UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU DE ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

COMPARING YOLO AND MIXNET ARCHITECTURES FOR IMAGE-BASED HUMAN DETECTION

Autor: Lukaz Martin Doehne

Director: Dr. Meysam Madadi
Realitzat a: Departament de Matemàtiques i Informàtica
Barcelona, 20 de juny de 2020

Contents

1. INTRODUCTION, MOTIVATION & GOALS

Object detection is a technique that allows computers to identify objects in images or videos. The technique most commonly used for this operation is called Convolutional Neural Network (CNN), because of its good performance.

Object detection had a big impact in the last two decades, because of its wide range of industries where it can be applied. Among which we can find autonomous driving where cars have to decide by their own when to accelerate, turn ,brake… face detection which can be used for unlocking phones or surveillance among others, object extraction of images, personal identification through iris code, smile detection for cameras, medical image processing tools and many more. We see the importance of finding ways to improve the way we teach computers to understand images, so we can have autonomous machines that are more accurate and reliable.

Our goal in this project is to study the performance of different architecture designs and techniques in the task of object detection. This thesis could help as a guide for future projects to observe how changes with data augmentation and different architecture designs can affect their model.

Convolutional Neural Networks (CNN) are one of the most promising branches of deep learning, not only because of its wide range of possible applications but also because of its scalability, performance and adaptability. Additionally, the way CNN algorithms understand objects is very interesting and leaves room to implementing new techniques. Studies showed models with a rate on face recognition of 98.3% for a dataset of 400 subjects [1]. On the CIFAR-10 dataset, which consists of 60.000 images with 10 classes and the objective is to categorize every image to their respective class, models achieved success rate of 99.37% [2,3].

With the advancements in technology, new image-based object detection techniques where originated, one of the pioneer techniques where Region-based CNNs (R-CNN) [10], which later was overcome by Faster-CNN [11]. The same year

a new object detection technique was developed (SSD [12]) which outperformed all existing algorithms. Later, another algorithm was developed with a different approach, called YOLO [6], which latest version (YOLOv3 [9]) outperformed in accuracy Faster-CNN and SSD.

Despite of the extraordinary progress made in hardware and software to detect objects in images, it is still a challenging task to achieve a reliable autonomous system for recognition. Therefore, it is very important to keep improving the algorithms that are being used, so they can be more accurate without sacrificing time by incrementing needlessly the computation that the machines are making.

In this 7-month project, we built a neural network from scratch for detecting hand written digits [5] to study different CNN architectures and designs. We implemented the Mixnet architecture [4] to our model, which combines different kernel sizes instead of the traditional one sized kernel, collected the results after training and compared the model with a version of YOLO. We worked with different types of data augmentation: horizontal flipping, cropping, scaling, rotation and 2 new types of data augmentation (Masking humans and Masking not humans). Finally, we trained our network with data augmentation and participated in an international competition [8] to analyze the results. Our start point for this project, was an already working image detection algorithm.

2. RELATED WORK

Faster-CNN: By 2016, this technique was one with the best performance in accuracy, obtaining the best mean Average Precision in the COCO dataset [13] with an accuracy of 41.3% [14]. Instead of using Selective Search, which uses the image structure to search for shapes and performs an exhaustive search as R-CNN [10], Faster-CNN is using Region Proposal Network (RPN) [11]. In the PASCAL VOC 2012 dataset [15], it obtained an accuracy of 75.9%. This algorithm showed a small accuracy advantage over SSD if real-time speed is not needed.

Single Shot Detection (SSD): This algorithm [12] can outperform Faster-CNN in accuracy with larger objects and speed, but performs worse on small objects and if real-time detection is not required. By testing with the PASCAL VOC 2012 dataset [15], it obtained a mean Average Precision of 82.2%, which is 6.3% more than Faster-CNN achieved. It is called Single Shot Detection because, as in YOLO [6], it's able to find objects in an image in one shot, and doesn't have to look at multiple regions of the image as R-CNN or Faster-CNN [10,11].

3. NEURAL NETWORK

Neural network is a computational model used primarily in deep learning for image, video or voice recognition.

3.1 Introduction to Neural Network

Artificial Neural Networks are inspired by how biological neural networks in the human brain process information. A human brain contains around 86 billion neurons, since working with this number of neurons would be too expensive for any computer, Artificial Neural Networks emulate the behavior with fewer neurons.

A Neural Network consists of several connected layers that apply convolutional filters of one or more dimensions. The layers are one input layer, multiple hidden layers and one output layer. A layer is formed by neurons, and a neuron is basically a variable that contains a numerical value. In case of image recognition, a neuron from the input layer would be a pixel.

3.2 Architecture of Neural Networks

A Neural Network processes all the values of the input and transforms them to an output value. To do this, each neuron in the input layer will be connected through a weight to each neuron in the next layer. The weight that connects two neurons is a number that represents the influence the first neuron has on the second. After this, we apply an activation function (e.g. ReLU or Sigmoid) to add non-linearity to our model so that it can solve more complex tasks.

Let's take a look at a possible input layer. In *Figure 1*, we have an example of an input image which has 3 pixels height and 3 pixels width and 3 channels (Red, Green and Blue), making a total of 27 values for the image (or 27 neurons for the input layer). Each of these values represents the amount of color of their channel that is in this position. When these 3 channels overlap, we see the image how we are used to see it. This input image will form the first layer (input layer).

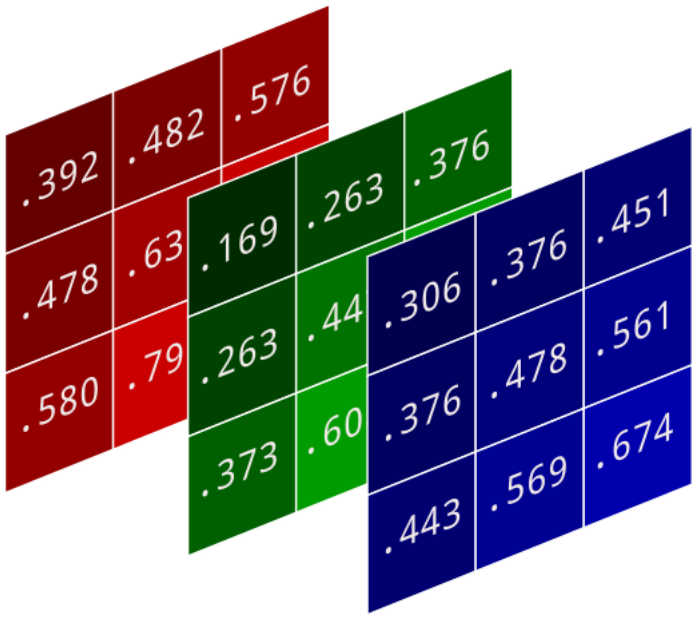


Figure 1. Example of an RGB input image. [1]

[1] https://brohrer.github.io/convert_rgb_to_grayscale.html

The second layer will be formed by neurons computed by the first layer. The output value of one neuron in the second layer will be each neuron in the first layer multiplied by the weight that connects both neurons. Also, we add a bias that determines how high the value of a neuron should be to be meaningful active.

Finally, we use the activation function. This operation will go until we reach the output layer, where the neuron with the highest activation or value will be the one the model predicts as the correct output.

Basically, each neuron in one layer will be multiplied by its associated weight and a bias will be added to form a neuron of the next layer. Then we apply the activation function. We show a neuron computation in equation (1) where *w* represents the weight, *a* is a vector of neurons in the previous layer, *b* is the bias and σ is the activation function.

$$\sigma\big((\boldsymbol{w_1 a_1} + \boldsymbol{w_2 a_2} + \cdots + \boldsymbol{w_n a_n}) + \boldsymbol{b}\big) \qquad (\mathbf{1})$$

Sigmoid and ReLU, two activation functions commonly used for neural networks, are shown below.

Sigmoid:

The sigmoid function normalizes the input to a range between 0 and 1, which uses the equation (2). We can observe an example at *Figure 2(a)*.

$$\boldsymbol{S(x)} = \frac{\mathbf{1}}{\mathbf{1} + \boldsymbol{e^{-x}}} \qquad (\mathbf{2})$$

Rectified Linear Units (ReLU):

This activation function is currently the most popular for deep neural networks. This function transforms the negative inputs to 0, for larger values we get a linear function. We can see the behavior in equation (3) and *Figure 2(b)*.

$$\boldsymbol{f(x)} = \mathbf{max}\,(\mathbf{0}, \boldsymbol{x}) \qquad (\mathbf{3})$$

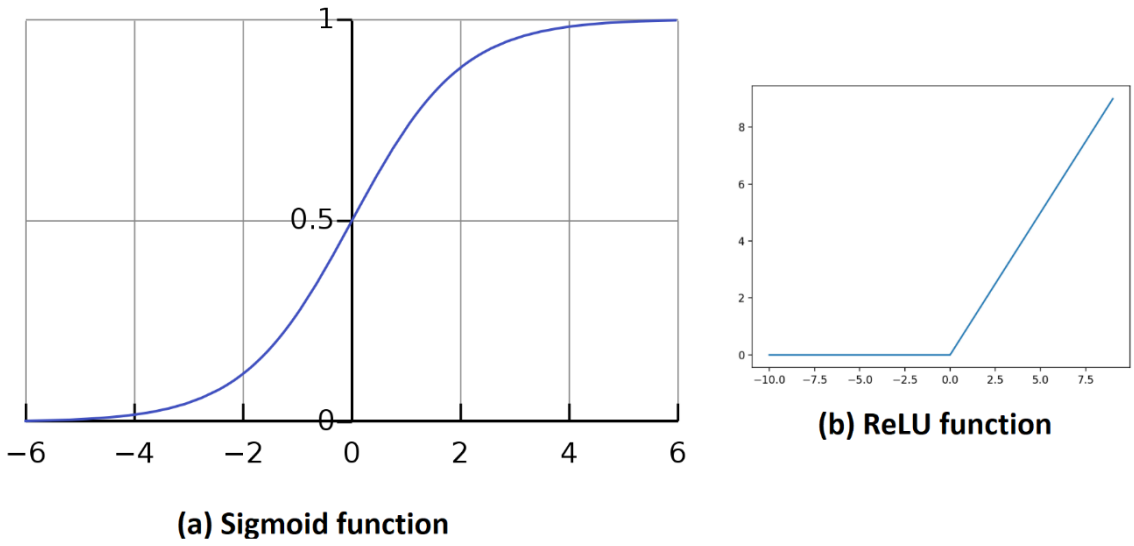


Figure 2. Sigmoid function (left) and Rectified Linear Units function (right). [2] [3]

In *Figure 3*, we have a simple neural network with 3 layers. If we take the image in *Figure 1* as our input layer after flattening (converting the data into a 1 dimensional array), and assume for this example that the second layer has a total of 10 neurons and the output layer has 5 neurons, we get a total of 335 parameters (27x10 + 10 + 10x5 +5).

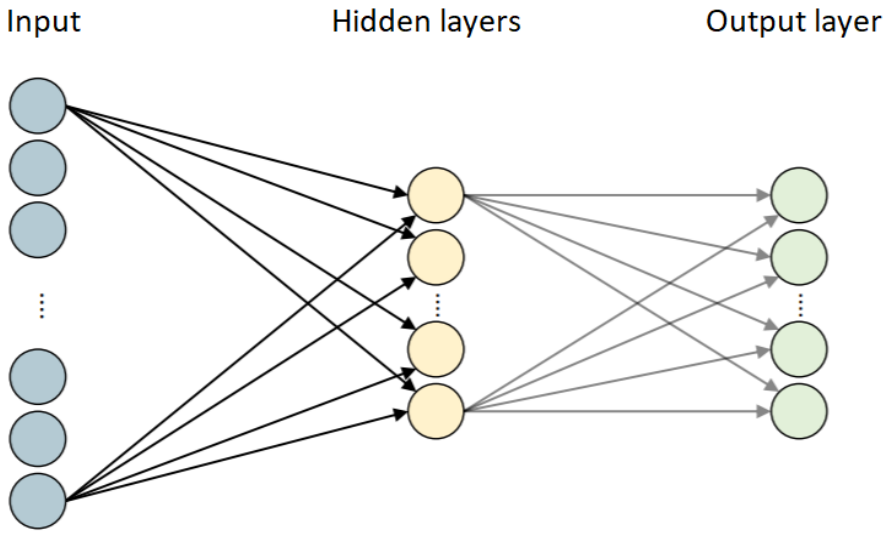


Figure 3. Example of a basic Neural Network. [4]

[2] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/
[3] https://en.wikipedia.org/wiki/Sigmoid_function
[4] http://www.mi.uni-koeln.de/wp-znikolic/wp-content/uploads/2019/06/11-Odenthal.pdf

Finally, we will see a technique to avoid overfitting. Overfitting occurs when our model is learning to correctly recognize the images given in the training, but memorizing them rather than learning the shape of the objects, so when it's given a new test set, it won't perform as well as with the training data. Our goal to avoid overfitting is that our model is not so depend on individual neurons, but rather more dependent on the collective of neurons. We accomplish this with a dropout layer.

A dropout layer ignores a given percentage of the neurons in the layer during the learning phase. Ignored neurons are randomly selected in each epoch of the training phase. In *Figure 4* we can see a Neural Network before and after using a dropout layer.

