The Calling of St Matthew by Caravaggio, in case you like art :)

RetinaNet: how Focal Loss fixes Single-Shot Detection

Object detection is a tremendously important field in computer vision needed for autonomous driving, video surveillance, medical applications, and many other fields. Have you heard buzzwords such as YOLO or SSD? In this article, I will explain to you several important advances recently made in state-of-the-art object detection using deep learning and lead you to great resources that demonstrate how to implement them. After reading this article you will have a good overview of the field, no prior knowledge in object detection required.

Fabio M. Graetz Follow
Nov 25, 2018 · 17 min read

This article is organized as follows:

First, I explain to you a simple model that is able to detect and classify *one single* object in an image. Then, I summarize different methods that have been used to detect and classify *several* objects in one image focusing on the comparison between *region-proposal* and *single-shot* methods. After that, I explain *in detail* how detectors like *SSD* or *YOLO*

work, why they are faster but also less accurate than *region-proposal* methods and finally how the recently developed *RetinaNet* fixes those problems.

. . .

Single-object detection

Neural networks can be used to solve *classification problems* (predict classes) and *regression problems* (predict continuous values). Today we will be doing both at the same time. We start with a simplified task: detect and classify *one* single object in an image instead of several objects.

Data

How does an object detection dataset look like? Well, the inputs to our model are of course images and the labels are typically four values that describe a ground truth bounding box plus a category the object in this box belongs two.

These four values could, for example, describe the x and y coordinates of the lower left and upper right corner of the bounding box. Or the coordinates of one specific corner and the height and width of the bounding box.

Architecture

Our neural network that takes the image as input has to predict *four values* that represent the *predicted* bounding box coordinates and that we will compare to the four *ground truth* bounding box coordinates (label).

In addition to that, the neural network has to predict a class probability for every of the n categories we want to classify. This gives us a total of $4 + n$ values the network has to predict.

To build our *single object detector* we could take any convolutional image classifier network such as VGG-16 or a ResNet (ideally pre-trained) and remove any classification layers at the top of the network, flatten the output of the truncated base network and add one or two linear layers that finally output $4 + n$ values. You should also remove any (adaptive) pooling layers prior to the classification layers because they destroy the spatial information we need to regress the coordinates of the edges of the bounding boxes.

Loss

So how would we train such a *single object detection* network that produces $4 + n$ values? Predicting the class of the object (n class probabilities) is a *classification* problem. Predicting the four coordinates

for the bounding box is a *regression* problem. We need a loss function that combines these two problems.

Let's start with the bounding boxes: The *localization loss* could be the *L1* loss of the predicted bounding box coordinates *x_pred* and the ground truth bounding box coordinates *x_label*:

$$L_{\text{loc}} = \sum_{i=1}^{4} |x_{\text{pred},i} - x_{\text{label},i}|$$

This is the sum of all the *absolute differences* between the four predicted and four ground truth bounding box coordinates. The more the predicted box differs from the ground truth bounding box, the larger this loss will be. You could use *L2* loss as well, however, the model will be more sensitive to outliers and will adjust to minimize single outlier cases at the expense of the normal examples that have a much smaller error.

What do we do with the *n* values the network outputs for the class probabilities? First, to convert these *n* values to probabilities, we apply the *softmax activation* function to them. Then we use the *cross-entropy loss function* to compare them to the label. We call this *confidence loss*.

The combined loss function is simply the weighted sum of *localization loss* (bounding boxes) and *confidence loss* (classes):

$$L = L_{\text{loc}} + a \cdot L_{\text{conf}}$$

Examples from the Pascal VOC 2007 dataset with a single detected object

What is *a*? In principle, the localization loss could be much larger than the confidence loss (or vice versa). In that case, the network would solely focus on learning to predict the bounding boxes while completely ignoring the classification task. You, therefore, have to look at the values of both losses and multiply one of them with a factor *a* that makes them approximately the same order of magnitude.

We now know how an object detection dataset might look, how we could set up a *single object detection* architecture and how to formulate the loss function. We have everything we need to train the network. If you would like to train such a model, I suggest you start by coding along to fastai lesson 8 :)

. . .

Ok, we discussed the minimal steps needed to detect one single object in an image. But that is not enough, right? In a real-world scenario we probably need to detect many objects (of an unknown number) in a single image. How can we do this?

Classical computer vision often used an approach called *sliding window*. A detector slides over an image and once it detects an object, a bounding box is drawn around the area the detector is currently looking at.

With the rise of Deep Learning so-called *two-stage* or *region-proposal* methods started to dominate: the first stage predicts a set of *candidate object locations* which should contain all objects but filter out most of the background. The second stage, a *convnet*, then classifies the objects in those candidate locations as one of the sought-for categories or as background. Google for R-CNN if you are interested in this technique.

Region-proposal methods produce state-of-the-art results but are often too computationally expensive for real-time object detection, especially on embedded systems.

YOLO-You only look once (Redmon et al. 2015) and *SSD-Single Shot MultiBox Detector* (Liu et al. 2015) are architectures that aim to solve this problem by predicting bounding box coordinates and probabilities for different categories in a *single forward pass* through the network. They are optimized for speed at the cost of accuracy.

Lin et al. (2017) recently published a beautiful paper: they explained why methods like SSD are less accurate than two-stage methods and proposed to address the problem by rescaling the loss function. With this improvement, single-shot methods are not only faster than two-stage methods but now also just as accurate allowing awesome new real-world applications of real-time object detection.

Now, I will explain to you in detail how SSD works, why single-shot methods are less accurate than two-stage methods and how RetinaNet and Focal loss fix this problem.

