INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

PFE

N° d'ordre : PFE-INFO-2016-19

Projet de Fin d'Études

présenté par

Hoel KERVADEC

Élève ingénieur de l'INSA Rennes

Spécialité INFO

Année universitaire 2015 - 2016

Deep Hough networks for object detection

Lieu du Projet de Fin d'Études

Inria Rennes

Tuteur du Projet de Fin d'Études

Yannis Avrithis

Correspondant pédagogique INSA Rennes

Bertrand Coüasnon

PFE soutenu le 02/09/2016

Inria Rennes

Contents

1 Introduction

This report will explain the work I did during the past six month in the Linkmedia team, under the supervision of Yannis Avrithis and Guillaume Gravier.

Linkmedia is doing content analysis of different kind of media (audio, video, text) to "link" them. For instance, semantic links between some news article and a video documentary. To achieve such multi-modal links, fundamental research in different fields are cohabiting in the team. Among them, computer vision, where the goal is to achieve high-level understanding of images and videos: what are the objects that appears in the image ? What is the action taking place, and where ? Who is this person in the image ? Is this the same in this other image ?

To solve those problems, machine learning is often used: instead of implementing an algorithm that would directly solve it, we give the machine ways to learn. By giving it enough time and examples, it will figure out recurring patterns, and be able to solve the task it was intended for. During the past years, one method gained a wide popularity: Deep Learning. It consist to train neural networks in a supervised fashion: we provide examples and their correct answers to the network. The term "deep" comes from the size of the networks, that are much bigger (and deeper, hence the name) that in the past, and from the number of training examples required: thousands of them, if not tens of thousands.

This is what we used for our task: object recognition. We will first reintroduce more context from the field, before explaining what we are trying to achieve. Then, we will focus on the work we did, and the results we were able to produce. Because this is still a work in progress, we will explain the future works that can be done and tested. Before concluding, we will explain how the code and experiments were managed to ensure meaningful and reproducible results.

2 Context

2.1 On the difference between detection and classification

To clarify the reading of this report, and to avoid confusion, we will first explain the difference between image classification, object detection and image segmentation.

Image classification The goal is to label an entire image, corresponding to its most prominent feature. Depending on the task, it could be the action taking place, or the object taking the most space in the image. For instance, in Figure 1, we can easily say that this is an image of an eagle. Under the image, are the top-5 labels proposed by a Convolutional Neural Network. It can be problematic when different objects are present

on the same image: which one should we prefer ? In some case, it can be difficult to decide, even for a human.


```
(0.94162601, 'n01614925 bald eagle, American eagle, Haliaeetus leucocephalus')
(0.025338238, 'n01608432 kite')
(0.011792503, 'n01819313 sulphur-crested cockatoo, Kakatoe galerita, Cacatua galerita')
(0.0077449828, 'n01829413 hornbill')
(0.0042824917, 'n02051845 pelican')
```

Figure 1: Example of image classification with a network trained for the ImageNet [4] dataset [10].

Object detection It can be seen as an extension of image classification. Instead of giving the main theme of the image, instances of objects are detected with their location (usually a simple rectangular box around it). This is more precise, and allows us to detect multiple objects from multiple classes. Or, as in Figure 2, only one object: a cat.

Object localization Similar to object detection, but there is only one object, from a class that is already known to us. It is therefore much easier to deal with false detection.

Image segmentation Bounding boxes are still quite limited, since a lot of the pixels contained in a box do not belong to the object, and small parts of the object may be outside of the box. This is what image segmentation is trying to achieve: pixel-wise classification. Figure 3 is perfectly illustrate this idea. We now have a clear delimitation of the detected object.

We can easily understand that image segmentation is harder that object detection, which is in turn harder than image classification. We will now present CNN, one method used to solve those problems.

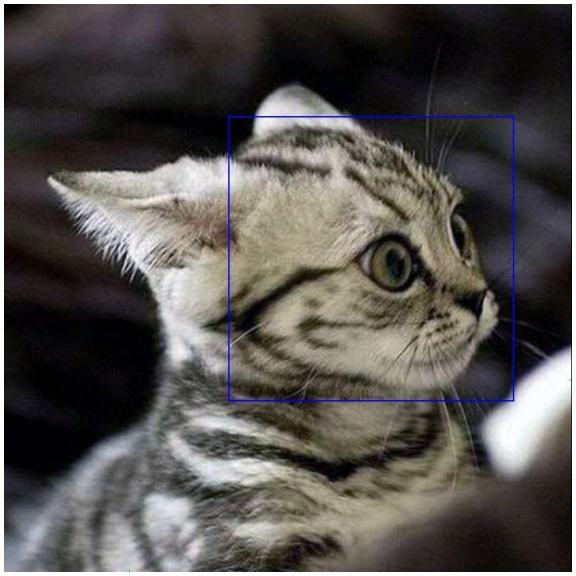

Figure 2: Object detection, done by one of our intermediate network. It successfully located the cat face.

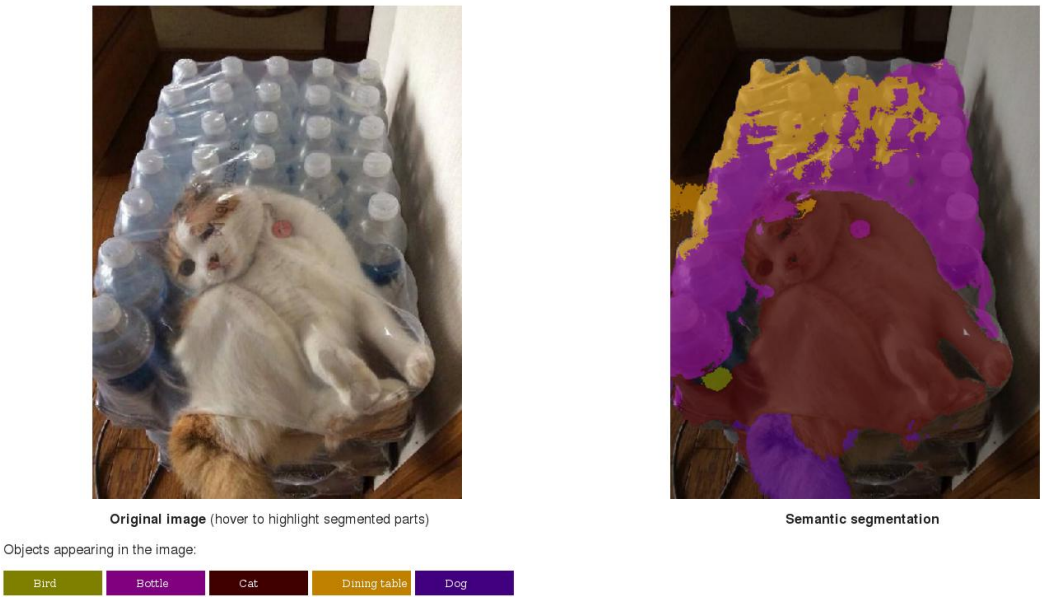

Figure 3: Image segmentation from [21], generated on their website. The users can upload their own images, proving they are not cherry picking their results.

2.2 Convolutional Neural Networks (CNN)

Neural network are a combination of neurons, which are a simplified model of biological neurons. A neuron (Figure 4) is composed of different scalar inputs, and only one scalar output. It is doing two things: first, a linear combination of its inputs (equation (1)), and then processing this combination through an activation function (equation (2)).

$$a = \sum_i w_i x_i \tag{1}$$

$$z = h(a) \tag{2}$$

x_i are the neuron inputs, and w_i are the weights for this particular input. They may evolve overtime, and change how the neuron react to a specific input. h is the activation function. According to the task the neuron is used for, different functions may be used (examples of usual functions in Figure 5). For binary decision, we may want saturating functions: 0 or 1, -1 or 1. For regression, no function or reLU might be more useful.

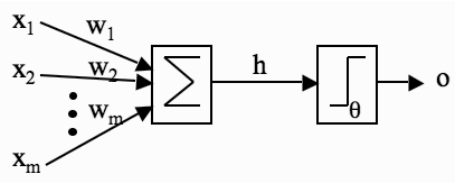

Figure 4: The model of a neuron used.

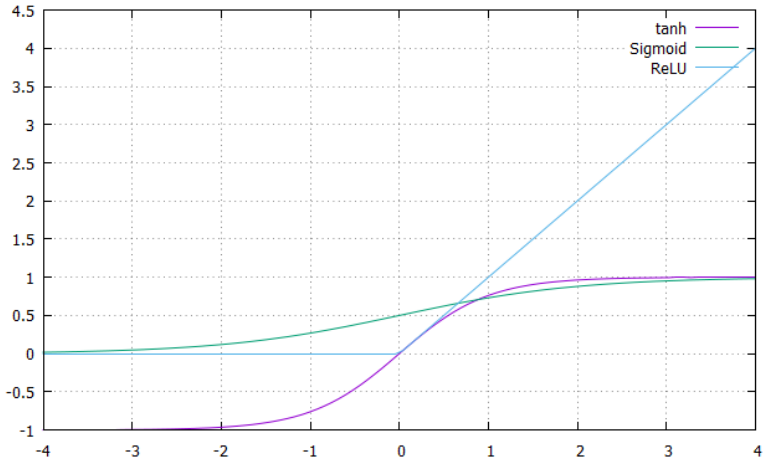

Figure 5: Three widely used activation functions. Tanh and Sigmoid are saturating functions: useful for classification. ReLU is useful for regression or for intermediate layers.

To build a network, we usually form layers of neurons: neurons are placed next to each others, and their output are fed to the next layer input. We can see in Figure 6 a

network with two layers: the hidden layer and the output. This architecture is referred as a multilayer Perceptron. We also say that the layers are fully connected: each neuron of a given layer is connected to each neuron from the previous layer.

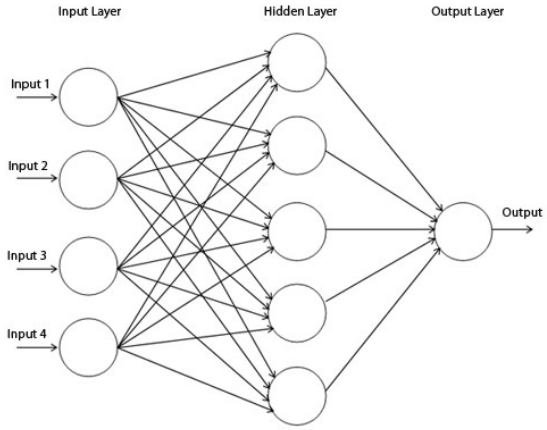

Figure 6: A multi layer perceptron, with one hidden layer. We can add as much hidden layers (of various size) as we want.

Convolutional neural networks introduced a new notion : feature maps [12]. They are a 2D array of neurons, feeding directly from the previous 2D layer (an image, or another feature map). But each neuron doesn't have their own set of weights: they all share the same, and each neuron takes a subpart of the image as an input. This is illustrated in Figure 7: the neuron computes a function from a 5x5 pixels input, and stores it in the feature map. The neuron next to it will do the same, but with a shifted input. Feature maps are heatmaps of the function defined by the weights, and produce translation covariant results. If the 0 in the figure shift on the left, then the feature map results will also shift on the left.

