BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

DEPARTMENT OF TELECOMMUNICATIONS AND MEDIA INFORMATICS

Gábor Révy

Object detection with YOLO deep learning algorithm

Bálint Gyires-Tóth, PhD

supervisor

Budapest, 2018

Összefoglaló

A GPU-k számítási kapacitásának fejlődésével, a rendelkezésre álló adattömeg növekedésével és a gépi tanulás terén elért kutatási eredmények nyomán különösen népszerűvé vált a mély tanulás (Deep Learning). Ezen jellemzően sokrétegű, mély neurális architektúrák ma már szerves részeit képezik a legkorszerűbb rendszereknek különböző tudományágakban, például a gépi látás és a beszédfelismerés (Speech Recognition) területén.

Az objektumfelismerés (Object Detection) a gépi látáshoz és a képfeldolgozáshoz kapcsolódó technológia, mely egy képen vagy videón található, egy adott osztályba tartozó objektumok helyének meghatározásával foglalkozik. Az eljárás számos területen hasznosítható. Például az arcfelismerés közösségi oldalakon népszerű alkalmazása ennek a metódusnak, de elengedhetetlen része az önvezető autóknak a járművek, táblák, vagy gyalogosok felismerésében is. Mozgóképen való objektumkövetéssel jó közelítést lehet adni az adott tárgy sebességére és meg lehet figyelni a mozgását.

Munkám célja egy olyan mély tanuló megoldás elkészítése a Keras keretrendszerben, mely képes egy képen található különböző tárgyak felismerésére és lokalizációjára, egyetlen kiértékelés (un. „one-shot”) alapján.

Dolgozatomban először bemutatom a különböző objektumfelismerési algoritmusokat és a YOLO (You Only Look Once) algoritmus verzióit. Ezután ismertetem a COCO (Common Objects in Context) adatbázisban található képek előkészítési lépéseit és a hozzá tartozó címkéket. Ez az adatbázis kb. 108 ezer képet tartalmaz, melyeken 80 különféle, a mindennapi életben körülöttünk lévő tárgyak találhatóak, pl. emberek, madarak, stop tábla, asztal, autók stb. A dolgozat további részében bemutatom a YOLO legújabb verziójának architektúráját és implementációs lépéseit, illetve az azt tanító algoritmust. Ezt követően a legnagyobb potosság elérése céljából különböző keresem az optimális hiperparaméter-beállítást. A tanítás során különféle módszereket alkalmaztam, melyek segítik a felismerés pontosságának növelését és a tanítás gyorsítását. Dolgozatom végén eredményeim objektív módon értékelem ki.

Abstract

With the evolution of the computing capacity of GPUs (Graphics Processing Unit), the increase in data volume and the research results of machine learning, deep learning has become particularly popular. These typically multi-layered, deep neural architectures nowadays form an integral part of state-of-the-art systems in various disciplines, for example in computer vision and speech recognition.

Object detection is a technology related to computer vision and image processing, that deals with localizing instances of objects of certain classes in digital images and videos. This method can be utilized in many areas. For example, face recognition is a popular use of this method on social networking sites, but it is also an essential part of self-driving cars in recognizing vehicles, signs or pedestrians. Object tracking on a video can give you a good approximation of the subject's speed, and one can observe its movement.

The aim of my work is to create a deep learning system in the Keras deep learning framework, that is capable of recognizing and localizing different objects on a picture based on a single (one-shot) evaluation.

In my paper, I first introduce the different object recognition algorithms and the YOLO (You Only Look Once) algorithm versions. Next, I describe the preparation steps for the images and their tags downloaded from the COCO (Common Objects in Context) database. This database consists of about 108.000 pictures, containing objects from 80 classes from our everyday life around us, for example, people, birds, stop table, table, cars, etc. In the next part of my paper, I present the architecture and implementation steps of the latest version of YOLO and its training algorithm. Then I am searching for the optimal hyperparameter settings to gain the highest accuracy. During the training, I used various methods to help increase the accuracy of recognition and accelerate training. At the end of my thesis, I objectively evaluate the results.

Contents

1 INTRODUCTION

I spent my summer internship at the Continental Automotive Hungary Kft. They concluded an agreement with the BME this year and are working there on a lot of projects that use deep learning algorithms.

I met deep learning about two years ago. From then on I was reading many things about it. High number of websites are available where you can read about the latest developments. This is no wonder because since about 2013 deep learning got a big hype. For example Medium[1] and Towards Data Science[2] present different algorithms and tricks in an entertaining but a little bit professional way. Archive Sanity Preserver[3] provides a useful interface for the arXiv[4] papers - the site, where the latest research results are released. On this preserver, you can sort these papers based on the number of mentioning on Twitter or by upload date.

I had some little projects only for myself, but I wanted to have a bigger project. My consultant recommended me the opportunity to create something at Continental using deep learning.

First I had to meet different image recognition algorithms to get an overall picture about them. It was very interesting to identify the similarities and the differences between them and to see how they developed. Based on the literature review I made, YOLOv3 is currently the fastest algorithm for object detection and in addition, its accuracy is acceptable compared to the other methods. YOLOv3 is originally written in the Darknet[5] framework and there is no Keras implementation available online. Even if there were, we should be careful because implementations available online are often inaccurate.

My main goal was to get to know different object detection algorithms and implement an accurate, adaptive YOLOv3 algorithm.

[1] www.medium.com *(accessed October 26, 2018)*
[2] www.towardsdatascience.com *(accessed October 26, 2018)*
[3] www.arxiv-sanity.com
[4] www.arxiv.org *(accessed October 26, 2018)*
[5] www.pjreddie.com/darknet *(accessed October 26, 2018)*

2 DEEP LEARNING

In this section my intention is to provide an overview of deep learning only at the level necessary to understand this paper. Machine learning is a field of artificial intelligence (AI) where we use statistical techniques to make the computer able to "learn", without explicitly programming the knowledge. There are three types of Machine learning algorithms:

- supervised learning: we have the input data and the desired output called ground truth The task is to find a well-generalized mapping from the input to the corresponding output.
- unsupervised learning: we have input data and no corresponding output variables The task is to find the underlying structure or distribution in the data in order to learn more about the data.
- reinforcement learning: we have an AI agent in an environment The agent tries to improve based on the received feedback – reward or punishment – from the environment for its decisions.

Deep learning is a machine learning method. Its models are inspired by information processing and communication patterns in biological nervous systems. It can be used for several purposes. Applications of deep learning include, among a lot of others, the following tasks:

- Classification is a process where you put things in categories.
- Using regression you can predict values based on previous experience (for example house prices).
- With Natural Language Processing (NLP) we can now preform speech recognition, natural language generation or text summaries.
- Using deep learning, Google's AlphaGO [1] could beat a professional Go player.

I am training my model in a supervised way, due to the type of the task and the data. I am using it for classification and regression.

2.1 Layer types

In my paper I will refer to the two main types of neural network layers: the fully connected layer and the 2 dimensional convolutional layer. Mostly we combine these two types in our

models. An overview of deep learning approaches, including the two main types of layers used in my work can be found in [2].

2.1.1 *Fully connected layer*

In a neural network with fully connected (FC) layers, we have connected layers containing neurons.

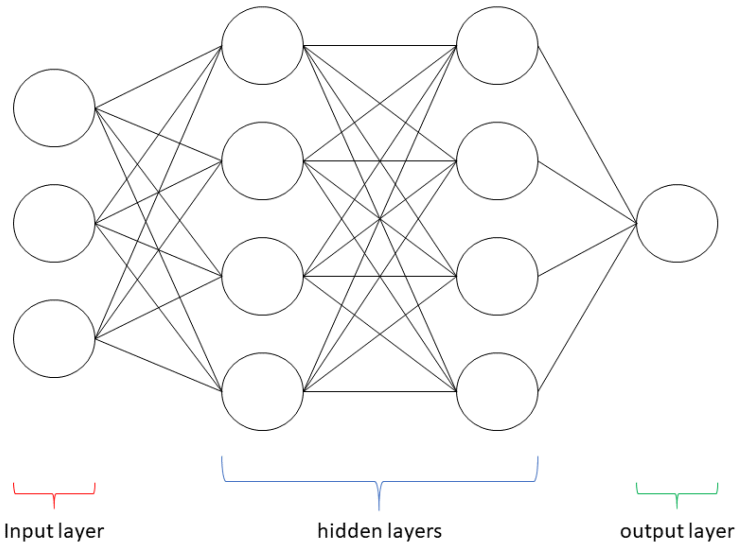


Figure 1 - A fully connected neural network

Figure 1 shows an example for a fully connected neural network. The circles represent the neurons and the lines are connections between them.

In deep learning, “deep” refers to having more than one hidden layer. In this example, we have an input layer (red), two hidden layers (blue), and an output layer (green). In the input layer, we have three neurons. We can imagine neurons like you can see in Figure 2.