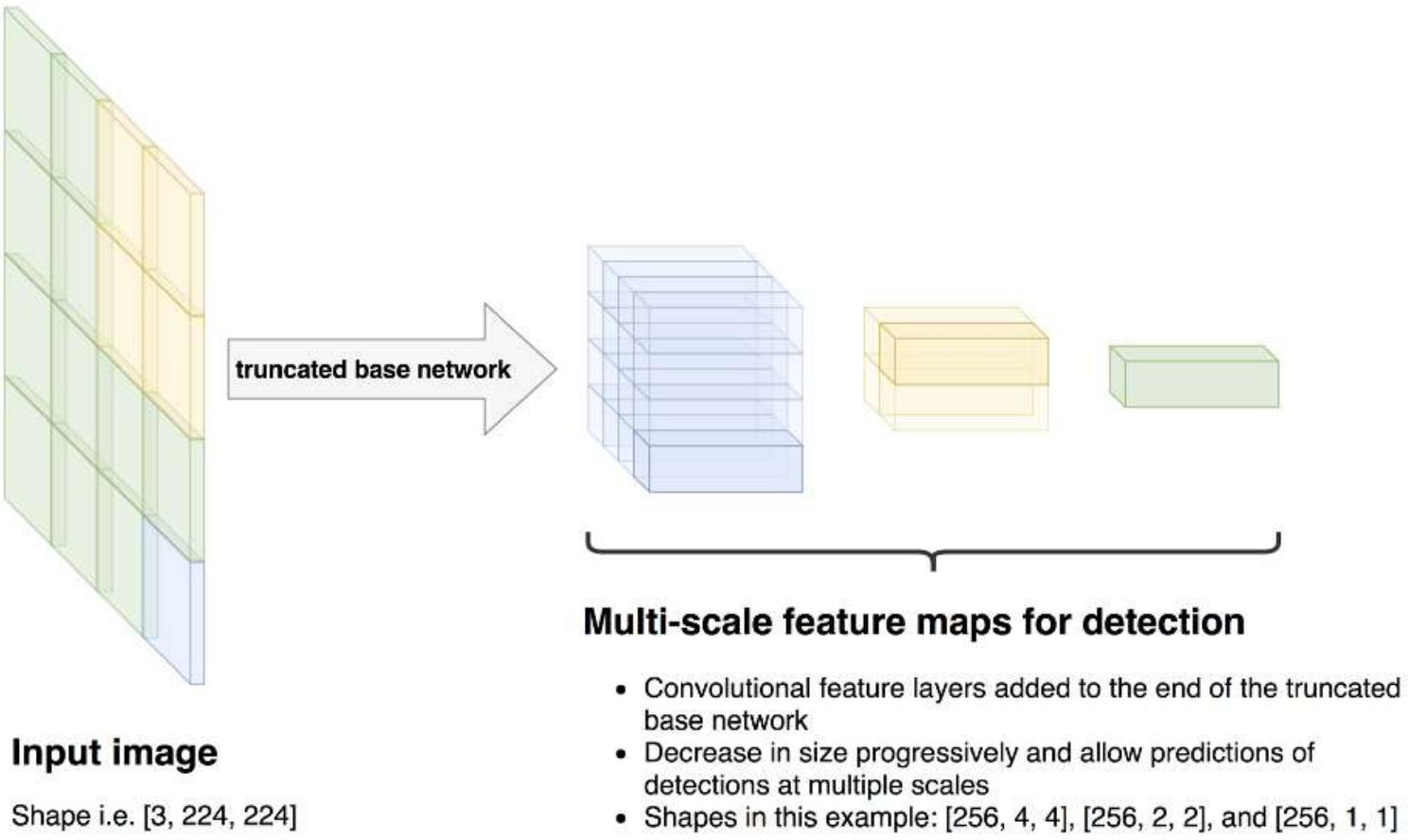
. . .

SSD: Single Shot MultiBox Detector

Let's take a look at the architecture of SSD (similar to the implementation in fastai) and discuss the concept of the *receptive field of an activation* along the way. YOLO works similarly to SSD with the difference that it uses fully connected layers instead of convolutional layers at the top of the network. We will focus on the superior SSD.

The image is fed to a standard architecture for high-quality image classification. Any classification layers at the end of the network are again *truncated*. The SSD paper uses a *VGG-16* network, but other networks such as R*esNets* work as well.

