University of Southern Denmark

AI4 project – Deep Learning

The Guide to YOLO

And Yolo-Strike

Niels Hvid
120293
nhvid13@student.sdu.dk

Mathias Thor
210393
mthor13@student.sdu.dk

Project period: February 1, 2017 - June 19, 2017

Contents

Chapter 1

Introduction

Performing real-time object detection and classification is a growing research area with a wide range of applications. In computer vision, classification is the process of finding out *what* is in the image, while detection is the process of finding out *where* it is in the image. When combined the expected output is a location in the image along with the classification of what object is present at that location.

YOLO (You Only Look Once) is a convolutional neural network that can do just that. It uses a single network to predict bounding boxes (locations) and class probabilities (classifications) directly from full images in one evaluation. Since the whole detection pipeline is a single network, it can be optimized end-to-end directly on detection performance [1]. This is different from competing approaches that often requires more than one network and/or network runs. Some of these approaches are explained in detail in appendix B.

The authors of YOLO have done a good job open sourcing their implementation, however for non experts in the field it can be complicated to start training the network with data different from that provided by the authors. Also knowing and understanding some details of the YOLO network may help improving the performance of the final network.

In this paper we will thoroughly explain the YOLO design structure as well as its strengths and weaknesses (chapter 2). Then we will, in a guide-like structure, explain how to train the network using DARKNET and other useful scripts (chapter 3). Finally we will apply the knowledge gained from the previous chapters to detect friendly and hostile players in the computer game Counter-Strike: Global Offensive (chapter 4). At the end of this report the reader should be able to apply YOLO to their own problem in order to come up with new and interesting use cases.

Chapter 2

The YOLO network

In this chapter the network structure of YOLO will be explained. Firstly we describe why YOLO is a good choice and how it works including a short section about convolutional neural networks in general. Then we explain the actual design of the network, including the improvements introduced in a second version of YOLO . This chapter may be skipped by readers that only desire to train YOLO with their own custom data and do not need to know the underlying theory.

2.1 Why YOLO?

YOLO (*You Only Look Once*) is a new approach to object detection presented in may 2016 by Redmon *et al.'s* [1]. YOLO is able to do real-time object detection at a high accuracy using only a single run-through in the convolutional neural network (*hence the name of the network*). The authors achieve this by re-framing object detection as a single regression problem, going from image pixels (*input*) straight to bounding box coordinates and associated class probabilities (*output*). This is very different from the other well-known detection algorithms discussed in appendix B, which evaluates object classifiers at various locations and scales in test images.

The new approach that YOLO uses is especially notable for four major strengths:

First, YOLO is extremely fast due to the simple and unified network model. Their version 1 (YOLO 1) network runs at 45 frames pr. second on a Titan X GPU, while their new version 2 (YOLO 2) is able to improve this by 22 frames resulting in a stunningly 67 frames pr. second [1, 2] (see table 2.1).

Second, YOLO 1 has a good mean average precision (mAP) when considering its speed and YOLO 2 is even better when compared to competing methods [3] (see table 2.1). mAP is a measure for the average precision (AP) for each class in your data set based on the detected bounding boxes and objects. We suggest reading `https://sanchom.wordpress.com/tag/average-precision/` for a detailed explanation of AP.

Detextion Frameworks	Train	mAP	FPS
Fast R-CNN	2007+2012	70.0	0.5
Faster R-CNN VGG-16	2007+2012	73.2	7
Faster R-CNN ResNet	2007+2012	76.4	5
YOLO 1	2007+2012	63.4	45
SSD300	2007+2012	74.3	46
SSD500	2007+2012	76.8	19
YOLO 2 288x288	2007+2012	69	91
YOLO 2 352x352	2007+2012	73.7	81
YOLO 2 416x416	2007+2012	76.8	67
YOLO 2 480x480	2007+2012	77.8	59
YOLO 2 544x544	2007+2012	78.6	40

Table 2.1 – Comparison of different object detection frameworks [2]. The different entries in YOLO 2 is the same network trained with different input image resolutions. See description of the other detectors in appendix B. All tests were done with a GTX Titan X

Third, YOLO has better global knowledge about the image when making predictions, since it considers the full image at training and test time. This results in less background errors where the detector confuses background patches for objects due to lag of context [1].

Fourth, the authors of YOLO have done a good job at making their training and testing code open source and easily accessible. This is especially seen in the thriving Google Group discussing YOLO [4] and the many projects that already uses YOLO [5, 6].

2.2 How does YOLO work?

As mentioned in the previous section, two versions of YOLO has been released. This section will mainly concern the design used in YOLO 1, as it is the foundation of their novel approach to object detection and classification. The improvements introduced by YOLO 2 are discussed in section 2.4 which is later to be used for detecting Counter-Strike players in chapter 4.

2.2.1 The predictions

A very simple illustration of the YOLO 1 architecture is shown in figure 2.1.

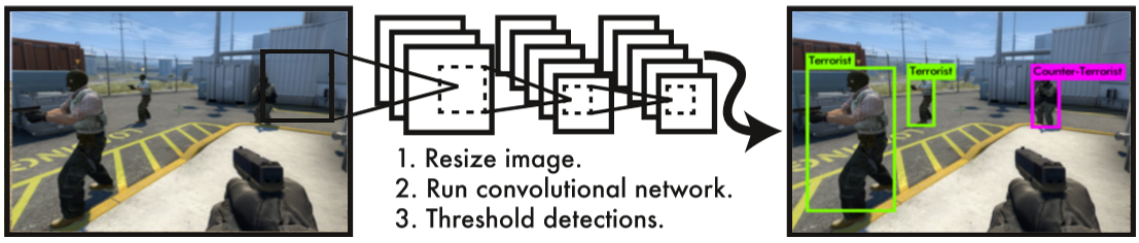


Figure 2.1 – Simple illustration of the YOLO detection system. Figure modified from [1]

The first step that YOLO 1 takes is to resize the input image to 448×448 and divide it

into $S \times S$ cells (Feature map). Each of these grid cells predicts B bounding boxes and B confidences. Each bounding box consists of 4 predictions: x, y, w, h, presenting the center of the box relative to the grid cell (x, y) and the size of the box (w, h). Each confidence score reflects how confident the model is that the predicted bonding box contain and fits the current object. If no object exists in a cell then the confidence should be zero, otherwise it will equal the intersection over union (IOU) between the predicted box and the ground truth (as shown in figure 2.2).