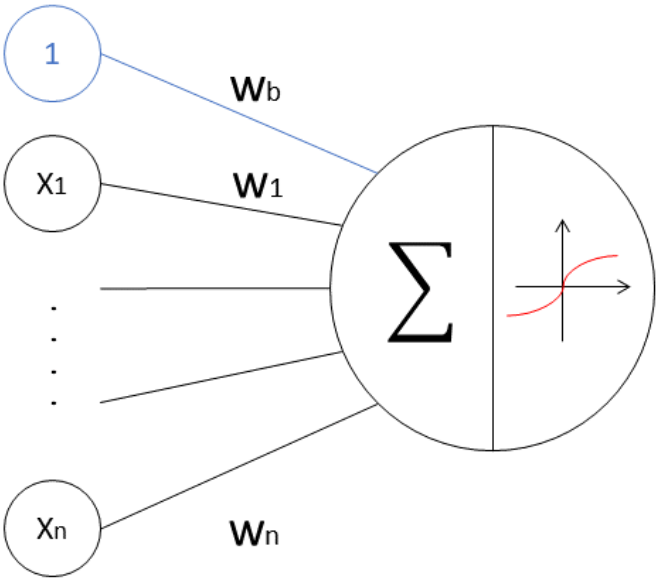


Figure 2 - A single neuron

We put the input data - represented as numbers - in these neurons. These layers are fully connected, because every neuron in a layer gets input from every neuron in the previous layer ($x_1, x_2 \ldots, x_n$). These inputs are multiplied by numbers called weights ($w_0, w_1 \ldots, w_n$), and are added. These are trainable parameters. Sometimes we add a one to the input called bias (also trainable). Bias helps to translate the activation functions by a learnable constant and therefore increases the flexibility of the model. Using an (usually non-linear) activation function basically we decide whether a neuron should be activated or not. Thus we get the activation value of one neuron.

2.1.2 *2D convolution*

Convolution is a data processing method specialized for grid-like data. For example we can process time series using one dimensional convolution, where each grid represents a measured value sampled at specific intervals. Two dimensional convolution is well known in image processing. Here, grids represent the pixels of the image. We use convolutional layers to extract features from the data. In the first layers, the model tries to recognize simple patterns, for example, lines or curves on the image. The more we go forward in the stack of layers, the more complex the patterns are recognized by the neural network.

First, I want to introduce 2D convolution through an example and then I will specify the terms and describe it more generally.

2.1.2.1. An example

In Figure 3 we can see three matrices. The first matrix represents the input data we want to convolve; the second is the convolutional kernel containing one filter we apply to the input and the third is the result. To compute the value of the first (grey) cell in the result we multiply the values highlighted in grey by the corresponding values in the filter. Then we sum these 9 values. To compute the value of the cell to the right we move the selected 3×3 area in the first matrix to the right by one. Similarly we can compute the value of the cell below. We have to move down the selected 3×3 area in the input matrix. If we have a bias value we add it to each cell then we apply the activation function to each cell individually.

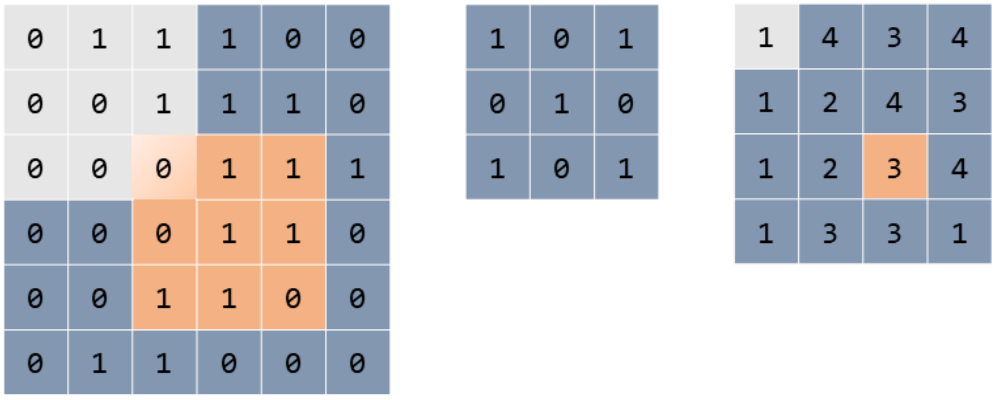


Figure 3 - 2D convolution

2.1.2.2. Generalizing, explanation

The input matrix can be represented by a three dimensional tensor. In Figure 3 it was $6 \times 6 \times 1$ but for example an RGB image is $n \times k \times 3$, where $n \times k$ is the resolution of the image and 3 is the number of channels.

The three dimensional kernel size can also be specified. Mostly we use 3×3 or 5×5 filters. And we can have more than one filters in the kernel. The number of filters is specified by the number of channels in the input tensor. For example if the input is an $n \times k \times 3$ RGB image then the number of filters is three. During training, the model learns the values in the filter, also called weights.

The result matrix can also be represented by a three dimensional tensor. In Figure 3 it was a $4 \times 4 \times 1$ tensor. Usually we have more than one output channels thus we can extract several type of features.

The number of kernels equals the number of channels in the output tensor, because every channel has its own kernel.

2.1.2.3. Stride and zero padding

We can specify the amount by which the filter shifts called stride. In Figure 3 the stride had the value of one, but to reduce dimensionality we can make bigger steps thus the size of the output matrix will be smaller. If we want to extract features without spatial dimension decrease, we can surround the input matrix with zeros, called zero padding.

2.2 Activation functions

We can see that neurons apply an activation function to provide nonlinearity. These non-linear activations make neural networks universal approximators. In this section I present some activation functions I refer to in this paper.

2.2.1 *Sigmoid*

Sigmoid is a function that has a target set between 0 and 1.

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^{-x}}{e^{-x} + 1}$$

If we want to make a binary classification where we want to decide if an object is in a class or not, we should use sigmoid. Using a threshold We can use its result as a probability: if the result is higher than the threshold then the object is in the class else it is not.

2.2.2 *ReLU*

ReLU refers to rectified linear unit. Especially in deep neural networks trained with gradient-based learning methods, there is a difficulty called vanishing gradients. In some cases, the problem with such methods is that the gradient is computed by the chain rule[6]. The traditional hyperbolic activation functions (for example $tanh$) have gradients in the range (0,1) and can be easily saturated. Multiplying these small gradients based on the chain rule can make the gradients in the first layers very small. Vanishing gradients make it difficult to know which direction the parameters should move to make the loss smaller. ReLU is meant to solve this problem since it has a gradient value of 1 if the input is higher than zero. It is an identity function if the input is higher than zero and its value is zero if the input is lower than zero.

$$ReLU(x) = \max(0, x)$$

2.2.3 *Leaky ReLU*

Leaky ReLU is similar to ReLU but it allows a small gradient when the unit is not active. That means that we can give the slope of the linear function below zero.

$$LeakyReLU(x, p) = \begin{cases} x, if\ x > 0 \\ p \cdot x, if\ x < 0 \end{cases}$$

[6] en.wikipedia.org/wiki/Chain_rule *(accessed October 15, 2018)*

2.2.4 *Softmax*

If we want to classify objects into more than one classes we mostly use softmax. This function transforms the values in a K dimensional vector into another K dimensional vector, where each entry is in the range of (0, 1), and all the entries add up to 1.

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}}, where\ x \in \mathbb{R}^K$$

2.3 Improvements

2.3.1 *Batch normalization*

When training we compute the gradients for each weight with respect to the loss function. When we update the weights we make the assumption that the other layers' weights don't change. But in practice we update all the layers simultaneously. After update unexpected results can happen because many functions composed together changed at the same time.

Batch normalization has been described by Szegedy and Ioffe [2]. It helps reparametrizing the activations by giving two learnable parameters to every layer to hold mean and variance of neurons at a given value (usually at 0 and 1). We compute mean and variance only per batch, thus it brings some noise in the system, so it won't overfit. It speeds up the learning and we can eliminate all other regularization forms, for example dropout layers.

2.3.2 *Residual block*

Residual blocks have been described by the Microsoft Research group [3]. Let x is the input and $H(x)$ is the output of a few stacked layers. If we hypothesize that multiple nonlinear layers can asymptotically approximate any function then it is equivalent to hypothesize that they can approximate our residual function $H(x) - x$. In this case, we can make a shortcut from the input and add it elementary to the output thus we get the original function. It gives no extra parameters to the training, only the input and outputs have to be the same size. It also helps in avoiding the vanishing gradient problem. The reason for this is that gradients are mapped one in one from

the output layers to the layer before the residual block. You can see this architectural solution in Figure 4. Here, the identity mapping is what we call residual connection.

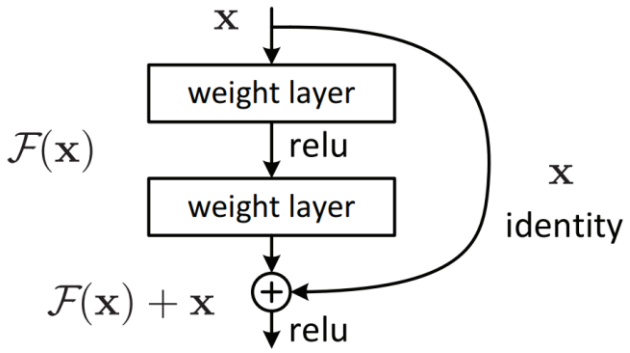


Figure 4 - A residual block (source: [3])

2.4 Training

We are training this network supervised, with (input, expected output) tuples. We are using the gradient descent algorithm. This algorithm tries to minimize the error between the networks output and the expected output. This error value is specified by the loss function. The optimizer updates the weights based on the gradient of the loss function. It computes the gradient of the loss function with respect to each weight in the neural net. Then it multiplies the gradient by a number called learning rate and subtracts it from the weight. This is a very high overview of the gradient descent algorithm, for deeper understanding I recommend reading chapter 4 and chapter 8 from the Deep Learning book [4].

The output can be computed parallel on a GPU so we are computing output on more than one input. This is called a batch. After every batch we update the weights of the network. This is called an iteration. One training on each training data is called an epoch. Thus we can calculate the number of iterations needed to learn one epoch with a given batch size.

$$n_{iterations} = \frac{n_{training\ data}}{size_{batch}}$$

3 METRICS

If we want to rate an algorithm, we have to introduce some metrics. Using these metrics during training we can track the evolution of our model and at we can compare different algorithms.