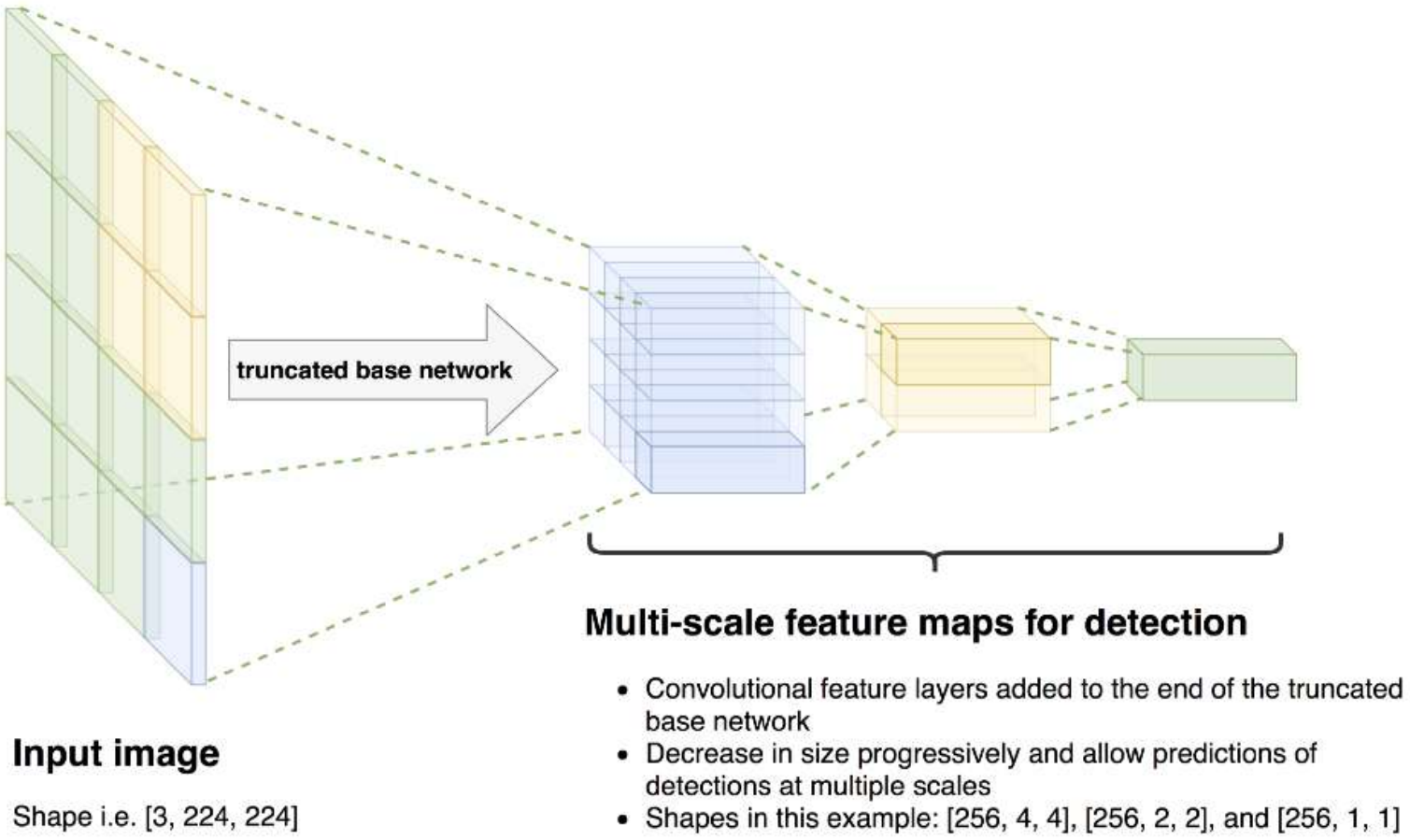
Let's say for instance that the input image has the shape [3, 224, 224] and that the output of the truncated network has the shape [256, 7, 7]. Through three more convolutions we create three *feature maps* at the top of the network with the shapes [256, 4, 4] (blue), [256, 2, 2] (yellow), and finally [256, 1, 1] (green):



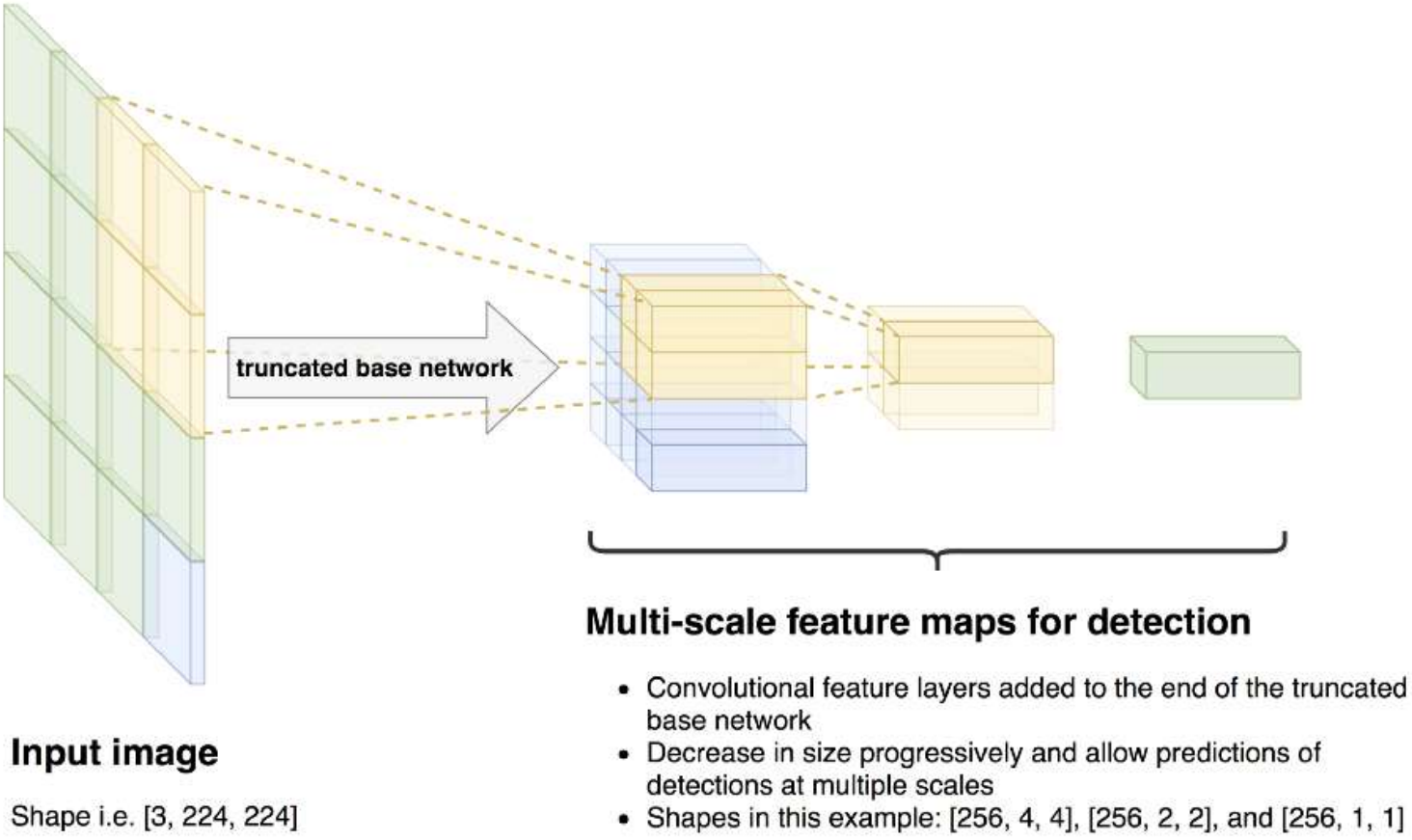
The architecture in the SSD paper has more feature maps at the top of the network but for demonstration purposes, we stick to three layers. The principle is exactly the same.