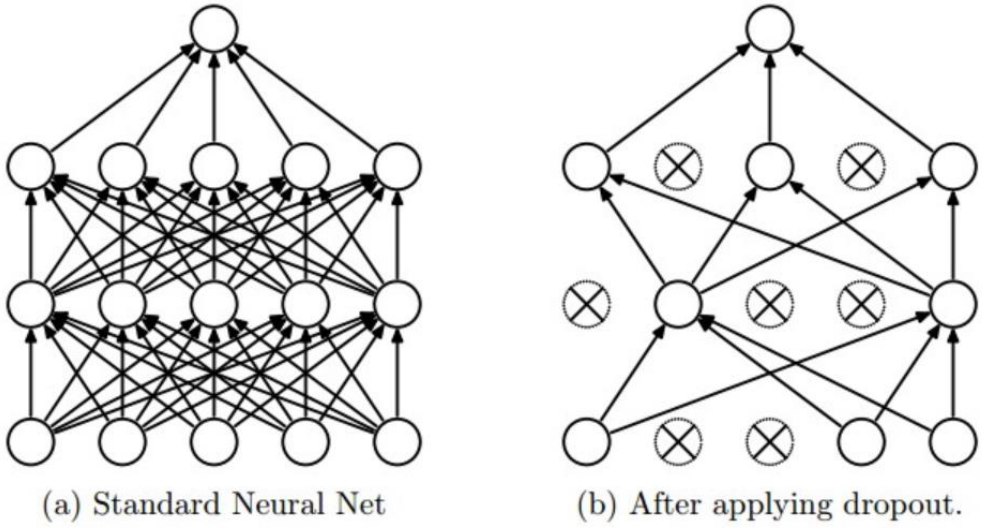


Figure 4. Standard Neural Network (a) and Neural Network using a dropout layer (b). [5]

3.3 Architecture of Convolutional Neural Networks

The main difference between a Neural Network and a Convolutional Neural Network (CNN) is that the neurons that form the hidden layers, or also known as convolutional layers, transform the input they get by convolution. This helps the model to detect patterns in the image. Also, unlike neural networks, the kernels are shared for the entire image on CNNs. Let's see some of the most used layer types.

[5] https://www.mdpi.com/2072-4292/11/16/1938?type=check_update&version=1

3.3.1 Convolutional layer

To do a convolutional operation we need an input image and a filter/kernel. A kernel consists of a tensor. A tensor is an algebraic object that can take various forms with different dimensions (See *Figure 5*). The traditional kernel used is a 3x3 size matrix.

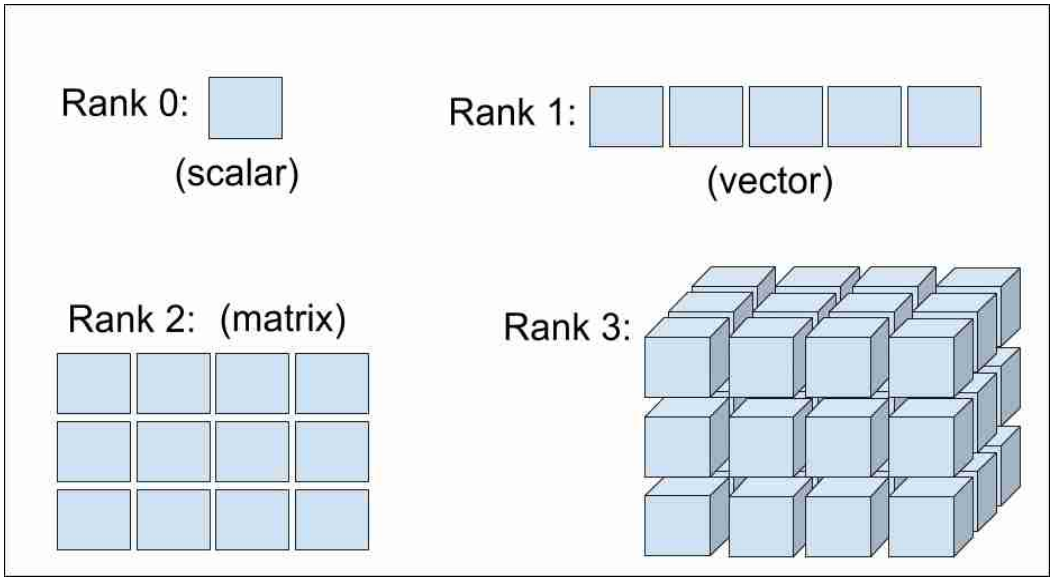


Figure 5. Example of the basic Tensor shapes. [6]

The kernel at the beginning is usually initialized with random numbers, which will be optimized in the process of learning of the model. For convolution, the kernel goes through the input image starting at the first position and moving across the width and height so that it can cover all the neurons to create a new convolved image, this technique is called sliding window.

In *Figure 6*, we see the first output of the convolutional operation of the image in the first position. This is the output obtained by adding the multiplications of the kernel on the image in the first position.

The output tensor is downsized compared to the input image. This need not always be the case. When calling a convolutional operation, it is possible to use strides to choose the steps that the kernel should move through the width and height when performing

[6] https://mc.ai/the-shape-of-tensor/

the sliding window technique, another option is to keep the same pattern as the input image so that the output tensor will have the same size.

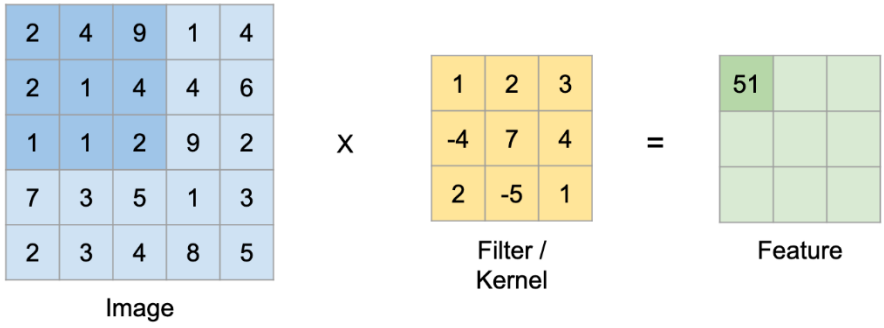


Figure 6. Example of a convolutional operation. [7]

3.3.2 Max-pooling layer

Another useful layer used is the max-pooling layer. The objective of this layer is to down sample the given tensor. For this, the max-pooling filter goes through the input tensor with the sliding window technique and in the overlapping region, takes the maximum value.

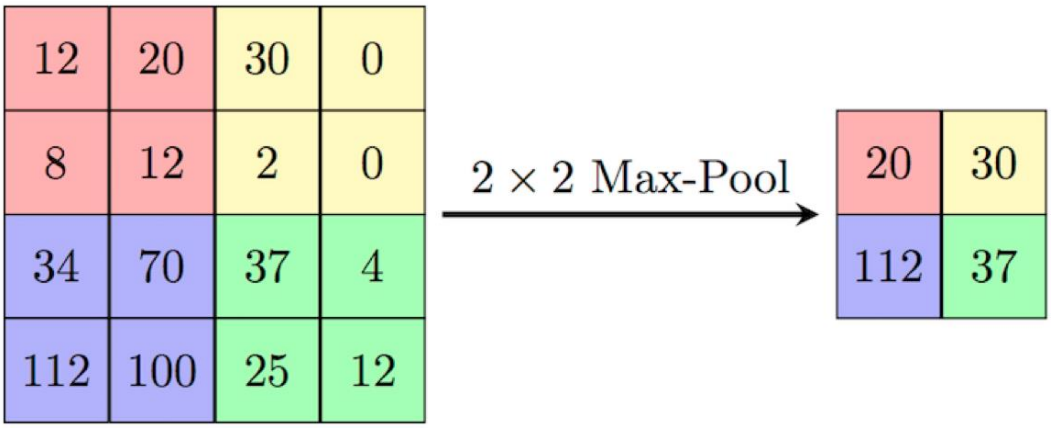


Figure 7. Example of a max-pooling over a matrix with one channel. [8]

[7] https://www.kaggle.com/inseltiger/plant-pathology-2020-with-custom-tensorflow-cnn
[8] https://amueller.github.io/COMS4995-s18/slides/aml-23-041818-convolutional-nets/#31

In the previous image, *Figure 7*, we take the maximum value without overlapping previously selected regions, so we don't hold any redundant values in the new tensor.

Inside a CNN model, the tensor usually has more than one channel, because when convolving we generally apply several kernels, so we have more parameters for more complex detections. A max-pooling operation on a CNN model would look as followed *Figure 8*.

There are two main down sample techniques (max-pooling and convolution with strides), which one we choose will depend on how we want to build our model. There are also variants such as average-pooling.

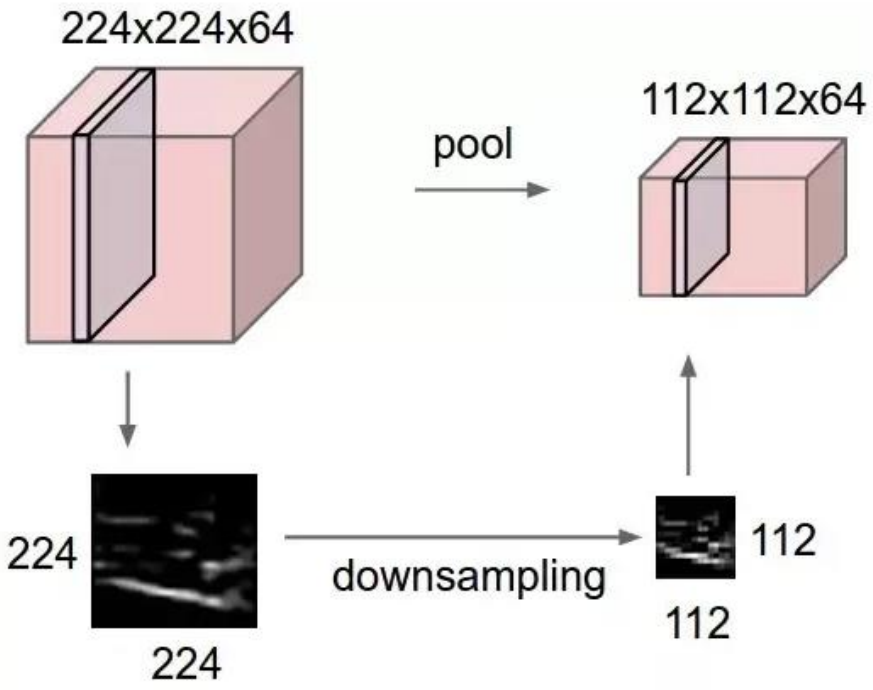


Figure 8. Max-pooling with multiple channels. [9]

3.4 How to train the network?

Let's say we have a simple model for recognition. At first, all weights and biases are randomly initialized, so when we are feeding the first image, the output will be a random guess. From here we calculate the cost function. There are different ways to calculate the cost function (e.g. the categorical cross entropy loss function or L2 loss function), but the idea is that given the guessed output, we calculate the difference to the

[9] https://computersciencewiki.org/index.php/Max-pooling_/_Pooling

expected output, from here we tune the neurons that should be more relevant when activated or, in other words, the weights and biases that should have a higher or lower value.

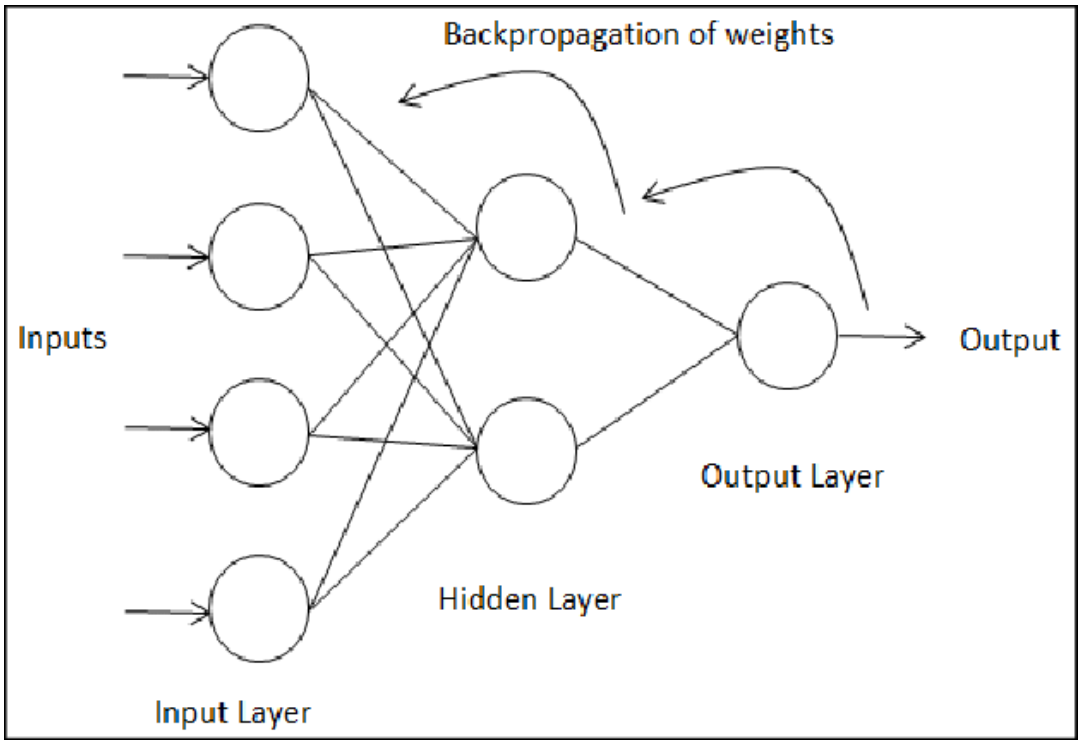


Figure 9. Backpropagation of weights. [10]

This process is called backpropagation, since it starts from the output and updates the weights and biases until it reaches the input layer. By training with a new image, the model has already learned something from the previous one and could perform better. We use optimizers to update the weights and biases (e.g. Adam optimizer or gradient descent optimizer). With the same technique we also update the kernel that convolves the image. Visualization of backpropagation at *Figure 9.*

A common way to train a model is to feed a batch of images as input rather than one at a time. This makes the training faster since we update the parameters once per batch instead of once per image.