3.1 Confusion matrix – true and false positives and negatives

Confusion matrix gives a lot of information about the relation of the model and the real facts. In a confusion matrix we can see the number of true positives (TP). This is the number of the objects the algorithm classified as positive and in fact they are positive. False positive (FP) is the number of objects the algorithm predicted as positive but they are negatives in the real world. False negative (FN) objects are from the negative class according to our model but they are in the positive class in the real world. Finally, true negative objects have been predicted as negatives and in fact they are negatives, too. We try to minimize the number of FPs and FNs, the objects that have been misclassified. Also we try to maximize the number of TPs and TNs, the number of correct predictions.

I present this in an example in Figure 5. Suppose we have cat and dog images. We want to retrieve all the cat images and none of the dog images. The cat images we get are true positives. If we get also dog images, they are false positives. The cat images we didn't retrieve are false negatives. And the dog images we didn't retrieve are the true negatives.

		algorithm predicted	
		cat	dog
in fact	cat	TP - true positive	FN - false negative
	dog	FP - false positive	TN - true negative

Figure 5 - Confusion matrix – number of true and false positives and negatives

3.2 Precision and recall

Precision and recall are usually not discussed in isolation, they complete each other. Precision gives information about the fraction of retrieved relevant instances among the retrieved instances, while recall shows the fraction of retrieved relevant instances among the total amount of relevant instances.

precision: the ratio of the true positives and the retrieved instances

$$precision = \frac{TP}{TP + FP}$$

recall: the ratio of true positives and the number of the relevant instances

$$recall = \frac{TP}{TP + FN}$$

By changing the threshold of what we want our system to consider as relevant, there will be more or less true positives - and also false positives(!). With this method we can characterize a classifier: we look at how precision and recall change as we change the threshold.

3.3 Average precision

Instead of comparing curves, it's easier to compare a single number that characterizes the system. It is very useful since all the famous algorithms are benchmarked using this metric. Average precision is computed by averaging precision across all values of recall between 0 and 1 (the area under the curve).

$$\text{average precision} = \int_0^1 p(r)\ dr$$

In practice we approximate it with a sum:

$$\sum_{k=1}^{N} P(k)\Delta r(k)$$

N: total number of images

$P(k)$:precision at a cutoff of k images

$\Delta r(k)$:difference between recall at cutoff $k-1$ and cutoff k

3.4 IoU - Intersection over Union

Intersection over union is an evaluation metric used to measure the accuracy of an object detector on a particular dataset. A bounding box is a rectangle that closely encloses the object. Our system will predict bounding boxes and in the test data, we know where were originally bounding boxes. IoU is the rate of the area of the intersection and the area of the union of the two boxes like you can see in Figure 6.

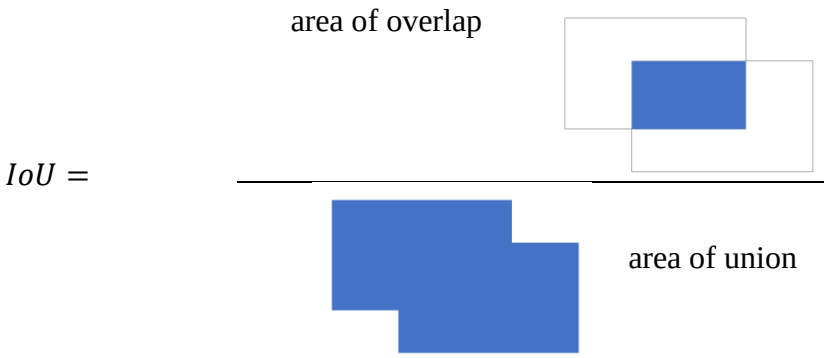


Figure 6 - IoU - Intersection over union

3.5 Mean average precision - mAP

With our system, we can classify objects into several classes. So we will define average precision for all classes and take the mean of these average precisions. That's mean average precision (mAP). A lot of object detection algorithms' mAP benchmark result is available in their paper, for example, SSD [5], Faster R-CNN [6] or YOLOv3 [7].

In my paper I will mention two types of mAPs. PASCAL VOC (Visual Object Classes) style mAP means that prediction is considered positive if IoU > 0.5. If multiple detections of the same object are detected it is negative. The AP is calculated for each class and these values are averaged over the classes. On the PASCAL VOC challenge they ranked the detection algorithms based on this value.

The other metrics is the COCO style mAP. They compute precision at multiple IoU thresholds from 0.5 to 0.95 with a step size of 0.05. These results are then averaged over the thresholds and then over the classes.

COCO style mAP metric is used in my paper because it better represents the accuracy of the algorithm.

4 OBJECT DETECTION ALGORITHMS

4.1 R-CNN

R-CNN [8] is an object detection method using the region proposal algorithm (R refers to region and) and a convolutional neural network (CNN). It consists of three parts.

The first module is responsible for generating 2000, category-independent regions where it found objects. For this, we are using the Selective Search [9] algorithm. In this algorithm we first obtain initial regions using another method [10]. Then we take all neighbouring region pairs in this set and compute similarity based on color, texture, size and shape. We merge regions with high similarity. Then we recalculate similarities and continue this process until the whole image becomes a single region. At every iteration we get bigger regions as shown in Figure 7. Then we compute bounding boxes from all regions.

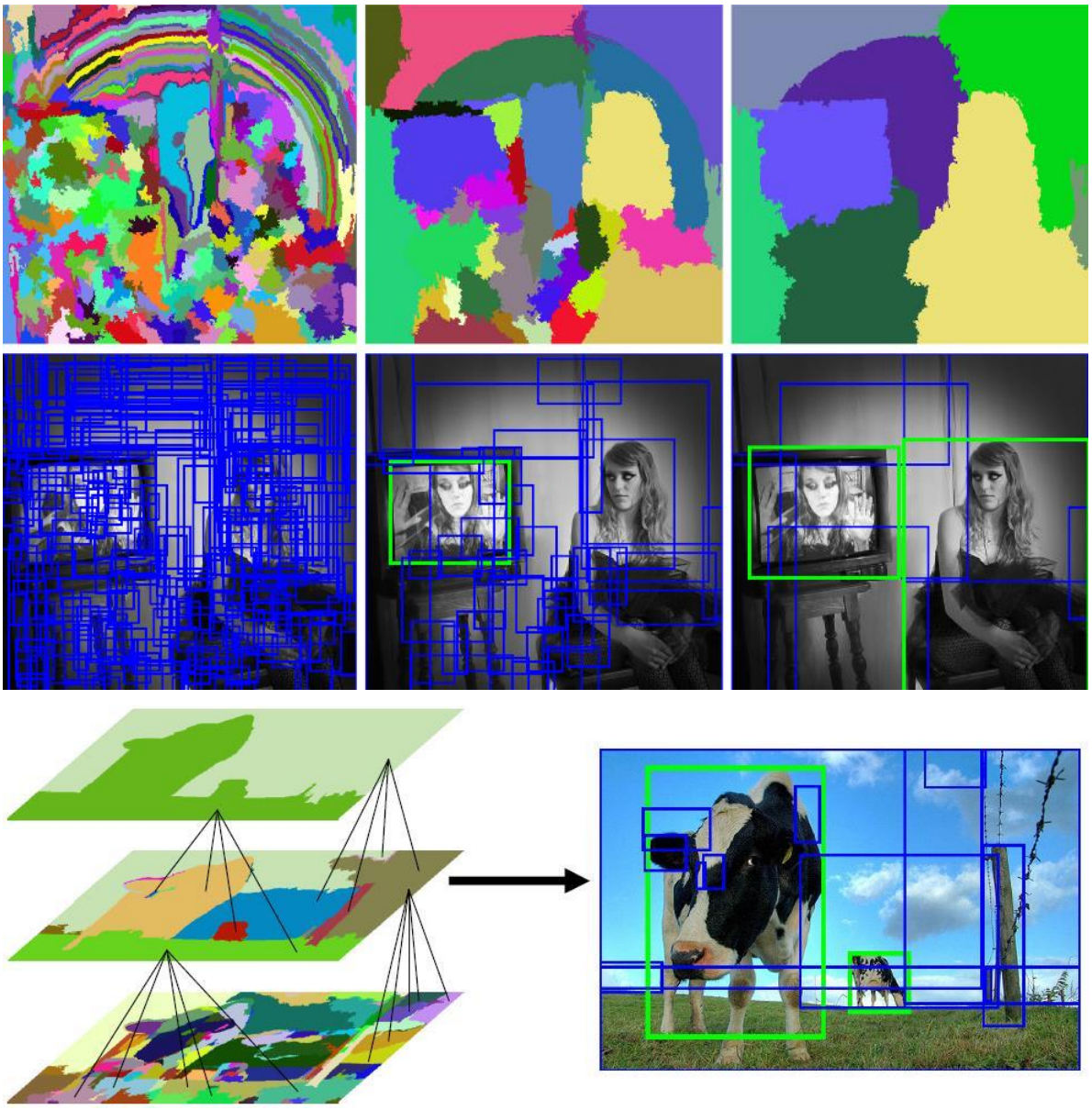


Figure 7 - Iteration steps of Selective Search algorithm for region proposal (source: [9])

The second module is a convolutional feature extractor called AlexNet described by Krizhevsky et al [11]. The regions are warped into a square and fed into the CNN. The CNN predicts 4096 feature values for each region we computed previously. It expects a 227×227 RGB image, has five convolutional layers and one fully connected layer.