Let's take a look at the *receptive field* of activations in those different layers:

We start with the final layer (green). The activations in the final green layer have dependencies on all activations in the previous layers and the receptive field is thus the entire input image:



Let's now look at the penultimate (yellow) layer. Notice, that the receptive field of an activation in the yellow layer is only one quarter of the input image:

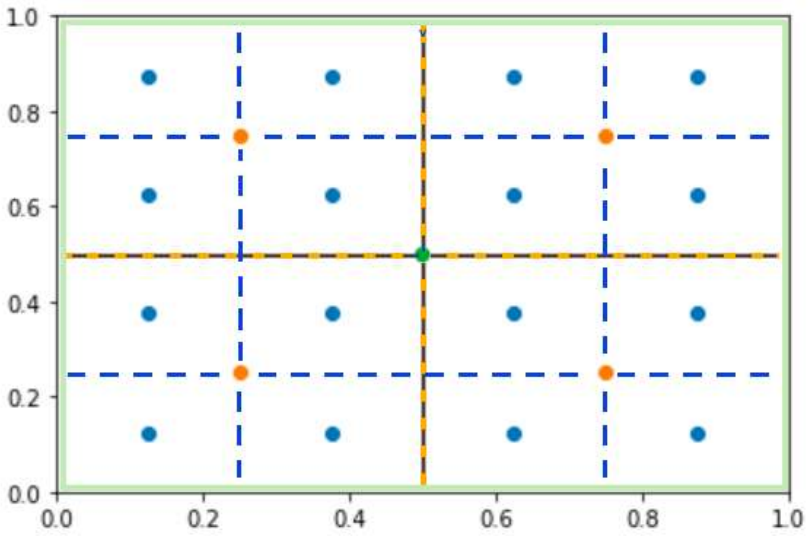


The receptive field of an activation in the blue layer is one 16th of the input image.

Please note that the illustrations I made are simplified for demonstration purposes and for example do not account for the fact that an activation has more dependencies coming in from the center of its receptive field compared to the surrounding regions. Look here for a great explanation.

The idea I want to bring across, however, is that it makes sense that an object that almost fills the entire image should best be detected by activations in the last (green layer) and an object that approximately fills the lower left quarter of the input image should be detected by activations in the yellow layer.

We, therefore, define a number of so-called *default boxes* or *anchor boxes*:



The dots are the so-called *anchors* and mark the centers of their respective *default* or *anchor boxes*. The green anchor in the middle is the center of the green box around the entire image which is the default box corresponding to the last convolutional layer (green).

The yellow/orange anchors define the centers of the four yellow/orange boxes which belong to the penultimate convolutional layer (yellow) and the 16 blue default boxes belong to the blue layer in the illustrated SSD architecture.

You might well ask yourselves now how this would look like in code. You could, for example, create an array in which every line defines one default box and that has four columns, for example the x and y coordinates of the lower left and top right corners or the coordinates of one corner plus height and width.

What do we do next? When we looked at how we can detect one single object in an image, we predicted 4 values for the coordinates of the bounding box and one probability for each of the n classes giving us a total of $4 + n$ numbers returned by our neural network. Keep in mind that we now also want to classify *background* so n should be the number of categories in your dataset + 1.

This is what we now need *per default box*: Every default box needs n values that represent the probabilities that a certain class was detected in that box and 4 values that now are not absolute coordinates of the predicted bounding box but rather the offset to the respective default box.

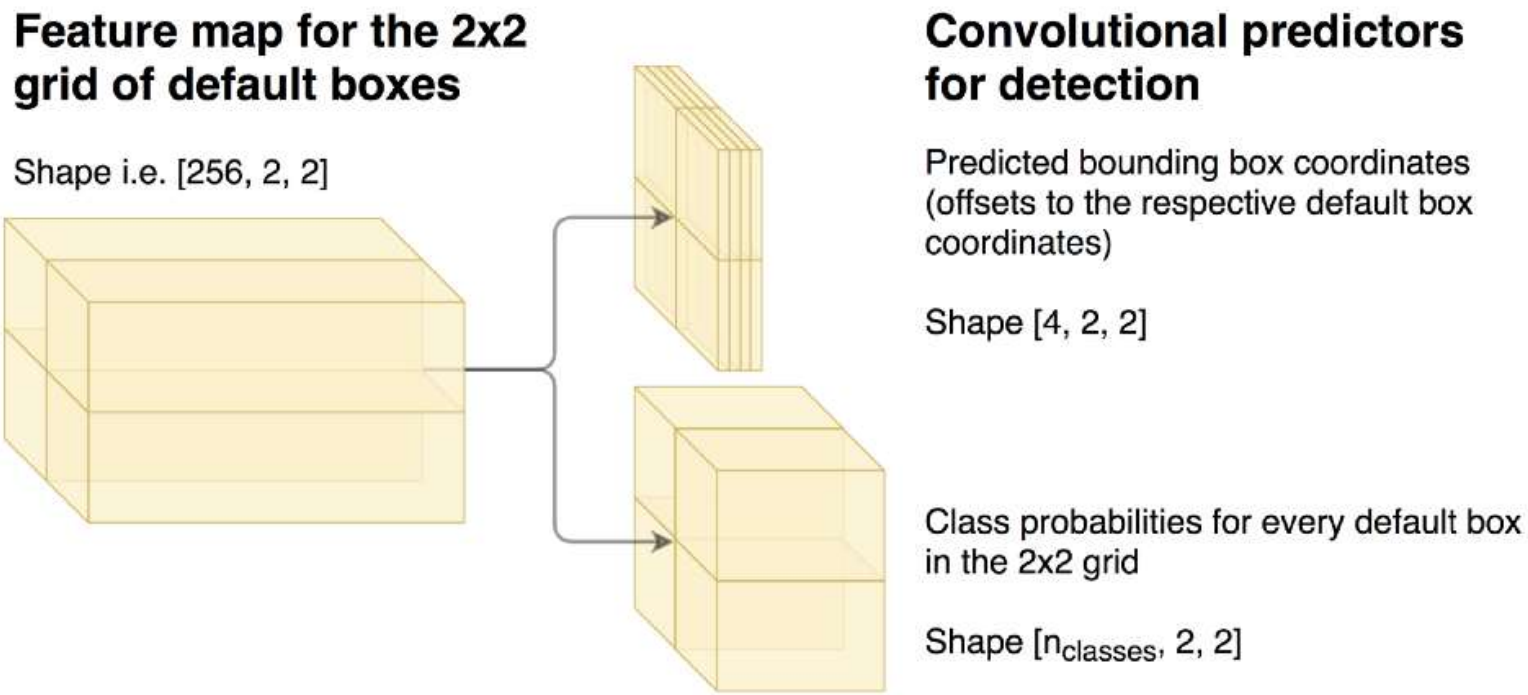
For the 4x4 blue default boxes, we need a total of 16 times 4 values to calculate the coordinates of the predicted bounding boxes and 16 times n probabilities for n categories we try to classify.