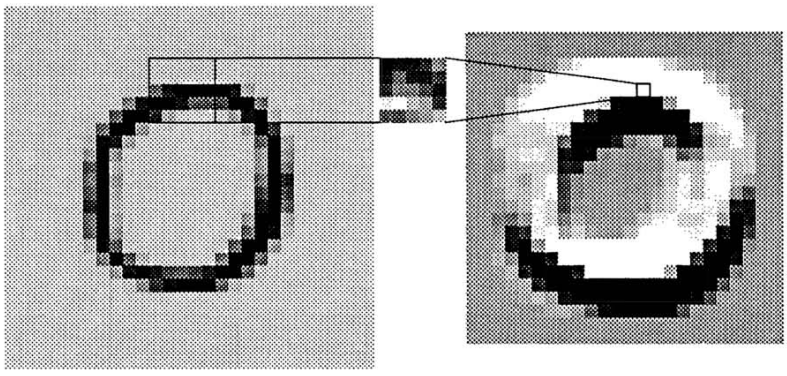

Figure 7: The feature map [12]: in the middle is the set of neurons, acting as a function. On the left its input, and on the right its output.

To build a CNN, feature maps are arranged in layers. The first layers will feed directly on the input image, and the next layers will feed on the previous feature maps. In Figure 8, we have four convolutional layers. We can notice that in the third layer, feature maps are feeding from two feature maps. This is actually the same: instead of having a 12×12 input, they have a $12 \times 12 \times 2$ input, and their weights are of size $5 \times 5 \times 2$. The last layer is fully connected: the network shown was used for digits recognition, and each neuron represent the confidence for one number. Those confidence can then be sorted, and the maximum one is the detected number.

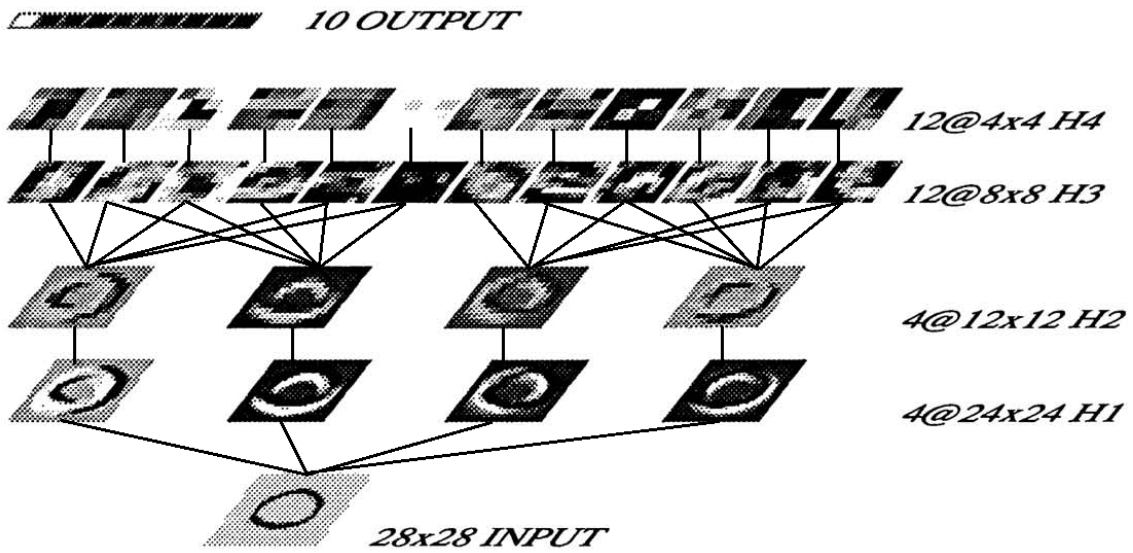

Figure 8: Network architecture from [12], which introduced the concept of feature maps and CNN. The neurons from a feature maps share the same weights, but each feature map will have different weights: they will all detect different properties of the input image.

Training a network consist to adapt the weights so the network does the right predictions. This is an optimization problem: we feed an input to the network, and with its output and the ground truth, we compute a loss that we want to minimize. The loss function depends of the problem. For classification, subtracting each class output to its corresponding ground truth (0 for all, except for one) might be sufficient. Then, we need to back-propagate this error, and to adjust the weights accordingly. Since this is difficult to know which weights are responsible for the misclassification (it can be at the last layer, or somewhere in the middle of the network), the gradients of the weights are computed: this is done by back-propagation. We first compute the gradient from the output units, and then the previous layer will compute its own gradients using the previous layer gradient. Figure 9 is useful for clarification, since we will reuse its notation. In it, the neuron is computing its gradient δ_j from the already computed gradients $\delta_k ... \delta_1$. Equation (3) shows

how the gradient for a specific neuron is computed.

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{3}$$

h' is the derivative of the activation function, and a_j is the linear combination of the neuron inputs (equation (1). w_{kj} are the weights between neuron j and neuron k, and δ_k is the gradient from neuron k. We continue with this formula through all the network.

Once this is done, those gradient can be used to compute the error for each weight, as in equation (4).

$$\frac{\delta E}{\delta w_{ji}} = \delta_j z_i \tag{4}$$

E is the error computed by the loss function, and z_i is the input from neuron i. The value computed will then be subtracted from the corresponding weight (in this case, w_{ji}), and the network response should now be closer to the ground truth. This is the Stochastic Gradient Descend (SGD).

The training is repeated many times. We use all training examples once (one epoch), and then shuffle their orders before starting again. We measure the performances (accuracy, loss, recall) over the number of epochs.

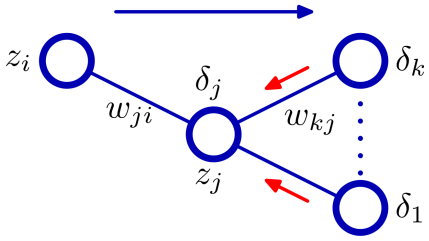

Figure 9: Figure illustrating the notation from 3 [2].

CNN, and neural network in general, are subject to overfitting: during training, the error for the training set keeps improving, but it starts increasing for the validation set. There is many ways to prevent this, but we won't talk about it in this section. The rest of the report will still be understandable, as we are not making use of those notions.

2.3 Related work

Using CNNs for object detection is not novel.

[7] used CNNs with a combination of region proposal system and SVM for it. The initial system was extracting region of the image that may contain an object, and feeding it to the CNN. The output of the last convolutional layer was fed to a SVM which was doing the final classification. The box coordinates were the ones from the initial region. This work was improved both by the same author [6] and other researchers [14]. The two

papers were published in a little time interval, and were doing similar things: instead of extracting regions and then process them in the CNNs, it was much more efficient to process the whole image and extract at the last convolutional layers the responses corresponding to the region. Girshick also replaced the SVM by a softmax classifier, common in image classification. More recently, [18] removed the region proposal mechanism by replacing it by a dedicated CNNs, the Region Proposal Network. The clever part is to share the first convolutional layers with the classifying network: this reduce a lot the overhead of having a dedicated network for the regions.

[8] is using the spatial pyramid pooling from [11] to create CNNs that takes input of arbitrary size. The architecture is simple: first, convolutional layers produce features for the whole image. It is then fed to the Spatial Pyramid Pooling layer. Fully connected layers then do the classification. To use it for object detection, SPP can be applied to different regions of the last convolutional layers, as in [6].

[3] is similar to [18]. Region proposal is done by a RPN sharing the convolutional layers. The difference is how the region is then classified. In this paper, the regions are divided into $k \times k$ bins. The classification is then done with the content of each bin. This is similar in some aspects to SPP.

But not all papers use a two stage processing (region proposal and then classification). [19] threat the boxes as a regression problem. On top of fully connected layers that does classification, they are predicting boxes coordinates. [17] is also using regression to predict boxes. The input image is split in a grid, and each cell will predict B boxes with their confidence and the classes probabilities. Some system will then suppress low probabilities boxes and duplicates.

2.4 Hough transform

Hough transform [9], and its generalizations [1, 13], use a voting mechanism. When detecting a part belonging to a bigger object, votes are cast (in a 2D or 3D space) to where the object center could be. If many parts cast votes to the same area, we can conclude that an object is centered there. By using the parts locations that casted the votes, we can infer a bounding box.

Hough transform has several interesting properties. First, it is efficient. Once the parts are detected, they are then processed only once. We don't have to run the same computations for each subwindow. Secondly, the voting space may take into account the scale of the image. The first two dimensions are the x and the y of the image, but a third dimension can be the scale. Now, to be detected, each part should correspond to the same scale. A 3D voting space in illustrated in Figure 10. The voting space is splitted in cells, and a threshold first select the ones with enough votes. Then, in the remaining cells, close votes are regrouped. Each group represent one detected object.

We can also mention that it is resilient to partially hidden objects (for instance, a

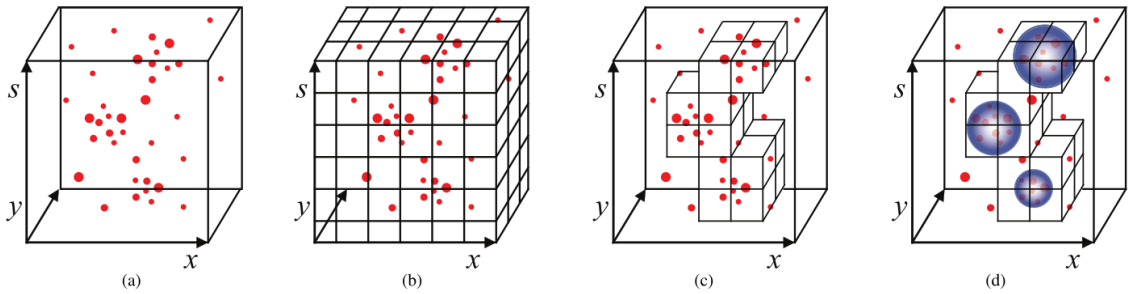

Figure 10: A 3D voting space, and the grouping detecting the objects from [13].

person standing in front of a car), or with different exposure across its parts (shadows overlapping a parked car). This is because we are detecting parts of the object, and we don't need to have all of them, but a sufficient number of votes.

2.5 Motivations

We didn't talk about how the parts are detected in the Hough transform voluntarily, mostly because it is specific to each application and method using it.

In our case, we want to use the power of CNNs to do it. After training, the convolutional layers acts as features (or parts) detector (each feature map showing the location and intensity of a specific feature). This is similar to vote, but in a dense fashion. Since networks have thousands of feature maps as an output, it can be a great basis for the Hough transform. We would only need to learn the scale on top of the location, and the set of features defining an object.

Since we didn't had a "Deep Hough Network" architecture already defined, we instead started from existing architectures and added gradually what we needed. We will present those steps in the next section. Since we didn't had time to complete everything, we will discuss in section 6 the future steps, trials and probably errors that we would need to try to achieve it.

3 CNN: From classification to detection

3.1 Pre-trained networks

CNN training is very long (for instance, it takes 3 weeks of continuous training for the AlexNet architecture [10]), even with modern GPUs. This is why many researchers [7, 6, 17, 19, 16] starts with standard architectures, or even pre-trained networks, and base their work on those. The main advantage is to skip long training time, and to have a reliable

and proven basis for their work. Networks trained on the ImageNet dataset have learned a huge number of features, usable for other purposes and datasets.

We started with the Caffenet architecture (shown in Figure 11), almost identical to the AlexNet Architecture. The difference is mostly technical: back in 2012, GPUs did not have sufficient memory to host the whole network, and Alex Krizhevsky had to split it in two parallel GPUs, overcoming the limitation of the time. Today's GPUs memory have sufficiently increased, and can host much deeper and shallow networks.