The third module is a set of linear SVMs (Support Vector Machines) [12]. Each SVM is trained for scoring the previously computed feature vectors for a given class. Thus we have SVMs for each class we want to predict. It also predicts new bounding boxes using class specific bounding box regressors.

R-CNN achieves a mAP of 31.4 on the 200-class ILSVRC2013 detection dataset. The inference time is about 49 seconds on GPU.

4.2 Fast R-CNN

Fast R-CNN [13] is the improved version of R-CNN. The approach is very similar to R-CNN. But instead of feeding each region of the picture one by one into the feature extractor, we feed the whole image into it. We use the AlexNet [11], or the VGG [14] feature extractor described by Oxford's Visual Geometry Group.

Then, with a so-called RoI pooling layer, we transform the regions of interest from the feature map into a fixed size feature matrix. First, we select the proposed region and divide it into $n \times k$ parts, where n and k are the sizes of the result matrix. Then we select the highest values from each of these parts. Thus we have $n \times k$ values. We can follow the whole algorithm in Figure 8. Here, the region of interest is a 5×7 rectangle. We want to have a 2×2 result matrix so we divide the rectangle into 2×2 parts. From the resulting parts we pick the highest values, thus we have the fixed size 2×2 feature matrix.

Each feature vector is fed into a sequence of fully connected layers that branch into two output layers. The first predicts class probabilities. The second predicts 4 values for refining proposed bounding boxes.

The inference time is 2.3 seconds (~0.5 FPS, frames per second) on GPU. The PASCAL-style mAP is 35.9, the new COCO-style mAP, is 19.7.

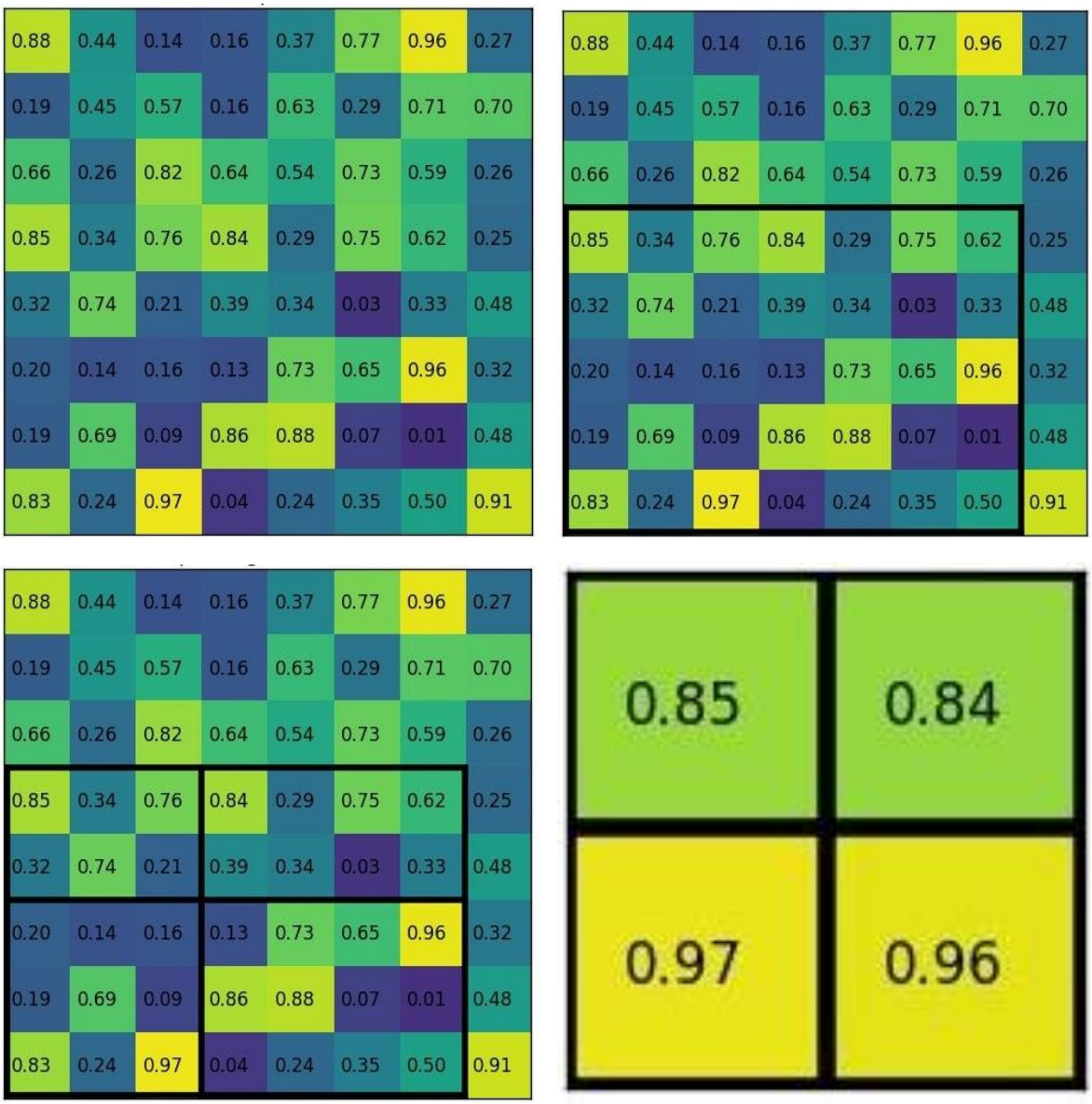


Figure 8 - Steps of RoI pooling layer (source:DeepSense[7])

4.3 Faster R-CNN

Faster R-CNN [6] improves Fast R-CNN by replacing the Selective Search algorithm. It is using a deep fully convolutional network to propose regions.

This region proposal network (RPN) predicts a set of rectangular boxes with objectness scores. To increase the speed, we share the computation between the feature extractor and the RPN. Thus they have 5 (Zeiler and Fergus model) or 13 (VGG-16) shareable layers. A small network slides over the feature map and creates lower dimensional vectors. Then we feed the vectors into two sibling fully connected layers which predict bounding boxes and class

7 https://deepsense.ai/region-of-interest-pooling-explained/ *(accessed October 15, 2018)*

probabilities. We reshape the predicted proposals using the RoI pooling layer, which classifies the object in the region and predicts offset values for the bounding boxes.

This algorithm could reach 48.4 mAP (PASCAL-style) and 27.2 mAP (COCO-style) at 0.2 seconds (5 FPS) on GPU.

4.4 SSD

SSD is another famous object detector described by Szegedy et al [5]. It consists of two parts as you can see in Figure 9: a feature extractor and auxiliary convolutional layers to detect objects.

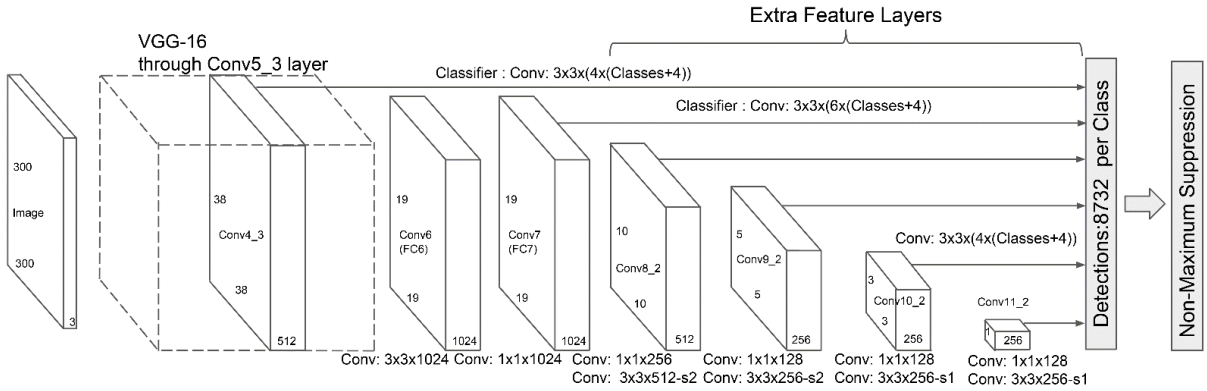


Figure 9 - The architecture of SSD (source: [8])

It is using the VGG-16 [14] feature extractor that has 16 convolutional layers. Six additional convolutional layers are connected to VGG-16. These layers downscale the feature map to be able to detect bigger objects. One auxiliary structure is connected to a layer in the VGG-16, the others are connected to the five last convolutional layers. These structures predict $n \times n \times (k \cdot number\_of\_classes + 4)$ values where n refers to the number of the grid cells we predict objects in; k refers to the number of anchor boxes with different aspect ratio which is 6 in three predictions and 4 in the others. Since many predictions contain no object SSD reserves a class prediction for class "nothing". In Figure 10 you can see a prediction at a given scale: the picture is divided into grid cells and the center of the object is predicted into the corresponding cell.