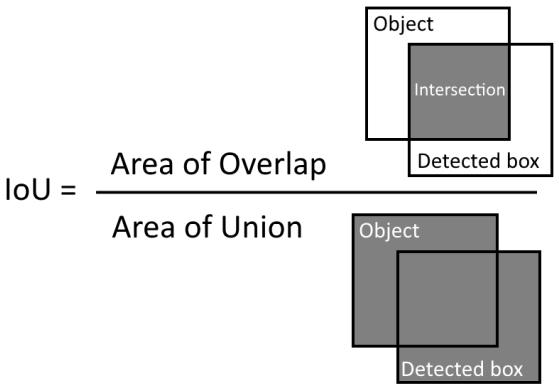


Figure 2.2 – Illustration of the intersection over union (IOU)

The last things that are predicted in each grid cell are the class probabilities. These probabilities are conditioned on the grid cell containing an object ($Pr(Class_i|Object)$), meaning that if a bounding box contains an object (which is calculated with the confidence score for each box) then the object must be the one with the highest class probability. The resulting output tensor for a system with $S = 7$, $B = 2$ and 20 classes is shown in figure 2.3. The total number of predictions is more generally given by $S \times S \times (B \cdot 5 + Classes)$, meaning that the tensor in figure 2.3 has a total of 1470 outputs. Note that the value of B is found experimentally and in the YOLO 1 paper they use $B = 2$ as it works better than 1 and not much worse than 3. Setting B to a large number will of course lower the error, but also slow down the network.

2.2.2 The architectural design

YOLO 1 consists of 24 convolutional layers followed by 2 fully connected layers and is able to do real-time detection and classification when using a high-end graphics card. If one would like to use YOLO on a system without such a graphic card, a fast version of YOLO can be used. This version uses fewer convolutional layers resulting in less computations but a larger error (mAP). Both the input and the output to and from the network are otherwise the same as those of the standard YOLO.

The fully connected layers at the end of the YOLO 1 network are used to predict the output probabilities and coordinates discussed in the previous section, while the initial convolutional layers are used as feature extractors/classifiers. Only the final layer uses

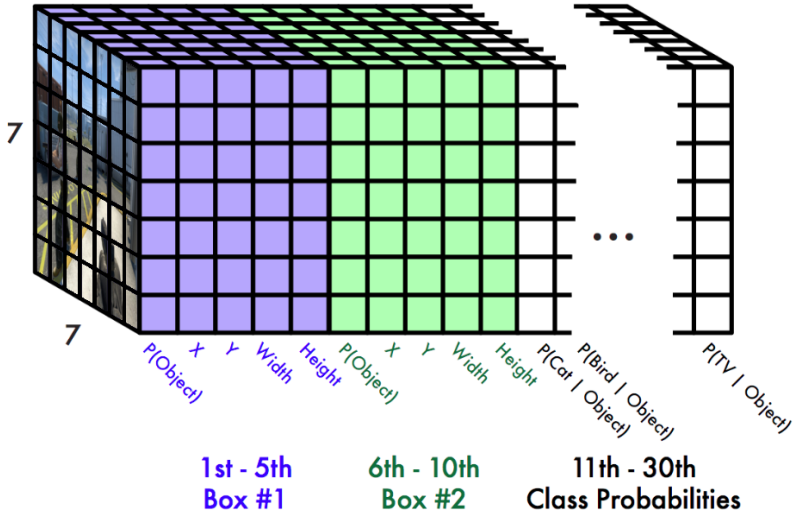


Figure 2.3 – The resulting output tensor from the network, which contains all the parameters that the network has to predict. Figure modified from [7]

a linear activation function while all others layers use the following leaky rectified linear activation function [1]:

$$\theta(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.1x, & \text{otherwise} \end{cases} \tag{2.1}$$

To avoid overfitting YOLO 1 uses dropout [1]. Overfitting happens when a network is trained too much on the same data so it becomes too complex. By building a very complex network it is easy to perfectly fit the training data, but when new and unseen data is presented to the network it performs poorly. In other words, the network does not *generalize* well. The key idea behind dropout is to randomly remove/drop connections and units from the neural network during training. This prevents the units from co-adapting too much, which can result in overfitting [8].
To additionally prevent overfitting extensive data augmentation can be used. Data augmentation is a method for extending the original training set by making small changes to the input images. In YOLO it is possible to randomly scale, translate and rotate the input image, as well as randomly adjusting its exposure and saturation [1].

There are generally two ways of initializing a convolutional neural network; with pre-trained or random weights. It is often preferred to use pre-trained models as small miscalibrations of the initial weights may lead to vanishing or exploding gradients, as well as poor convergence properties [9]. For YOLO the first 20 convolutional layers of the network are pretrained on the ImageNet 100-class competition data-set for object classification [1]. This training takes around a week and that is also why various pretrained models are available on the official YOLO website [3]. The pretrained model is then converted to perform detection by adding four convolutional layers and the two fully connected layers to the network, all with

randomly initialized weights [1].

The YOLO 1 network trains by optimizing a multi-part loss function which has the following properties [1][1]

- The loss function only penalizes classification error if the current grid cell contains an object. This is to avoid the fact that confidence scores close to zero (i.e. grid cells that do not contain any objects) overpowers the gradient from cells that do contain objects, which may cause instability.
- The used error metric reflects that small deviations in large bounding boxes matter less than for small boxes. This is done by predicting the square root height and width of the boxes instead of the height and width directly.
- Remember that YOLO 1 predicts B bounding boxes per grid cell. However, the loss function only penalizes bounding box coordinate error for the box with the highest IOU in the current grid cell. This leads to specialized bounding boxes, that each gets good at predicting a certain type of objects.

Throughout training it is possible to specify the batch size, momentum, decay and learning rate.
Batch size defines the number of training examples that are going to be propagated through the network. The advantages of training with batches is that it requires less memory and that it often trains faster because the weights are updated after each propagation (also known as mini-batch learning).
Momentum adds a fraction of the previous weight update to the current one. This helps the network avoid local minimums or saddle points and to increases the speed of convergence of the system. Too high momentum can, however, result in overshooting the desired minimum which causes the system to become unstable.
Decay (also known as weights decay) is an additional term used when updating the weights of the network that causes the weights to exponentially decay towards zero if no other update is scheduled. This helps suppressing any irrelevant weights and static noise on the targets, which improves generalization [10].
The final and perhaps most important parameter is the learning rate. This parameter dictates how fast the network should learn. Too high learning rate will result in an unstable gradient while a low one will make the training process slow. During the training of YOLO an adaptive learning rate is used, which start low and then rises, to avoid an unstable gradient.

All of the network parameters discussed in this section can be set to fit the needs of the user. To find out how to set them see chapter 3.

[1]The actual equation for the loss function can be found in [1] page 4.

2.3 Convolutional neural networks

Convolutional Neural Networks (CNNs) spread through computer vision like a wildfire, impacting almost all visual tasks imaginable [9]. CNNs aim to model animal visual perception and can thus be used for any visual recognition tasks. CNNs generally consists of 5 types of layers, each of which is described below [11, 12].

Convolutional layer

The convolutional layer is the core of CNNs, the layer works by having a set of small learnable filters that run through the full image. Each filter detects a specific type of feature (See figure 2.4). The feature(s) to be detected is learned by the network itself during training, so there is no need to find, calculate or guess what features are good to look for.

Let us for example say, that we have a 9 by 9 black and white image of an **X**. The network is trained to determine whether there is an **X** in the image or not. A useful filter in this context would be one that could detect diagonal lines. A 3 by 3 filter with a diagonal line is then iterated through the image. At each position, the value of the filter is multiplied with the value of the image and then averaged. This gives a 7 by 7 image with a value between -1 and 1 that indicates how well the diagonal filter matches the image (See figure 2.4). Likewise, a filter with the other diagonal line would probably be useful features to look for when detecting an **X**.