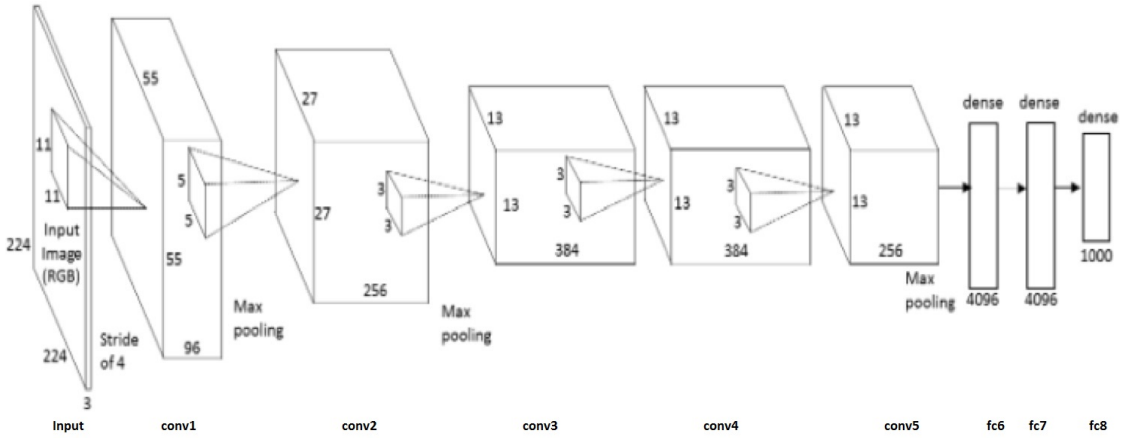

Figure 11: The CaffeNet architecture, with the layers labels. As in AlexNet, it has five convolutional layers, and three fully connected layers. The last layer has one prediction per class, that can be sorted to have the top predictions.

Note that the selected network was trained for image classification, and not object detection. The next section will explain how we make use of it.

3.2 Convolutionify

To do its classification, the network takes as an input a resized image (in our case, 227×227) and produce only one output, a vector of length n, n being the number of classes. From there, we can (apply a softmax and) sort them to have the top predictions.

Now, if we think of the last layers as convolutional layers (i.e., sliding the same function on subparts of an image), we can say that they are producing an array of dimensions $1 \times 1 \times n$. It just happens that their window (227×227) perfectly matches the input image size (227×227).

With convolutional layers we don't have any limit for the image size, and we will obtain more predictions for bigger images[1], each one being the classification of a 227×227

[1]It wouldn't work with smaller images, but it was always the case anyway.

window (with a stride[2] of 32 pixels between each window). The result will now be an array of dimension $x \times y \times n$, where x and y depends on the image dimension (equation (5)).

$$x = (image\_length - kernel\_size)/stride \tag{5}$$

With Figure 12 as an input, we now have a heatmap for each class (Figure 13), or, the other way around: a prediction for each subwindow of the image (Figure 14).

Figure 12: Input image for Figure 13 and 14

It is also quite straightforward to go from a prediction coordinates (the cell position in the heatmap) to a box coordinates (actual pixels in the image): we need to multiply it by the stride between each window (equation (6)) for the x_{min}, and to add the kernel size as well for x_{max} (equation (7)).

$$x_{min} = i \times stride \tag{6}$$

$$x_{max} = i \times stride + kernel\_size \tag{7}$$

(i, j) are the cell from the predictions. i is only used when computing the xs, and j is used for ys in a similar fashion. In our case, $stride = 32$ and $kernel\_size = 227$.

By combining this network with a threshold Non Maxima Suppression from Section 4, we can already obtain qualitative results, as shown in Figure 15. This proves the network is already able to detect objects and suppress duplicates or low responses, but also shows

[2]The stride is the number of pixel between each tested window.

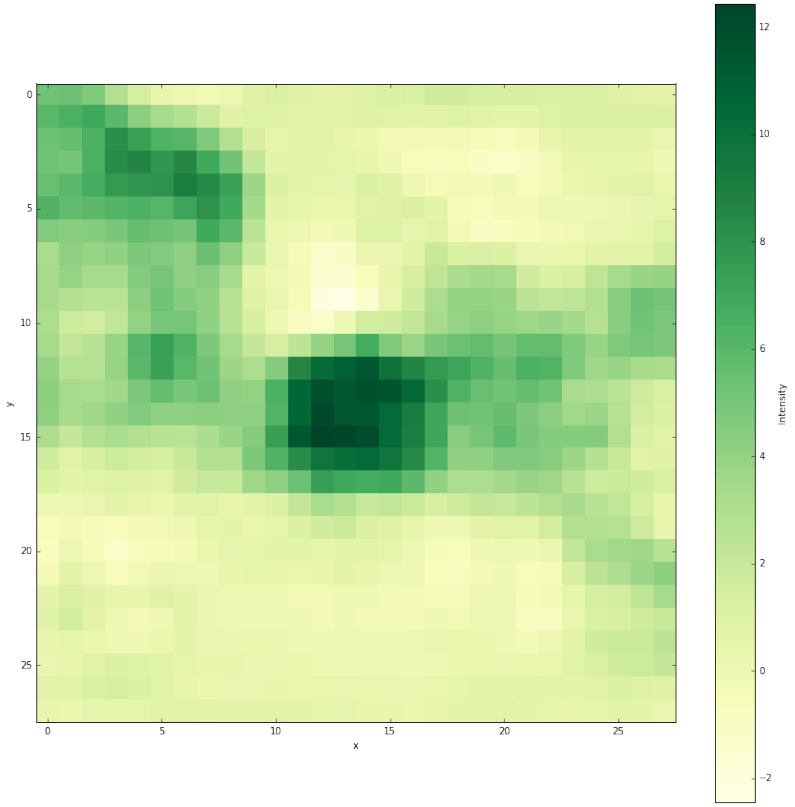

Figure 13: Heatmap of the tabby cat class. We can easily see that the higher responses are located around the cat, or around the confusing dogs to a lesser extent.

two problems we will have to solve: classes for the classification, and scale. Classes from object detection dataset (as the VOC2007 [5]) do not match the classes from the ImageNet, and we cannot use right away the network predictions. The next section will focus on how we solved this problem.

3.3 Retrain the last layer

The solution is to retrain the last layer (fc8, the one doing the predictions for the 1000 classes). We can see the first layers as a big feature generator, and the last layer is merely aggregating the features for each class.

Since the ImageNet dataset has so many examples (1.2 million images in the training set) and so many classes, we can expect to be able to reuse the features learned during the initial training. Some of them may not be relevant, but we expect the training to ignore them.

To do this fine tuning [6, 7], we load the network with the learned weights, and then replace the last layer by a smaller one, with less outputs. We then use the same training

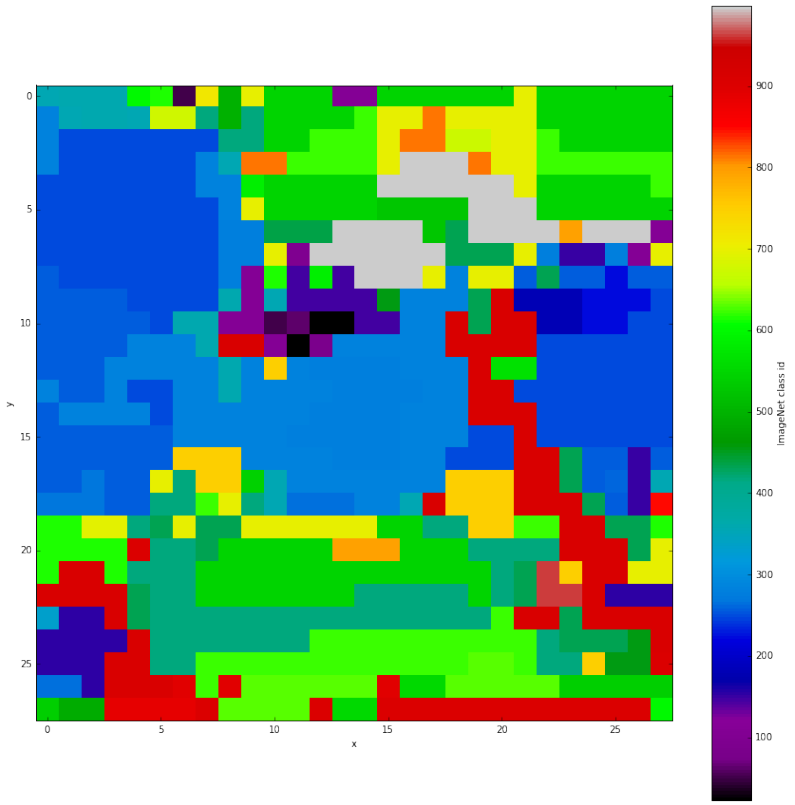

Figure 14: Top prediction for each subwindow. Knowing that the tabby cat ID is 281, it explains the blue area in the middle of the image.

process, with our new dataset. We tried two methods: one with the 20 VOC2007 classes, and one with 20+1 classes.

3.3.1 20: Object classes

Dataset To retrain the last layer, we first needed new dataset. We tried two methods.

First we retrained the network in the exact same manner as in its original training: feeding the whole resized image, with the object class as a label. The network will figure out by itself what actually define the label provided as ground truth. But this was problematic for images with several objects [3] : we would give two different truths for the same image, contradicting ourselves, and probably confusing the network.

This is why we decided to use the ground truth for pre-processing, and crop the objects directly from the images. This way, each image would appear only once during training. But since the image is resized to a square before being fed to the network, is would change

[3]That is, pretty much any image in the VOC2007 dataset.

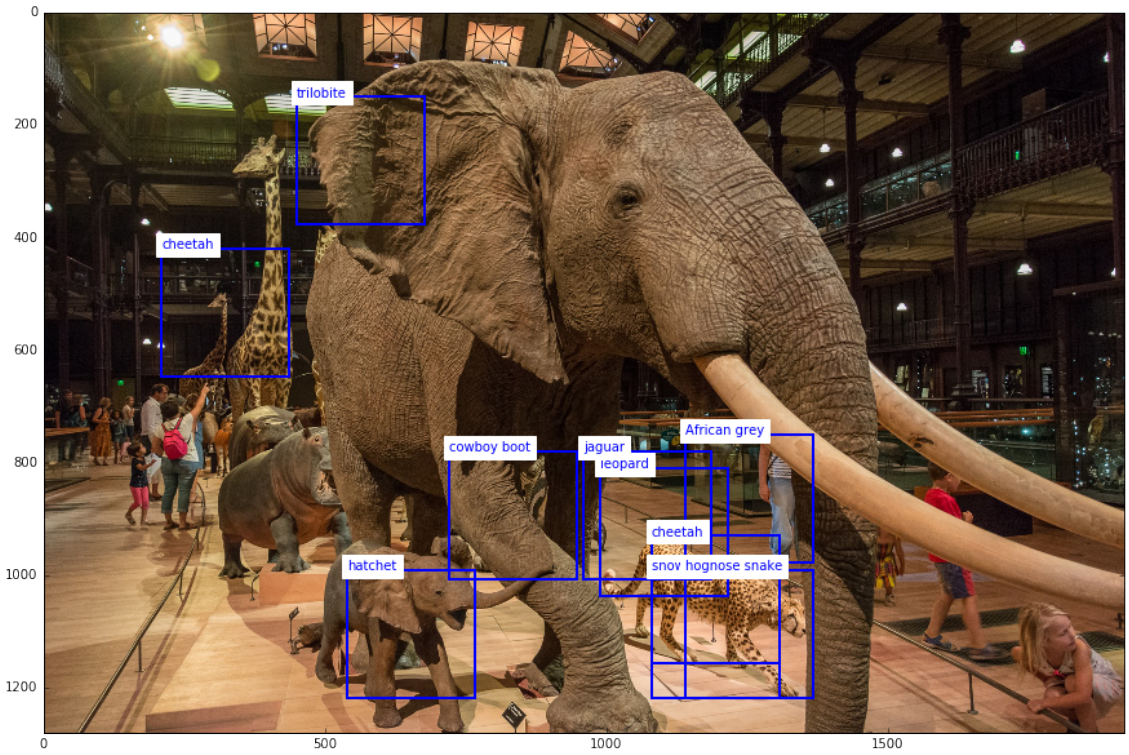

Figure 15: Quick preview of a fully convolutional network with basic Non Maxima Suppression. Due to scale problems and number of classes from the ImageNet dataset, some results are not making much sense.

the aspect ratio of the object (Figure 16). This is usually not a concern for classification, but since we intent to use this trained network for object detection later on, we want the training samples to be closer to its future use case. Therefore, we cropped squares matching the object maximal edge, centred on the object.

