it is also an approach that worth pursuing.

4.2.3 Generate Meta Training Sets based on Alternative Metrics and Strategies

The generation of our meta training sets relied on F1 score, which is calculated based on the resulting detections versus ground truth. Even though it is a well-used and standard measure, in some scenarios it might not be the best strategy. For instance, in cases where the F1 score of two face detectors yields the exact same result, which one should we choose? Depending on the application, one could choose the one that is faster to execute or the one that uses less memory, for example. In our case, we did not consider images where two or more face detectors had the same results. We performed simple experiments (not reported in this dissertation) with all images, using such cases too. To choose the face detector, we randomly choose one of the winning ones, weighting the chance according to the results over the remaining set of images. That is, we run all face detectors and computed the percentage each of them was the best alone. Afterwards, we used such percentage to weight the randomness factor. Our preliminary experiments showed worst accuracies and results for meta classifiers. Therefore, we believe that other criteria besides F1 score could be used like, for instance, a combination of different measures or a combination of F1 with confidence level of the detections.

4.2.4 Train Meta Classifiers in Different Datasets

We trained our meta classifiers only with IJB-A training data. Even though IJB-A is an interesting dataset, it contains few instances. Recently, large datasets became available, such as MegaFace [56] and IMDB-WIKI [94]. MegaFace is a benchmark dataset for face recognition at scale, and it is comprised of one million photos that capture more than $690,000$ different individuals. IMDB-WIKI is dataset targeted for biological age prediction. It contains $524,230$ face images, which were crawled from IMDB and Wikipedia. Both datasets contain bounding boxes for faces and could be used for a scenario as ours. Hence, one could make use of such datasets to both improve the meta classifiers providing more data and as benchmark for comparisons (in the same way FDDB is used).

4.2.5 Evaluation with Larger IoU Thresholds

During all our evaluation process, we evaluate the models based on a fixed IoU threshold of 0.5. It means that for a detection to be considered correct, its area must overlap with more than 50% of the ground truth's area. This threshold seems reasonable, but it affects the precision of the detection with respect to the face. That is, the detected bounding boxes might be shifted a little from the face, which may cause undesirable results depending on the type of application. Additionally, deep learning approaches are trained based on these small thresholds and hence, could be improved as well. We believe that an evolution similar to the one that happened in object detection evaluation might happen here, where the threshold was enlarged from time to time. Nevertheless, it is important to mention that the shift on the detection is not always a problem, which justifies the flexibility currently available.

4.2.6 Implement Light-Weight and Faster Face Detectors

Real-time face detection and recognition is a broadly studied subject. Before the seminal work of Viola Jones, most of the face detection algorithms were not suitable to run in devices with less computational resources. Even though many approaches were able to outperform Viola-Jones in terms of accuracy, there is still room for improvement for real-time face detection that could be used in, for instance, edge devices (low-power devices that can perform local computations instead of going to the cloud). YOLO is one implementation that focused on reducing both the execution time and the computational resources needed. Hence, one approach could be to train YOLO for face detection.

Another alternative that we envision is to work on models that are light-weight. For instance, *XNOR-Net* [90] is an implementation that focuses exactly on this goal. The implementation

is based on a a convolutional neural network for image classification that is trained using binary weights, filters, and inputs. The implementation results in $58\times$ faster convolutional operations and $32\times$ memory savings [90]. Other recent approach is *DoReFa-Net* [134], which is able to handle arbitrary bit-width in weights, activations, and gradients. Therefore, one could implement face detection methods based on deep neural networks with binary representations, aiming to produce light-weight and efficient models.

4.2.7 Evaluate the Impact of Improved Detectors in Face Recognition

Finally, a promising research venue could be to evaluate the impact of improved detectors in face recognition. By improved we mean those based on the suggestions we proposed in this section. The classic face recognition pipeline is comprised of four stages: face detection, face alignment, feature extraction and classification/recognition [46]. Face detection is the first and potentially the most important step of the pipeline. A small change on the precision of a given face detector could lead to alignment of a region that does not cover the whole face, incorrect feature extraction, and finally, poor recognition results. We believe that to assess the impact of these recent face detectors in face recognition could lead to future improvements.