[10] https://www.researchgate.net/figure/The-structure-of-single-hidden-layer-MLP-with-Backpropagation-algorithm_fig2_234005707

4. YOLO

Yolo (You Only Look Once) [6] algorithm is a single neural network that predicts bounding boxes and class probabilities in only one evaluation.

4.1 YOLO introduction

The original paper [6], achieved results where the model processes images at 45 frames per second. We observe, that this network is very promising since the speed performance is similar to the human one. A human brain understands objects in the environment instantly, and with this speed is doing YOLO too, although not with the same accuracy yet. We show an example of YOLO detection in *Figure 10*.

Figure 10. Real output of a YOLO network. [11]

4.2 Architecture of YOLO

YOLO divides each image into an SxS grid. In our models we use a grid size of 19x19. In *Figure 11*, we can see an example of how YOLO divides an image with a grid size of 13x13.

[11] https://www.thepythoncode.com/article/yolo-object-detection-with-opencv-and-pytorch-in-python

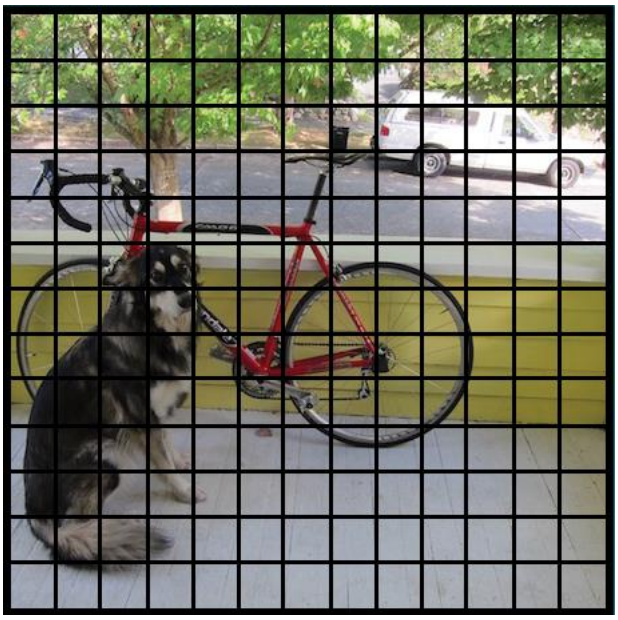


Figure 11. Image divided into a grid of 13x13. [12]

In each grid, the model predicts the possible bounding boxes and their probability. Since it can happen that a grid cell contains the center point of more than one object, we use anchor boxes to allow detection of multiple overlapping objects. By defining anchor boxes, we prevent each grid cell from being forced to choose one of the classes it thinks it contains. In our models we use 5 anchor boxes. These anchor boxes have different scales and different aspect ratios, so the model can detect multiple objects of different shapes.

For the experiments carried out in this project, we use an architecture called MyYOLONet, which is a modification of YOLOv2 [7]. We decided to use MyYOLONet because it can significantly improve accuracy compared to YOLOv2.

In MyYOLONet we resize keeping the same width and height ratio by using zero padding. In YOLOv2, an image resizes by warping, therefore one axis of the image may not scale homogeneously.

We also apply different types of data augmentation to our model, which can be found in the data augmentation section.

[12] https://arxiv.org/pdf/1506.02640.pdf

One problem with YOLOv2 is that it doesn't have a specific layer to detect objects with different scales. This is important because we can benefit if the model learns features of objects with different scales.

To solve this, MyYOLONet extends the last layer into 5 different branches, one for each anchor size and therefore each branch is learning independent features for each anchor. We do this to have a better estimation for objects with different scales. During the training phase, each branch is feeded with objects that correspond to its branch anchor size. In *Figure 12*¸we have the MyYOLONet architecture, where at the end we see the branch divisions for each anchor size, where **b** is the batch size, **w** and **h** are the number of cells of width and height and **v** is a vector containing a confidence probability. In YOLOv2 there is only one branch for detections.

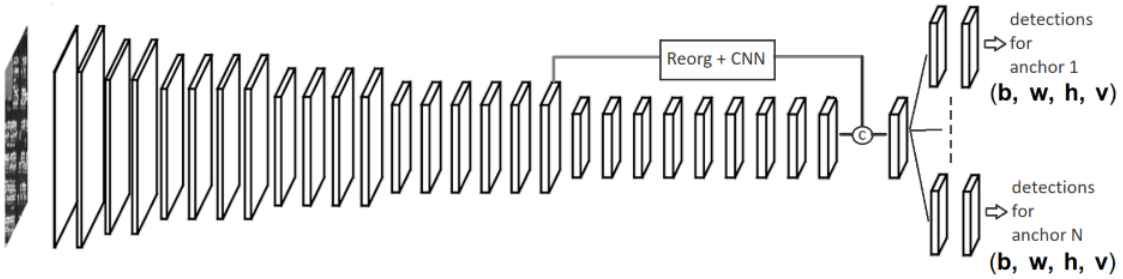


Figure 12. Architecture of MyYOLONet.

Another difference is that YOLOv2 calculates the probabilities with Softmax. This can lead to slow convergence. For this reason, in MyYOLONet we use Sigmoid for possible faster convergence. Also, with Softmax we cannot apply multi-label detections, because Softmax forces the network to detect only the object class per cell size.

MyYOLONet uses a weighted L2 loss function, which has a weight for classes and a weight for offsets. This helps us when the model predicts a bounding box with a high probability but it is a False Positive, then the penalization is less toward zero probability.

In *Figure 13*, we have the first part of YOLOv2 or MyYOLONet architecture. As mentioned before, the last layers differ from each other. We see that the layer 26 is a reorg layer. This layer is used to reorganize the features so they fit into the last layer. Reorg layer takes a tensor of shape **[B, C, H, W]**, where B is the batch size, C the number of channels, H the height of the tensor and W the width of the tensor. Then it transforms it to a tensor with size $\mathbf{[B, Cs^2, \frac{H}{s}, \frac{W}{s}]}$. In our case **s** is equal to 2. (B, 512x2^2=2048, 26/2=13, 26/2 =13). We are also using route layers. The route layer, at position 27, will concatenate the layers 24 and 26.

layer		filters	size	input		output
0	conv	32	3 x 3 / 1	416 x 416 x 3	>	416 x 416 x 32
1	max		2 x 2 / 2	416 x 416 x 32	>	208 x 208 x 32
2	conv	64	3 x 3 / 1	208 x 208 x 32	>	208 x 208 x 64
3	max		2 x 2 / 2	208 x 208 x 64	>	104 x 104 x 64
4	conv	128	3 x 3 / 1	104 x 104 x 64	>	104 x 104 x 128
5	conv	64	1 x 1 / 1	104 x 104 x 128	>	104 x 104 x 64
6	conv	128	3 x 3 / 1	104 x 104 x 64	>	104 x 104 x 128
7	max		2 x 2 / 2	104 x 104 x 128	>	52 x 52 x 128
8	conv	256	3 x 3 / 1	52 x 52 x 128	>	52 x 52 x 256
9	conv	128	1 x 1 / 1	52 x 52 x 256	>	52 x 52 x 128
10	conv	256	3 x 3 / 1	52 x 52 x 128	>	52 x 52 x 256
11	max		2 x 2 / 2	52 x 52 x 256	>	26 x 26 x 256
12	conv	512	3 x 3 / 1	26 x 26 x 256	>	26 x 26 x 512
13	conv	256	1 x 1 / 1	26 x 26 x 512	>	26 x 26 x 256
14	conv	512	3 x 3 / 1	26 x 26 x 256	>	26 x 26 x 512
15	conv	256	1 x 1 / 1	26 x 26 x 512	>	26 x 26 x 256
16	conv	512	3 x 3 / 1	26 x 26 x 256	>	26 x 26 x 512
17	max		2 x 2 / 2	26 x 26 x 512	>	13 x 13 x 512
18	conv	1024	3 x 3 / 1	13 x 13 x 512	>	13 x 13 x 1024
19	conv	512	1 x 1 / 1	13 x 13 x 1024	>	13 x 13 x 512
20	conv	1024	3 x 3 / 1	13 x 13 x 512	>	13 x 13 x 1024
21	conv	512	1 x 1 / 1	13 x 13 x 1024	>	13 x 13 x 512
22	conv	1024	3 x 3 / 1	13 x 13 x 512	>	13 x 13 x 1024
23	conv	1024	3 x 3 / 1	13 x 13 x 1024	>	13 x 13 x 1024
24	conv	1024	3 x 3 / 1	13 x 13 x 1024	>	13 x 13 x 1024
25	route	16				
26	reorg		/ 2	26 x 26 x 512	>	13 x 13 x 2048
27	route	26 24				

Figure 13. First part of YOLOv2 architecture.

5. MIXNET ARCHITECTURE

5.1 Mixnet introduction

The Mixnet architecture [4] replaces single convolutional kernels with different kernels of different sizes that can lead to better accuracy and efficiency.

Conventional kernels are 3x3 in size, but studies showed that combining multiple kernels with sizes of 3x3, 5x5, 7x7 and 9x9 can potentially improve performance.

Larger kernels do not always achieve better results, the accuracy will depend on the dataset and the class of the objects. Another thing to keep in mind is that larger kernel sizes increase considerably the model size with more parameters, so the computation will take longer.

On the official paper of Mixnet (Mixed Depthwise Convolutional Kernels) [4], authors have studied the accuracy and efficiency with different kernels and observed that the performance dropped when the kernel size is larger than 9x9. See *Figure 23* for a visualization of the accuracy over the ImageNet Top-1 dataset, where model size is represented by point size.

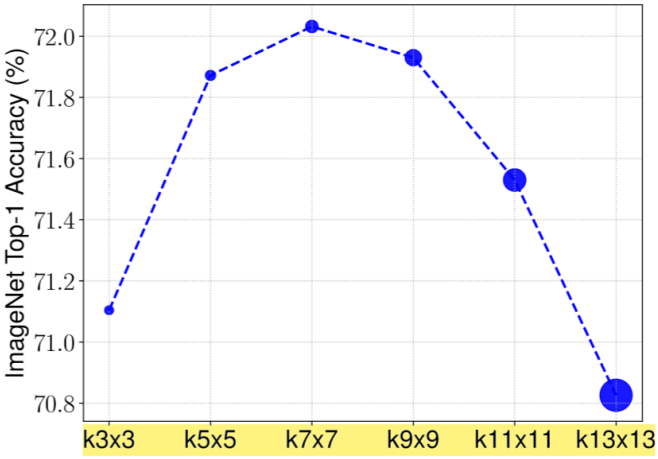


Figure 23. Accuracy when implementing different kernel sizes over ImageNet Top-1.

The bottom line is that it can improve model performance if we use large and small kernels to detect high-resolution and low-resolution patterns.

Mixnet-L, which is the largest of the models proposed in the Mixnet paper [4], achieves a state-of-the-art of 78.9% in ImageNet top-1 under standard mobile metrics.

The kernel sizes used in each model start with 3x3 and increases by 2 every time we make a split on the tensor to apply a new kernel. For example, if we want to apply 3 different kernel sizes in the convolution, the sizes will be 3x3, 5x5 and 7x7.

For each kernel in a layer, we can use the following equation (13) to find out the size of the kernel:

$$\boldsymbol{Kernel_{size} = 2i + 1} \qquad (\mathbf{13})$$

Where $\boldsymbol{i}$ goes from 1 to group size. Therefore, a tensor with splits or groups of 3 channels would have 3 kernels with sizes 3x3 (2x1+1), 5x5 (2x2+1) and 7x7 (2x3+1).

5.2 Architecture of Mixnet

In the official paper [4], there are 3 different models proposed: Mixnet-S, Mixnet-M and Mixnet-L.

5.2.1 Mixnet-S

Mixnet-S [4] has more than 4.1 million parameters that achieve a state-of-the-art of 75.8% in ImageNet top-1.

The proposed architecture consists of 17 convolutional layers with 5 max-pooling layers and kernel sizes from 3x3 to 11x11. See *Figure 24*.

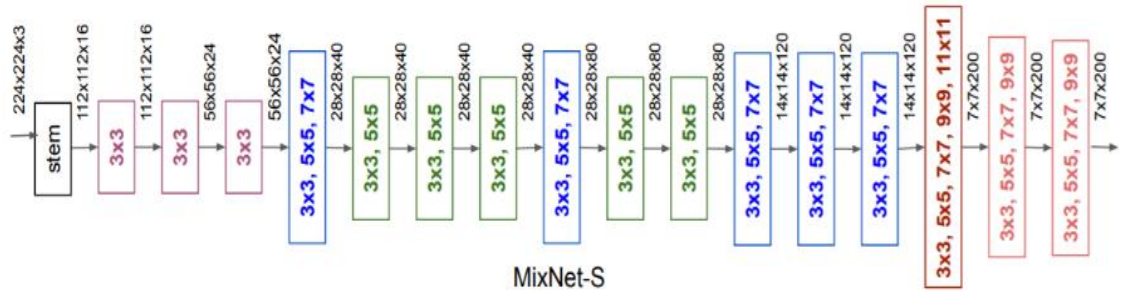


Figure 24. Original Mixnet-S architecture. [13]

Our network expects an input image size of 608x608 and has a grid size of 19x19, so we changed the original proposal to one that fits our model. The architecture we choose is in *Figure 25*.