As expected, the second method, on top of making much more sense, had much better results. We can see the loss and the accuracy of both methods in Figure 17.

Layers to retrain Then we wondered if it made a big difference between training only the last layer and training every layers. We could expect the later to be as least a little bit better, but maybe not that much, since the network contains already so much knowledge.

The results are shown in Figure 18. Training the previous layers indeed increase the accuracy by 2/3%, which is not that negligible.

Confusion matrices We now have a dataset and a list of layers to retrain, and need better metrics to analyse our results. We especially want to check if all classes are detected

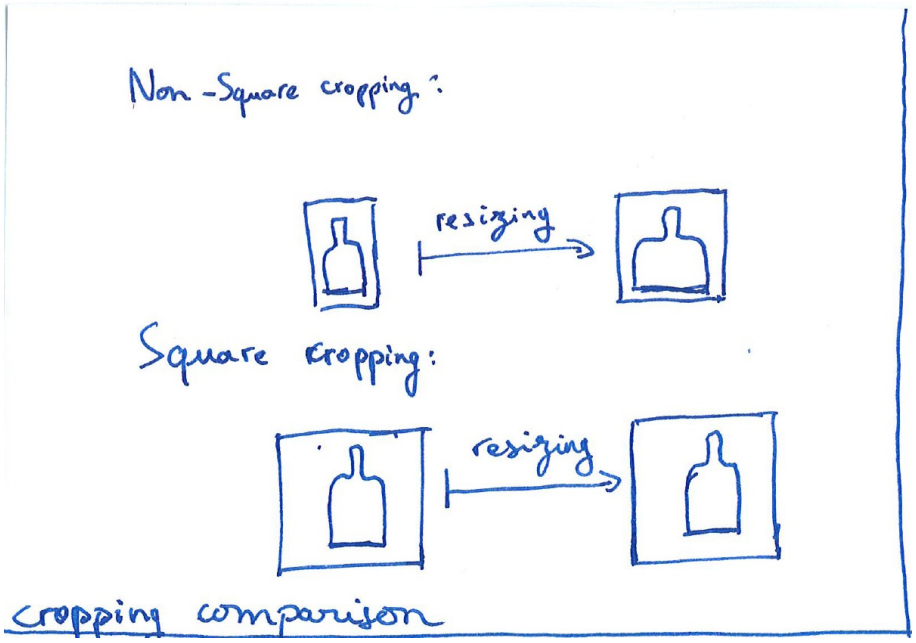

Figure 16: We want to avoid deforming the objects, to match the aspect ratio of objects during detection.

with similar scores. We know that the classes in the VOC2007 dataset are not balanced (with much more persons than the rest combined), and don't want to have a great accuracy for this class and gibberish for the rest.

If we divide each number in the confusion matrices by the number of ground truth class (so that each column adds to one), we obtain the probability of classification for each class. In Figure 19, we can see that the results are consistent across all classes. The two weak classes are 8 and 17, chair (1432 instances) and sofa (425 instances) respectively, of a total of 15662 objects. We can see that most objects are misclassified as a person more than anything else. It could be reduced by balancing the training set, but it does not invalidate our model.

3.3.2 21: With backgrounds

Image classification must always find a label. However, when it comes to object detection, most of the time there is nothing to detect, and we want our network to be able to say so. To ease our future steps, we can add a background class (hence the $20 + 1$ classes). This way, the network will have a way to say explicitly that there is nothing [4].

[4] On the network point of view, the background class is no different than any other, it is only our post-processing that differs.

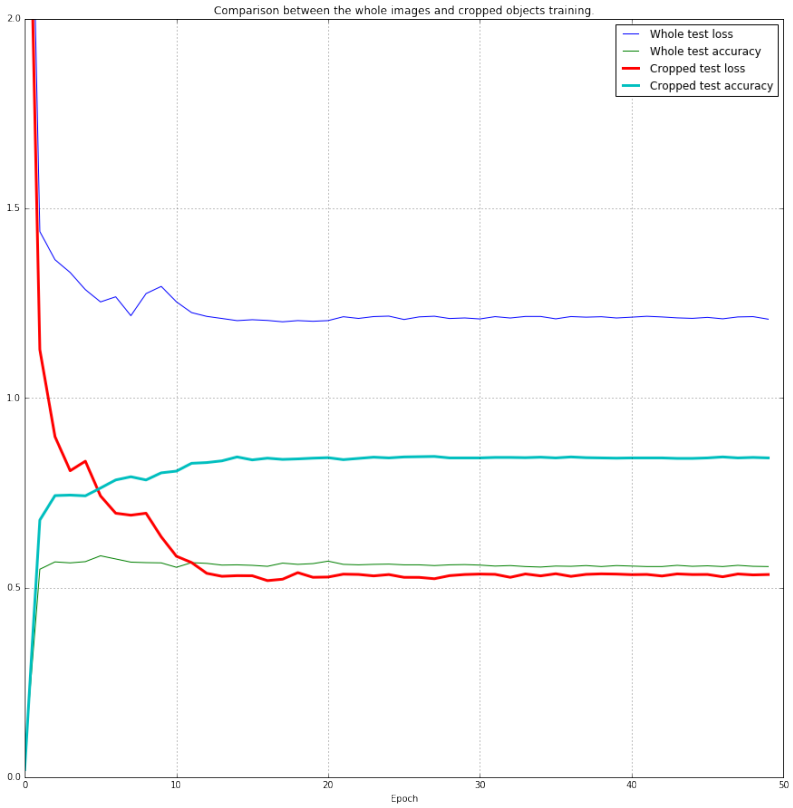

Figure 17: Isolating the objects greatly improve our results, because we stopped contradicting ourselves by giving several ground truths for the same image.

Background definition Backgrounds are random patches of images not containing or overlapping objects. We consider the objects overlap the patch if at least one object has a IoU [5] over a threshold t. We tried different thresholds: 0.5, 0.25 and 0.0. Figure 20 shows the accuracy over epoch when training a network $20 + 1$ classes.

Lowering the overlap between background examples and objects doesn't change much in terms of accuracy. However, it makes less and less sense considering what we want to do with the trained network: when doing object detection on an image, we will have a lot of overlap between the object and the rest of the image. Therefore, we will keep the 0.5 value for the threshold definition.

Positive and negative balance Positive examples are patches containing an object, and negative examples are background patches. We now need to define how many positive and negative examples we want to use. What is actually relevant is not their sheer number, but the ratio between the two. We tried three values: 90/10, 80/20, 50/50. The results are shown on Figure 21.

[5]Intersection over Union, a widely used metric for box matching. The area of the boxes intersection is divided by the area of their union.

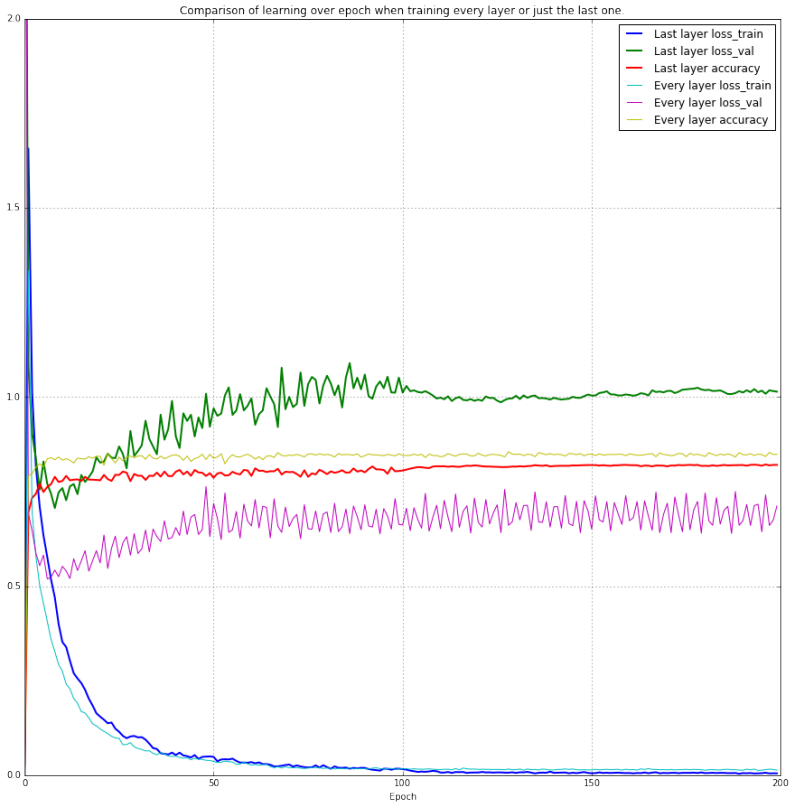

Figure 18: Loss and accuracy for the training and validation set. Accuracy for the training set is unfortunately not available.

We can see that the 90/10 and 80/20 ratios are really similar in term of accuracy. The 50/50 ratio is doing a little bit better, even if it takes quite some time to achieve it. The 50% accuracy for its first epochs could be misclassification for most objects as background, before the predictions finally improves. Since the results are similar for each ratio, we will keep the 50/50 one: it is closer to the future use case, where most subparts of an image are background.

Training parameters Once the backgrounds and ratio were defined, we had to tweak the learning parameters. Starting from the original network parameters, we tried to lower the learning rate[6], lower the momentum[7] and increase the weight decay[8]. The results are shown on Figure 22.

We can see that our instinct was right about the parameters, and really improved the training. However, we didn't had time to continue tweaking those parameters. Nonetheless, it shows how we must not overlook small differences in configuration files, and still provide

[6]How much we take the gradient into account during each weight update.
[7]When a weight start increasing or decreasing, it gains momentum in this direction.
[8]Weights that are not often updated lose their value over time.