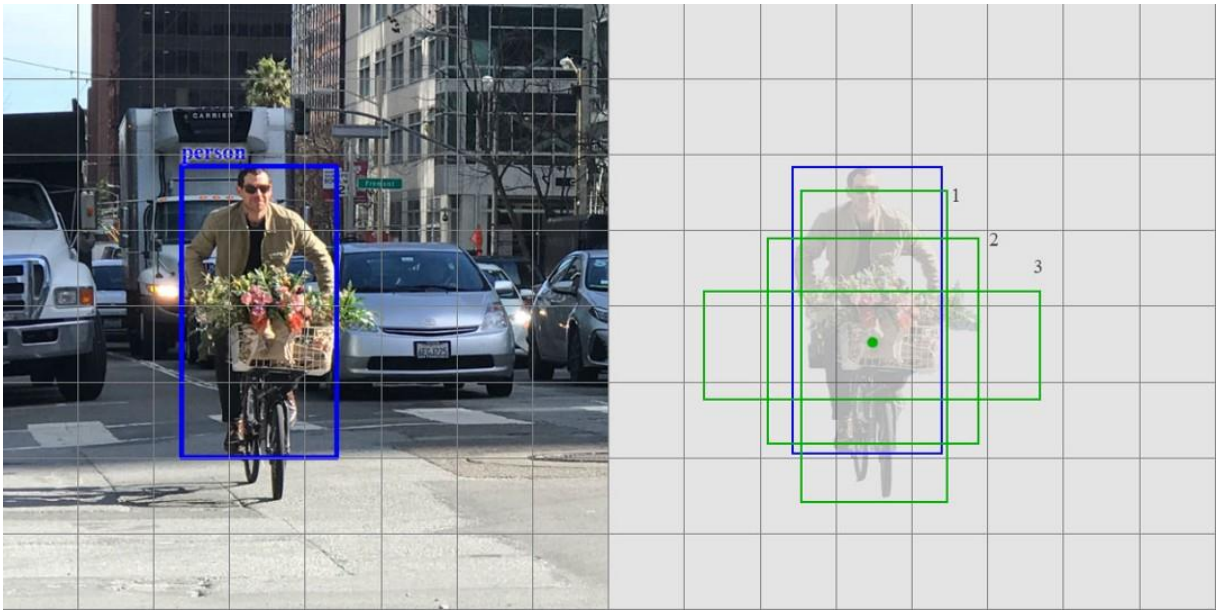


Figure 10 - Prediction of SSD at a scale (source: Medium[8])

The network can be trained end-to-end, we just have to assign the ground truth to a specific output of the detector.

SSD makes predictions with 26.8 mAP (PASCAL-style) and 46.5 mAP (COCO-style) and 76.8 FPS at 512×512 resolution.

4.5 RetinaNet

FAIR (Facebook AI Research) released a paper [15] where they write about feature pyramid networks. Using this architecture they introduced their object detector called RetinaNet. Its architecture is very similar to the YOLOv3s.

Feature pyramids are very important components in neural networks that recognize objects at different scales. Feature extractors compute a feature hierarchy layer-by-layer. In the first layers we have low-level features and by going deeper we get-higher level features. With

[8] https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06 *(accessed October 26, 2018)*

subsampling this hierarchy we get a pyramidal shape as you can see in Figure 11. However high resolution feature maps have low representational capacity for object detection.

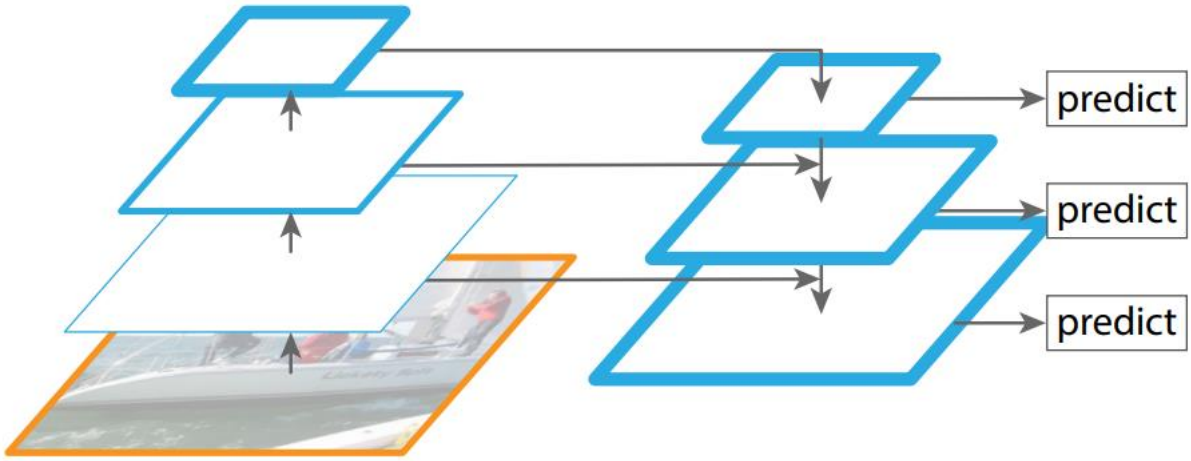


Figure 11 - Feature pyramids in RetinaNet (source: [15])

In the first part of the backbone we extract features by downscaling. In the second part we upscale the low-resolution, semantically strong features and concatenate them with the high resolution, semantically weak feature maps.

We predict at five scales and we use three anchor boxes at every location. We are using convolutional layers to predict class probabilities bounding boxes.

4.6 YOLOv1

YOLO [16] stands for You Only Look Once. It's a neural network architecture for real-time object detection.

A single neural network predicts bounding boxes and class probabilities to them - in one evaluation. That makes the algorithm run faster. Unlike other traditional methods that look only at a part of an image, YOLO sees the entire image during training, so it can learn contextual information, too. Yolo can also learn generalizable representations of objects. It can process 45 frames a second (on GPU) with 63.4 mAP.

The system divides the picture into an $S * S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each cell predicts B bounding boxes and confidence scores to them. Each bounding box consists of 5 values:

- x, y – the coordinates of the center of the box relative to the bounds of the cell
- w, h – width and height relative to the full image

- confidence score – shows how accurate the model thinks the box is (IoU), and how confident it is that the box contains an object: $\Pr(obj) \cdot IoU_{pred}^{truth}$

Each cell predicts C conditional class probabilities (only one per cell), $Pr(Class_i|Object)$, conditioned on the grid cell containing an object. When testing we multiply the probabilities and confidence predictions so that we get a class confidence score for each box:

$$Pr(Class_i|Object) \cdot Pr(Object) \cdot IoU_{pred}^{truth} = Pr(Class_i) \cdot IoU_{pred}^{truth}$$

Now we can see, that the result of the model is an $S \times S \times (B \cdot 5 + C)$ tensor.

To avoid localizing an object multiple times we use non-max suppression. First, we eliminate all boxes with $Pr(Object) < 0.6$. Then we select the bounding box with the highest $Pr(Object)$ and eliminate other boxes that have $IoU \geq 0.5$ with that box. We repeat this algorithm.

The network has 24 convolutional layers - responsible for feature extraction -, followed by 2 fully connected layers - responsible for prediction. The full architecture can be seen in Figure 12.

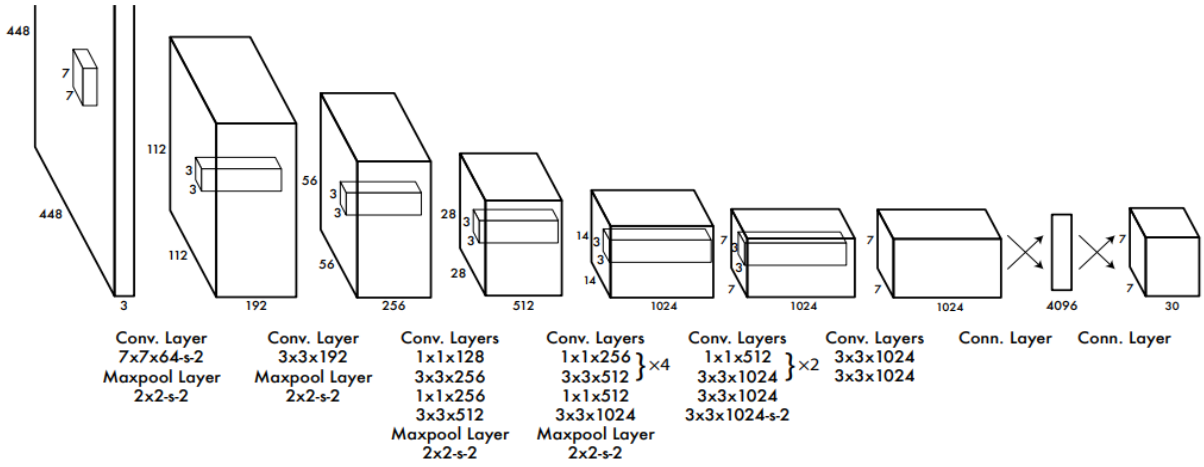


Figure 12 - Architecture of YOLOv1 (source: [16])

1×1 convolutional layers are used to reduce feature space from the previous to the next layer. The final layer is using a linear activation function and all the other layers use leaky ReLu.

When training we use enhanced sum of squares as loss function, because weighting localization error equally with classification error may not be ideal. Equation 1 shows the parts of the loss function.

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2$$

Equation 1 - Loss function of YOLOv1

λ_{coord} and λ_{noobj} are constants to weight parts of the loss function. That's because many grid cells don't contain any object. This would push the confidence scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. The parts of the function:

- We compute the loss related to the predicted bounding box position (x, y). 1_{ij}^{obj} is 1 if there's an object in cell i and the jth bounding box is responsible for it.
- We compute the loss related to the predicted box width / height. We use square root to reflect that small deviation in large boxes matters less than in small boxes.
- We compute the loss associated with the confidence score for each bounding box predictor. We compute it separate if there's an object and weighted if there isn't.
- We compute the sum-squared error for classification if there's an object (hence the conditional class probability).

4.7 YOLO9000 (YOLOv2)

YOLO9000 [17] has some improvements compared to YOLO, so it could reach 21.6 mAP (COCO style) at 67 FPS on the COCO dataset. It can detect over 9000 objects.