References

[1] "Image scaling using deep convolutional neural networks". Available at: http://engineering.flipboard.com/2015/05/scaling-convnets. Accessed: 2015-11-03.

[2] "Understanding convolution in deep learning". Available at: http://timdettmers.com/2015/03/26/convolution-deep-learning. Accessed: 2015-10-30.

[3] "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups", *Signal Processing Magazine*, vol. 29–6, 2012, pp. 82–97.

[4] Acasandrei, L.; Barriga, A. "Embedded face detection application based on local binary patterns". In: International Conference on Embedded Software and Systems, ICESS 2014, 2014, pp. 641–644.

[5] Ahonen, T.; Hadid, A.; Pietikäinen, M. "Face recognition with local binary patterns". In: European Conference on Computer Vision, ECCV 2004, 2004, pp. 469–481.

[6] Amit, Y. "2D Object Detection and Recognition: Models, Algorithms, and Networks". Cambridge, MA, USA: MIT Press, 2002.

[7] Amodei, D.; Anubhai, R.; Battenberg, E.; Case, C.; Casper, J.; Catanzaro, B.; Chen, J.; Chrzanowski, M.; Coates, A.; Diamos, G.; Elsen, E.; Engel, J.; Fan, L.; Fougner, C.; Han, T.; Hannun, A. Y.; Jun, B.; LeGresley, P.; Lin, L.; Narang, S.; Ng, A. Y.; Ozair, S.; Prenger, R.; Raiman, J.; Satheesh, S.; Seetapun, D.; Sengupta, S.; Wang, Y.; Wang, Z.; Wang, C.; Xiao, B.; Yogatama, D.; Zhan, J.; Zhu, Z. "Deep speech 2: End-to-end speech recognition in english and mandarin", *CoRR*, vol. abs/1512.02595, 2015.

[8] Bartlett, M. S.; Littlewort, G.; Frank, M.; Lainscsek, C.; Fasel, I.; Movellan, J. "Recognizing facial expression: machine learning and application to spontaneous behavior". In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005, 2005, pp. 568–573.

[9] Bell, S.; Bala, K. “Learning visual similarity for product design with convolutional neural networks”, *ACM Transactions on Graphics*, vol. 34–4, 2015, pp. 98:1–98:10.

[10] Berg, T. L.; Berg, A. C.; Edwards, J.; Forsyth, D. A. “Whos in the picture”. In: Neural Information Processing Systems, NIPS 2004, 2004, pp. 137–144.

[11] Bledsoe, W. W.; Chan, H. “A man-machine facial recognition system—some preliminary results”, *Panoramic Research, Inc, Palo Alto, California., Technical Report PRI A*, vol. 19, 1965, pp. 1965.

[12] Bradski, G.; Kaehler, A. “Learning OpenCV: Computer Vision in C++ with the OpenCV Library”. O’Reilly Media, Inc., 2013, 555p.

[13] Bromley, J.; Bentz, J. W.; Bottou, L.; Guyon, I.; LeCun, Y.; Moore, C.; Säckinger, E.; Shah, R. “Signature verification using A "siamese" time delay neural network”, *International Journal of Pattern Recognition and Artificial Intelligence, IJPRAI*, vol. 7–4, 1993, pp. 669–688.

[14] Carcagnì, P.; Coco, M. D.; Mazzeo, P. L.; Testa, A.; Distante, C. “Features descriptors for demographic estimation: A comparative study”. In: Video Analytics for Audience Measurement, VAAM 2014, 2014, pp. 66–85.

[15] Chatfield, K.; Simonyan, K.; Vedaldi, A.; Zisserman, A. “Return of the devil in the details: Delving deep into convolutional nets”, *CoRR*, vol. abs/1405.3531, 2014.

[16] Chopra, S.; Hadsell, R.; LeCun, Y. “Learning a similarity metric discriminatively, with application to face verification”. In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005, 2005, pp. 539–546.