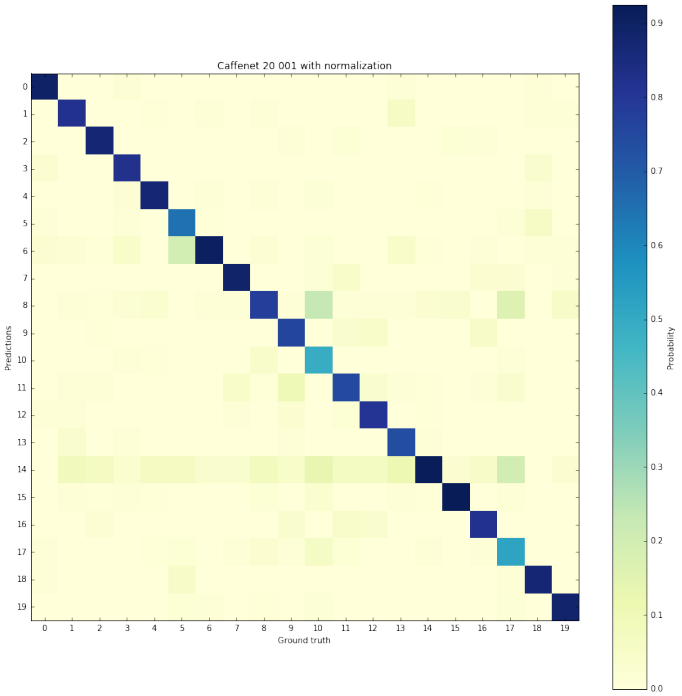

Figure 19: We can clearly see the bias for the class 14 ("person" in the dataset). The results for the other classes remains acceptable.

really satisfactory results.

Confusion matrices We now have every parameters we want for our network, and we can check the confusion matrix, to ensure once again that the results are consistent for all classes. As we can see in Figure 23, it remains acceptable. As in Figure 19, most classes are misclassified as a background, the most present class in the training set. But there is no way to fix the balance between classes, as we explained earlier. The results still remains completely acceptable, and should be a good basis for object detection.

Now that we have defined, created and fine-tuned our base network, we can explain how we are selecting boxes.

4 Non-Maxima Suppression

We now have a network producing heatmaps of different classes, we need to select the responses that correspond to an object. There is actually two different things to do: filter the low responses (that would produce false positives), and regroup close cells (to avoid duplicates).

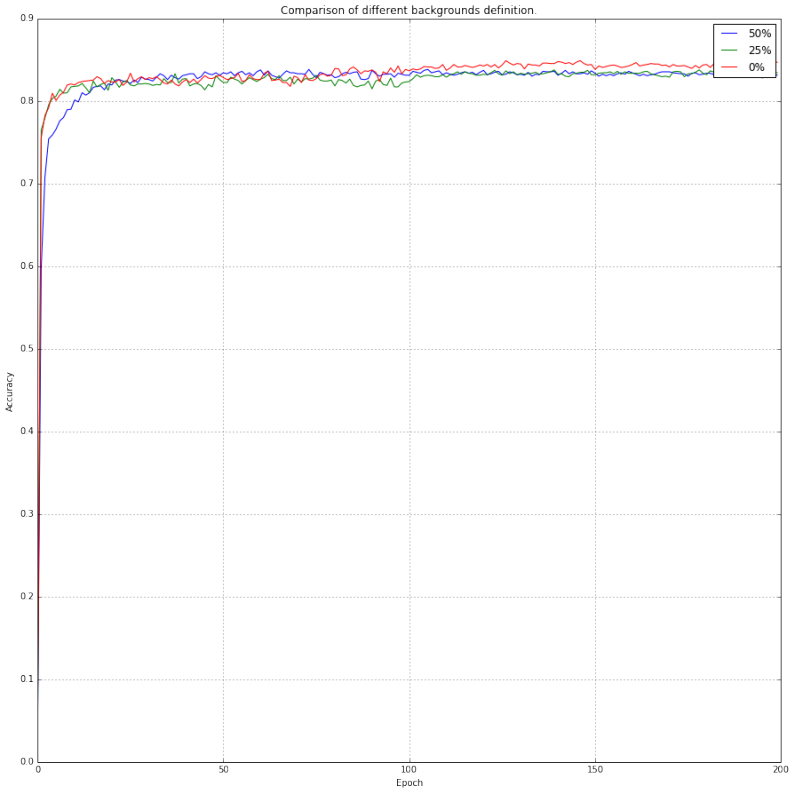

Figure 20: The difference between the different threshold is negligible.

4.1 Filtering step

Sometimes there is nothing to detect (for a given class). What we want to do is to discard the low responses of the heatmap (Figure 13). There is two common solutions to tackle this problem.

Threshold We defined a minimal value (that we found experimentally) of the network response, that each cell should attain or exceed. If not, its response is set to zero. Finding a good threshold value is difficult, and is not subject to any learning. Instead, the responses would have to adapt around this threshold. It works, but it is difficult to find a good value for the threshold.

Background class Having a background class solve this problem: if the background class is higher in a given cell, then we can say that there is nothing in this cell. The learning is much easier, since the background class is treated by the network as a regular class.

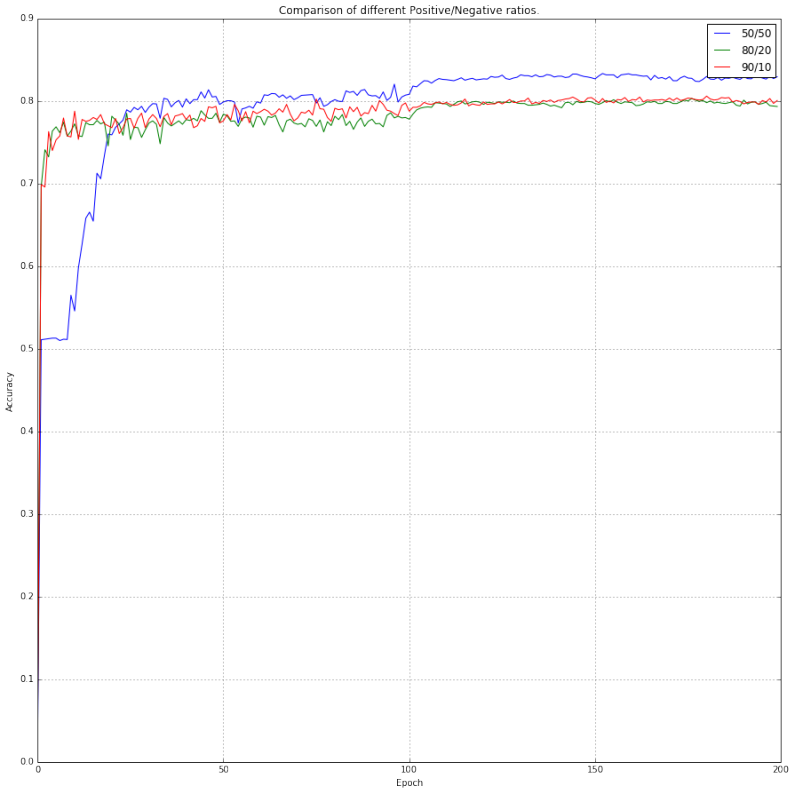

Figure 21: Although starting painfully, the 50/50 ratio yield noticeable improvements over time.

4.2 Regrouping

The other thing to do is to go from a set of neighbour responses to only one response: we don't want to have many boxes that correspond in fact to the same object. Figure 24 illustrate the problem: several boxes are indicating the same cats.

The solution is to compare a response to its neighbourhood. Usually, a neighbour is a box with an IoU superior to a certain threshold, usually 0.5. In our case, neighbours are the close cells on the x, y axis (position in the image), regardless of the value on the c axis (classes for a given position).

If a response is the maximum one among its neighbourhood, we keep it. If not, we set it to zero. Figure 25 shows the result that we can obtain.

4.3 Testing

To test the NMS layer plus the fine tuning, without taking into account the scale problem (when an object is bigger or smaller from the network kernel), we decided to resize images

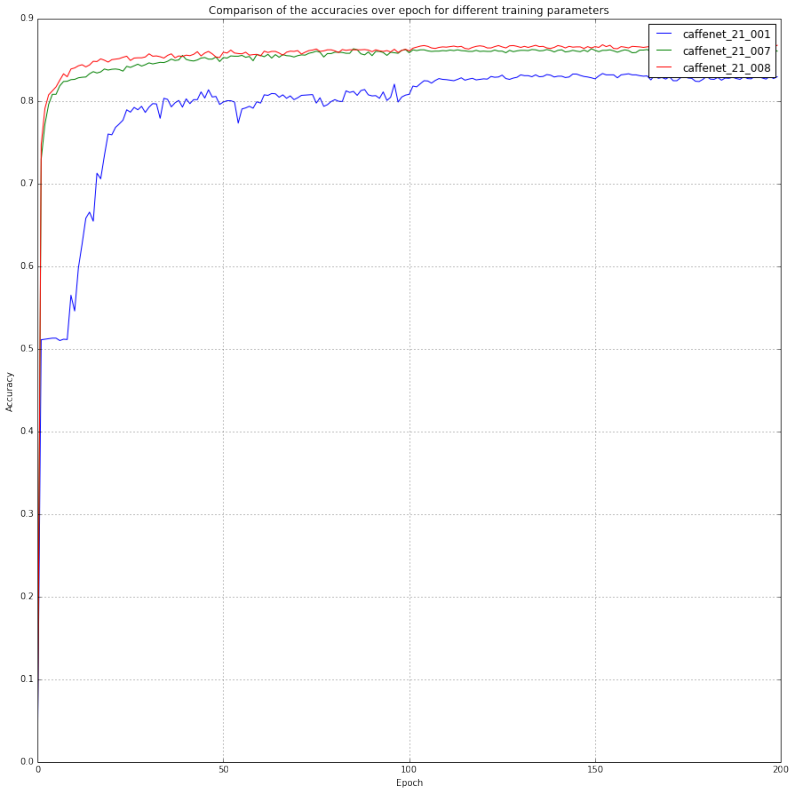

Figure 22: Our new parameters greatly improve the results from the original parameters.

so the objects can fit the detection window. Dealing with the scale is left for future work, and section 6 explains different approach that could be tried.

We can now compare the results between the two kinds of NMS. The one with the threshold use a network trained on 20 classes, whereas the background one use a network trained with 21 classes. Figure 26 shows an image with a threshold filtering: we can see that the labeled object are detected, but there is many false positives in the image. On the contrary, Figure 27 produce much less false positive. We left the background boxes (although in another color) to help understand what is happening.

5 End-to-end training

Now that we have a complete network for detection, we can try to train it, and improve its performance. Since this is still an optimization problem, we need to define a function minimize: the loss function. First, we will explain how the loss function we chose works, and then we will explain how back-propagation works in our network.

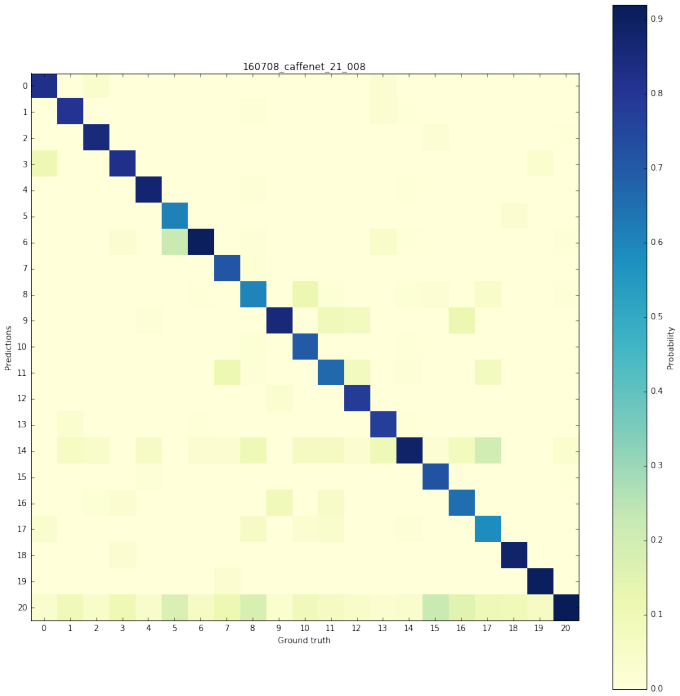

Figure 23: Confusion matrix of the trained network we we used for the next step.