4.7.1 *Convolutional with anchor boxes and dimension clusters*

From YOLO we remove the fully connected layer and use anchor boxes to predict bounding boxes. Anchor boxes are boxes with predefined height, width. Our system predicts offset and scale to the anchor boxes to get bounding boxes. We also decouple the class prediction from the spatial location and predict class and IoU for every anchor box.

We can notice that there are boxes with similar aspect ratio but instead of hand-picking these ratios, we run ***k-means clustering*** on the training set bounding boxes to automatically find good priors. In order to larger boxes doesn't generate more error we use the following metric in the k-means algorithm:

$$d(box, centroid) = 1 - IOU(box, centroid)$$

Here, $k = 5$ seemed to be a good value. The network predicts 5 bounding boxes at each cell with 5 coordinates for each bounding box, t_x, t_y, t_w, t_h and t_o. If the cell is offset from the top left corner of the image by (cx, cy) and the bounding box prior has width and height p_w, p_h, then the predictions correspond to:

$$b_x = \sigma(t_x) + c_x$$

$$b_y = \sigma(t_y) + c_y$$

$$b_w = p_w e^{t_w}$$

$$b_h = p_h e^{t_h}$$

$$P_r(object) \cdot IoU(b, object) = \sigma(t_o)$$

We are using batch normalization to make training faster.

4.7.2 *Multi-scale training*

To make the system more robust, every 10 batches the network changes its dimension. Since the model downsamples by 32, it will choose from $\{320, 352, \ldots, 608\} \times \{320, 352, \ldots, 608\}$.

4.7.3 *Fine-Grained Features*

Predictions on a 13×13 feature map is efficient for large object, but not so much for small objects. To better predict small bounding boxes, we add a pass-through connection from a 26×26 layer from an earlier layer.

4.7.4 *Classification*

Labels are structured in a tree: hyponyms are children of a node. To perform classification with this tree we predict conditional probabilities at every node for the probability of each hyponym of that synset given that synset.

For example, at the “cat” node we predict:

P_r(Persian|cat)

P_r(tabby|cat)

If we want to compute the absolute probability for a node we just have to multiply these probabilities through that path on the tree. For example on Figure 13 you can see: if you want to compute the probability of an object being a persian cat you just have to compute: $P(physical\ object) \cdot P(animal|phisical\ object) \cdot P(cat|animal) \cdot P(Persian|cat)$.

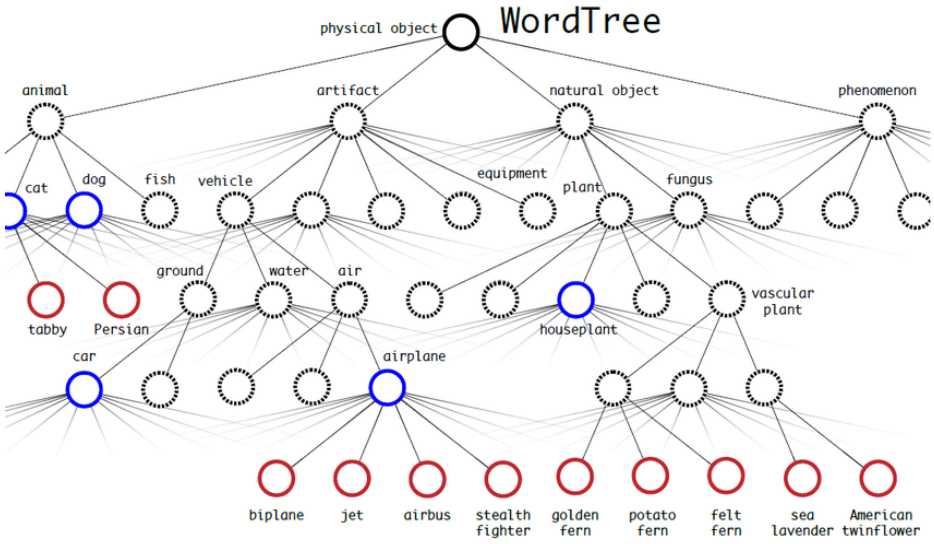


Figure 13 - WordTree

4.8 YOLOv3

YOLOv3 [7] is the enhanced version of YOLOv2 (YOLO9000). It can predict boxes for 80 different classes with 31.0 mAP (COCO style) at 29 FPS.

4.8.1 *Bounding box prediction*

We still predict 4 coordinates. During training, we use sum of squared error loss. We also predict objectness values which is 1 if that prediction overlaps the ground truth object by more than any other bounding box priors. Any other bounding box priors that overlap that ground truth above a threshold (0.5) will be ignored. If a bounding box prior is not assigned to a ground truth object it incurs no loss for coordinate or class predictions, only objectness.

4.8.2 *Class prediction*

We don't use softmax like in YOLOv2 but rather only independent logistic classifiers. Labels with scores higher than a threshold are assigned to the image. We use cross-entropy loss for the class predictions.

4.8.3 *Predictions across scales*

We predict boxes at 3 different scales: 13×13, 26×26 and 52×52. The first prediction is at the 82nd layer at 13×13. It's performed by a $13 \times 13 \times B \cdot (Cl + Co + O)$ convolutional layer.

B: $number\ of\ anchor\ boxes$

Cl: $number\ of\ classes$

$Co = 4$: $number\ of\ coordinates$

$O = 1$: $objectness$

Prediction at the 94th layer at 26×26 scale is performed by upsampling the 79th layer and concatenating with the 61st feature map layer. Similar, prediction at the 106th layer at 52×52 scale is performed by upsampling the 91st layer and concatenating with 36th feature map layer.

5 SOFTWARE AND HARDWARE ARCHITECTURE

5.1 Software

I wrote the algorithm in Python language. I was using the PyCharm integrated development environment (IDE) which is available for free for students on BME. The main tool for creating my system was Keras[9] library. I recommend it for everyone who wants to make fast prototypes or even systems ready for industrial usage. It is a library providing a high level interface to create models, write loss functions or train the models. One can even create custom layers and run them on GPU. A lot of callbacks help the developers work during training, for example, early stopping, TensorBoard or learning rate scheduler. We can use Keras only with a backend

[9] https://keras.io/ *(accessed October 21, 2018)*

that make the computations behind the high level definitions. Keras currently supports TensorFlow[10], Theano and CNTK.

I was using the TensorFlow (TF), which currently is the most popular deep learning framework. TF was first developed by Google Brain. It is an open source library for high performance numerical computation. TensorFlow creates a computation graph and the data "passes" through it. The best part of TF is, that gradients of the functions are written in advance so we don't have to write them. I was using TensorFlow also directly when defining the loss function.

For postprocessing I was using the NumPy[11] library. It is a package for scientific computing in Python. It can handle high dimensional arrays which is useful when you want to compute with tensors.

5.2 Hardware

At Continental, I was using professional workstations. They are equipped with 64 GB RAM . I ran my trainings on dedicated Nvidia GPU-s. Every workstation contained four Nvidia GeForce GTX 1080 Ti 11GB GPUs. That means 11.3 TFLOPs (FLoating point Operations Per Second) of FP32 (32 bit floating point) performance. Using Keras multi-GPU models I could make the training faster. In this case namely, GPUs share their computational power and train on a batch taking it into samples for every GPU. Using this tool training time could be reduced to one third of the original.

I ran the tests on an Nvidia GeForce GTX TITAN X GPU having 11 TFLOPs. This hardware was granted by the Department of Telecommunication and Media Informatics.

6 PROPOSED WORK

In this section I describe the process of the implementation. Instead of using the original version written in the Darknet framework I decided to implement my own version. This solution is more flexible and I can control the whole process.

[10] https://tensorflow.org/ *(accessed October 21, 2018)*
[11] http://www.numpy.org/ *(accessed October 21, 2018)*

The starting point was a scientific article [7] on the internet. This description was not detailed enough for programming so I made an implementation plan for myself. I realized that I have to get acquainted with two main frameworks like Keras's functional API[12] and with TensorFlow. The steps of my work can be traced in the following chapters.

6.1 Model

First I implemented the first part of the network: the darknet-53. It wouldn't be necessary to implement the network in two parts but it will be useful later if we save the weights from that. Since darknet-53 is a convolutional feature extractor, it can be built into other networks. If we freeze the weights of the feature extractor we can train it for other purposes, too.

Implementing the network is pretty simple using the Keras functional API. We can find patterns, so we can simplify layer types in different functions. For example, a 2D convolutional layer has always leaky ReLU activation after batch normalization. Another example is the residual block.

In the darknet-53 we have "bottleneck" residual units. As you can see in Figure 4 we have an input, through 2 layers we downsample it and then add the result to the input with a shortcut. I put the convolutional unit into a function as well as the residual block. Later I just had to call these functions.

Since I've had the converted weights from the Darknet implementation I wanted to make my network the same so that I can load them into my network. I needed some time until I realized that there's no bias in the convolutional layers in the Darknet implementation. But after it, I could load the weights.

It's important to save the 36th and 61st layer into a variable and return it because we have to concatenate it with other layers in the second (detection) part of the network.

In the detection part, we have 2D convolutional units (like in the feature extraction part), upsampling layers. We have three output layers: for large, medium and small objects. Right before the predictions, upsampled layers are concatenated with layers from the feature extractor. This method helps preserve the fine grained features which help in detecting small objects. In the end, we reshape the output in the network, so it's easier to process features.

[12] https://keras.io/getting-started/functional-api-guide/ *(accessed October 21, 2018)*

6.2 Postprocessing

6.2.1 *Process features*

First, we have to convert the raw output to x, y coordinates, width, height, objectness and class probabilities. It can be confusing using multiplication in 4 dimensions. Currently, I'm using NumPy to compute these values but later maybe it would be more efficient to implement this algorithm in TensorFlow to run faster on GPU.