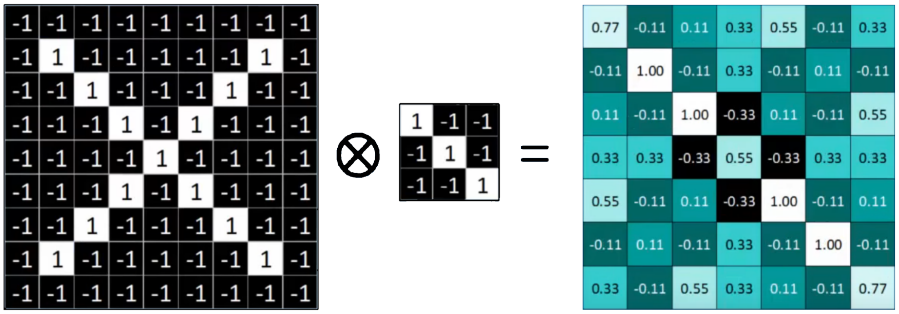


Figure 2.4 – Simple example of a convolutional layer with a 3x3 filter

Pooling layer

The pooling layer is used to shrink the size of the image, this is useful when working with large images. Several implementations of pooling exists, the most common is *max pooling* where the image is partitioned into smaller non overlapping grids, the highest value of each grid is stored in the output image. (See figure 2.5).

ReLU Layer

The ReLU layer (short for Rectified Linear Units) is an elementwise operation, used to change negative values in the image to zero. This is done to introduce nonlinearity to the network, which allows the network to learn more complex relations. Compared to other functions (like sigmoid and tanh), just changing negative values to zero is preferred,

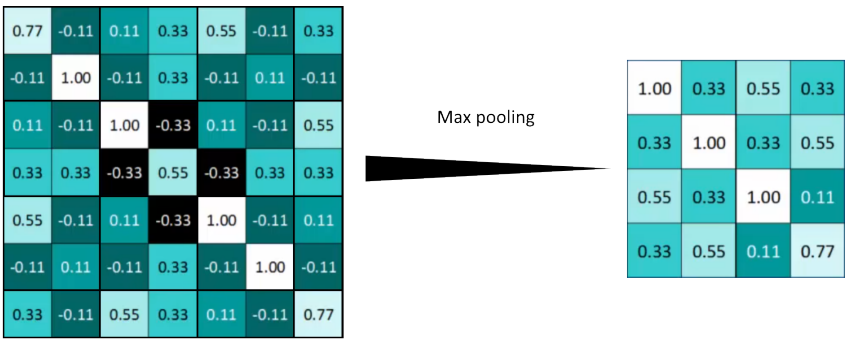


Figure 2.5 – Simple example of a 2x2 pooling layer with a stride of 2

because it results in the network training several times faster, without making a significant difference to generalization accuracy [13].

Fully connected layer

The fully connected layers is where the reasoning starts to show. All neurons in a fully connected layer are connected to all neurons in the previous layer. Fully connected layers are typically placed near the end of the network, after all the features has been extracted (with several convolutional layers) and the image has been compressed (with several max pooling layers).

Loss layer

The last layer in a CNN network is the loss layer, the loss layer is used to specify how errors are penalized during training of the network. Depending on the task, several different loss functions can be used (Softmax, Sigmoid cross-entropy or Euclidean).

YOLO

The structures of YOLO 1 and YOLO 2 can be seen in table A.1 in the appendix.

2.4 The improvements of YOLO 2

YOLO 2 was released approximately half a year after the first version yielding further improvements. Its aim was to fix some of the shortcomings that YOLO 1 suffered from. When compared to other state-of-the-art detection systems, YOLO 1 makes a significant number of localization errors and has less recall (see the mAP scores in table 2.1). Recall is the percentage of relevant objects that are retrieved as shown on figure 2.6.
A general trend when trying to improve the performance of a network, is to use larger and more complex networks. The authors of YOLO did, luckily, not follow this trend as they instead of scaling up the network, simplified the network and made the representation easier to learn. The following five sub-sections explains the most important changes made in the YOLO 2 network.

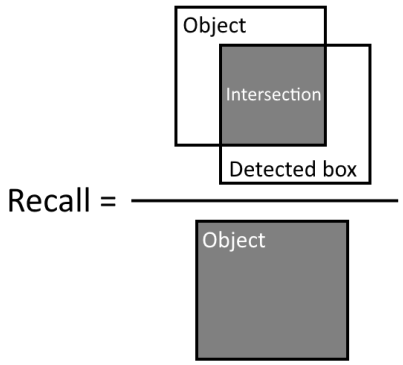


Figure 2.6 – Illustration of the Recall

Batch Normalization

Improvement(s): mAP+2% | Removal of droput
Batch normalization is a technique to provide any layer in a Neural Network with inputs that are zero mean and unit variance. However, a layer may later learn during training that any other distribution might be better and change to that instead. By doing this, one can use a higher learning rate and eliminate the need for Dropout. This is because batch normalization also helps regulating the model and thus prevents overfitting [14]. In YOLO 2 batch normalization is applied to every layer in the network and dropout is completely removed [2].

High Resolution Classifier

Improvement(s): mAP+4%
The YOLO 1 network pre-trains the classifier network using 224×224 images from ImageNet (as talked about in section 2.2.2). It then increases the resolution to 448 for detection, meaning that the network has to adjust to the new resolution and switch to object detection simultaneously. YOLO 2 lets the network adjust for 10 epochs in the pre-training phase before switching to object detection, thus giving the network time to adjust its filters.

Anchor Boxes

Improvement(s): mAP+4.9% | More predictions | Increased recall
YOLO 1 uses fully connected layers to predict bounding box coordinates. In YOLO 2 anchor boxes are used which enables the removal of the fully connected layers. Predicting offsets rather than coordinates also simplifies the problem and makes it easier for the network to learn [2]. Anchor boxes can be regarded as a bunch of prior boxes with different scales and aspect ratio. For a $W \cdot H$ feature map there is a total of $W \cdot H \cdot k$ anchor boxes, i.e. k anchor boxes for each grid cell.

The problem with using anchor boxes is possible model instability and a decreased mAP score. The authors of YOLO addressed this by introducing dimension clusters and direct

location predictions:

- The idea behind dimension clusters is not to use handpicked priors for the box dimensions. Instead good priors are found by running k-means clustering[2] on the ground truth bounding boxes from the training set.
- The idea behind direct location predictions is to predict location coordinates relative to the location of the grid cell, instead of to the entire image. This constrains the location prediction, which makes the parametrizatiation easier to learn and thus the network more stable.