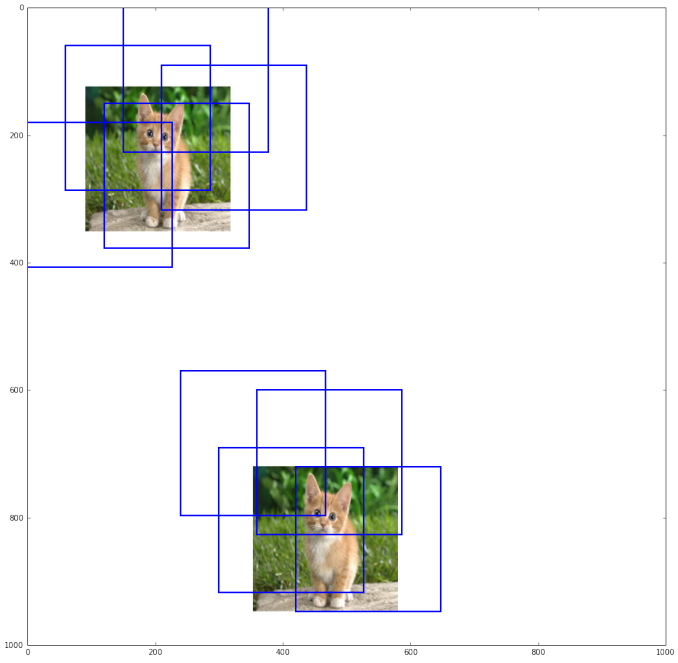

Figure 24: Boxes kept after a low threshold filtering: each remaining box is true, but they are duplicates of each others.

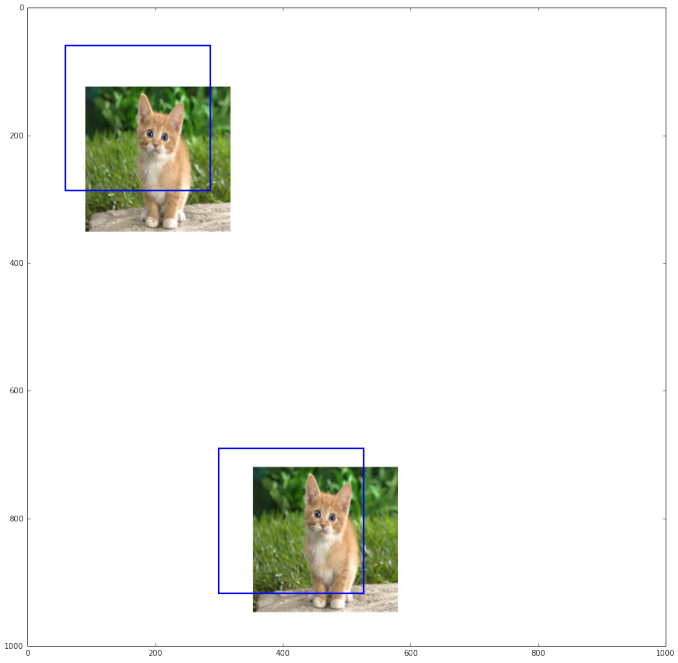

Figure 25: After regrouping, the whole NMS keeps only one box per cat.

Figure 26: By having low threshold, the filtering step produce too many false positives.

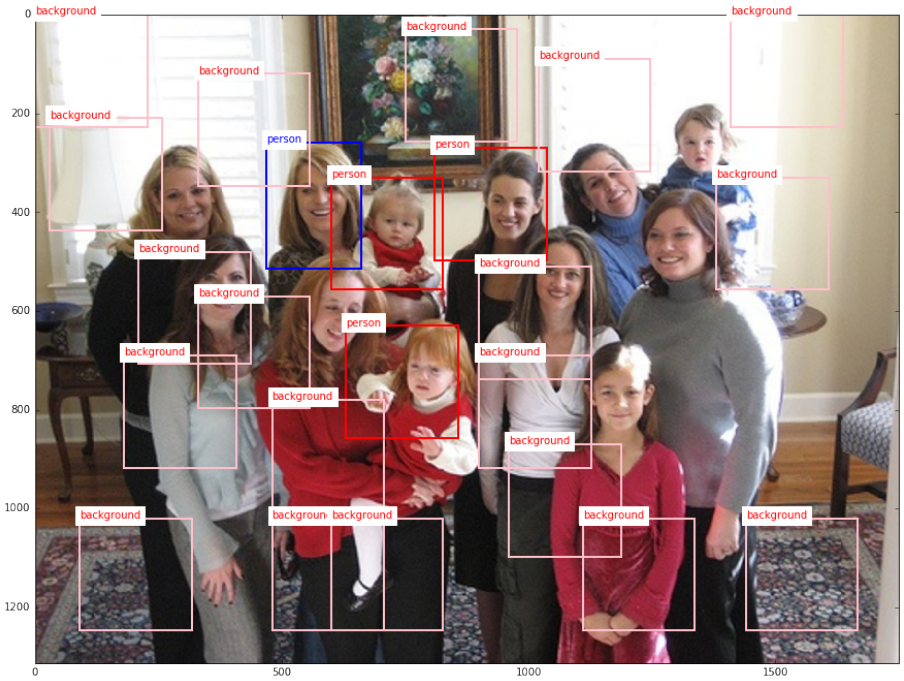

Figure 27: We can see that there is much less false detection when using a background filtering.

5.1 Loss function

We use the function defined in [20]. Their motivation was to take into account the NMS layer when computing the loss. In Figure 28, three boxes are detected for the same person. The blue one is of course closer to the ground truth, but if the red or green boxes have higher responses, it would be suppressed. To fix this, the network must be trained so that the blue response is the highest one.

Instead of using only the boxes selected by the NMS layer (A), they use the predictions from fc8 (B) as well. Figure 29 shows our network architecture with such loss function. A' is the set of boxes before NMS that match the ground truth, with only one box per ground truth. Algorithm 1 develops how to compute it. *overlapping_boxes* compute the

Algorithm 1 The algorithm used to compute A'.

```
A' = empty_set
for box in ground_truth do
    neighbours = overlapping_boxes(box, B)
    best_box = argmax_box(neighbours)
    A'.add(best_box)
end for
```

set of boxes that overlap the box given as first argument. The argmax will then select the

Figure 28: If the responses from the green and red boxes are higher than the blue box, the NMS layer would keep them and not the blue one.

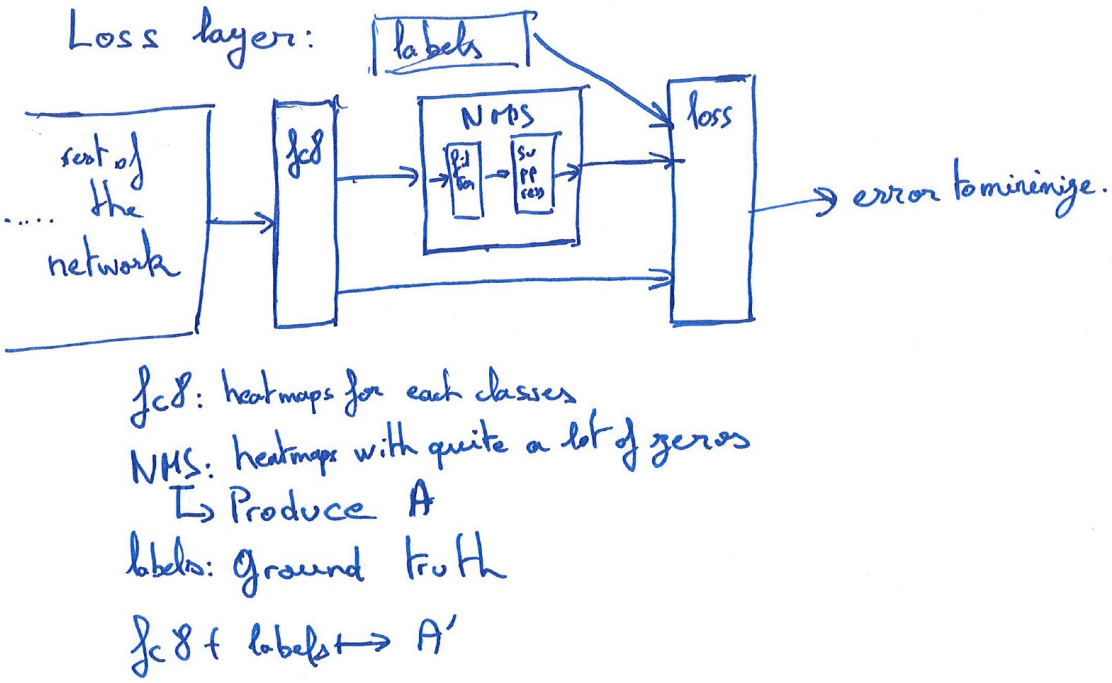

Figure 29: A temporary illustration of the architecture for the loss layer.

neighbour with the highest response. In the end, we have one box per ground truth.

Then, the cost of each set A and A' is computed, and we want to minimize the difference

between the two costs. $C(A)$ is computed in (8).

$$C(A) = \sum_{(b_i,y_i,r_i)\in A} H(r_i, y_i) + \sum_{(b_j,y_j,r_j)\in S(A)} H(r_j, 0) \tag{8}$$

$$H(r, y) = I_{y=0} max(0, r + 1)^2 + I_{y>0} max(0, 1 - r)^2 \tag{9}$$

$$S(A) = B \backslash neighbours(A) \tag{10}$$

b, y, r are the box coordinate, its class and its response. $y = 0$ correspond to the background class. I is equal to 1 if its boolean is true, 0 otherwise. $C(A')$ is computed similarly. The idea of this cost is to measure not only the cost induced by the boxes, but also by the background.

Ideally, A and A' will match: they will therefore have the same values, and cancel each other. Since we are subtracting the two costs, the final loss can be simplified to (11).

$$L(A, A') = \sum_{A'} H(r', y') - \sum_{A} H(r, y) + \sum_{N\backslash N'} H(r, 0) - \sum_{N'\backslash N} H(r', 0) \tag{11}$$

N and N' are respectively A and A' neighbours. Figure 30 gives an example of the sets used in this formula. Because of the loss, we can expect the responses from A to lower during training, and the responses from A' to increase. If it is successful, NMS will now produce a set A identical to A', and the loss will be zero.