[17] Cortes, C.; Vapnik, V. “Support-vector networks”, *Machine Learning*, vol. 20–3, 1995, pp. 273–297.

[18] Dai, J.; He, K.; Sun, J. “Instance-aware semantic segmentation via multi-task network cascades”. In: Conference on Computer Vision and Pattern Recognition, CVPR 2016, 2016, pp. 3150–3158.

[19] Dalal, N.; Triggs, B. “Histograms of oriented gradients for human detection”. In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005, 2005, pp. 886–893.

[20] de Bruin, T.; Kober, J.; Tuyls, K.; Babuska, R. “Improved deep reinforcement learning for robotics through distribution-based experience retention”. In: International Conference on Intelligent Robots and Systems, IROS 2016, 2016, pp. 3947–3952.

[21] Deng, J.; Dong, W.; Socher, R.; Li, L.; Li, K.; Li, F. “Imagenet: A large-scale hierarchical image database”. In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2009, 2009, pp. 248–255.

[22] Déniz, O.; Bueno, G.; Salido, J.; la Torre, F. D. "Face recognition using histograms of oriented gradients", *Pattern Recognition Letters*, vol. 32–12, 2011, pp. 1598–1603.

[23] Dieleman, S.; Schrauwen, B. "End-to-end learning for music audio". In: International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014, 2014, pp. 6964–6968.

[24] Dollár, P.; Wojek, C.; Schiele, B.; Perona, P. "Pedestrian detection: A benchmark". In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2009, 2009, pp. 304–311.

[25] Domingos, P. M. "A few useful things to know about machine learning", *Communications of the ACM*, vol. 55–10, 2012, pp. 78–87.

[26] Donmez, P. "*Introduction to Machine Learning*, 2nd ed., by ethem alpaydın. cambridge, MA: the MIT press 2010. ISBN: 978-0-262-01243-0. \$54/£ 39.95 + 584 pages", *Natural Language Engineering*, vol. 19–2, 2013, pp. 285–288.

[27] dos Santos, C. N.; Gatti, M. "Deep convolutional neural networks for sentiment analysis of short texts". In: International Conference on Computational Linguistics, COLING 2014, 2014, pp. 69–78.

[28] Everingham, M.; Gool, L. J. V.; Williams, C. K. I.; Winn, J. M.; Zisserman, A. "The pascal visual object classes (VOC) challenge", *International Journal of Computer Vision*, vol. 88–2, 2010, pp. 303–338.

[29] Farabet, C.; Couprie, C.; Najman, L.; LeCun, Y. "Learning hierarchical features for scene labeling", *Transactions on Pattern Analysis and Machine Intelligence*, vol. 35–8, 2013, pp. 1915–1929.

[30] Felzenszwalb, P. F.; Girshick, R. B.; McAllester, D. A.; Ramanan, D. "Object detection with discriminatively trained part-based models", *Transactions on Pattern Analysis and Machine Intelligence*, vol. 32–9, 2010, pp. 1627–1645.

[31] Fischler, M. A.; Elschlager, R. A. "The representation and matching of pictorial structures", *Transactions on Computers*, vol. 22–1, 1973, pp. 67–92.

[32] Foresti, G. L.; Micheloni, C.; Snidaro, L.; Marchiol, C. "Face detection for visual surveillance". In: International Conference on Image Analysis and Processing, ICIAP 2003, 2003, pp. 115–120.

[33] Freund, Y.; Schapire, R. E. "A desicion-theoretic generalization of on-line learning and an application to boosting". In: Computational learning theory, 1995, pp. 23–37.

[34] Girshick, R. B. "Fast R-CNN". In: International Conference on Computer Vision, ICCV 2015, 2015, pp. 1440–1448.

[35] Girshick, R. B.; Donahue, J.; Darrell, T.; Malik, J. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: Conference on Computer Vision and Pattern Recognition, CVPR 2014, 2014, pp. 580–587.

[36] Goodfellow, I. J.; Bengio, Y.; Courville, A. C. "Deep Learning". MIT Press, 2016, 800p.