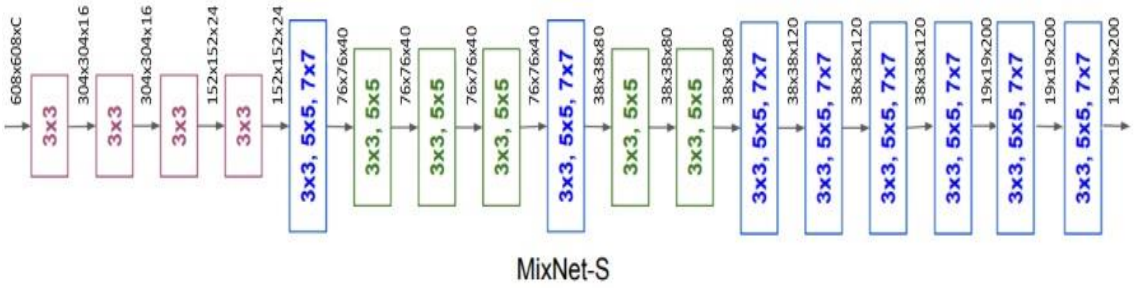


Figure 25. Modified Mixnet-S architecture.

[13] https://arxiv.org/abs/1907.09595

We made changes to the tensor size, but keeping the channels of the original proposal. We also used the max-pooling layer at the positions indicated in the source code. Also, keep in mind that the expected input is 608x608xC since we want to train our model with RGB or 1-channel images.

Finally, we removed larger 9x9 and 11x11 kernels due to their high computational cost.

5.2.2 Mixnet-M and Mixnet-L

Mixnet-M [4] has more than 5 million parameters achieving a state-of-the-art 77.0% on ImageNet top-1.

The proposed architecture consists of 20 convolutional layers with 5 max-pooling layers and kernel sizes from 3x3 to 9x9. See *Figure 26*.

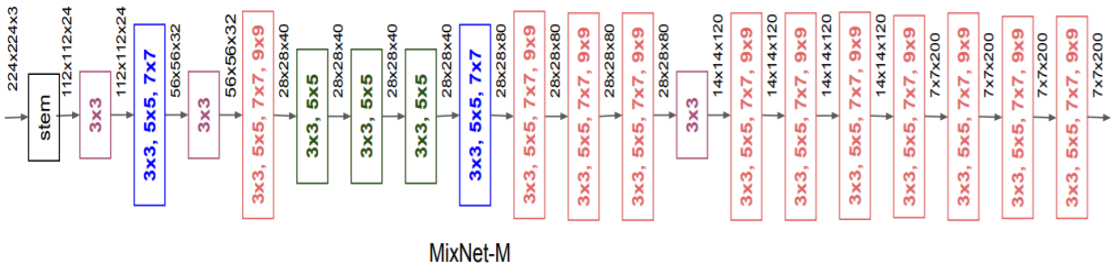


Figure 26. Original Mixnet-M architecture. [14]

As explained in Mixnet-S, our network expects a 608x608xC size input image, therefore we scaled the original Mixnet-M architecture to start with the expected input size and end with the expected output size of 19x19xC. The architecture we choose is at *Figure 27*.

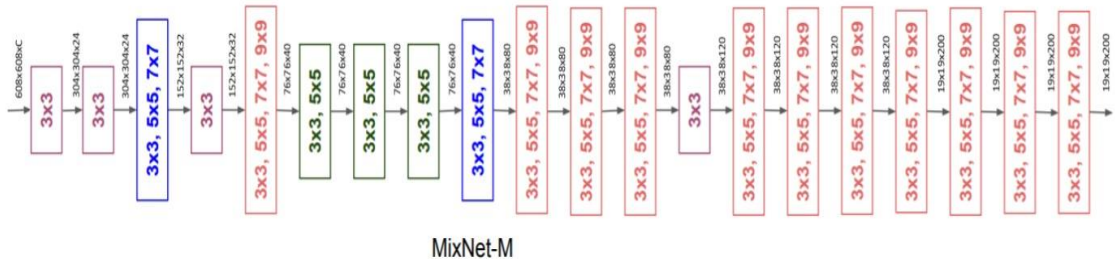


Figure 27. Modified Mixnet-M architecture.

[14] https://arxiv.org/abs/1907.09595

Mixnet-L [4] has over 7.3 million parameters that achieve a state-of-the-art 78.9% on ImageNet top-1.

The Mixnet-L architecture is the Mixnet-M architecture with a depth multiplier of 1.3 over the number of filters per layer. The final number of filters is rounded if it is decimal number. For example, if we apply 4 splits on the tensor channels in Mixnet-M, in Mixnet-L it would be 5 (4x1.3 = 5.2 ≈ 5). Therefore, layers with multiple kernels in Mixnet-M like the last one (3x3, 5x5, 7x7 and 9x9) would be transformed into layers with 5 different kernel sizes (3x3, 5x5, 7x7, 9x9 and 11x11).

Since these networks are very large, their computation is taking too long even by removing large kernels, so we decided to leave the Mixnet-M model similar to the original and make the tests with the Mixnet-S model.

We have also applied different data augmentation techniques to our models, which we will see in the next section.

5.3 Data augmentation

Data augmentation is a technique for expanding the dataset without obtaining new data. This is very useful to improve results and reduce overfitting.

The idea is to make minor alterations or modifications (e.g. flipping, rotating…) in the images of the dataset, so that our model can train with them as if they were new images. An important thing to keep in mind when we are doing data augmentation, is that we do not change the class of the objects in the image or delete objects, we are making small changes in the image so that the objects are still recognizable.

Another thing to take into consideration is to recalculate the ground-truth of the bounding boxes of each object in the image, as they might have changed after the modifications.

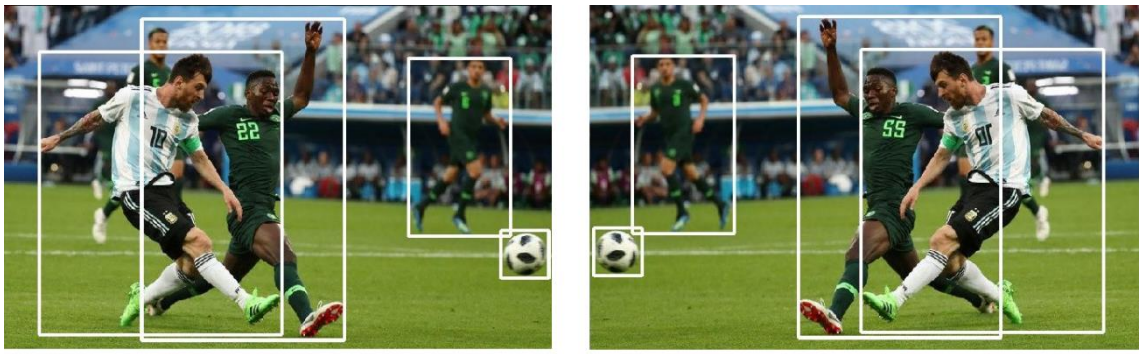
In our model we introduced horizontal flipping, cropping, scaling, rotation, masking with human patches and masking without human patches.

5.3.1 Horizontal flipping

Let's start by looking at a very basic and frequently used data augmentation technique. By using horizontal flipping, we transfer each pixel from the *'x'* axis to their opposite side. This effect is the same such as when we turn a page. We can see an example in *Figure 14*, where the original image 14(a) was flipped horizontally and turned into 14(b).

We also have to recalculate the new ground-truth of the image, because since the image has flipped the ground-truth will be new situated too.

Ground-truth annotations are usually given in a text file where each bounding box in the image is defined by 5 numbers. One defining the class (e.g. *0 for human*, *1 for football*), two to define the *x* and *y* positions and two to define the *width* and *height* of the bounding box ground-truth. The *x*, *y*, *width* and *height* coordinates are normalized on the image, so they represent the percentage where they are or how much they occupy and, in general, there are two representations, one where *x* and *y* represent the top left corner of the bounding box and one where *x* and *y* represent the center of the bounding box which is the one we use.



(a) Original Image (b) Image after horizontal flipping

Figure 16. Original image (left) and flipped image (right) with drawn ground-truth. [15]

In our case, as the *width*, *height*, *y* and *class* of the images are invariant, we just have to update the *x* coordinates. And since the coordinates are normalized, the new *x* coordinate will be the following (4):

[15] https://blog.paperspace.com/data-augmentation-for-bounding-boxes/

$$x_{new} = 1 - x_{original} \quad (4)$$

For vertical flipping we would do the same but changing the *y* coordinate.

5.3.2 Cropping

By cropping, as the name implies, we are removing part of the image. In our case, what we are removing is a random percentage of the image margin, but we want to keep a few things in mind.

First of all, we only want to remove or crop parts of the image that do not contain objects (humans in our case). Another thing we want is to avoid very narrow images (e.g. a large width but a poor image height), for this we put the boundaries to 25% of the image margin so that the cropping can just reach till there. And finally, we want to choose a random value between the edge of the image and the ground-truth with the values closest to the border of the image (or the boundaries in case the objects are situated at more than 25% away from the border). We can see an example of a cropped image in *Figure 15,* with original image being 15(a) and cropped 15 (b).

(a) Original Image

(b) Image after cropping

Figure 15. Original image (left) and cropped image with drawn bounding box (right).

The area ratio of the new bounding box will remain the same as the bounding box before cropping, so after cropping we have to normalize it again taking into consideration the new image size. Also, since after cropping we probably have new position coordinates, we have to calculate the new *x* and *y*.

The top left of the image has the coordinates (0,0) while the bottom right has the coordinates (W, H) where *W* is the image width and *H* is the image height. When we are cropping, we have to check whether we are cropping on the left or the top of the image, as cropping to the right or bottom of the image wouldn't make any changes to the coordinates of the new ground-truth. If we take a look at *Figure 15(b)*, we will see that it wouldn't matter if we cropped more of the image below or at the right of the human being, since the ground-truth would remain with the same coordinates.

For cropping on the top and left, we recalculate the new ground-truth by subtracting to the *x* and *y* coordinate the amount removed of the image. See equation (5).

$$\boldsymbol{x_{new} = x_{original} - x_{crop}} \qquad (\mathbf{5})$$
$$\boldsymbol{y_{new} = y_{original} - y_{crop}}$$

The *x* and *y* coordinates are those of the ground-truth bounding box, not the entire image.

5.3.3 Scaling

When we scale an image, we resize the given input image size to a new modified output size.

For scaling, we take a random percentage between 0 and 10% of the image, since if we take a larger range, the resized image will have lost a lot of resolution and would be difficult to recognize even for humans.

We selected a random percentage on each side which will extend the border by zero padding with black pixels. In *Figure 16*, we can see an example of the original image 16(a) and the one after scaling 16(b).

The new ground-truth coordinates have to add to the *x* and *y* coordinate the amount extended of the image. See equation (6).

$$\boldsymbol{x_{new} = x_{original} + x_{extend}} \qquad (\mathbf{6})$$
$$\boldsymbol{y_{new} = y_{original} + y_{extend}}$$

(a) Original image

(b) Image after scaling

Figure 16. Original image (left) and scaled image with drawn bounding box (right).

5.3.4 Rotation

The rotation we are doing is by making a circular movement of the image taking the center as reference point.

The rotation we are doing takes a random angle between -15 and 15, so the rotation can be clockwise or counter-clockwise.

The most challenging part of the rotation is calculating the new ground-truth of the objects in the image. We don't want to compute every new ground-truth, since after rotation it could happen that an object rotates out of the image and is barely recognizable, so we want to keep the ground-truth bounding boxes of the objects that have at least 50% of the area of them inside the image.

Let's first look at how to calculate the ground-truth of objects within the image.

Counter clockwise rotation.

We have to compute each new vertex position after the rotation, and after we have the 4 vertices recalculate the area of the ground-truth.

$$y_{min} = \left(-\left(x_2 - \frac{width}{2}\right) * \sin\theta\right) + \left(y_1 - \frac{height}{2}\right) * \cos\theta + \frac{height}{2} \quad (7)$$

$$x_{max} = \left(x_2 - \frac{width}{2}\right) * \cos\theta + \left(y_2 - \frac{height}{2}\right) * \sin\theta + \frac{width}{2}$$

$$x_{min} = \left(x_1 - \frac{width}{2}\right) * \cos\theta + \left(y_1 - \frac{height}{2}\right) * \sin\theta + \frac{width}{2}$$

$$y_{max} = \left(-\left(x_1 - \frac{width}{2}\right) * \sin\theta\right) + \left(y_2 - \frac{height}{2}\right) * \cos\theta + \frac{height}{2}$$

To apply this equation, we convert the degrees to radians. We use equation (8) to achieve this.

$$\theta = abs(angle) * \frac{\pi}{180} \quad (8)$$

We use the absolute value of the angle, since the rotation equation will change if it is clockwise or counter-clockwise. We can see an example in *Figure* 17, the original image is 17(a) and the one after counter clockwise rotation 17(b).

In equation (7), *x1*, *x2*, *y1*, *y2* represent the original left, right, top and bottom of the ground-truth, in the respective order. *Width* and *height* are the width and height of the entire image, not the ground-truths. In the equation we calculate the new vertices after rotation, where y_{min} is the vertex at bottom left**,** x_{max} is the vertex of the bottom right, x_{min} is the vertex at the top left and y_{max} is the vertex at the top right. We call them y_{min}, x_{max}, x_{min} and y_{max} because we don't need the exact coordinate of each vertex, we just want to know where the bottom, right, left and top are. From here we can calculate the new ground-truth, which we can see at equation (9).