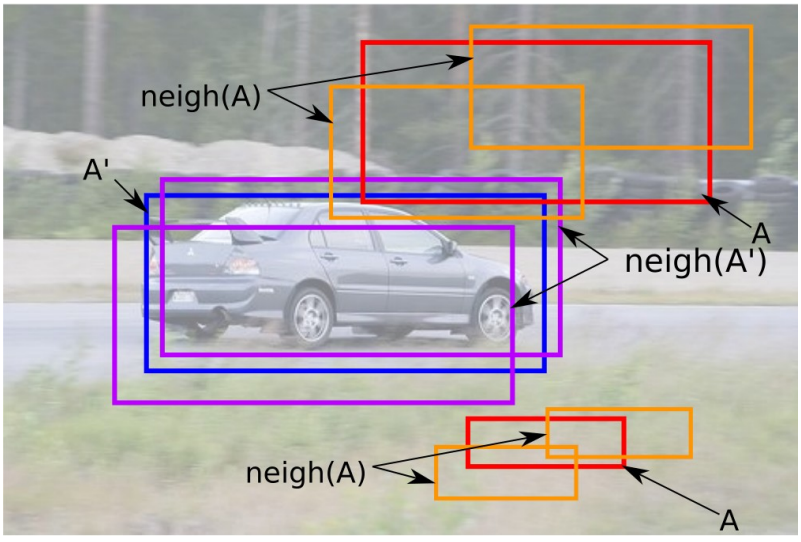

Figure 30: In this example, A and A' do not match, and we expect to have a high loss. Moreover, using N and N' instead of $S(A)$ to compute the cost greatly reduce the complexity.

5.2 Back-propagation and derivatives

To be able to back-propagate the error, and therefore train the network, we need first need to compute the gradients. Gradients from the loss layer are fairly straightforward: it is the loss derivative wrt the network response. From (11) and (9), we can write easily (12) and (13).

$$\begin{aligned}\frac{dL(A,A')}{dr} =& \sum_{A'} \frac{dH(r',y')}{dr} - \sum_{A} \frac{dH(r,y)}{dr} \\ &+ \sum_{N \setminus N'} \frac{dH(r,0)}{dr} - \sum_{N' \setminus N} \frac{dH(r',0)}{dr}\end{aligned} \tag{12}$$

$$\begin{aligned}\frac{dH(r,y)}{dr} =& I_{y=0} max(0, 2(r+1)) \\ &+ I_{y>0} max(0, 2(r-1))\end{aligned} \tag{13}$$

The network responses that the loss function is using are actually a $x \times y \times c$ tensor, where x and y are the grid size, and c the number of class. What we call a box is actually the response r from a specific cell, with coordinates (in pixels) inferred from the x and y values ((i, j)). The gradients are stored in a similar way, in a $x \times y \times c$ tensor. To fill this gradient tensor, we do not need to test if each cell belongs to A, A', N or N'. It would be really long, and pretty wasteful since most responses belong to none of them.

What we can do instead is to initialize the gradient tensor with zeros, and when using a response in the forward pass, append its derivative to the corresponding gradient cell. Appending, and not setting, because some responses may be used twice: in A and A', or in A and N' for instance. Algorithm 2 shows an (uncomplete) forward pass algorithm to do it. *zeros* create a new tensor filled with zeros with the specified shape.

Algorithm 2 Compute the loss derivatives wrt the network responses.

```
A' = closest_responses(B, labels)          ▷ As shown in Algorithm 1
derivatives = zeros(B.shape)               ▷ B.shape is x, y, c.
for r,(i,j,y) in A do   ▷ (i, j, y) being the indexes of the responses tensor, and r the actual response.
    Loss += H(r, y)
    derivatives(i,j,y) += dH(r,y)/dr       ▷ This is new.
end for                                    ▷ Similar For loops for A', N and N'
```

Once we have all the gradient computed for the loss layer, they can now be back-propagated in the network. The NMS layer won't have any use of it, and it will be fed directly to the last convolutional layer, fc8. From fc8, the SGD will behave as usual, and the learning process will take place.

5.3 Evaluation and training

As we are writing this report, measurements and training are still ongoing, so we (unfortunately) do not have any result to present for this part. We, however, expect to have them for the final presentation in September.

6 Further developments

As stated several times, the next step would be to manage the scale of the objects. Several methods can be investigated.

6.1 Voting space

In section 2.4, we highlighted the difference between 2D and 3D voting space. At the moment, `fc7` is the last convolutional layer producing features, and is acting similarly to a 2D vote with its heatmap. We can consider that all of them are casted on the same scale.

We could add additional feature maps to `fc7`. For instance, another feature map for each existing one. The current is predicting the intensity of the object, and the second one could predict the scale. We would only need to change the activation function of the new feature maps, depending on how we want to represent the scale values (continuous, between 0 and 1 ?).

Also, we could add two other predictions per existing one: dx and dy. We are voting for the object center, and not the object positions. dx will represent the offset in the grid to the center for the x axis, and dy for the y axis. For those layers, we would need another activation function, that produce positive and negative integers.

Then, `fc8` would act as a mask: it selects a subset of features (r, s, dx and dy) and cast them in the 3D voting space (class, x, y, scale). Of course, would have n voting space (one per class), so we end up with a 4D voting space. The NMS layer could be updated to take into account the scale, and work in a similar way as in Figure 10.

The loss layer wouldn't need to be updated, apart from the minor tweaks in the way we represent the boxes internally. This architecture is summarized in Figure 31

6.2 Multiple images and aggregating

Another solution would be to feed the same image to the network with different sizes, and then aggregate the different outputs.

For instance, we could decide that a big box on top of smaller boxes would cancel

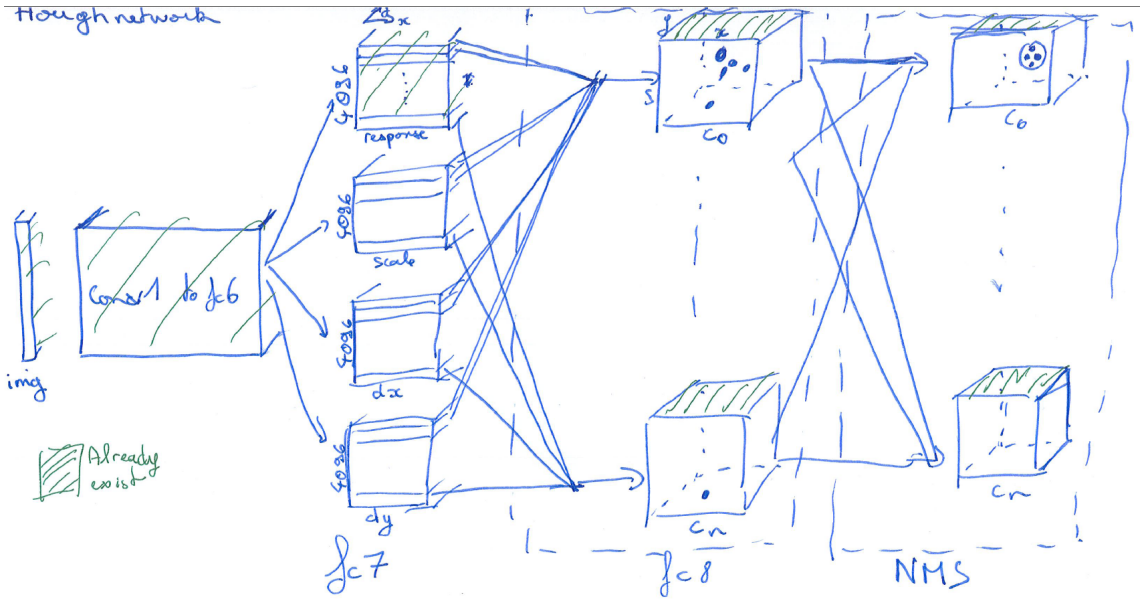

Figure 31: The architecture proposed to reproduce the Hough Transform after the CNN.

the small ones, depending on the responses difference, or that a set of small boxes can be regrouped as one big box.

The work in [15] could be a good basis for this, were the authors are merging boxes instead of using Non Maxima Suppression. [15] is even closer to our work, since in this work, objects are split is square that are merged at the last stage of the detection.

6.3 Box regression

A third solution would be to add four numbers for the predicting cells. At the moment, we only have square boxes, and we could use those numbers to deform this original square. We already talked about several papers [19, 17] that are doing boxes regression, and could be used as insights to our work.

7 Experiment reproducibility

This section will be more about the engineering side of my internship. It was motivated by some difficulties I faced in the past, during my previous internships in research, and that are never easily resolvable. Good science is sometimes difficult to define, depending of the field, but it usually rests on the following properties:

Predictability A good theory can explain the current measurements, but can also predict future results. If future measurements are in accordance with the predictions, the theory hold, otherwise, it is wrong. Some theories may hold a very long time (Newton's law of universal gravitation for instance) before being proven wrong. New theories arises (in this case, the general relativity) to explain the new data, but the old theory may still be kept for simplicity in some particular cases.

Falsifiability Results from an experiment could potentially invalidate the theory or the hypothesis tested. For instance, if I say that a giant undetectable Spaghetti Monster flies in the sky, we wouldn't be able to discard this theory ("Of course you cannot detect it, it's undetectable"), and wouldn't have a great value on a Scientific standpoint.

Reproducibility When doing the same experiment in the same initial conditions, we must obtain the same results. If not, it would mean there is some parameters that have not been identified, and it wouldn't be possible to formulate a reliable hypothesis based on the experiment. It can also mean that with new data, we should be able to have the same conclusion (this is especially true in epistemology, but often difficult due to the prohibitive cost to collect new data). In computer vision, in means that using a new dataset with the same properties should produce similar performances.

I will focus on how I tried to attain a good reproducibility of my experiments during this internship. I will first explain how I managed the experiment/data acquisition phase, and then I will talk about the data analysis. The solutions I will present are not universal. Computer vision has some particular requirements: dataset can be extremely huge (138 GB for the ImageNet training set), making archiving slightly more difficult than language processing, for instance. Machine learning is also specific: training data need to be stored somewhere, but not on the code repository using them. Simulation, for instance, may not have this problem: experiments configurations files are all that is needed, and their size are negligible.

My code is available on `https://gitlab.irisa.fr/00008A4G/ouf-networks`. You are invited to check it out while reading the next sections, to have a better overview of my work.

7.1 Datasets

We had to pre-process our datasets on a regular basis, sometimes with little difference between iterations (Changing the threshold for the overlapping, testing different mix ratios, among others). To avoid confusion [9] , I first wrote completely automated scripts to generate the data from a vanilla dataset. Hardcoding parameters could work, but it would then be harder to compare different datasets, especially when only a tiny parameter is changed.

[9] For instance: Which parameters I used to generate this dataset I used to train this network ? And is it the same used for this other network ?

Also, forking a dataset would be error prone, since it would be easy to forget one parameter in the script.

The solution was to put everything as script parameters, provided when calling the script. I then used wrapper scripts (in the form of Makefiles) to make those calls in a reliable fashion. Most of the work was made by a common script, and the Makefiles contained every parameters: the dataset definition was now a few lines in lengths, and are now easier to compare. Mistakes are also more easily avoided. Since everything is stored in the header, is it harder to forget to change one parameter that was defined elsewhere in the script.

I also tried to have a clear and non-ambiguous naming convention for my datasets iteration: `<dataset_name>_<iteration>`. For instance, we have `class_voc_20_001` to fine-tune with 20 classes, and `class_voc_21_xxx` to finetune with 20+1 classes.

7.2 Experiments

I used similar makefiles for my experiments: parameters were either specified in the makefile, or in other configuration files (such as Caffe configuration files). Either way, they are easily tracked by Git.