[37] Goodfellow, I. J.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A. C.; Bengio, Y. "Generative adversarial nets". In: Neural Information Processing Systems, NIPS 2014, 2014, pp. 2672–2680.

[38] Graves, A.; Jaitly, N. "Towards end-to-end speech recognition with recurrent neural networks". In: International Conference on Machine Learning, ICML 2014, 2014, pp. 1764–1772.

[39] Graves, A.; Mohamed, A.; Hinton, G. E. "Speech recognition with deep recurrent neural networks". In: International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, 2013, pp. 6645–6649.

[40] Hadid, A.; Heikkilä, J. Y.; Silvén, O.; Pietikäinen, M. "Face and eye detection for person authentication in mobile phones". In: International Conference on Distributed Smart Cameras, ICDSC 2007, 2007, pp. 101–108.

[41] Hadsell, R.; Chopra, S.; LeCun, Y. "Dimensionality reduction by learning an invariant mapping". In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2006, 2006, pp. 1735–1742.

[42] He, K.; Zhang, X.; Ren, S.; Sun, J. "Spatial pyramid pooling in deep convolutional networks for visual recognition". In: European Conference on Computer Vision, ECCV 2014, 2014, pp. 346–361.

[43] He, K.; Zhang, X.; Ren, S.; Sun, J. "Deep residual learning for image recognition". In: Conference on Computer Vision and Pattern Recognition, CVPR 2016, 2016, pp. 770–778.

[44] He, K.; Zhang, X.; Ren, S.; Sun, J. "Identity mappings in deep residual networks". In: European Conference on Computer Vision, ECCV 2016, Leibe, B.; Matas, J.; Sebe, N.; Welling, M. (Editors), 2016, pp. 630–645.

[45] Hjelmås, E.; Low, B. K. "Face detection: A survey", *Computer Vision and Image Understanding*, vol. 83–3, 2001, pp. 236–274.

[46] Hu, G.; Yang, Y.; Yi, D.; Kittler, J.; Christmas, W. J.; Li, S. Z.; Hospedales, T. M. "When face recognition meets with deep learning: An evaluation of convolutional neural networks for face recognition". In: International Conference on Computer Vision Workshop, ICCV 2015, 2015, pp. 384–392.

[47] Huang, G. B.; Ramesh, M.; Berg, T.; Learned-Miller, E. "Labeled faces in the wild: A database for studying face recognition in unconstrained environments", Technical Report 07-49, University of Massachusetts, Amherst, 2007.

[48] Hubel, D. H.; Wiesel, T. N. "Receptive fields and functional architecture of monkey striate cortex", *Journal of Physiology (London)*, vol. 195, 1968, pp. 215–243.

[49] Ioffe, S.; Szegedy, C. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: International Conference on Machine Learning, ICML 2015, 2015, pp. 448–456.

[50] Jain, V.; Learned-Miller, E. "Fddb: A benchmark for face detection in unconstrained settings", Technical Report UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[51] Ji, S.; Xu, W.; Yang, M.; Yu, K. "3d convolutional neural networks for human action recognition", *Transactions on Pattern Analysis and Machine Intelligence*, vol. 35–1, 2013, pp. 221–231.

[52] Jiang, H.; Learned-Miller, E. G. "Face detection with the faster R-CNN", *CoRR*, vol. abs/1606.03473, 2016.

[53] Joulin, A.; Grave, E.; Bojanowski, P.; Mikolov, T. "Bag of tricks for efficient text classification", *CoRR*, vol. abs/1607.01759, 2016.

[54] Kaelbling, L. P.; Littman, M. L.; Moore, A. W. "Reinforcement learning: A survey", *Journal of Artificial Intelligence Research*, vol. 4, 1996, pp. 237–285.

[55] Kawulok, M.; Celebi, M. E.; Smolka, B. "Advances in Face Detection and Facial Image Analysis". Springer, 2016, 434p.