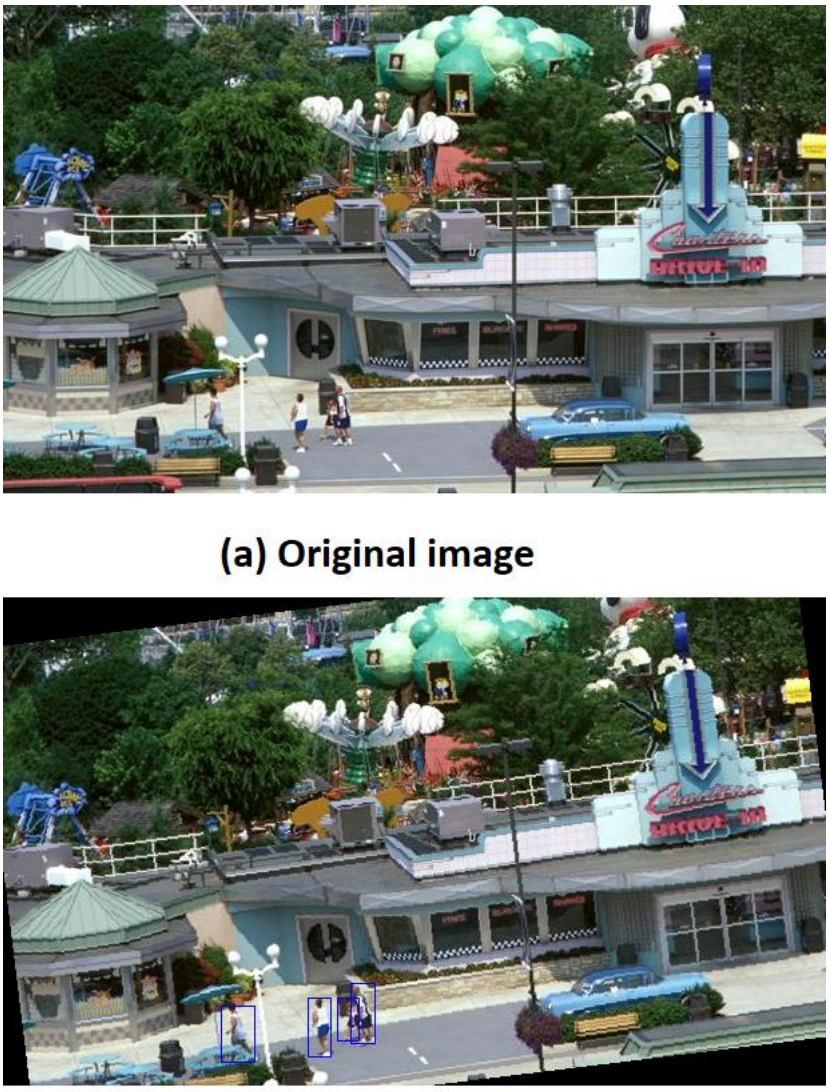


Figure 17. Original image (left) and counter clockwise rotated image with drawn bounding box (right).

Our notation for bounding boxes takes the center as reference point, so we have to calculate them (x_{center}, y_{center}).

We note that the new ground-truth is slightly larger than the original, this decision is made because it is better to have a little more ground-truth information without humans than less information and risk eliminating part of the human.

$$G_{width} = x_{max} - x_{min} \quad (9)$$
$$G_{height} = y_{max} - y_{min}$$
$$x_{center} = x_{min} + \frac{G_{width}}{2}$$
$$y_{center} = y_{min} + \frac{G_{height}}{2}$$

Clockwise rotation.

We go through the same process as with counter clockwise rotation, we only change the applied equation. See *Figure* 18, for the original image 18(a) and the one after clockwise rotation 18(b).

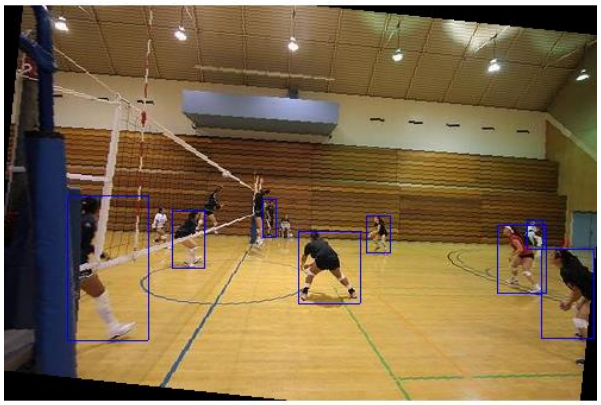
$$y_{min} = \left(x_1 - \frac{width}{2}\right) * \sin\theta + \left(y_1 - \frac{height}{2}\right) * \cos\theta + \frac{height}{2} \quad (10)$$
$$x_{max} = \left(x_2 - \frac{width}{2}\right) * \cos\theta - \left(y_1 - \frac{height}{2}\right) * \sin\theta + \frac{width}{2}$$
$$x_{min} = \left(x_1 - \frac{width}{2}\right) * \cos\theta - \left(y_2 - \frac{height}{2}\right) * \sin\theta + \frac{width}{2}$$
$$y_{max} = \left(x_2 - \frac{width}{2}\right) * \sin\theta + \left(y_2 - \frac{height}{2}\right) * \cos\theta + \frac{height}{2}$$

In equation (10) we can see how we calculate the new vertices of the bounding box. In this case, y_{min} is the vertex of the bottom right, x_{max} is the vertex of the top right, x_{min} the one bottom left and y_{max} the one at top left.

The only thing left is to check if more than 50% of the bounding box is inside the image. We perform this by calculating the area of the ground-truth bounding box and comparing it to the area of the ground-truth outside the image to see if it exceeds 50%. See equation (11).

If $G_{area_{inside}}$ is bigger than 50% of the total area ($\frac{G_{area\_total}}{2}$), then the ground-truth is valid for the training and we are cutting the parts of the ground-truth outside the image.

(a) Original image



(b) Image after clockwise rotation

Figure 18. Original image (left) and clockwise rotated image with drawn bounding box (right).

$$G_{area} = G_{width} * G_{height} \quad (11)$$

$$G_{area_{inside}} = G_{area_{total}} - G_{area\_outside}$$

5.3.5 Masking with human patches

Masking with human patches is a new data augmentation technique. The idea is to improve the model performance by training it with images that have human patches that are not associated with the environment.

With this technique, we are not only providing the model with more data to train, but we are also helping the model detect only human objects by providing patches that are not connected to their surroundings.

For this, we take the image with which the model is training and add in a free place the ground-truth of a random image. We can see an example at *Figure 19,* with the original image 19(a) and the one after masking with human patches 19(b).

There are a few things to take into consideration when applying this technique. First, we want the new patches to have an area similar to their actual ground-truth bounding box of the image. For this, before applying data augmentation to the model, we review our dataset and check the area of each bounding box and group them based on that. In our case, we created 3 groups, small patches (with an area less than 1000 pixels), medium patches (with an area greater than small patches but less than 10.000 pixels) and big patches (with an area greater than 10.000 pixels). The threshold used to make the groups is tuned based on the dataset. We want to do this, because it is more real to life to have humans in an image with a similar size.

Once this is done, we look for a good position to place our human patch. We don't want it to overlap with an existing bounding box because we would cover relevant information. For this, we first select a random human patch from a random image and apply it a random scale (from 0.9 to 1.1), so it also has the possibility of being somewhat different from the image from which we took the patch. Next, we create a matrix of ones over the original image, so that the matrix will have one position with the value 1 for each pixel of the original image at the beginning. We will use this matrix to find a position for the human patch, where each position in the matrix will represent the starting point at the top-left of the human patch. The representation will be, positions with the value 1 for the available spots and positions with the value 0 for the unavailable spots. We put zero to the positions where if we were placing the patch it would overlap with a bounding box or it would be outside the image.

We already know from the beginning some places that will not be available. We can already put these spots in the matrix with the value 0 and thus improve the speed performance of this algorithm. In equation (12), we can see how we calculate which spots of the matrix can be discarded as valid positions. Since the new patch must be completely inside the image, we know a priori that positions with a distance less than the patch to the border of the image will not be valid. In the equation, $\boldsymbol{image_{target}}$ is the matrix of the image where we want to put the patch (e.g. a matrix of ones over *Figure 19(a)*), $\boldsymbol{image_{patch}}$ is the patch we want to put on the image and the $\boldsymbol{height}$ and $\boldsymbol{width}$ variables are the height and width of the entire image or patch, depending on associated variable. In the first part of the equation, we are setting all positions that are too close to the bottom of the image to 0. The second part sets all positions too close to the right of the image to 0.

(a) Original image

(b) Image after masking with human patches

Figure 19. Original image (left) and masked image with human patches (right).

$$\boldsymbol{image_{target}}\left[\boldsymbol{image_{target_{height}}} - \boldsymbol{image_{patch_{height}}} + \mathbf{1} : \boldsymbol{image_{target_{height}}}, :\right] = \mathbf{0} \quad (\mathbf{12})$$

$$\boldsymbol{image_{target}}\left[:, \boldsymbol{image_{target_{width}}} - \boldsymbol{image_{patch_{width}}} + \mathbf{1} : \boldsymbol{image_{target_{width}}}\right] = \mathbf{0}$$

Next, we also set all positions of the matrix that overlap with a ground-truth bounding box to 0. In addition, we look at the positions above and to the left of each bounding box, and we discard the positions where it would overlap with the bounding box, considering that the position is the starting point of the top left pixel of the human patch.

(a) Original image

(b) Image after masking with human patches

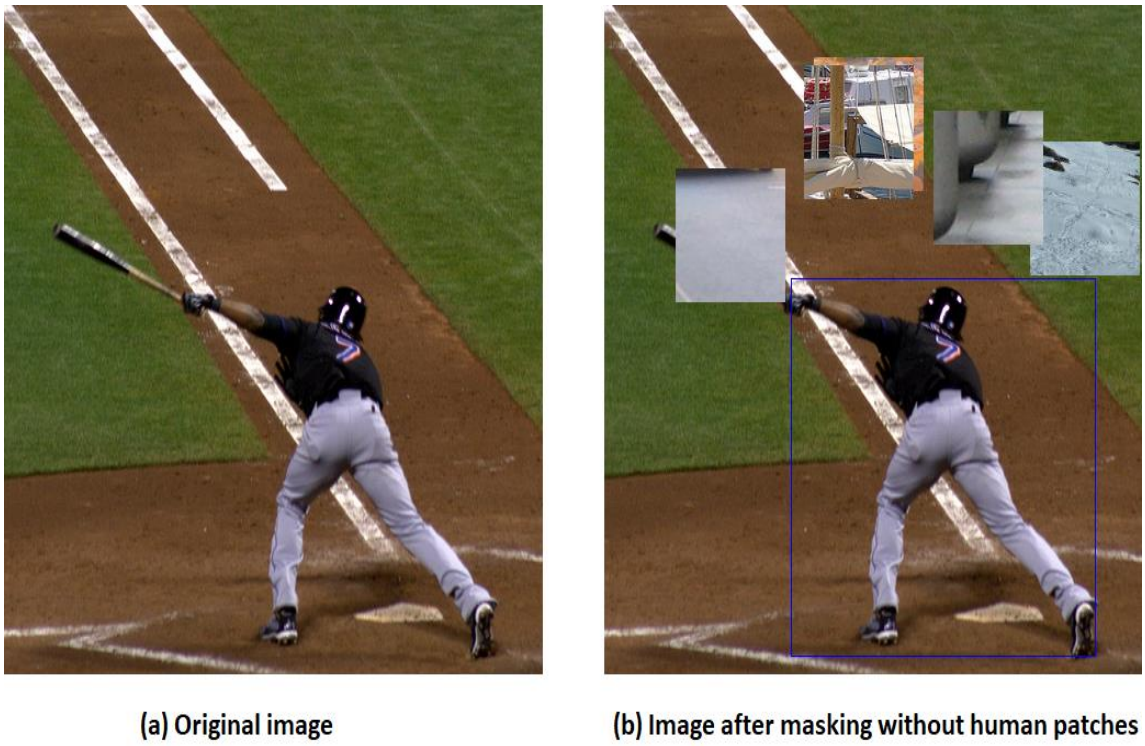
Figure 20. Original image (left) and masked image with human patches (right).

Finally, we choose an available random position that will be the starting point from where we put the human patch. It may happen that any spot is available, e.g. with very large bounding boxes. If after the first iteration of searching for an available spot a free position is not found, we change the human patch to another that belongs to a smaller area group (if the patch is already from the small group, we omit human masking in this image). We can see another example at *Figure 20*, with 20(a) as original image and 20(b) the image after masking.

The annotations of the new ground-truth will be those of the original image (e.g. *Figure 20(a)*) plus the bounding box of the infiltrated human patch.

5.3.6 Masking without human patches

This technique is similar to the masking with human patches. We use patches that do not contain any relevant human information. This can improve the performance of our model by training to recognize humans without taking into consideration the environment, as our model should be good at understanding where humans are and where they are not. We see an example at *Figure 21,* with the original image 21(a) and the one after masking without human patches 21(b).



(a) Original image (b) Image after masking without human patches

Figure 21. Original image (left) and masked image without human patches (right).

We want the patches to be a similar size, so we don't get a very large patch or a very narrow one. For this, we gave the patches an area of 10.000 pixels. On this area, we apply a random scale of 0.9 to 1.1. We always keep the same length in width and height of the patches without humans, so they are square.

Just like masking with human patches, we have to search for available spots, but in this case, we have to do it twice. One to choose a random patch that does not contain any bounding box information, and another to choose an available spot to place the new patch. In both cases, we create a matrix on the target images (image where we want to place the patch and image where we want take the patch) and set the pixels as available (one) or unavailable (zero) as masking with human patches. Finally, we choose an available random position from both matrices.