I tried to have as little code as possible in the experiment directories. Common utilities were stored elsewhere, and in our case, most layers were standard Caffe layers that didn't needed specific code. For our layers development, however, it was better to save them into the directory itself. Otherwise, we would have difficulties to maintain back-compatibility when tweaking their behaviour, and would compromise the whole reproducibility thing.

Experiment results were stored in two places. Log files, and small text files were saved directly in the folder, but ignored by Git. Is was therefore easy to avoid confusion. For bigger files, such as the network weights snapshots, folders were automatically created in a dedicated drive based on the experiment name. This prevented confusion, and overwriting as well.

The naming convention was similar as the one for the datasets:
`<date>_<network_name>_<iteration>`. Our networks for fine-tuning were named `caffenet_20` and `caffenet_21`, while our network was named `ouf_networks`. Future experiments on the Deep Hough network architecture could include the tested property as well:
`ouf_networks_ETE_training` or `oufnetworks_new_NMS` for instance.

7.3 Results analysis

Even with properly named results, it can sometimes get confusing to know how we generated some diagrams. Which code was used, which results ? If we are not entirely sure, we have to redo everything from scratch.

To solve this simple yet tricky problem, I used the Jupyter software. It is a webserver running "notebooks": they contains cells of code, and display after them their output. Markdown[10] can also be used to explain the experiments, the analysis done and comment the results. Figure 32 shows a screenshot of a notebook, and my notebook HTML exports are available at `http://oufnetworks.gforge.inria.fr`.

We therefore have the inputs and outputs of the analysis, in a readable way. If we want to update the graph with new data, we just need to rerun each cells.

7.4 Improvements

There still is plenty of room for improvement to my workflow. Those solution were devised on the fly, since I didn't had much time to go back and refactor my previous work. For instance, I could have added a file generated by the Makefile, tracking each software installed and its version, with as well the commit hash of the code when the experiment was run. It can be especially useful for the far future, if some libraries introduce or fix bugs in new releases. It would then be feasible to reinstall the exact same environment, and ensure the difference was due to some third party dependency.

But in any case, I've really seen the difference with this workflow, compared to my first research internship three years ago, where I literally had to rerun a whole bunch of simulations because I had lost track of the parameters producing them.

Future work will allow my to add more automation in my scripts and makefile, and reduce the room for human errors and genuine mistakes. I think this is a continuous work throughout the whole life of a researcher, as new problem arises when tools, field or solutions evolves.

[10] A minimalist markup language.

jupyter Prepare voc Last Checkpoint: 06/14/2016 (autosaved)

File Edit View Insert Cell Kernel Help Python 2

Markdown CellToolbar

In [13]:
```
percentages = [[] for _ in range(10)]
for f in range(10):
    test_count = lambda box_size: box_size*f < kernel_size
    percentages[f] = [sum(test_count(b)/len(c) for b in c) for c in per_class_combined]
```

In [14]:
```
for f in range(2, 10):
    print("Upscale factor: %d, mean: %d%%" % (f, 100*np.mean(percentages[f])))
    print(" ".join(["{0:.0%}".format(p) for p in percentages[f]]))
```

```
Upscale factor: 2, mean: 36%
29% 39% 48% 50% 69% 29% 54% 7% 53% 47% 14% 13% 13% 28% 41% 60% 63% 9% 14% 55%
Upscale factor: 3, mean: 23%
21% 20% 33% 34% 47% 15% 40% 3% 29% 27% 5% 6% 6% 17% 26% 42% 50% 4% 7% 28%
Upscale factor: 4, mean: 15%
14% 10% 23% 25% 31% 8% 30% 2% 16% 18% 2% 1% 4% 7% 19% 27% 43% 1% 4% 16%
Upscale factor: 5, mean: 10%
8% 7% 16% 17% 22% 6% 22% 1% 9% 15% 1% 0% 3% 6% 13% 18% 36% 0% 2% 10%
Upscale factor: 6, mean: 7%
8% 5% 10% 11% 15% 5% 16% 1% 5% 11% 0% 0% 1% 4% 9% 12% 29% 0% 1% 5%
Upscale factor: 7, mean: 5%
6% 4% 7% 7% 11% 4% 11% 0% 3% 7% 0% 0% 1% 3% 7% 8% 26% 0% 1% 4%
Upscale factor: 8, mean: 4%
5% 3% 4% 5% 7% 4% 9% 0% 2% 5% 0% 0% 0% 2% 5% 4% 22% 0% 0% 2%
Upscale factor: 9, mean: 3%
5% 3% 3% 4% 4% 3% 7% 0% 1% 4% 0% 0% 0% 2% 4% 3% 19% 0% 0% 2%
```

In [15]:
```
percent_pc = [[percentages[f][c] for f in range(10)] for c in range(20)]

for c, c_id in zip(percent_pc, class_id.keys()):
    plt.plot(c, label=c_id)

average_percent = [np.mean(percentages[f]) for f in range(10)]
plt.plot(average_percent, label="Average", linewidth=3)
plt.grid(which="major")
plt.legend()
```

Out[15]: <matplotlib.legend.Legend at 0x7f0df98fff90>

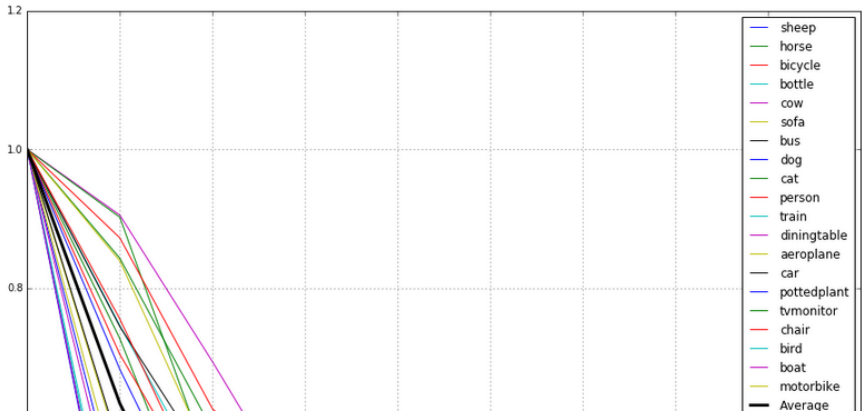

Figure 32: A screenshot of a notebook used to analyze object size in the VOC2007 dataset.

8 Conclusion

We presented how we transformed a CNNs made for classification to a CNNs for object detection, and how to train it. We also explained different methods that could be used to deal with the scale problem.

Although the results are not as satisfactory as I would have liked (because of reasons), we still have obtained valuable work. Not the kind of valuable that can be published right away, but the kind of valuable that can be extended, pursued and may be published if it isn't a dead end.

This is why I really tried to produce reusable code and experiment: it became clear pretty quickly that we wouldn't be able to complete everything, and I didn't wanted Yannis to restart from scratch, and struggle as I did sometimes.

On a personal level, I liked the past six months, and want to pursue in this field. I don't know how exactly yet, since I don't want to rush a Ph.D The field is quite big, and I want to have a better overview before choosing a thesis subject. Working in the as an engineer in computer vision for a while might be a good middle-ground to sharpen my technical skills and improve my knowledge.

A Schedule

February	Setup, bibliography
March	Installed Caffe framework on local machine, learned to use it
April	Convolutionify and NMS
May	Medical leave due to health issues
June	Installed GPU, fine tuning
July	Loss layer, back-propagation, report

References

[1] Dana H Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.

[2] Christopher M Bishop. Pattern recognition. *Machine Learning*, 2006.

[3] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. *arXiv preprint arXiv:1605.06409*, 2016.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[5] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[6] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

[7] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 37(9):1904–1916, 2015.

[9] Paul VC Hough. Machine analysis of bubble chamber pictures. In *International Conference on High Energy Accelerators and Instrumentation*, volume 73, 1959.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2169–2178. IEEE, 2006.

[12] B Boser Le Cun, John S Denker, D Henderson, Richard E Howard, W Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*. Citeseer, 1990.

[13] Bastian Leibe, Aleš Leonardis, and Bernt Schiele. Robust object detection with interleaved categorization and segmentation. *International journal of computer vision*, 77(1-3):259–289, 2008.

[14] Karel Lenc and Andrea Vedaldi. R-cnn minus r. *arXiv preprint arXiv:1506.06981*, 2015.

[15] Shu Liu, Cewu Lu, and Jiaya Jia. Box aggregation for proposal decimation: Last mile of object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2569–2577, 2015.

[16] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[17] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *arXiv preprint arXiv:1506.02640*, 2015.

[18] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015.

[19] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[20] Li Wan, David Eigen, and Rob Fergus. End-to-end integration of a convolution network, deformable parts model and non-maximum suppression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 851–859, 2015.

[21] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1529–1537, 2015.

Résumé

Pendant les six derniers mois, j'ai effectué un stage de recherche à l'Inria Rennes, un institut de recherche public. Notre objectif était de tester des nouvelles méthodes de détection d'objets, combinant des réseaux de neurones à la transformée de Hough.

Les réseaux neuronaux convolutionnels (CNN en anglais) sont courants en apprentissage profond appliqué à la classification d'images. Ils sont aussi de plus en plus utilisés en détection d'objets. La transformée de Hough est une catégorie d'algorithmes, qui utilise des votes pour prédire où un object pourrait se situer dans l'image.

Afin de faciliter la lecture de ce rapport, nous commencerons par présenter les notions utiles à sa compréhension. Nous en profiterons par ailleurs pour parler de travaux similaires à notre objectif. Nous présenterons ensuite ce que nous avons réalisé, étape par étape. Nous sommes partis d'une architecture de réseau de neurones standard, et avons ajouté de nouvelles fonctions, de nouvelles couches pour nous rapprocher de notre objectif. Le réseau a d'abord été re-entrainé afin de correspondre à notre nouveau jeu de test. Puis, nous avons transformé le réseau afin qu'il puisse considérer des sous-parties d'une image, et non l'image dans sa globalité. Ensuite, nous avons rajouté des couches pour supprimer les faux positifs et les duplicats lors de la détection. Enfin, nous présenterons le méthode que nous allons utiliser pour entrainer le réseau de bout en bout. Ce travail n'étant pas terminé, nous parlerons ensuite de différentes méthodes qui doivent être testées dans le futur afin de continuer nos recherches.

Abstract

During the past six months, I've been working at the Inria Rennes, a public research institution in computer science. Our goal was to try new methods for object detections combining already existing tools and algorithms, Convolutional Neural Networks (CNN) and the Hough transform.

CNNs are a standard tool in deep learning for image classification, and is increasingly used for object detection as well. The Hough transform is a category of algorithms that uses votes to predict where potential objects could be.

We will start with some background to help to understand this report, and will discuss related works. Then, we will present our work, step by step. Starting with a neural network standard architecture, we gradually added new functions, layers that would get us closer to our goal. We first retrained parts of the network to fit our new training data. Then, we changed the network so it would consider subparts of images and not images as a whole. Next, we added layers to limit false positive and duplicates during detection. At last, we created a end-to-end method to train the network. Since this is still an ongoing work, we finally discuss methods that will be tested in the future.

INSA

INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839 - 35708 Rennes Cedex 7
Tél. +33 (0) 2 23 23 82 00 - Fax +33 (0) 2 23 23 83 96
www.insa-rennes.fr