[56] Kemelmacher-Shlizerman, I.; Seitz, S. M.; Miller, D.; Brossard, E. "The megaface benchmark: 1 million faces for recognition at scale". In: Conference on Computer Vision and Pattern Recognition, CVPR 2016, 2016, pp. 4873–4882.

[57] Kemelmacher-Shlizerman, I.; Shechtman, E.; Garg, R.; Seitz, S. M. "Exploring photobios", *ACM Transactions on Graphics*, vol. 30–4, 2011, pp. 61:1–61:10.

[58] King, D. E. "Dlib-ml: A machine learning toolkit", *Journal of Machine Learning Research*, vol. 10, 2009, pp. 1755–1758.

[59] Klare, B. F.; Klein, B.; Taborsky, E.; Blanton, A.; Cheney, J.; Allen, K.; Grother, P.; Mah, A.; Burge, M. J.; Jain, A. K. "Pushing the frontiers of unconstrained face detection and recognition: IARPA janus benchmark A". In: Conference on Computer Vision and Pattern Recognition, CVPR 2015, 2015, pp. 1931–1939.

[60] Krizhevsky, A.; Sutskever, I.; Hinton, G. E. "Imagenet classification with deep convolutional neural networks". In: Neural Information Processing Systems, NIPS 2012, 2012, pp. 1106–1114.

[61] Kulis, B. "Metric learning: A survey", *Foundations and Trends in Machine Learning*, vol. 5–4, 2013, pp. 287–364.

[62] Kumar, A.; Irsoy, O.; Su, J.; Bradbury, J.; English, R.; Pierce, B.; Ondruska, P.; Gulrajani, I.; Socher, R. "Ask me anything: Dynamic memory networks for natural language processing", *CoRR*, vol. abs/1506.07285, 2015.

[63] Lazebnik, S.; Schmid, C.; Ponce, J. "Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories". In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2006, 2006, pp. 2169–2178.

[64] LeCun, Y.; Bengio, Y.; Hinton, G. E. "Deep learning", *Nature*, vol. 521–7553, 2015, pp. 436–444.

[65] LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. "Gradient-based learning applied to document recognition". In: Intelligent Signal Processing, 2001, pp. 306–351.

[66] LeCun, Y.; Chopra, S.; Hadsell, R.; Ranzato, M.; Huang, F. "A tutorial on energy-based learning", *Predicting structured data*, vol. 1, 2006, pp. 60.

[67] LeCun, Y.; Kavukcuoglu, K.; Farabet, C. "Convolutional networks and applications in vision". In: International Symposium on Circuits and Systems, ISCAS 2010, 2010, pp. 253–256.

[68] Lee, H.; Grosse, R. B.; Ranganath, R.; Ng, A. Y. "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations". In: International Conference on Machine Learning, ICML 2009, 2009, pp. 609–616.

[69] Lenc, K.; Vedaldi, A. "R-CNN minus R". In: British Machine Vision Conference 2015, BMVC 2015, 2015, pp. 5.1–5.12.

[70] Li, F.-F.; Karpathy, A.; Johnson, J. "Cs231n: Convolutional neural networks for visual recognition, available at: http://cs231n.github.io/", June 2015.

[71] Liao, S.; Zhu, X.; Lei, Z.; Zhang, L.; Li, S. Z. "Learning multi-scale block local binary patterns for face recognition". In: International Conference on Advances in Biometrics, ICB 2007, 2007, pp. 828–837.

[72] Lienhart, R.; Maydt, J. "An extended set of haar-like features for rapid object detection". In: International Conference on Image Processing, ICIP 2002, 2002, pp. 900–903.

[73] Lin, M.; Chen, Q.; Yan, S. "Network in network", *CoRR*, vol. abs/1312.4400, 2013.

[74] Lin, T.; Maire, M.; Belongie, S. J.; Bourdev, L. D.; Girshick, R. B.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C. L. "Microsoft COCO: common objects in context", *CoRR*, vol. abs/1405.0312, 2014.