For the 2x2 yellow default boxes, we need a total of 4 times 4 values for the coordinates and 4 times n probabilities for the n classes.

How do we do this? More convolutional layers!

The last three feature maps in the sketched SSD architecture—the blue, yellow, and green layers—have the shapes [256, 4, 4], [256, 2, 2], and [256, 1, 1]. They correspond to the 4x4 grid of blue default boxes, the 2x2 grid of yellow default boxes and the one single green default box spanning the entire image in the grid shown above.

Every one of the three feature maps is input to two more convolutional layers. I know that it gets confusing but bear with me, we are almost there :)



Let's take a look at the yellow feature map (2x2 grid). We have [256, 2, 2] activations and need an output of shape [4, 2, 2] and a second output of shape [*n*, 2, 2] (where *n* is the number of classes). So the width and height of the feature map must not be changed but the two convolutional output layers need to reduce the feature map from 256 to 4 and *n* filters, respectively.

For the blue feature map (which corresponds to the 4x4 grid), we need two output layers that reduce the [256, 4, 4] feature map to a [4, 4, 4] and a [*n*, 4, 4] tensor.

The green default box that looks at the entire image corresponds to the green feature map that has the shape [256, 1, 1]. We finally need two more convolutional output layers that take the green feature map as input and produce [4, 1, 1] and [*n*, 1, 1] shaped outputs.

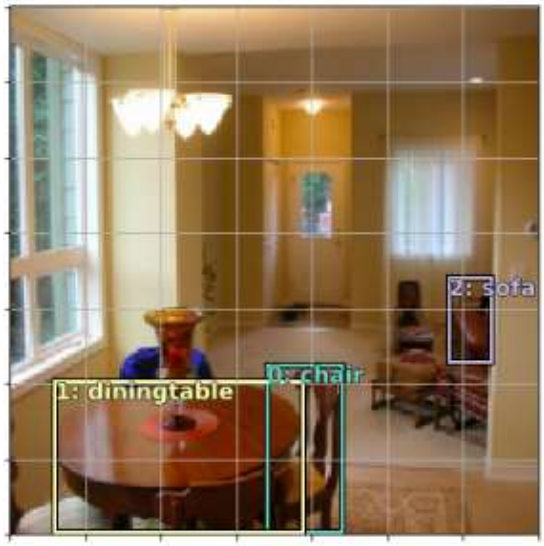
I know this was a lot. The important idea is that we do for every default box in the differently sized grids what we did when predicting one single object in an image.

. . .

Ok, let's summarize:

1. We defined several grids of differently sized default boxes that will allow us to detect objects at different scales in *one single forward pass*. Many previous architectures detected objects at different scales for example by passing differently sized versions of the same picture to the detector which is computationally more expensive.

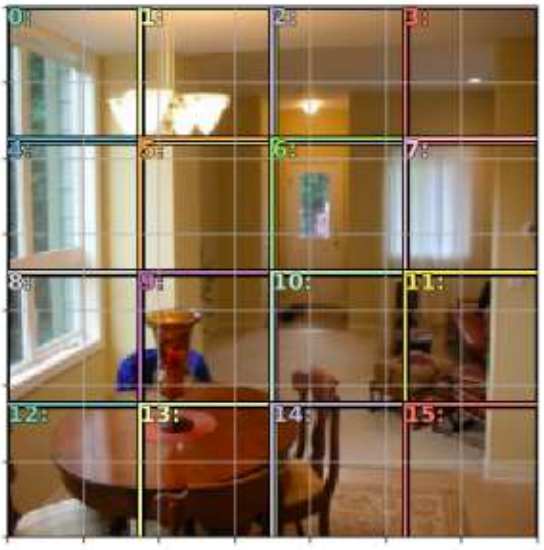
2. For each default box in every grid, the network outputs n class probabilities and 4 offsets to the respective default box coordinates which give the predicted bounding box coordinates.



Ground truth bounding boxes

So how do we train the network?

On the left, you see an example from the Pascal VOC 2007 dataset for object detection. There are three ground truth bounding boxes for a dining table and two chairs.