Multi-Scale Training & Dynamic Feature Map

Improvement(s): Possibility to trade between mAP and FPS
In YOLO 1, the size of the feature map was simply hardcoded (default 7) and the input image resolution is always 448×448. YOLO 2 does not use a fixed input image resolution and the user is thus able to adjust this to fit their needs. Also, the size of the feature map is calculated by dividing the input resolution with 32, meaning that to change the size of the feature map, simply change the size of the input. The default input resolution of YOLO 2 is 416×416, resulting in a 13×13 feature map. A lower resolution can still run at decent speeds even on low end graphics cards, while high accuracy *and* high speed can be achieved on high end graphics cards (See table 2.1).

During training, YOLO 2 can also train at a random resolution for 10 batches and then pick a new random resolution. The network downsamples the input by a factor of 32, so the randomly selected input resolutions are also a factor of 32. The resolutions range from 320×320 to 608×608. This forces the network to work on a variety of different resolutions increasing its precision.

We will experiment with various input sizes in chapter 4, to give an idea of what impact this may have.

Fine-Grained Features

Improvement(s): mAP+1%
The default 13×13 feature map is pretty good at detecting larger objects, but smaller features can be more difficult to extract and utilize for detecting smaller objects. In the network the feature map starts at a higher resolution and is pooled throughout the network. A passthrough layer is used to concatenate higher resolution features in the 13×13 feature map. The high resolution is preserved by storing adjacent features in different channels. This turns the $26 \times 26 \times 512$ feature map into a $13 \times 13 \times 2048$ feature map instead. This allows the network to detect small features even on a smaller feature map.

[2] using the following distance metric: *d(box,centroid)=1-IOU(box,centroid)*

Chapter 3

Training YOLO on a custom dataset

In this chapter we will in a guide-like structure describe how to train the YOLO network with your own data. We will furthermore present a software package that includes various useful programs for training YOLO and validating the results.

3.1 Workspace setup

The first thing you have to do is to setup the YOLO workspace. YOLO uses a framework called DARKNET which is an open source neural network framework written in `C` and `CUDA`. It is fast, easy to install, and supports CPU and GPU computation [15]. You can find the source code at `https://github.com/pjreddie/darknet` or at `https://github.com/MathiasThor/YOLO-Strike`. The second link is to our fetch of DARKNET (called YOLO-Strike), which is modified to ease the setup and contains all sorts of useful scripts. One of these scripts is the `./init_workspace.sh`. It automatically edits the neural network files based on your number of classed, as well as compiling DARKNET with your desired preferences.
DARKNET can be compiled with `CUDA` and `CUDNN` if you have an NVIDIA GPU. If you do not have such a GPU then go buy one, as it is way to slow to train and detect with a CPU. DARKNET can also be compiled with `OpenCV`, which it uses when testing on images, web-cam streams and videos. We have, however, encountered some problems when using `OpenCV` while training and validating our networks and we therefore suggest to do these parts without `OpenCV` enabled.

If you wish to do the setup manually you need to follow these steps:

1. Specify compilier options in the beginning of the `Makefile` and compile DARKNET
2. Set up your `.names` file with the names of your classes
3. Set up your `.data` file with the number of classes, path to the training data set, path to test data set, path to backup directory, and path to `.names` file.
4. Set up your `.cfg` file parameters (specified later in this chapter) and change the last

occurrence of `filters` to equal $(\#\text{classes} + 5) \cdot \#\text{anchors}$

A large part of the workspace initialization is also to add your own custom data. The data consists of images and corresponding label files, all of which are explained in the following sub-sections.

Training data generation

To train any type of neural network, lots of training data is needed. Training data is data where the ground truth is already known and with YOLO, the training data needs to be accompanied by a `.txt` file containing:

$$\text{<object class> <x> <y> <height> <width>} \tag{3.1}$$

The object class is a number that corresponds to a class. The X and Y is the objects position in the image, and the height and width is the size of the bounding box. Both position and size is relative to the image. Each image used as training data must have it's own `.txt` file with the same name.

To make the process of marking the ground truth in our training data easier, we have developed our own annotation tool (called AnnoTool) that allows us to draw boxes around the object. AnnoTool was developed in Processing, a small java based sketchbook used to code within the visual arts. This was chosen as the tool is rather simple and not many features were needed. The fact that it is java based means that it will easily run on any computer regardless of operating system and specifications which means that, if needed, we could reach out to friends, family, fellow students or even people on the Internet to help us classify the images.

AnnoTool requires the folder structure shown in figure 3.1. This structure is automatically set up if you choose to use YOLO-Strike.

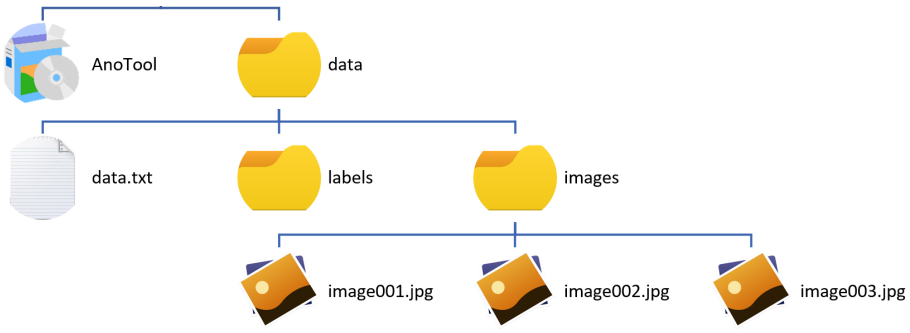


Figure 3.1 – File structure

In the `data.txt` file each line must contain the name of an image in the *images* folder (without file extension). The `data.txt` file dictates what images are opened and in what order. All images in the *images* folder must be in the `.jpg` format. After all the files listed

in `data.txt` has been annotated, the program will terminate itself. The `data.txt` can be generated with the `make_train_validation_sets.sh` script (using the `-d` flag) in the scripts folder.

Using the AnnoTool is pretty simple:

Use the mouse to draw bounding boxes around objects in the image. The most recent bounding box is the *active* bounding box, the object class of the active bounding box can be changed with the `scroll wheel`. Which bounding box is active can be changed with the `left` and `right arrow key`, the selected bounding box will be bold. Make sure the most recently drawn bounding box is active when drawing a new bounding box. Pressing `C` will remove all bounding boxes drawn in the image. Pressing `space` will save all bounding boxes in the image and load the next image. Pressing `backspace` will load the previous image (used if space is pressed accidentally). The `plus` and `minus` keys can be used to change the size of the crosshair. If some of the images are not suitable for training, pressing `D` will delete the image. Pressing `S` will remove entries from `data.txt` that has been annotated, which can be seen as saving the progress. Pressing `Q` will terminate the program.

To summarize:

- `Mouse`: Draw bounding box
- `Scroll wheel`: Select object class
- `Arrow keys`: Select bounding box
- `C`: remove all bounding boxes
- `Space`: Save bounding boxes and load next image
- `Backspace`: Load previous image
- `Plus and Minus`: Change crosshair size
- `D`: delete image
- `S`: Save progress
- `Q`: Terminate the program

In the top left corner of the screen, an index of the currently loaded image as well as an FPS counter is written. A video demonstrating the program is available at: `https://youtu.be/JlusjULGlS0`.