[75] Lin, Y.; Lv, F.; Zhu, S.; Yang, M.; Cour, T.; Yu, K.; Cao, L.; Huang, T. S. "Large-scale image classification: Fast feature extraction and SVM training". In: Conference on Computer Vision and Pattern Recognition, CVPR 2011, 2011, pp. 1689–1696.

[76] Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S. E.; Fu, C.; Berg, A. C. "SSD: single shot multibox detector". In: European Conference on Computer Vision, ECCV 2016, 2016, pp. 21–37.

[77] Lowe, D. G. "Object recognition from local scale-invariant features". In: International Conference on Computer Vision, ICCV 1999, 1999, pp. 1150–1157.

[78] Mitchell, T. M. "Machine learning". McGraw-Hill, 1997, 432p.

[79] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. A. "Playing atari with deep reinforcement learning", *CoRR*, vol. abs/1312.5602, 2013.

[80] Mohan, A.; Jayabalan, S.; Mohan, A. "Autonomous quantum reinforcement learning for robot navigation". In: International Conference on Intelligent Computing and Applications, ICICA 2017, 2017, pp. 351–357.

[81] Nielsen, M. A., "Neural networks and deep learning", 2015, book in preparation, Source: http://neuralnetworksanddeeplearning.com.

[82] Ojala, T.; Pietikäinen, M.; Harwood, D. "A comparative study of texture measures with classification based on featured distributions", *Pattern Recognition*, vol. 29–1, 1996, pp. 51–59.

[83] Papageorgiou, C.; Oren, M.; Poggio, T. A. "A general framework for object detection". In: International Conference on Computer Vision, ICCV 1998, 1998, pp. 555–562.

[84] Parkhi, O. M.; Vedaldi, A.; Zisserman, A. "Deep face recognition". In: British Machine Vision Conference 2015, BMVC 2015, 2015, pp. 41.1–41.12.

[85] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; VanderPlas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E. "Scikit-learn: Machine learning in python", *CoRR*, vol. abs/1201.0490, 2012.

[86] Perronnin, F.; Liu, Y.; Sánchez, J.; Poirier, H. "Large-scale image retrieval with compressed fisher vectors". In: Conference on Computer Vision and Pattern Recognition, CVPR 2010, 2010, pp. 3384–3391.

[87] Pietikäinen, M.; Hadid, A.; Zhao, G.; Ahonen, T. "Computer Vision Using Local Binary Patterns". Springer, 2011, *Computational Imaging and Vision*, vol. 40.

[88] Radford, A.; Metz, L.; Chintala, S. "Unsupervised representation learning with deep convolutional generative adversarial networks", *CoRR*, vol. abs/1511.06434, 2015.

[89] Ranjan, R.; Patel, V. M.; Chellappa, R. "Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition", *CoRR*, vol. abs/1603.01249, 2016.

[90] Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: European Conference on Computer Vision, ECCV 2016, 2016, pp. 525–542.

[91] Redmon, J.; Divvala, S. K.; Girshick, R. B.; Farhadi, A. "You only look once: Unified, real-time object detection". In: Conference on Computer Vision and Pattern Recognition, CVPR 2016, 2016, pp. 779–788.

[92] Redmon, J.; Farhadi, A. "YOLO9000: better, faster, stronger", *CoRR*, vol. abs/1612.08242, 2016.

[93] Ren, S.; He, K.; Girshick, R. B.; Sun, J. "Faster R-CNN: towards real-time object detection with region proposal networks". In: *Neural Information Processing Systems, NIPS 2015*, 2015, pp. 91–99.

[94] Rothe, R.; Timofte, R.; Gool, L. J. V. "DEX: deep expectation of apparent age from a single image". In: International Conference on Computer Vision, ICCV 2015, 2015, pp. 252–257.

[95] Rui, Y.; Huang, T. S.; Chang, S. "Image retrieval: Current techniques, promising directions, and open issues", *J. Visual Communication and Image Representation*, vol. 10–1, 1999, pp. 39–62.

[96] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J. "Learning representations by back-propagating errors", *Cognitive modeling*, vol. 5–3, 1988, pp. 1.