4x4 grid of default boxes

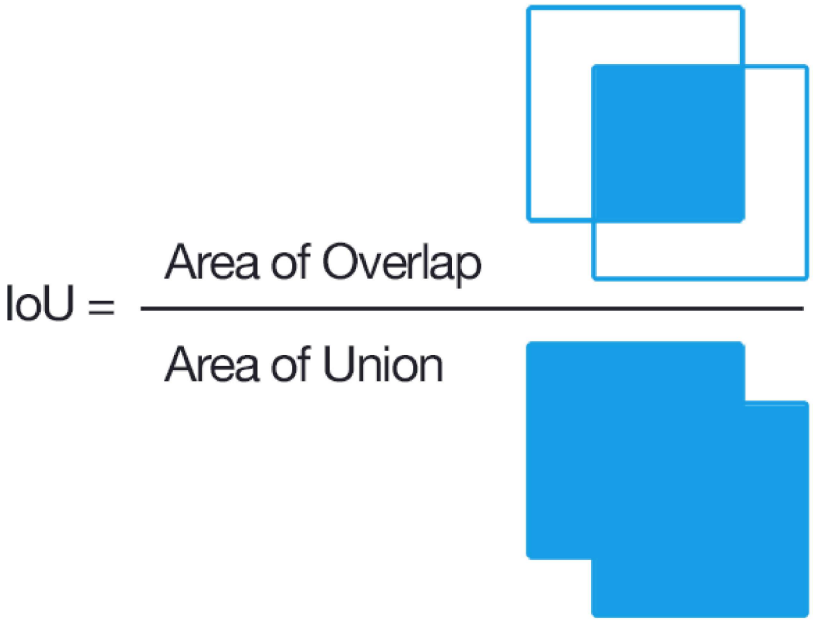
In the picture below you see the 4x4 grid of default boxes (The 2x2 and 1x1 grids are not shown).

The network predicts 4 coordinates (offsets) and n class probabilities for every of the 16 default boxes. But which of those predictions should we compare to the ground truth default boxes in our loss function while training?

We need to match each of the ground truth bounding boxes in our training example to (at least) one default box.

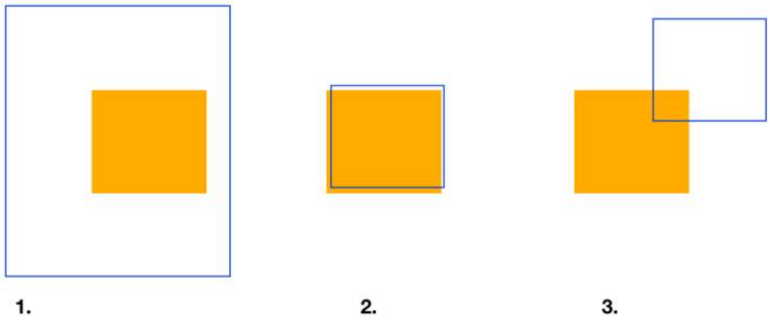
Matching problem

We want to match the ground truth bounding box to a default box that is “as similar” to it as possible. What do I mean with similar? Two boxes are similar when they overlap as much as possible while having as little as possible area that is non-overlapping. This is defined by the *Jaccard* index or *IoU* index:



Source

Let's look at a few examples:



1. The overlapping area (intersection) is the entire orange area, so overlap reached a maximum. But the union (blue area) is much larger. This is not a good match which results in a low Jaccard index.
2. This is the best match: the area of overlap and the area of union almost have the same size. The Jaccard index should be close to 1.
3. Here again, the overlap is much smaller than the entire area of the two boxes (union) which gives a low Jaccard index.

We calculate the Jaccard index for *every* ground truth bounding box in our training example with *every* default box in our 4x4, 2x2 and 1x1 grids.

A ground truth bounding box is matched to the default box it shares the highest Jaccard index with and to every other default box with which it has a Jaccard index that is larger than a certain threshold.

I know that these details might sound confusing. The important idea is that we want to compare the ground truth bounding boxes in the training

example to predictions made by default boxes that are already very similar to the ground truth bounding boxes.

We now know…

1. … 4 predicted coordinates (offsets to default boxes) and n predicted class probabilities for *every* default box and …
2. … which ground truth bounding box is matched to which default box.

Now we do the same thing as we did when we wanted to detect only one single object.

The loss is again a weighted sum of *localization loss* (bounding boxes) and *confidence loss* (classes): We calculate how much the predicted bounding box (default box coordinates + predicted offsets) differs from the matched ground truth bounding box (L1 loss) and how correctly the default box predicted the class (binary cross entropy). We add those two values for every matched default box and ground truth bounding box pair together and have a loss function that will return lower values when the predicted boxes are closer to the ground truth bounding boxes and when the network did a better job at classifying the objects.

When not a single class probability crosses a certain threshold, the network will consider the object in that box as background. In reality, you will find that for every real object in the image the network will produce several bounding boxes that almost lie on top of each other. A technique called *non-maximum suppression* is used to reduce all boxes that predict the same class and have a Jaccard index that is larger than a certain threshold to one single predicted box per detected object.

• • •

There is one small detail I omitted for the sake of understandability. For every anchor, SSD defines k default boxes of different aspect ratios and sizes instead of just one as discussed in this article. This means that we also need 4 times k predicted offsets to the respective default box coordinates instead of 4 and n times k class probabilities instead of n. More boxes of different sizes and aspect rations = better object detection. This detail is not too important for the general understanding, but you can look at Fig. 1 in Liu et al. 2015 for illustration if you are interested in this detail.

• • •

SSD and YOLO need only one forward pass through the network to predict object bounding boxes and class probabilities. Compared to *sliding windows* and *region proposal* methods they are much faster and

therefore suitable for *real-time* object detection. SSD (that uses multi-scale convolutional feature maps at the top of the network instead of fully connected layers as YOLO does) is faster and more accurate than YOLO.