3.2 Train the network

Before we can start training the network, we will have to do some pre training setup, which includes: parameter settings, training set generation, and validation set generation.

3.2.1 Pre training setup

Training set, test set and validation set generation

To train a YOLO network you will need a training set. This can easily be generated with the `make_train_validation_sets.sh` script, from which you may also generate a validation set. In chapter 4 we show how we used a validation set to evaluate our trained networks.

Parameter settings

All of the user specified training parameters are located inside `yolo-obj.cfg`. The ones that we recommend changing to fit your needs are presented in listing 3.1 and explained subsequently.

Listing 3.1 – User specified training parameters

```
[net]
batch=64
subdivisions=12
height=416
width=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation=1.5
exposure=1.5
hue=.1

learning_rate=0.0001
max_batches=45000
policy=steps
steps=100,25000,35000
scales=10,.1,.1
...
[region]
anchors=1.08,1.19,  3.42,4.41,  6.63,11.38,  9.42,5.11,  16.62
    ↪ ,10.52
...
random=0
```

Batch and subdivisions are a mechanism to fit multiple input images into available GPU memory. Batch decides how many images to load at once, while subdivisions decides how many times to break the batch into smaller pieces. $\frac{\text{batch}}{\text{subdivisions}}$ thus decides how many images that are loaded and fed to the network. It is of course desirable to feed a large portion of images to the network, as it leads to a more precise gradient approximation (i.e. faster training). The training will, however, crash if the size of this portion is larger than the available GPU memory.

Height and width are used to scale the input images to the network. As discussed in chapter 2 section 2.4, there exists a trade off between the size of the input image and the speed of the trained network.

Channels can be used for training either gray scale images (channels=1) or color images

(channels=3).

Momentum and decay are as discussed in chapter 2 section 2.2.2. We do not recommend changing these, as they were found by the authors of YOLO to work well.

Angle, saturation, exposure and hue are all used for data augmentation and are more or less self-explanatory.

The learning rate is as discussed in chapter 2 section 2.2.2, where **policy, steps and scales** are used to make the learning rate adaptive. We recommend not to change this.

Max_batches specifies when the training should terminate. We recommend setting this to a large number, as you are able to stop the training yourself at any time.

Towards the end of the `.cfg` file you will find the line specifying the number of **anchors** and their sizes (discussed in chapter 2 section 2.4). The network is by default using 5 anchor boxes that is generated from the VOC data set. You can choose to use these boxes or to generate them yourself using the `anchor_tool.sh` script[1] located in the workspace script folder. The script is able generate custom made anchors that fits your training set. It takes as input the path to your training file and the number of clusters that you want to generate. The amount of clusters should reflect the amount of different shapes you have in your data set. Note that if you chose an amount of clusters different from 5, you should change the `filters` line in the `.cfg` file like specified in the bottom of section 3.1. The script will place an anchor `.txt` file in the `output_data` directory from which you will have to copy the generated anchor box dimensions into the `.cfg` file. The script can furthermore visualize your anchors as shown in figure 3.2.

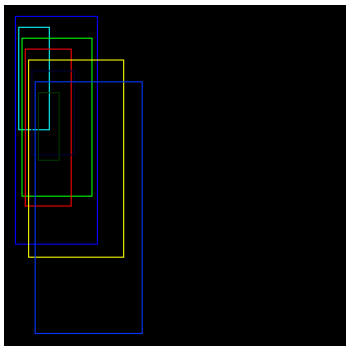


Figure 3.2 – Visualization of eight anchor boxes

Finally, at the very end of the `.cfg` file you will find the line `random=0`. By setting this equal to 1 YOLO will train with random image resolutions between 320×320 and 608×608, as discussed in chapter 2 section 2.4. We will look closer in to the input image size in chapter 4, where we experiment with various settings.

[1] Thanks to Jumabek Alikhanov from the Darknet Google Group for providing most of the code for this

3.2.2 Begin training

The training can be started by executing the following command in your root workspace folder containing the `darknet` executable.

Listing 3.2 – Command for start training

```
./darknet detector train obj.data yolo-obj.cfg < path-to-
    ↪ weights > -gpus 0,1 | tee log.txt
```

The first part of the command (`darknet detector train obj.data yolo-obj.cfg`) specifies that darknet should train a network as well as the files containing information about the network that it is about to train. These files should all be correct if the init script from the section 3.1 is used.

The "`< path-to-weights >`" can either point at pretrained weights or on one of your backup weights. These backup weights are generated for each 1000 iteration (number of batch) and are placed in the backup directory specified in `obj.data`. This makes it possible to stop and restart the training and to validate earlier stages in the training process. The pretrained weights can be downloaded from `http://pjreddie.com/media/files/darknet19_448.conv.23` or `https://pjreddie.com/darknet/imagenet/#darknet19_448` and are used when training a network for the first time.

The "`-gpus 0,1`" tells DARKNET to use two GPU's. If you only have one GPU then just delete this part of the command.

Finally, the "`| tee log.txt`" in the end of the command pipes the output from DARKNET into a log file that is needed for the validation scripts explained in section 3.3.

3.2.2.1 Training on ABACUS

It is very resource-intensive to train a YOLO network and it may therefore be preferable to use a supercomputer like ABACUS at SDU. ABACUS is a state-of-the-art solution optimized for a wide range of applications in computational science and technology. It is open to all Danish researchers and industry, and to get access you will have to apply on their website `https://abacus.deic.dk/`.

The first thing you will have to do when on your ABACUS user is to setup the workspace as explained in the previous sections. In order to get `CUDA` and `CUDNN` support you will first have to execute `module add cuda/7.5 cudnn/5.0` before running the init script (do not use `OpenCV`). When DARKNET has been compiled you are ready to train on your data. To do so you will have to submit a `SBATCH` job, as ABACUS uses Slurm for scheduling jobs. Such a job is shown in the following listing.

Listing 3.3 – Job script for ABACUS

```
#! /bin/bash
#
#SBATCH --account sdubats_gpu      # account
#SBATCH --nodes 1                  # number of nodes
#SBATCH --time 23:59:00            # max time (HH:MM:SS)

module add cuda/7.5 cudnn/5.0

./darknet detector train cs.data yolo-cs.cfg < path-to-weights
   ↪  > -gpus 0,1 | tee log.txt
```

The three `SBATCH` lines describes from what account the GPU usage should be billed, how many nodes the job should use (you will never need more than 1), and the amount of time the job should run for (24 hours is maximum).

3.3 Validate the network

We have implemented three different methods for validating your trained YOLO network, all of which are explained in the following three sub-sections. We will furthermore show how to use these measures in chapter 4.

Training Error

To plot the training error you can use the script called `plot_error.sh`. The input to the script is the training `log.txt`, that we showed how to get in the previous section. The error plot can be used as a stopping criterion for the training process. A steep slope typically indicates that the network has not finished training, where as a more flat slope indicates that the network finished and continuous training risks overfitting the network to the training data.