Since one patch doesn't cover much of an image, we choose the number of patches we use randomly (between 1 and 5, all from different images). Other patches can overlap as long as they don't take exactly the same position. We can see another example at *Figure 22*, with 22(a) as the original image and 22(b) as the image after masking.

(a) Original image

(b) Image after masking without human patches

Figure 22. Original image (left) and masked image without human patches (right).

Since there is no overlapping among patches and ground truth boxes, ground truth remains unchanged.

6. EXPERIMENTS

In this section we will see all the experiments carried out in this project.

6.1 Datasets

Here we will find the datasets used.

Identity-preserver Human Detection dataset

This dataset was provided by an international competition in which we participated, called ChaLearn LAP Challenge "Identify-Preserving Human Detection" [8].

The provided dataset included nearly 100.000 training images and 15.000 test images for each depth and thermal images.

The depth data consists of 1-channel images with a size of 1280x720 where each pixel represents the distance to the camera in millimeters. In *Figure 28*, we have an example of an image from the depth dataset. The image is color mapped to ease visualization. The red square represents a predicted bounding box.

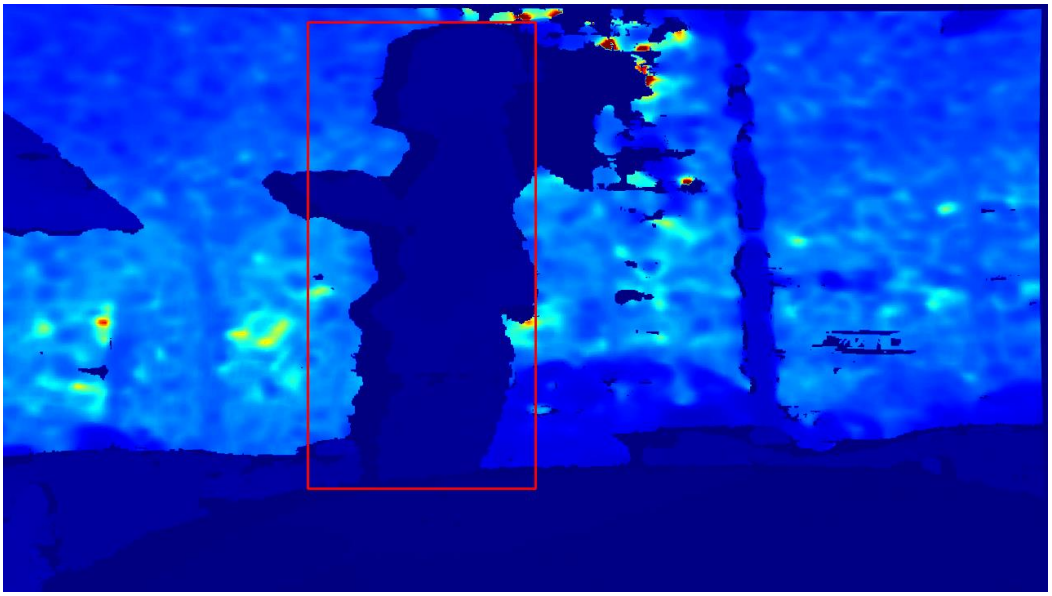


Figure 28. Image example of the depth dataset. [16]

Thermal data consists of 1-channel images with a size of 213x120 where each pixel represents the absolute temperature in Kelvin degrees multiplied by 100. In *Figure 29*, we have an example of an image from the thermal dataset. Here we also color map the image to ease visualization.

[16] http://chalearnlap.cvc.uab.es/dataset/34/description/

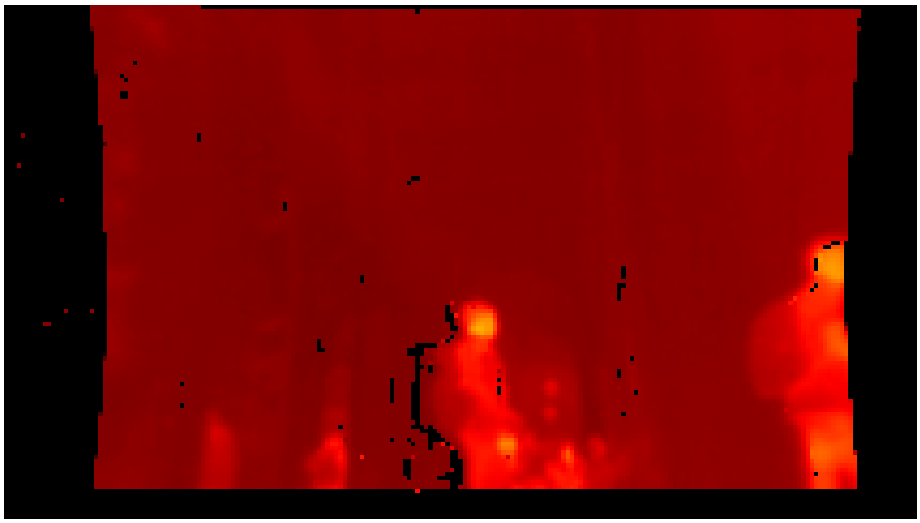


Figure 29. Image example of the thermal dataset. [17]

MNIST dataset

The MNIST dataset is a database of handwritten digits [5]. The official dataset consists of 60.000 training and 10.000 test images. Each image is grayscale, has the digit centered, is normalized and 28x28 pixels in size.

The digits were drawn by over 250 different writers, setting them apart for training and test samples, to make sure that the model learns to recognize the digit and not personal calligraphy. In *Figure 30*¸ we have two examples of handwritten digits from this dataset.

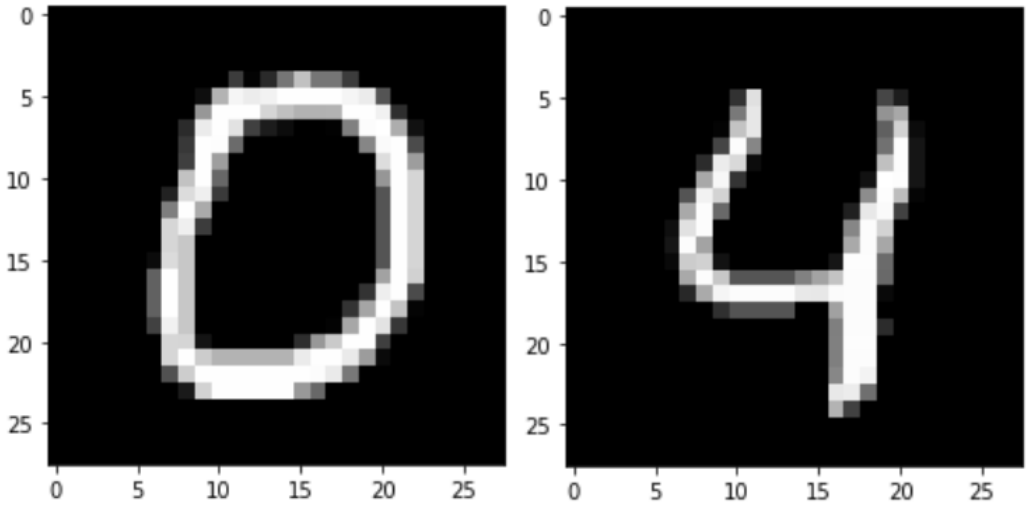


Figure 30. Two examples of handwritten digits from the MNIST dataset. [18]

[17] http://chalearnlap.cvc.uab.es/dataset/34/description/
[18] http://yann.lecun.com/exdb/mnist/

PASCAL VOC dataset

We wanted our models to train with RGB images too, so we introduced over 20.000 RGB images that contain human information from the PASCAL VOC dataset. In *Figure 31*¸ we have an example of an RGB image from this dataset.

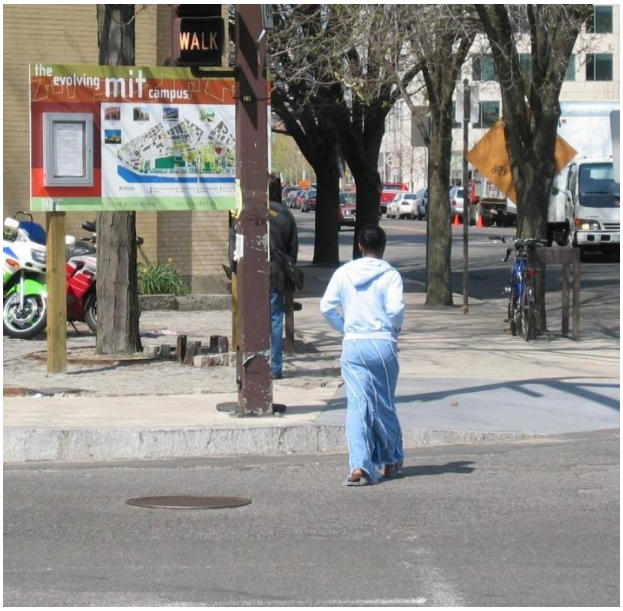


Figure 31. Example of RGB image from the PASCAL VOC dataset.

6.2 Metrics

A popular metric for measuring the accuracy of object detectors is mean Average Precision (mAP), which calculates the normalization of the accuracy between 0 and 1.

To calculate the mAP, we must first understand Intersection over Union (IoU). This metric helps us identify the correctness of predicted bounding boxes, even if they do not exactly overlap with the ground-truth bounding box. It is highly unlikely that the predicted bounding box fits perfectly over the ground-truth and just because it is moved a few pixels does not mean it is a wrong detection. In *Figure 32* we can see an example of IoU, we use equation (14) to calculate the value of IoU.

$$\boldsymbol{IoU} = \frac{\boldsymbol{area_{overlap}}}{\boldsymbol{area_{union}}} \qquad (\mathbf{14})$$

Figure 32. Example of IoU. [19]

With the IoU parameter we can calculate the True Positives/Negatives and the False Positives/Negatives, which we will need to calculate the Precision and Recall of the model.

A commonly used threshold for IoU is 0.5, so if the IoU of a predicted bounding box is above 0.5 it is considered a True Positive, otherwise it is considered a False Positive.

Now that we have the True Positives and False Positives, we can calculate the Precision from the model. Precision calculates the correct detections of the model. See equation (15), where TP stands for True Positives and FP for False Positives:

$$\boldsymbol{Precision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FP}} \tag{15}$$

Recall calculates the number of True Positives found. See equation (16), FN stands for False Negatives:

$$\boldsymbol{Recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}} \tag{16}$$

Finally, we can calculate the Average Precision which finds the area under the precision-recall curve, which contains the Precision and Recall of each image. See equation (17).

[19] https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

$$AP = \int_0^1 p(r)dr \tag{17}$$

Since we only have one class (humans), we already have the mAP. In case of more classes, we would add the AP of each class divided by the total number of classes.

6.3 Non-Maximal Suppression (NMS)

Non-maximal suppression (NMS) is a technique we are using in the detection algorithms that allows us to discard all bounding boxes that do not reach a selected threshold of probability of containing an object. So, for example, if we want our network to detect every single possible object this parameter should be very low but we probably will get a lot of false positives. In *Figure 33* we see an image before and after removing the bounding boxes with lower confidence than the NMS threshold.

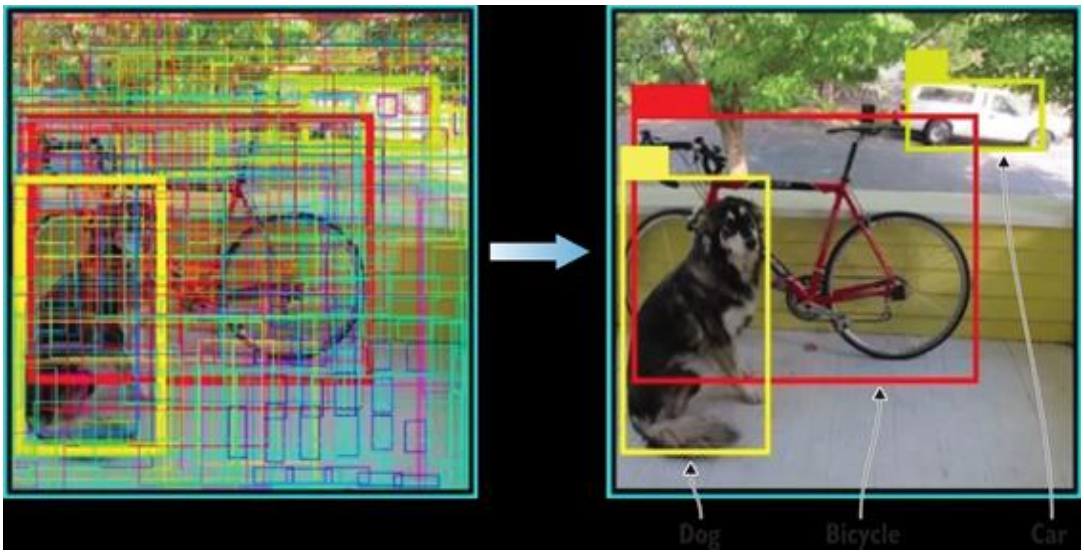


Figure 33. Removing bounding boxes with low confidence using NMS. [20]

6.4 Results

Results in ChaLearn LAP Challenge "Identify-Preserving Human Detection"

[20] https://arxiv.org/pdf/1506.02640.pdf

This competition was organized in the context of the 15th IEEE International Conference on Automatic Face and Gesture Recognition. The goal was to develop a computer vision method for human recognition in depth and thermal images.

We participated in two categories: depth and thermal recognition. For each category, we trained a model with the data corresponding to its category. We also trained a model with a mix of both dataset and adding RGB images, but the results were much lower in precision when comparing to the other two models.