Only remaining problem: region proposal methods such as R-CNN are more *accurate*.

Focal Loss

The advantage that two-stage methods have is that they first predict *a few* candidate object locations and then use a convolutional neural network to classify each of these candidate object locations as one of the classes or as background. The emphasis here is on *a few candidate locations*.

Methods like SSD or YOLO suffer from an extreme *class imbalance*: The detectors evaluate roughly between ten to hundred thousand candidate locations (way more than the 4x4, 2x2 + 1 default boxes in our example here) and of course most of these boxes do not contain an object. Even if the detector easily classifies these large number of boxes as negatives/background, there is still a problem.

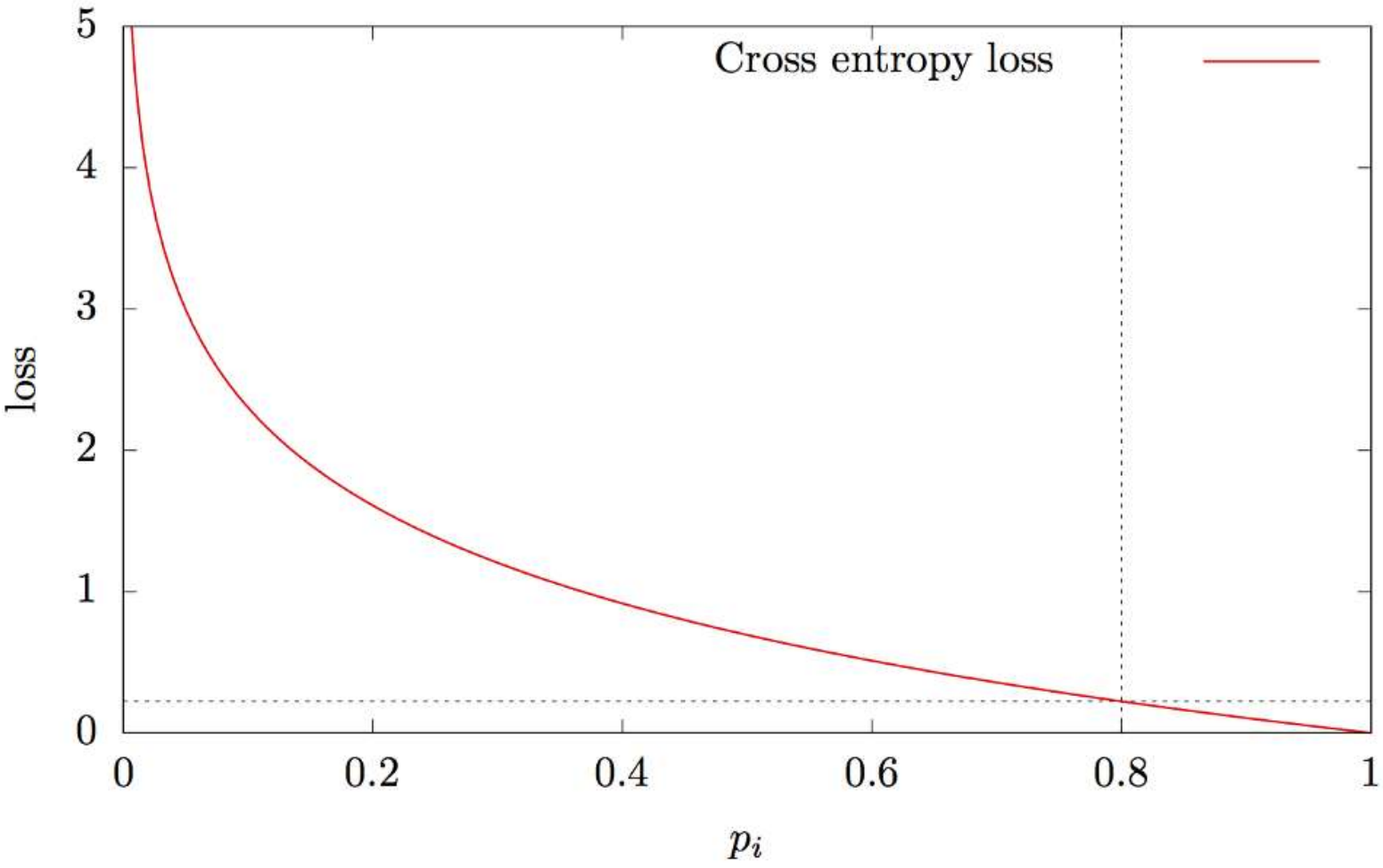
I'll explain to you why:

This is the cross entropy loss function where *i* is the index of the class, *y_i* the label (1 if the object belongs to class *i*, 0 otherwise), and *p_i* is the predicted probability that the object belongs to class *i*.

$$C(p, y) = -\sum_{i} y_i \ln p_i$$

We will look at the plot, don't worry too much about the equation :)

Let's say a box contains background and the network is 80% sure that it actually is only background. In this case *y(background)=1*, all other *y_i* are 0 and *p(background)=0.8*.



You can see that at 80% certainty that the box contains only background, the loss is still ~0.22.

Let's say for example that we have ten actual objects in the image and the network is not really sure to which class they belong so that their loss is i.e. ~3. This would give us around 30 *(all numbers are absolutely fictional and chosen for demonstration purposes by the way)*.

All the other ~10,000 default boxes are background and the network is 80% sure that they are just background. This gives us a loss of around 10,000*0.22 = 2200.