Average Intersection Over Union (IOU)

For the network to be good it has to have a high average IOU, as it indicates how good it is at bounding the object of interest, see figure 3.3.

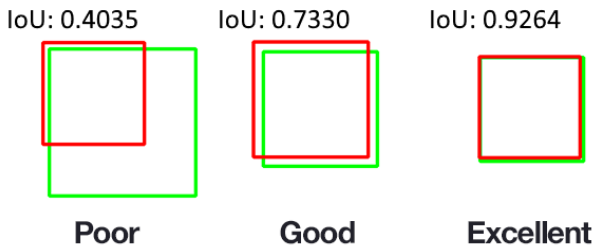


Figure 3.3 – Example of three different IOU values. The red box is the ground truth, and green box is the one proposed by the network

The average IOU for a set of weights can be calculated with the command shown in

listing 3.4. Note that this only works if you use YOLO-Strike, as we have modified the source code of `detector.c` to calculate the avg. IOU.

Listing 3.4 – Calculate average IOU

```
./darknet detector recall obj.data yolo-obj.cfg backup/yolo-
    ↪ obj_XXX.weights
```

By calculating the avg. IOU for several weights and/or parameter settings you will be able to find the most optimal setting of your network. In the YOLO 2 paper they used this method to find the best amount of clusters for their anchor boxes (see figure 3.4).

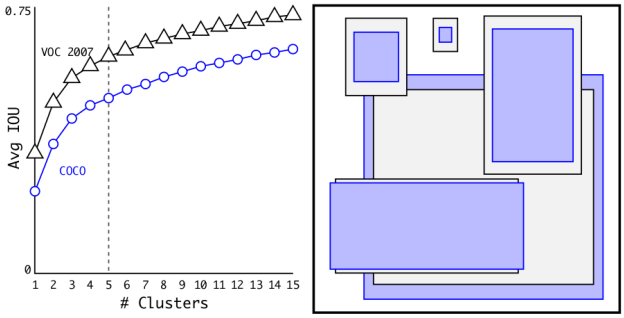


Figure 3.4 – YOLO 2 using avg. IOU to find the best amount of clusters. Left image shows the anchor boxes used for the VOC 2007 and COCO dataset. Image from [2]

Recall and Precision

Recall and precision are both interesting measures as we explained earlier. These are calculated with the same command as for the average IOU and you thus also need to use YOLO-Strike, to calculate these.

3.4 Using the network

There are several ways to use your YOLO network, all of which are listed below. At this point we advice to recompile DARKNET with `OpenCV`.

Listing 3.5 – Use YOLO on image

```
./darknet detector test obj.data yolo-obj.cfg backup/yolo-
    ↪ obj_XXX.weights <path to image>
```

Listing 3.6 – Use YOLO on webcam video stream

```
./darknet detector demo obj.data yolo-obj.cfg backup/yolo-
    ↪ obj_XXX.weights
```

Listing 3.7 – Use YOLO on video

```
./darknet detector demo obj.data yolo-obj.cfg backup/yolo-
    ↪ obj_XXX.weights <path to video>
```

Chapter 4

YOLO-STRIKE

In this chapter we will train YOLO to detect players in the computer game Counter-Strike: Global Offensive. This is to show how YOLO could be used in practice and what to be aware of when using the network.

Counter-Strike is a multiplayer first person shooter video game in which two teams of terrorists and counter-terrorists battle to, respectively, execute an act of terror (by detonating a bomb) and preventing it (by defusing the bomb). Simply eliminating the opposing team before the bomb is planted will also result in a victory.

Valve Corporation, the owner of Counter-Strike, has managed to create a big competitive community around the game, with several annual tournaments with million-dollar prize pools and over a million viewers watching live.
Like any other competitive activity, artificially enhancing one's performance is something frowned upon by most players and viewers. Getting caught cheating in a major, Valve sponsored, tournament also results in a permanent ban from these tournaments. But like other competitive activities, people still do it.
For this reason, Valve has created a small piece of software, VAC (Valve Anti Cheat), designed to detect and ban people who cheats. The VAC system looks for any third-party programs designed to give one player an advantage over another. This includes modifications to a game's core executable files and dynamic link libraries [16].

An implementation of YOLO to detect players on the screen does not need to modify the game's core executables or dynamic link libraries. This means that VAC will not be able to detect this. Once YOLO has detected the presence of a player from the opposing team, simple scripts to move the cross-hair using the coordinates of the proposed bounding boxes could be implemented to make what is known in the Counter-Strike community as an aimbot.

4.1 Method

Generating the images used to train the network is relatively straight forward. Open the game and run around taking a lot of screenshots of the different playermodels for the network. All screenshots are captured at a 1280×1024 resolution, with the in-game video settings set to high.

Counter-Strike is, along with many other games developed by Valve, built on their Source game engine. A useful feature in the source engine, is it allows users to open an in-game console to enter commands such as changing different graphical options, changing server properties (if this is allowed by the server), adding and removing bots changing or removing HUD (Heads Up Display) etc. We used the console to ensure we played with the same graphical settings to have consistency in the generated images. We also used the console to add and remove bots among various other things. All of these things could be done without a console, but a console made the task a lot easier.

During this part of the project we discovered, which in hindsight seems obvious, that the best performance of the network is achieved when the training images are near ideal images. Meaning clear visible objects with no occlusion. We thus advice you to also use close to ideal images to train your network should you attempt to make your own implementation.

After taking 4000 screenshots (2000 of terrorists and 2000 of counter-terrorists) we used our annotation tool (AnnoTool) to annotate all 4000 images (we recommend a cup of coffee and some relaxing music if you want to replicate this). After the lengthly process of annotating all the images, we were ready to start training our network.

We trained four different YOLO networks to see which one performed the best. The difference between each network is just the resolution of the input image during training. The networks used the following input resolutions[1]:

- 416×416 (416 network)
- Random resolutions between 320×320 and 608×608 (Random network)
- 736×736 (736 network)
- 992×992 (992 network)

The prior knowledge for the anchor boxes for each network is calculated with `anchor_tool.sh` for five clusters. The rest of the network parameters are set to the default ones specified in the `.cfg` file when first downloading YOLO-Strike.

The networks has finished training when the slope of the error is close to 0, we check this using our `plot_error` script. We will subsequently evaluate the performance of the network using three different validation sets. Each validation set contains images of both terrorists and counter-terrorists. The difference between the validations sets is just the distance to the object. The distances used in the validations sets are 150, 300 and 450

[1] Note that all the input sizes are multiple of 32

units. A unit in the Source game engine is roughly equivalent to one inch or 25.4 mm. Note that the distance to the player models in the training set of 4000 images is approximately 150 units.

(a) **(b)** **(c)**

Figure 4.1 – Images from the three validation sets. a) 150 units from the player model. b) 300 units from the player model. c) 450 units from the player model

Apart from the four networks mentioned above, we will also evaluate the random network when it is given 992×992 images (called the upscaled network). This is done by using the `.cfg` file of the 992 network with the `.weights` of the random network. The pure random network will otherwise use an input image size of 416×416.