6.2.2 *NMS*

We use non-maximal suppression to eliminate bounding boxes with low confidence and keep only one box from highly overlapping boxes. I implemented it to compute NMS class by class avoiding to eliminate overlapping boxes from different classes. Both confidence threshold and overlapping rate (IoU) threshold values can be changed.

6.2.3 *Drawing bounding boxes*

Over the full postprocess, I used x, y, width and height like if the picture height and width would be 1. It's easier using these values because at the end we only have to upscale them to the size of the image.

6.3 Preprocessing

We're generating training data on the fly. Since I'm using Keras and have a database containing 118 thousand pictures, I am using Keras's *fit_generator()* function. It trains the model batch by batch created by a generator. It's very useful because we don't have to store all the data in the memory only a little part of it (depending on the batches size). This data generating can run (on the CPU) parallel to the training (on the GPU). It loads the input picture and generates the expected output from the annotations: 3 tensors for the 3 scales with the data of coordinates, shifts, confidence and classes. To be able to select which scale to apply for a given object we calculate IoU between the object's bounding box and each anchor boxes of the three scales. The anchor box having the highest IoU value defines the scale.
We normalize the input picture so that the model can fit faster.

6.4 Loss function

This is the most difficult and most important part of the training. Using TensorFlow's functions it makes computing easier.

I wrote the loss function as written in the paper. The 4 coordinates - x, y, w, and h - are in grid scale so that it's easier to compute IoU. In the ground truth I put the coordinates and class in the label of all three boxes but set confidence 1 only at the best. With this solution I can use confidence as a mask for the best box when computing loss and ignoring boxes that aren't the best but overlap with the bounding box over a threshold. X, y, w, h and class loss is calculated only on those cases where the mask value equals one, i.e. object can be found in the appropriate scale.

Since COCO has multi-class labels - class prediction is a one-hot vector - a general solution would be using categorical cross entropy loss. But in the original paper they write that it is faster to use binary cross entropy since they aren't using softmax.

Objectness is calculated on every cell. Its value at one position can be either a one-hot vector or a zero vector (if there isn't any object at the given cell). So here I'm using binary cross entropy.

7 TRAINING

7.1 The dataset

For training I am using the COCO[13] (Common Objects in Context), which is a large-scale object detection, segmentation, and captioning dataset They have a database of 118K pictures labeled for object detection. They also provide 5K pictures for validation. On the pictures you can see complex everyday scenes containing everyday objects in their natural context. The annotation contains the coordinates of the bounding boxes for 80 different categories. A very big advantage of this dataset is that the pictures are varied, which helps the model not to overfit.

[13] http://cocodataset.org *(accessed October 20, 2018)*

7.2 Data augmentation

Augmentation is a method to make your learning data “larger” and your model more robust. For example using data augmentation model will be able to generalize. Without data augmentation my model wasn’t able to recognize object upside down. I’ve found imgaug, a very good OpenCV and skimage based augmentation library. Since modifying input image can also modify the labels it was very important to have augmentation that also modifies bounding boxes the same way it modifies images. I was using flipping (upside-down and left-right), Gaussian blur, grayscale and add as shown in Figure 14.

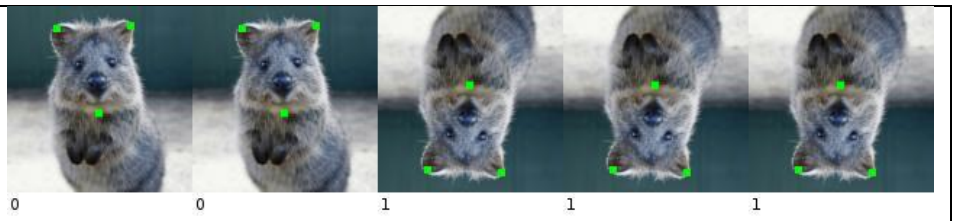
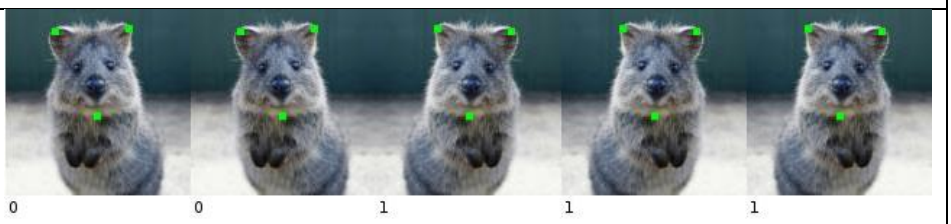
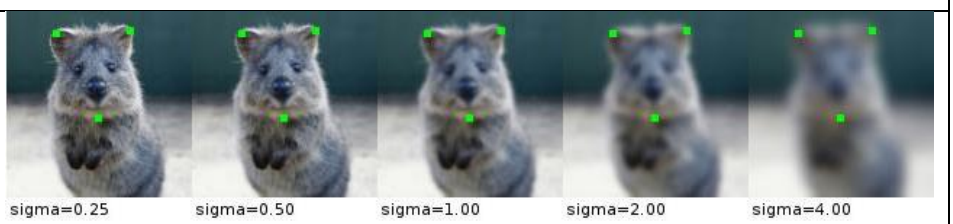
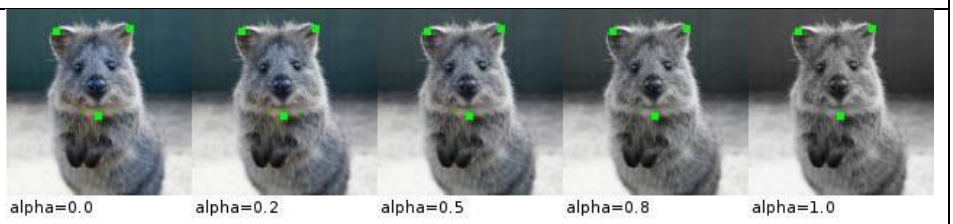
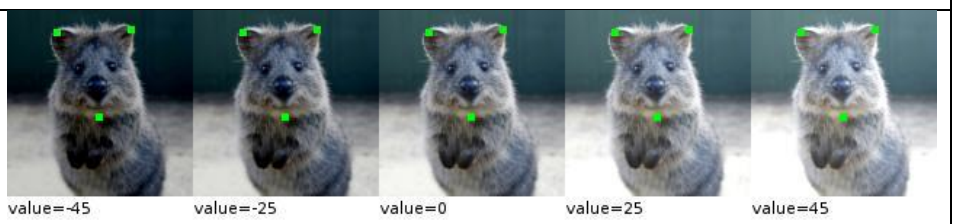
Flip vertical	
Flip horizontal	
Gaussian blur	
Grayscale	
Add	

Figure 14 - Augmentation techniques

First I was using color inverter and dropout, too but the loss went very high and the results were worse.

7.3 Hyperparameters

I tried different learning rate values. Using high (0.001-0.1) learning rate the gradients were too high so the loss value was NaN. That is the so called exploding gradient problem. I had to reduce the learning rate. Selecting some test learning rate values 3e-5 resulted to be optimal and the decrease of the loss values stabilized. The best tool to monitor the loss and accuracy values is TensorBoard[14]. Keras callback is to be set which logs training results per epoch. Values could be traced on a graph in this visualization toolkit.

I used basically Adam optimizer and for the fine tuning the SGD optimizer was applied. To decide from when to use SGD I set another callback called EarlyStopping. This callback stops the training if value of validation loss remains the same after a given number of epochs. The whole training took 2.5 days.

It is recommended using batch size of power of 2 to take advantage of maximum performance. This has hardware reasons. During training we have to store the activation values for the backpropagation algorithm. Since the network is big, I couldn't set batch size bigger than 16. However it was big enough for the training algorithm to stay stable.

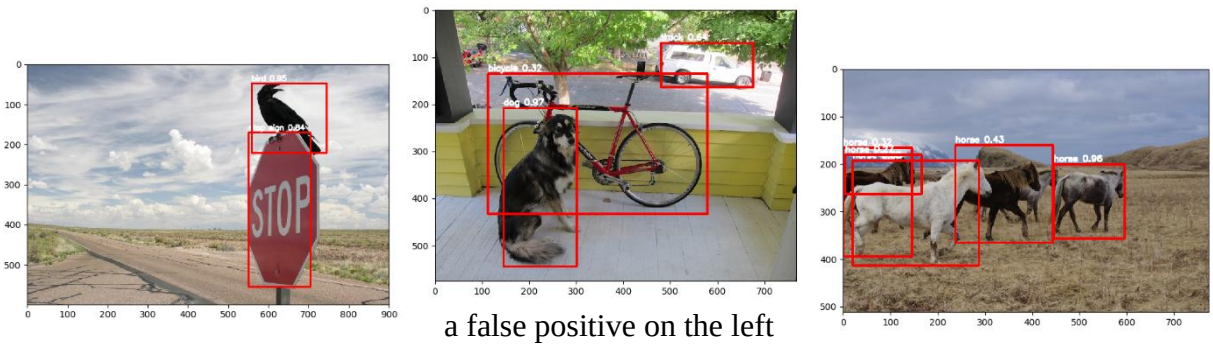
After the training box score threshold and IoU threshold must be set. Box threshold indicates how confident the algorithm is that the object belongs to the given class. If this threshold is too high good detections with low score are ignored. However, if it is set too low inappropriate results are received. IoU threshold shows that over how high IoU value of two bounding boxes from the same class should be dropped the box with the lower score. I made a lot of iterations until I got the appropriate results.