Depth competition

First we submitted in the learning phase a model without data augmentation and with a small number of trained epochs and we obtained a mAP of 0.196 where the Intersection over Union (IoU) is at least 0.5, so the ground-truth bounding box and the predicted bounding box are sharing at least 50% of the area.

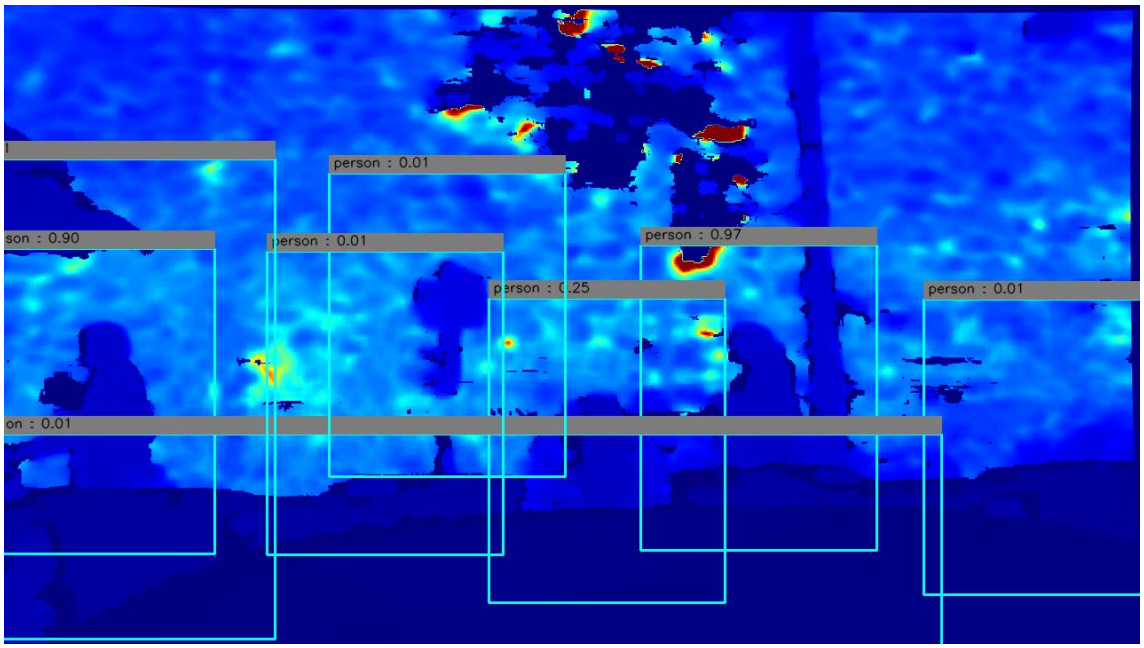


Figure 34. Example of depth image with predicted bounding boxes with low NMS.

In the submission, we were training with the Identity-preserver Human Detection dataset, with 80% as the training set and 20% as the validation set. The validation dataset provides an unbiased evaluation of our model while tuning the hyperparameters during training. The dataset provided consists of several separated videos in frames for

the images. We decided for future trainings to place only the frames belonging to 2 random selected videos to the validation set (700 images) and the rest as training set. So, our model has more images to train.

After training for 23 epochs and applying data augmentation, we obtained a mAP of 0.318 with IoU at 0.5 and 0.533 with IoU at 0.25. For this result, we also internally fine-tuned NMS and IoU threshold parameters. The NMS parameter for the first submission was very low, resulting in a large number of False Positives, as we were predicting a bounding box even if the confidence was low. In *Figure 34*, we can see that because we chose a low NMS threshold, the model made many false predictions, resulting in a lower precision. Above each bounding box we see confident the model has on each prediction.

For the submission with 23 epochs, we fine-tuned the NMS threshold parameter to 0.65, which means that it must have at least 0.65 confidence to predict the bounding box. We set the intern IoU threshold parameter for mAP calculation to 0.5. In *Figure 35*, we have a better prediction of an image than with the model for the first submission.

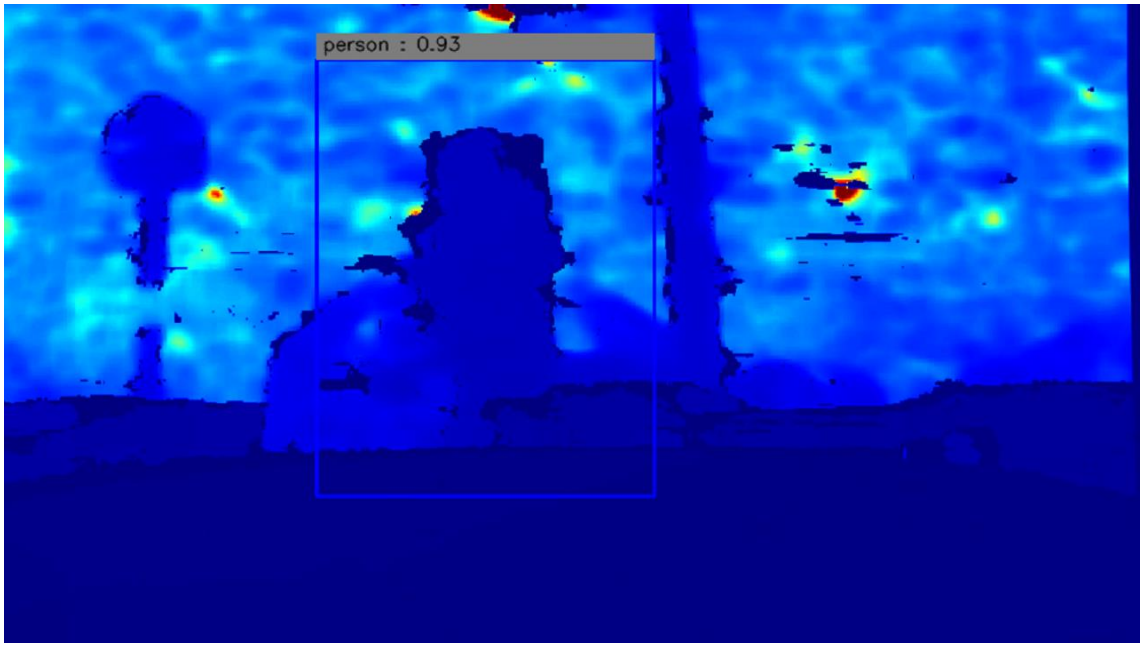


Figure 35. Example of depth image with predicted bounding boxes with fine-tuned NMS.[21]

[21] Competition leaderboard: https://competitions.codalab.org/competitions/21926#results , username: Lukaz

Despite the fact that these results are still far from perfect, we achieved an improvement of more than 60% compared to the first model (IoU at 0.5) thanks to data augmentation, more training and fine-tuning parameters.

In the final evaluation on the test depth data we obtained a mAP of 0.259 with IoU at 0.5 and 0.594 at 0.25.

Thermal competition

Just like in the depth competition, we submitted a model without data augmentation, a small number of epochs and without fine-tuning parameters. Our score for this model was a mAP of 0.302 with IoU at 0.5.

Here we also went from the 80/20 split for the training and validation set, to just placing the frames that belong to 2 videos for the validation set (2419 images).

After training for 59 epochs and applying data augmentation, we obtained a mAP of 0.364 with IoU at 0.5 and 0.556 with IoU at 0.25. As in the depth competition, we fine-tuned NMS parameter, obtaining the best results with NMS threshold set to 0.6. In *Figure 36*, we have a bounding box prediction of an image.

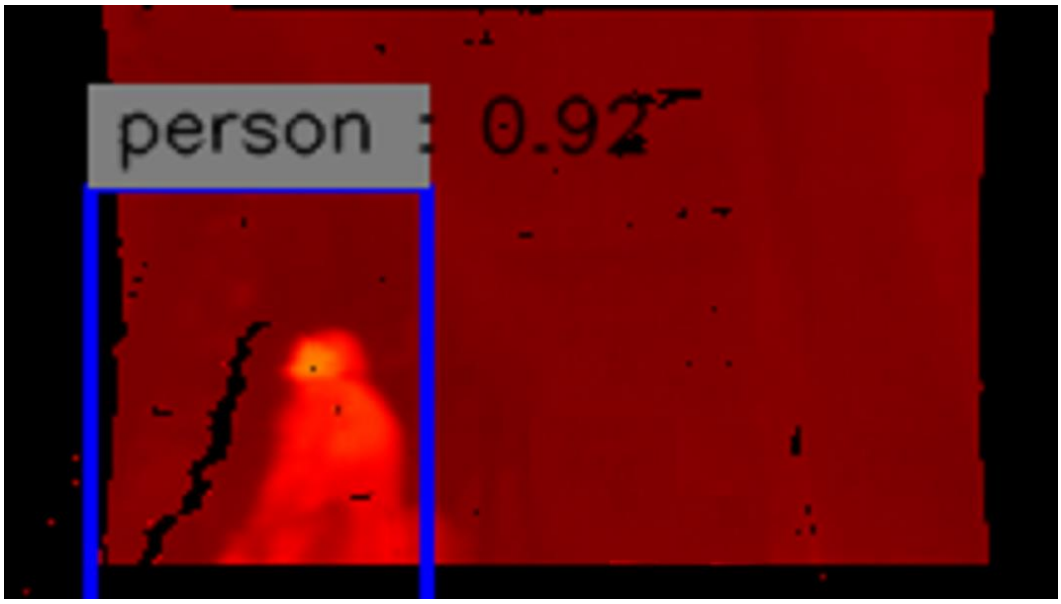


Figure 36. Example of thermal image with predicted bounding box. [22]

[22] Competition leaderboard: https://competitions.codalab.org/competitions/21927#results , username: Lukaz

This represents an improvement of more than 20% with respect to the first model (IoU at 0.5).

In the final evaluation on the test thermal data we obtained a mAP of 0.304 with IoU at 0.5 and 0.684 with IoU at 0.25.

Results with MNIST dataset

The objective for the MNIST dataset is to classify each digit. The best classifiers on the official MNIST page, achieve an accuracy of 99.77% using CNNs.

We build our own model implementing Mixnet [4]. We tried different number of layers, optimizers, loss functions and kernel sizes. Our final and best model achieved a performance of 99.1% accuracy over the test data.

The model we built consisted of 3 convolutional layers, where the first layer had the traditional 3x3 kernel and the second and third where each using two different kernels with different sizes in the layer (3x3 and 5x5). The last two layers are fully connected layers.

We also down sampled the tensor with maxpooling, used ReLU as activation function and we used a dropout layer. As loss function, we used the categorical cross entropy loss and as an optimizer we used AdamOptimizer. We can see the architecture of our model at *Figure 37*.

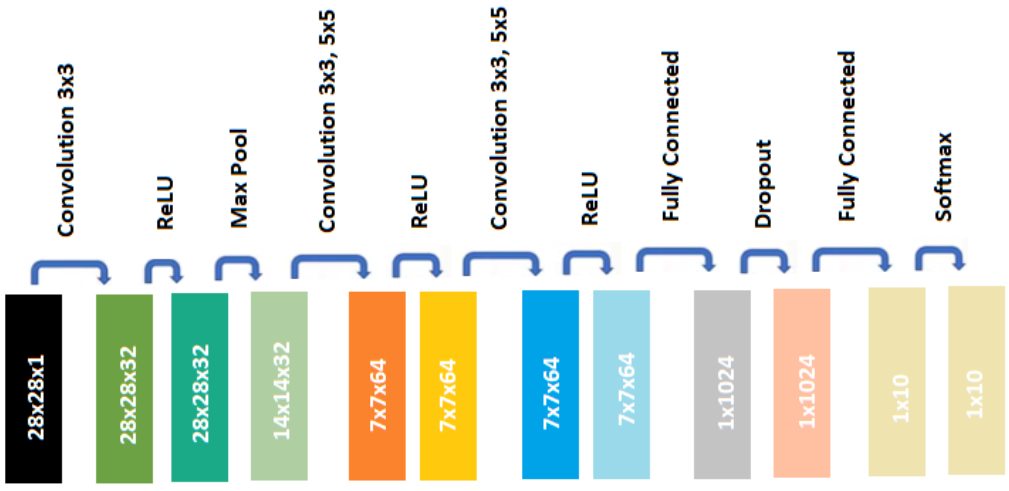


Figure 37. Architecture of the model for MNIST dataset experiments.

In previous models, we tested one convolutional layer that used multiple kernel sizes. Here we obtained 91% accuracy after one training epoch and 97% accuracy after several

training epochs. By using 2 Mixnet convolutional layers, we achieved 99.06% accuracy after 3 training epochs, which increased slightly after 12 epochs to 99.1%.

We wanted to compare this model with one that uses the exact same architecture, but changing layers with multiple kernel sizes to layers with one 3x3 kernel. Here we got a small disadvantage in accuracy comparing to the Mixnet model. We obtained a maximum of 97% accuracy after 12 epochs (2.1% less than the Mixnet model).

We also tested the proposed Mixnet-S architecture of the Mixnet paper on the MNIST dataset, but obtained very low results (0.097% accuracy after 1 epoch and 0.12% accuracy after 6 epochs). These results are low because the proposed Mixnet-S architecture is very deep with 17 convolutional layers and is not intended to recognize simple objects such as the handwritten digits in the MNIST dataset. We have a summary of the results at *Table* 1.

We observe a better performance using multiple kernels on the MNIST dataset, if the depth of the network is according to the dataset.

	SIMPLE MIXNET MODEL (3 epoch)	**SIMPLE CNN MODEL** (3 epoch)	**SIMPLE MIXNET MODEL** (12 epoch)	**SIMPLE CNN MODEL** (12 epoch)	**ORIGINAL MIXNET-SMALL** (6 epoch)
MNIST DATASET	99.06%	96.5%	99.1%	97%	12%

Table1. Results on MNIST dataset.