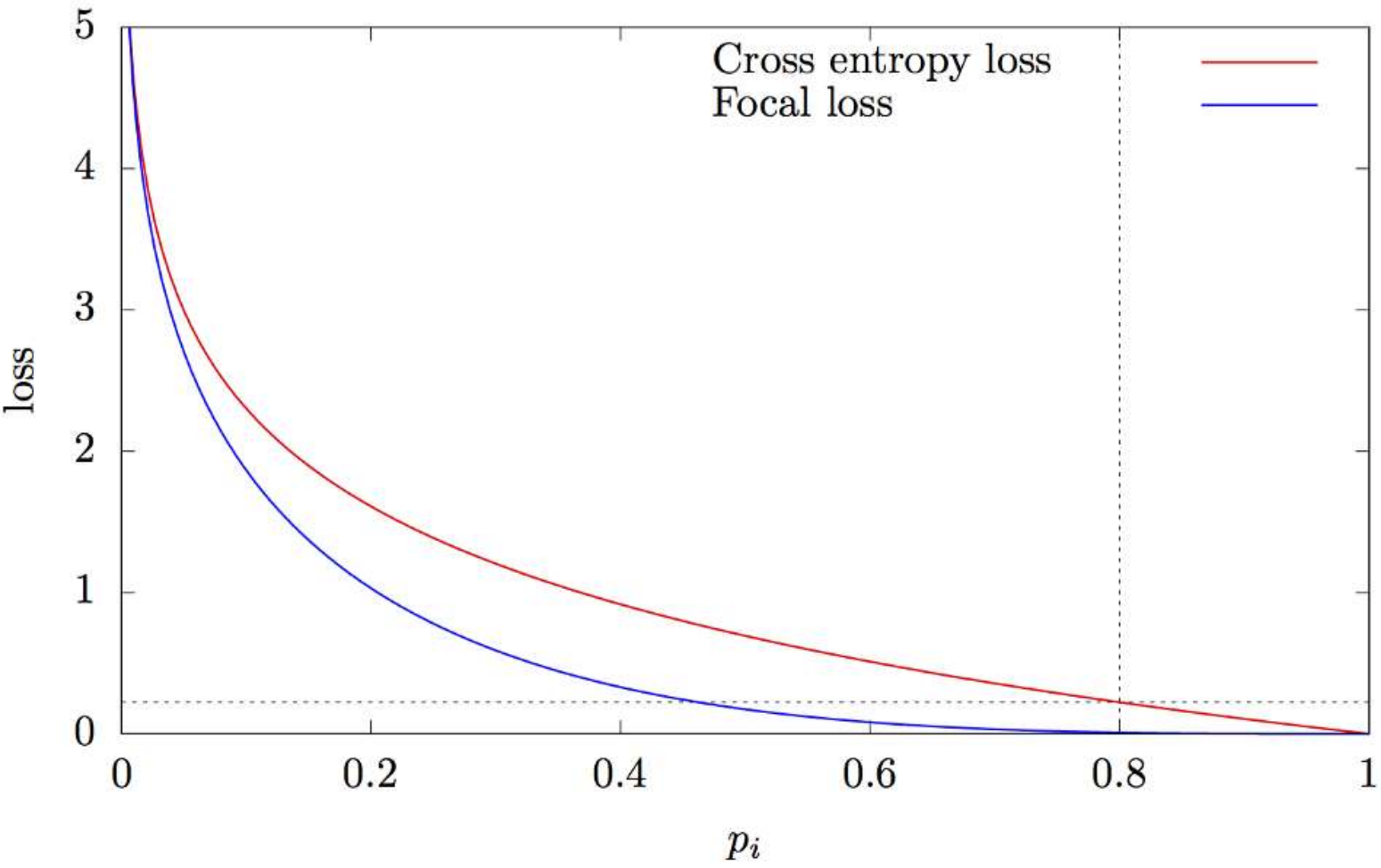
So tell me, what dominates? The few true objects the network actually has difficulties to classify? Or all the boxes that the network easily classifies as background?

Well, the large number of easily classified examples absolutely dominates the loss and thus the gradients and therefore overwhelms the few interesting cases the network still has difficulties with and should learn from.

So what do we do? Lin et al. (2017) had the beautiful idea to scale the cross entropy loss so that all the easy examples the network is already very sure about contribute less to the loss so that the learning can *focus* on the few interesting cases. The authors called their loss function *Focal loss* and their architecture *RetinaNet* (note that RetinaNet also includes Feature Pyramid Networks (FPN) which is basically a new name for *U-Net*).

$$C(p, y) = -\sum_i y_i (1 - p_i)^\gamma \ln p_i$$

According to the paper gamma = 2 works best. Again, if you don't want to, don't worry too much about the equation, you will understand the difference this makes once you see the plot:



Notice that when the network is pretty sure about a prediction, the loss is now significantly lower. In our previous example of 80% certainty, the cross entropy loss had a value of ~0.22 and now the focal loss a

value of only 0.009. For predictions the network is not so sure about, the loss got reduced by a much smaller factor!

With this rescaling, the large number of easily classified examples (mostly background) does not dominate the loss anymore and learning can concentrate on the few interesting cases.

. . .

With this powerful improvement object detectors that need only a single forward pass through a network are suddenly able to compete with two-stage methods regarding accuracy while easily beating them with respect to speed. This opens many new possibilities for accurate real-time object detection even on embedded systems. Awesome!

. . .

If you are as fascinated by this stuff as I am, I highly recommend you implement a single-shot object detection network yourself. I suggest you code along to Lesson 8 and 9 from the great course *fastai—Cutting Edge Deep Learning for Coders*. You find the corresponding notebook here. In this notebook, all functions regarding the matching problem are just written in a cell and their output is shown only later. This reduces the understandability of the code a lot. I reimplemented the notebook, added more comments and brought everything (especially the matching problem) in an order that hopefully allows you to read the notebook top to bottom while gradually building an understanding of how the ground truth bounding boxes are matched to the default boxes before proceeding to implement functions for this purpose. Feel free to use my notebook for additional reference as well.

. . .

I hope you learned something interesting from this article :) If any part is unclear or you need additional explanations, please leave a comment, I'd love to help you understand.