[97] Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M. S.; Berg, A. C.; Li, F. "Imagenet large scale visual recognition challenge", *International Journal of Computer Vision*, vol. 115–3, 2015, pp. 211–252.

[98] Sainath, T. N.; Mohamed, A.; Kingsbury, B.; Ramabhadran, B. "Deep convolutional neural networks for LVCSR". In: International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, 2013, pp. 8614–8618.

[99] Sak, H.; Senior, A. W.; Rao, K.; Beaufays, F. "Fast and accurate recurrent neural network acoustic models for speech recognition". In: Conference of the International Speech Communication Association, INTERSPEECH 2015, 2015, pp. 1468–1472.

[100] Sakai, T.; Nagao, M.; Kanade, T. "Computer analysis and classification of photographs of human faces". Kyoto University, 1972.

[101] Schmidhuber, J. "Deep learning in neural networks: An overview", *Neural Networks*, vol. 61, 2015, pp. 85–117.

[102] Schroff, F.; Kalenichenko, D.; Philbin, J. "Facenet: A unified embedding for face recognition and clustering". In: Conference on Computer Vision and Pattern Recognition, CVPR 2015, 2015, pp. 815–823.

[103] Shelhamer, E.; Long, J.; Darrell, T. "Fully convolutional networks for semantic segmentation", *Transactions on Pattern Analysis and Machine Intelligence*, vol. 39–4, 2017, pp. 640–651.

[104] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; Hassabis, D. "Mastering the game of go with deep neural networks and tree search", *Nature*, vol. 529–7587, 2016, pp. 484–489.

[105] Simonyan, K.; Zisserman, A. "Very deep convolutional networks for large-scale image recognition", *CoRR*, vol. abs/1409.1556, 2014.

[106] Sobel, I.; Feldman, G. "A 3x3 isotropic gradient operator for image processing", *a talk at the Stanford Artificial Project in*, 1968, pp. 271–272.

[107] Srivastava, N.; Hinton, G. E.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. "Dropout: a simple way to prevent neural networks from overfitting", *Journal of Machine Learning Research*, vol. 15–1, 2014, pp. 1929–1958.

[108] Sun, Y.; Liang, D.; Wang, X.; Tang, X. "Deepid3: Face recognition with very deep neural networks", *CoRR*, vol. abs/1502.00873, 2015.

[109] Sutskever, I.; Vinyals, O.; Le, Q. V. "Sequence to sequence learning with neural networks". In: Neural Information Processing Systems, NIPS 2014, 2014, pp. 3104–3112.

[110] Szegedy, C.; Ioffe, S.; Vanhoucke, V.; Alemi, A. A. "Inception-v4, inception-resnet and the impact of residual connections on learning". In: Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence, AAAI 2017, 2017, pp. 4278–4284.

[111] Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. "Going deeper with convolutions", *CoRR*, vol. abs/1409.4842, 2014.

[112] Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. "Rethinking the inception architecture for computer vision", *CoRR*, vol. abs/1512.00567, 2015.

[113] Taigman, Y.; Yang, M.; Ranzato, M.; Wolf, L. "Deepface: Closing the gap to human-level performance in face verification". In: Conference on Computer Vision and Pattern Recognition, CVPR 2014, 2014, pp. 1701–1708.

[114] Tan, P.-N.; et al.. "Introduction to data mining". Pearson Education India, 2006, 769p.

[115] Uijlings, J. R. R.; van de Sande, K. E. A.; Gevers, T.; Smeulders, A. W. M. "Selective search for object recognition", *International Journal of Computer Vision*, vol. 104–2, 2013, pp. 154–171.

[116] van de Sande, K. E. A.; Uijlings, J. R. R.; Gevers, T.; Smeulders, A. W. M. "Segmentation as selective search for object recognition". In: International Conference on Computer Vision, ICCV 2011, 2011, pp. 1879–1886.

[117] van den Oord, A.; Dieleman, S.; Schrauwen, B. "Deep content-based music recommendation". In: Neural Information Processing Systems, NIPS 2013, 2013, pp. 2643–2651.