Results of Mixnet-S and MyYOLONet on human detection

We trained a model with the Mixnet-S architecture [4] for 48 epochs without data augmentation. The dataset we used was a combination of the depth and thermal images provided by the Identity-preserver Human Detection challenge (84818 images for each modality) and the images with human information we had from the PASCAL VOC dataset (23284 RGB images).

Since the combination of all these datasets contained almost 200.000 images, we trained our model with 10% randomly selected images from the entire dataset Our validation set contained over 2.500 images. The ratio of each modality, and therefore the probability of selecting an image of one of the 3 modalities, is 44% for depth, 44% for thermal and 12% for RGB images.

On the evaluation of the model on the depth dataset provided by the Identity-preserver Human Detection challenge (almost 13.000 images in the validation set) we achieved a mAP of 0.49% with IoU at 0.5 and NMS at 0.65.

On the evaluation of the model on the thermal dataset (also from the Identity-preserver Human Detection challenge with almost 13.000 images as validation set) we obtained a mAP of 0.42% with the same parameters as in the depth evaluation.

If we compare these results with those we obtained in the International Competition, we observe an increase of more than 50% on depth (0.49 mAP on Mixnet and 0.318 on MyYOLONet) and an increase of more than 15% on thermal (0.42 mAP on Mixnet and 0.364 on MyYOLONet), although we have to take into account that the Mixnet-S model trained for more epochs than MyYOLONet and that Mixnet trained one model over the entire dataset (depth, thermal and RGB) and MyYOLONet trained one model with depth dataset and one with thermal dataset.

We also trained MyYOLONet for 48 epochs with the same dataset Mixnet-S was training. In *Table 2*, we have the results.

	MIXNET-SMALL MODEL (1 epoch)	**MYYOLONET MODEL** (1 epoch)	**MIXNET-SMALL MODEL** (48 epoch)	**MYYOLONET MODEL** (48 epoch)
DEPTH DATASET	14%	12%	49%	44%
THERMAL DATASET	15%	9%	42%	13%

Table 2. Results on depth and thermal datasets with Mixnet-S and MyYOLONet.

We can see a small accuracy advantage from the Mixnet model when testing with the depth dataset and even greater advantage in the thermal dataset. In the depth dataset, we achieved an accuracy improvement of more than 10% with the Mixnet model and an improvement of more than 320% over the MyYOLONet algorithm.

Let's take a look at Mixnet's results compared to MyYOLONet models from the international competition. See *Table 3*. In *Figure 38*, we can see some predictions made on the validation set from both models.

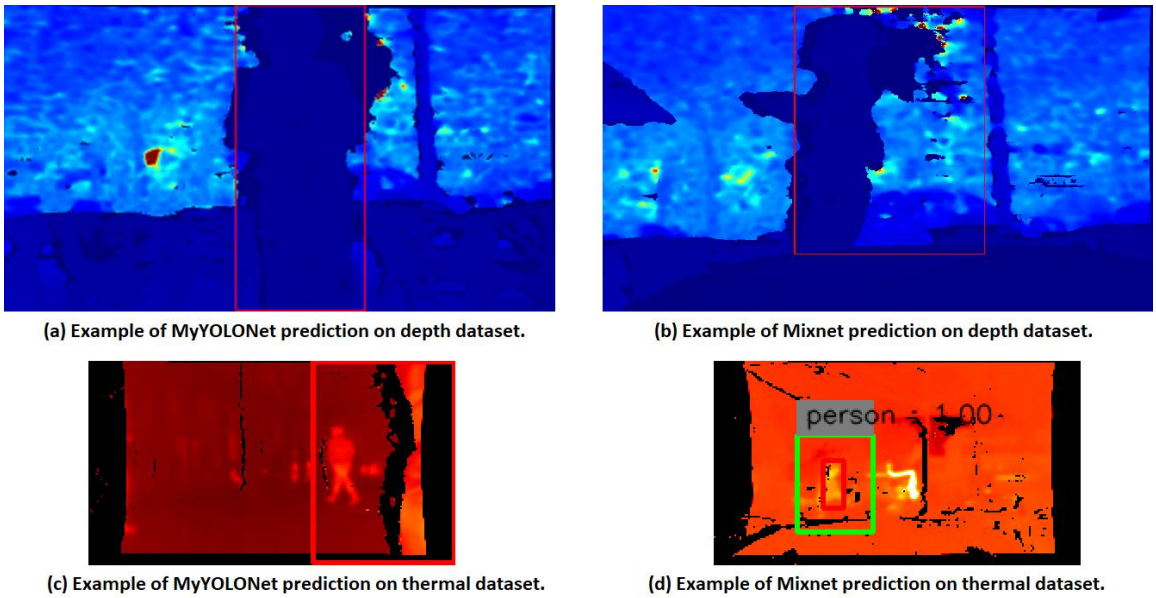
Please note that the models have been trained with different dataset and different number of epochs. What they have in common is that they evaluated the same validation set (almost 2600 depth and thermal images).

	MIXNET-SMALL MODEL	**MYYOLONET MODEL WITH DATA AUGMENTATION**
DEPTH DATASET	49%	40%
THERMAL DATASET	42%	45%

Table 3. Results on depth and thermal datasets with Mixnet-s and MyYOLONet with data augmentation.

When applying data augmentation, we have to take into consideration that it has a higher computational cost. For horizontal flipping, we are reversing each row of the matrix, so we have a time complexity of **O(N*M)** where N is the number of rows and M the number of columns. For scaling and cropping, the time complexity is related to the number of bounding boxes that the image has **O(B)**. For rotation, the time complexity is **O(N*M+B)**, where N is the number of rows of the image, M is the number of columns and B is the number of bounding boxes of the image. For masking with human patches, we have **O(N*M*B + B)**, where N are the rows, M the columns and B the bounding boxes of the image, this is because we first go through the bounding boxes of the image to calculate the area and thus know which human patch with a similar area we can choose, and then, we go through width and height of the targeted image, discarding spots that are not available for the patch due to the bounding boxes. For masking without human

patches, the time complexity is **O(P*G*N*M + B)** where P is the number of patches without humans we are going to put ,G is the number of ground-truth bounding boxes of the image where we take the patches from and B,N and M are the same as in masking with human patches. Here we also go first through the bounding boxes of the target image to disable positions that are occupied, and then for each non-human patch we check the ground-truth bounding boxes of the image to take the patch without cutting any relevant information.



(a) Example of MyYOLONet prediction on depth dataset.

(b) Example of Mixnet prediction on depth dataset.

(c) Example of MyYOLONet prediction on thermal dataset.

(d) Example of Mixnet prediction on thermal dataset.

Figure 38. Example of predictions on the depth and thermal datasets from MyYOLONet and Mixnet models.

6.5 Setup

Here we will show the hyperparameters used for the experiments. All models used Adam as optimizer with a learning rate of $1e^{-4}$.

Setup for ChaLearn LAP Challenge "Identify-Preserving Human Detection"

On the international competition, we trained our model on a server with Python 2.7.12 and tensorflow 1.12. For the data augmentation we used the libraries math, numpy and cv2.

In the depth category, we trained a model for 23 epochs with MyYOLONet architecture, which took nearly 200 hours of computational training. As previously mentioned, the training set contained almost 100.000 images and the validation set almost 700, all

depth images. We used different type of data augmentation. We configured it so that it has a 75% probability of having data augmentation. If we apply data augmentation to the image, it has a 50% probability to flip horizontally, a 50% probability to mask (if we mask, we have 50% probability of masking with human patches and 50% to mask without), a 50% probability of rotation and a 50% probability of resizing (if we resize, we have 50% probability to crop and 50% probability to scale). The IoU threshold is 0.5 and the NMS threshold is 0.65. We also use a batch size of 10. Model evaluation is usually much faster and take less than an hour.

In the thermal category we trained a model for 59 epochs with MyYOLONet architecture, which took almost 175 hours. The training set consisted in 95.000 images and the validation set in 2.500 images. We apply the same probabilities for data augmentation as in the depth category. The IoU threshold is 0.5 and the NMS threshold is 0.6. We used a batch size of 20.

Setup for MNIST dataset experiments

We used Python 3.6.10. We build the model on Jupyter-Notebook with version 6.0.3, running on Anaconda with version 4.8.1. We also used tensorflow 1.8.0 and the libraries mnist and numpy. Training the model for 12 epochs took nearly 4 hours.

Setup for the results of Mixnet-S and MyYOLONet on human detection

Here we also used Python 2.7.12 and tensorflow 1.12.

The Mixnet-S model trained for 48 epochs, which took nearly 550 hours. IoU threshold is 0.5 and NMS threshold 0.65. We used a batch size of 10.

The MyYOLONet model trained for 48 epochs, which took nearly 65 hours. The parameters are the same as for the Mixnet-S model.

We can see that even Mixnet-S in some modalities had a better mAP than MyYOLONet, the training process takes much longer.

7. CONCLUSIONS

In this project, we saw different architecture designs. We built a model with different kernels in the layers (Mixnet) and we achieved an almost perfect model to classify handwritten digits (91.1% accuracy). The model without the multiple kernels remained

97%. We also train a modified Mixnet-S model with a combination of 3 different datasets and test its accuracy with depth and thermal images. We did the same with the MyYOLONet model and saw that the Mixnet-S model had a small accuracy advantage in depth and thermal images, but the training process took much longer. Finally, we tested two MyYOLONet models that were learning with data augmentation over depth and thermal images, in a separated way. With these two models we participated in a competition. In addition, we evaluated them on the validation set in which Mixnet-S was tested for comparison. Here we got a slight advantage for Mixnet-S in depth images and a slight advantage for MyYOLONet in thermal images.

There are still many challenges for object detection. For training, a model needs thousands of images with their corresponding ground-truth. Labeling images with a ground-truth bounding box of the exact position of the object is a slow task, and some dataset do not have enough data for the model to learn to detect objects well. That's also one reason why we mix 3 datasets for training our models. Another difficult task for object detection is to detect large and small objects as well. Objects in real life have different W/H ratios and scales, so they are in the images the model trains with. This is a reason why MyYOLONet implements multiple branches for each anchor size, where each branch is learning to recognize objects of a specific size. The model could then detect objects in multiple scales. However, object detection for small images remains a challenge. We also have challenges with unbalanced data, for example in the MNIST dataset there are 25% more images with digit 1 than with digit 5. This leads the model to learn more about features of digit 1 and this can be counterproductive for the accuracy. Note here that we got good accuracy with the MNIST model anyway, but MNIST is a simple dataset to recognize and we are classifying each image instead of detecting the exact position of the object, making the task easier. Another obstacle is that our models also have a slow convergence. The network takes a long time to learn more or less precisely the features of the objects in the images. It is likely that in the future we will benefit from advances in neural networking for a faster convergence time.

Possible implementations for future work are combinations of MyYOLONet with Mixnet. We could test the mAP of the MyYOLONet architecture on different datasets, but applying multiple kernels instead of one in the layers. This could lead to better accuracy as we are using multiple branches to detect images of different sizes and multiple kernels for detecting high-resolution and low-resolution patterns.

Image-based human detection algorithms offer good results, but are still far from human-level performance in most areas. But object detection has grown in popularity in recent years due to its wide range of opportunities, so the direction it is heading is promising to achieve reliable and autonomous machines.

8. References

[1] Patrik Kamencav, Miroslav Benco, Tomas Mizdos, Roman Radil, *A New Method for Face Recognition Using Convolutional Neural Network,* Advances in Electrical and Electronic Engineeering, 2017.

[2] Alex Krizhevsky, Vinod Nair and Geoffrey Hinton, *CIFAR-10 and CIFAR-100 datasets.*

[3] Alexander Kolesnikow, Lucas Bayer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvian Gelly and Neil Houlsby, *Big Transfer (BiT): General Visual Representation Learning, cs.CV;cs.LG,* arXiv:1912.11370, 2019.

[4] Mingxing Tan and Quoc V. Le, *MixConv: Mixed Depthwise Convolutional Kernels*, cs.CV; cs.LG, arXiv:1907.09595, 2019.

[5] Yann LeCun, Corinna Cortes and Christopher J.C. Burges, *The MNIST database*.

[6] Joseph Redmon, Santosh Divvala, Ross Girsbick and Ali Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, cs.CV, arXiv:1506.02640, 2015.

[7] Joseph Redmon and Ali Farhadi, YOLO9000: Better, Faster, Stronger, cs.CV, arXiv:1612.08242, 2016.

[8] Identity-Preserving Human Detection (IPHD), ChaLearn, 2020.

[9] Joseph Redmon and Ali Farhadi, *YOLOv3: An Incremental Improvement,* cs.CV, arXiv:1804.02767, 2018.

[10] Ross Girshick, Jeff Donahue, Trevor Darrell and Jitendra Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, cs.CV, arXiv:1311.2524, 2014.

[11] Shaoging Ren, Kaiming He, Ross Girschick and Jian Sun, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, cs.CV, arXiv:1506.01497, 2015.

[12] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu and Alexander C. Berg, *SSD: Single Shot MultiBox Detector*, cs.CV, arXiv:1512.02325, 2015.

[13] Tsung-Yi Lin, Genevieve Patterson, Matteo R. Ronchi, Yin Cui, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girschick, James Hays, Pietro Perona, Deva Ramanan, Larry Zitnick and Piotr Dollar, *COCO dataset*.

[14] Jifeng Dai, Yi Li, Kaiming He and Jian Sun, *R-FCN: Object Detection via Region-based Fully Convolutional Networks*, cs.CV, arXiv:1605.06409, 2016.

[15] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn and Andrew Zisserman, *The PASCAL VISUAL OBJECT CLASSES (VOC)*.