We use four metrics to measure the performance of the network:

1. **Average IoU:** Intersection over Union is a measure of how accurate the predicted bounding box fits the ground truth bounding box, the higher IOU the better.
2. **Average Recall:** Recall is a measure of how many of the objects in the image are detected, the higher the recall the better.
3. **Average Precision:** Precision is a measure of how many of the detected objects are correctly detected, the higher the precision the better.
4. **FPS:** As higher input resolution will likely lead to better performance, it will be interesting to see how much an increase in resolution penalizes the evaluation time, therefore FPS is an important metric to factor into the evaluation of the network.

4.2 Results

The following figures shows the average error for the networks at different iterations (number of batch) during training.

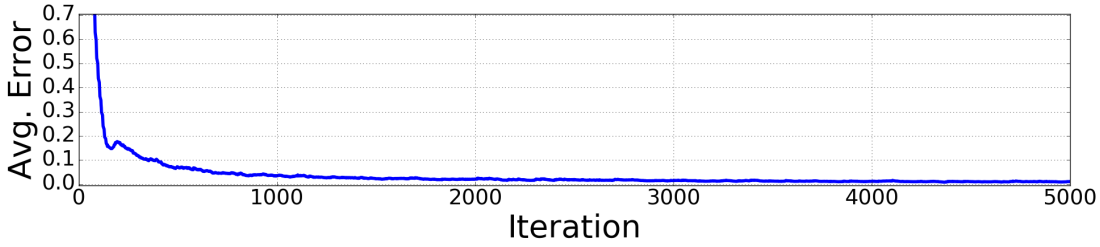


Figure 4.2 – Avg. training error for the 416×416 network

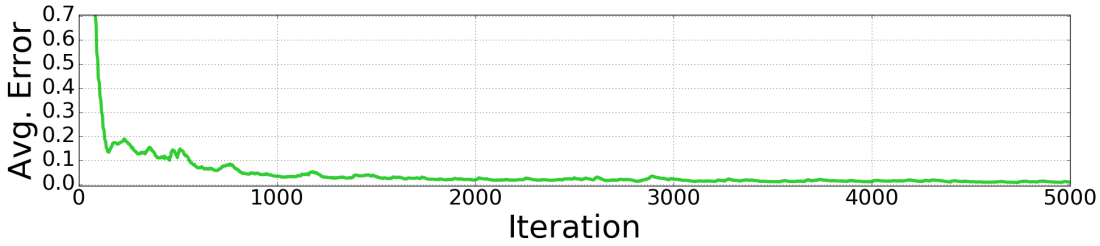


Figure 4.3 – Avg. training error for the random resolution network

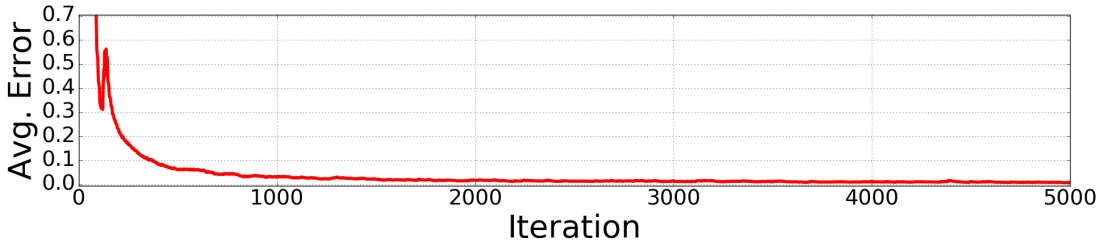


Figure 4.4 – Avg. training error for the 736×736 network

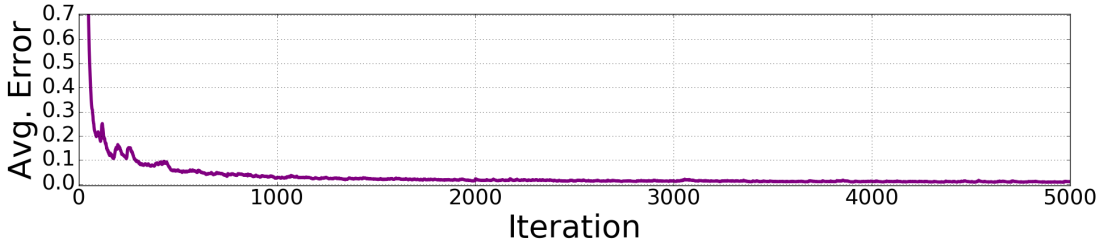


Figure 4.5 – Avg. training error for the 992×992 network

The following figures shows the avg. IoU for the networks at different iterations (number of batch) during training. Similar figures for recall and precision are shown in appendix C.

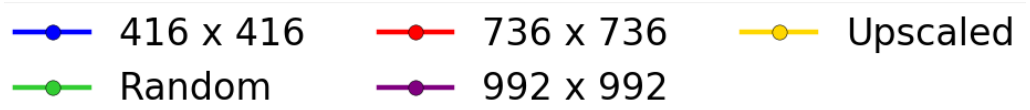
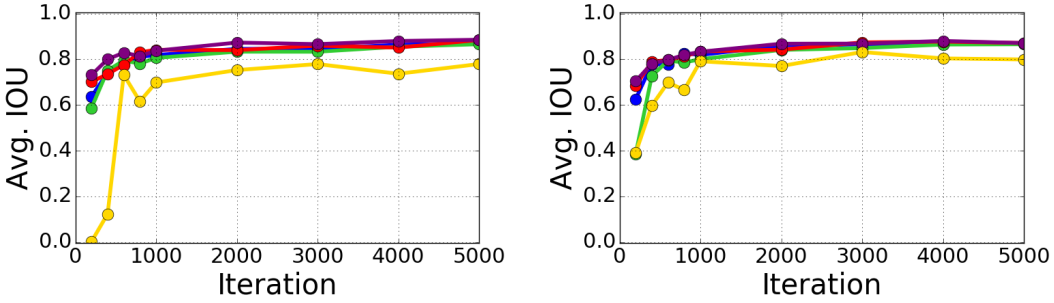
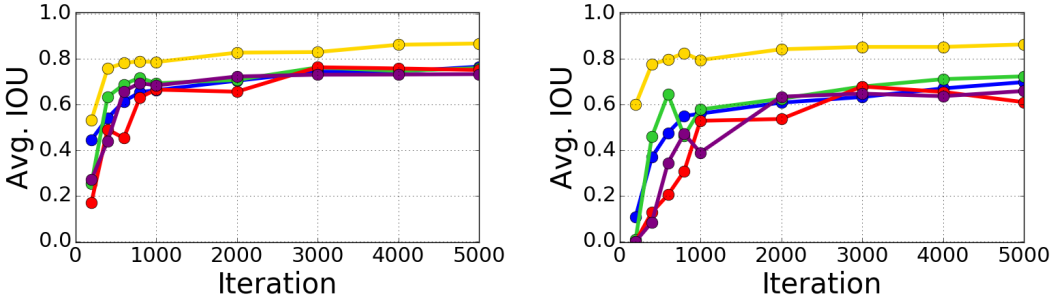