7.4 Results

Some of my results are shown on Figure 15, on the test pictures I got almost the same accuracy like with the original implementation. Sometimes the network makes a bounding box for the whole body and the face separately. Sometimes it makes "person" false positives.

[14] https://www.tensorflow.org/guide/summaries_and_tensorboard *(accessed October 20, 2018)*

a false positive (clock)



a false positive on the left (person)

Figure 15 - Predictions in my implementation

I tested my network on the COCO validation dataset: my implementation reached 30.49 mAP and the original implementation 23.56 mAP (COCO style). COCO style AP is calculated for each class. These values can be seen in the rows in Figure 16. To get the mAP value the average of the AP values were calculated. Graph on the left side shows my results while on the right the Darknet implementation's. My results are approximately 7% better than the original's as an average. For example the zebra on the left side has 70 AP while on the right side it has 57 AP. Although their results are better at certain lines (see the diagram on the right) as the average is higher in my results the overall values are better on the left. The diagrams are composed so that the highest values are on the top and as the average value is higher on the left the decrease of the bars on the left are slower than on the right. That is the picturesque explanation of this figure. Frankly, I don't know why my results are better because I am not familiar with their method in detail. I suppose the main algorithm is the same and difference could lay in the fact that I applied fixed sized pictures while training or I devoted more time on fine tuning.

I ran performance tests on 100 pictures. The critical hardware in this case is also the GPU which was an Nvidia GeForce GTX TITAN X GPU resulted in an inference time of 1.2 seconds as an average. The best value is 1.13 while the worst is 1.43.

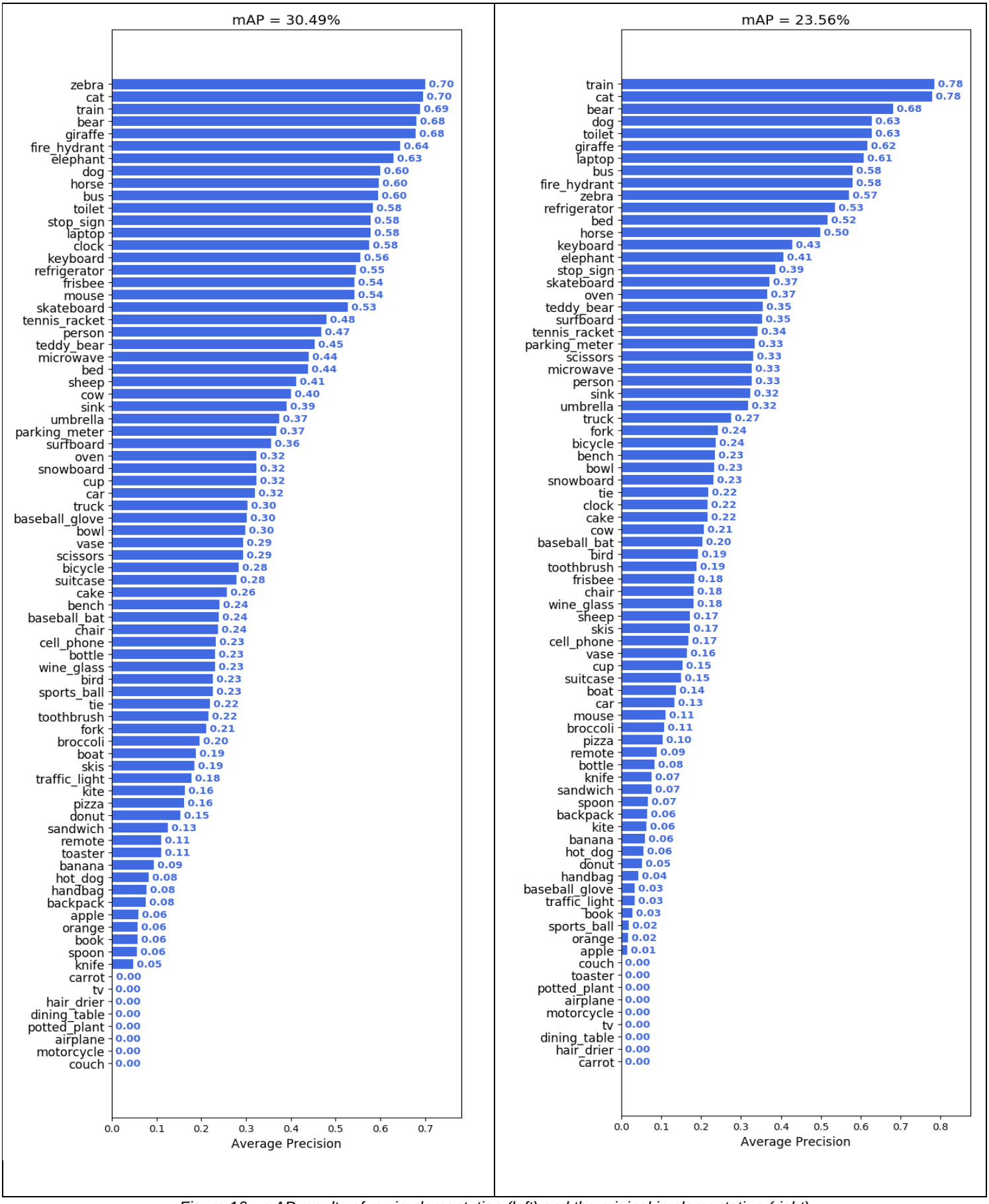


Figure 16 - mAP results of my implementation (left) and the original implementation (right)

8 SUMMARY

Based on the article I started to plan and after a literature review I finalized the concept. Then came the preparatory work, literature review and I could write code. I spent a little bit more time with the necessary debugging than I planned. Using the right tools of the IDE this part could be reduced to three days. I spent another week with finding the appropriate hyperparameters. The training went quite quickly - considering the size of the neural network - in 2.5 days. I really enjoyed each and every part of the work. I appreciated to work on a state-of-the-art technology.

The goal was to write a program in the Keras framework which is

- capable to recognize any kind of object after training
- easy to modify and parametrize
- at least as accurate as the program in the article.

I can state that I have managed to meet the goals. I trained the program on the COCO database containing common objects. Now it recognizes the corresponding objects on the pictures. To get the best accuracy the parameters were fine tuned with iteration. I had quite experience during the fine tuning if one parameter is modified in the wrong way multiply object recognition happens. Practically it means that one object is recognized several times but certainly only one is correct. If you realize that this malfunction is caused by a wrong parameter setting you have to step back and correct. With this example I just wanted to make you feel how sensible is the system for the parameters.

8.1 Future work

I am really glad that my program reached high accuracy but the runtime should be improved. The easiest way to introduce parallel program execution on each parts of the algorithm. Presently majority of the program runs in parallel mode but not all. If the program is accelerated could be used for online applications as well. Another possibility is to reduce the neural network size. Up until now I trained the program only on the COCO dataset. I am planning to run it on a different dataset e.g. KITTI[15]. I am ready to continue this work within the framework of Continental and the university.

[15] http://www.cvlibs.net/datasets/kitti/ *(accessed October 20, 2018)*

ACKNOWLEDGEMENT

First of all I would like to express my thanks to Dr. Gyires-Tóth Bálint Pál from the Department of Telecommunications and Media Informatics at Budapest University of Technology and Economics as my advisor. He continuously supported my work no matter it was a working day or a holiday, night or day. He really helped me in writing this paper knowing that this is my fist scientific try in writing in English.

My second big thank goes to the deep learning team of Continental Hungary Kft. especially to András Kohlmann, Arne Haak, Csaba Gór, Péter Láng, Raul Trenka and Zoltán Kárpáti. I got a strong professional support from this team. And a really strong hardware.

REFERENCES

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai and A. Bolton, "Mastering the game of Go without human knowledge," *Nature,* vol. 550, no. 7676, p. 354, 2017.

[2] M. Zahangir Alom, T. M. Taha, C. Yakopcic, S. Westberg, M. Hasan, B. C. Van Esesn, A. A. S. Awwal and V. K. Asari, "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches," *arXiv preprint arXiv:1803.01164,* 2018.

[3] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448-456..

[4] K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770-778..

[5] I. Goodfellow, Y. Bengio and A. Courvill, Deep Learning, MIT Press, 2016.

[6] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, 2016, pp. 21-37.

[7] S. Ren, K. He, R. Girshick and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal network," in *Advances in neural information processing systems*, 2015, pp. 91-99.

[8] A. F. Joseph Redmon, "YOLOv3: An Incremental Improvement," *arXiv preprint arXiv:1804.02767,* 2018.

[9] J. D. T. D. J. M. Ross Girshick, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[10] K. E. A. v. d. S. T. G. A. W. M. S. J. R. R. Uijlings, "Selective search for object recognition," *International journal of computer vision,* vol. 104, no. 2, pp. 154-171, 2013.

[11] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient graph-based image segmentation," *International journal of computer vision,* vol. 59, no. 2, pp. 167-181, 2004.

[12] I. S. G. H. A. Krizhevsky, "Imagenet classification with deep convolutional neural networks," *NIPS,* pp. 1097-1105., 2012.

[13] A. Ben-Hur, D. Horn, H. T. Siegelmann and V. Vapnik, "A support vector method for clustering," in *Advances in Neural Information Processing Systems*, 2001, pp. 367-373..

[14] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440-1448.

[15] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556,* 2014.

[16] T.-Y. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan and S. J. Belongie, "Feature Pyramid Networks for Object Detection.," in *CVPR*, 2017.

[17] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779-788..

[18] A. F. Joseph Redmon, "YOLO9000: Better, Faster, Stronger," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2017, pp. 6517-6525..

FIGURES