[118] Vatamanu, O. A.; Frandes, M.; Lungeanu, D.; Mihalas, G. "Content based image retrieval using local binary pattern operator and data mining techniques". In: Medical Informatics Europe, MIE 2015, 2015, pp. 75–79.

[119] Vilalta, R.; Drissi, Y. "A perspective view and survey of meta-learning", *Artificial Intelligence Review*, vol. 18–2, 2002, pp. 77–95.

[120] Viola, P. A.; Jones, M. J. "Rapid object detection using a boosted cascade of simple features". In: Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2001, 2001, pp. 511–518.

[121] Wang, J. Z.; Li, J.; Wiederhold, G. "Simplicity: Semantics-sensitive integrated matching for picture libraries", *Transactions on Pattern Analysis and Machine Intelligence*, vol. 23–9, 2001, pp. 947–963.

[122] Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M.; de Freitas, N. "Dueling network architectures for deep reinforcement learning". In: International Conference on Machine Learning, ICML 2016, 2016, pp. 1995–2003.

[123] Weston, J.; Bordes, A.; Chopra, S.; Mikolov, T. "Towards ai-complete question answering: A set of prerequisite toy tasks", *CoRR*, vol. abs/1502.05698, 2015.

[124] Xiong, W.; Droppo, J.; Huang, X.; Seide, F.; Seltzer, M.; Stolcke, A.; Yu, D.; Zweig, G. "Achieving human parity in conversational speech recognition", *CoRR*, vol. abs/1610.05256, 2016.

[125] Xiong, Y.; Zhu, K.; Lin, D.; Tang, X. "Recognize complex events from static images by fusing deep channels". In: Conference on Computer Vision and Pattern Recognition, CVPR 2015, 2015, pp. 1600–1609.

[126] Xu, K.; Ba, J.; Kiros, R.; Cho, K.; Courville, A. C.; Salakhutdinov, R.; Zemel, R. S.; Bengio, Y. "Show, attend and tell: Neural image caption generation with visual attention". In: International Conference on Machine Learning, ICML 2015, 2015, pp. 2048–2057.

[127] Yang, M.; Kriegman, D. J.; Ahuja, N. "Detecting faces in images: A survey", *Transactions on Pattern Analysis and Machine Intelligence*, vol. 24–1, 2002, pp. 34–58.

[128] Yang, S.; Luo, P.; Loy, C. C.; Tang, X. "WIDER FACE: A face detection benchmark". In: Conference on Computer Vision and Pattern Recognition, CVPR 2016, 2016, pp. 5525–5533.

[129] Yang, Z.; Nevatia, R. "A multi-scale cascade fully convolutional network face detector", *CoRR*, vol. abs/1609.03536, 2016.

[130] Zafeiriou, S.; Zhang, C.; Zhang, Z. "A survey on face detection in the wild: Past, present and future", *Computer Vision and Image Understanding*, vol. 138, 2015, pp. 1–24.

[131] Zhang, G.; Huang, X.; Li, S. Z.; Wang, Y.; Wu, X. "Boosting local binary pattern (lbp)-based face recognition". In: Chinese Conference on Biometric Recognition, SINOBIOMETRICS 2004, 2004, pp. 179–186.

[132] Zhang, K.; Zhang, Z.; Li, Z.; Qiao, Y. "Joint face detection and alignment using multitask cascaded convolutional networks", *Signal Processing Letters*, vol. 23–10, 2016, pp. 1499–1503.

[133] Zhou, B.; Khosla, A.; Lapedriza, À.; Torralba, A.; Oliva, A. "Places: An image database for deep scene understanding", *CoRR*, vol. abs/1610.02055, 2016.

[134] Zhou, S.; Ni, Z.; Zhou, X.; Wen, H.; Wu, Y.; Zou, Y. "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients", *CoRR*, vol. abs/1606.06160, 2016.

[135] Zitnick, C. L.; Dollár, P. "Edge boxes: Locating object proposals from edges". In: European Conference on Computer Vision, ECCV 2014, 2014, pp. 391–405.