Figure 4.6 – Common plot legend



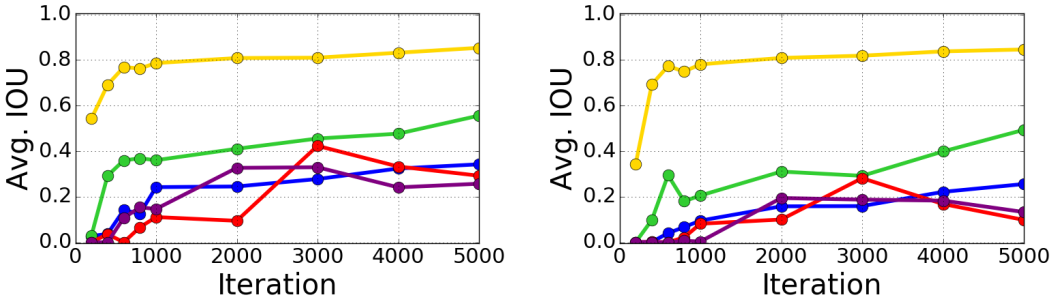
(a) counter-terrorist **(b)** terrorist

Figure 4.7 – Avg. IoU for the 150 unit validation set.



(a) counter-terrorist **(b)** terrorist

Figure 4.8 – Avg. IoU for the 300 unit validation set



(a) counter-terrorist **(b)** terrorist

Figure 4.9 – Avg. IoU for the 450 unit validation set

The following figures shows some examples from detecting on images:

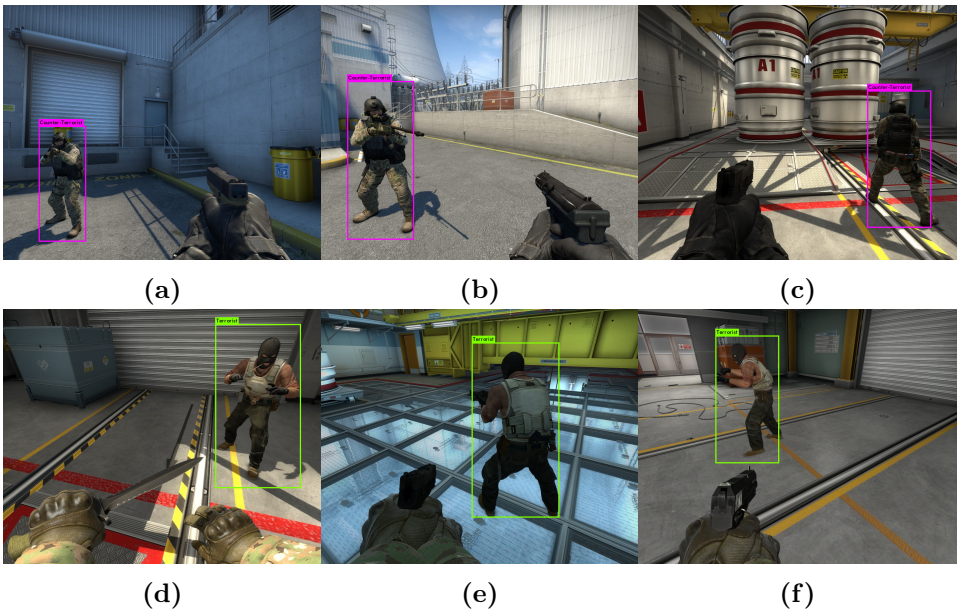


Figure 4.10 – 416 network tested on different images. (a-c) Detecting counter-terrorists (d-f) Detecting terrorists

The following links shows the networks when used on a in-game video sequences. The average FPS that each network could detect with is showed after the video links. All networks were executed on a PC with 16 GiB RAM, a 4.2 GHz Intel i7 processor with four cores (i5-7700k) and a NVIDIA GeForce GTX 970.

- **416 network**: `https://youtu.be/XlmljDd7Vho` (Avg. FPS 32.8)
- **Random network**: `https://youtu.be/3H3Fmu73gow` (Avg. FPS 31.2)
- **736 network**: `https://youtu.be/usqIicsS6i0` (Avg. FPS 16.6)
- **992 network**: `https://youtu.be/zR7x6tx2Qzk` (Avg. FPS 10.7)
- **Upscaled network**: `https://youtu.be/jqjcvi2Q3Io` (Avg. FPS 10.8)

4.3 Discussion

A general rule of thumb with YOLO when training a network is 2000 iterations per object class [4], better yet is to watch the error indicators and stop the training when the error no longer decreases. In our case we saw a stable performance after approximately 2000 iterations, we suspect this to be caused by the near ideal environment found inside a video game.

The average IoU plots shows that almost every network has similar performance at 150 and 300 units, except the upscaled one. The reason for the good performance at 150 units is presumably because this distance was the approximate distance used in the training set. The upscaled network performs slightly worse at 150 units but a lot better at 300 and 450 units. We suspect this is because the ratio between object size and grid cell size changes

less when moving the object away *and* upscaling the network at the same time (see figure 4.11). The 992 and 736 networks struggles at larger distances as these networks learned to detect large objects in a dense feature map and would thus need an even denser feature map to detect objects at large distances.

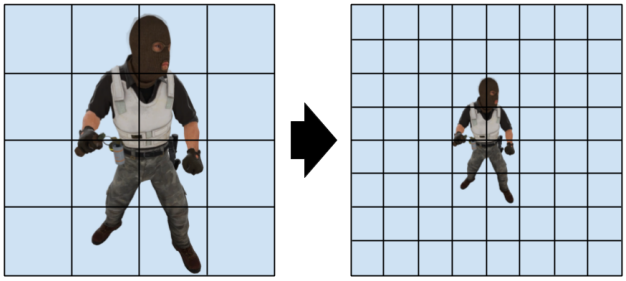


Figure 4.11 – Network upscaled from sparse feature map (small input image size) to dense feature map (large input image size)

From the precision, recall and IoU plots it is also possible to see that it is easier for the networks to detect counter-terrorist when compared to terrorists. This is most likely due to the training set that might contain better images of counter-terrorists. The precision and recall plots do not reveal much more information when compared to the IoU plots. Future experiments may therefore only include the average IoU plots.

The five videos where the networks detects players on an in-game video sequence furthermore shows the sovereignty of the upscaled network. It detect players that are close and far away as well as making a minimum of mistakes. The detection speed (avg. FPS) is as expected, where a larger input image results in a slower detection.

Future experiments could try to train with a training set that included images at all three distances to see whether the networks performance would increase. This effect is somewhat observed with the network trained on random image sizes, as it artificially changes the object size relative to the grid cells. Results from this experiment could maybe justify the use of a network with large input size, as they, for now, only slows down the training process and are slower at detecting.

4.4 Conclusion

It can be concluded that it is preferable to train a network with small or random input images, as the network appears to suffer no loss from being subsequently upscaled. It hereby offers a nice trade-off between detection speed and detection accuracy for small objects.

We do, however, suspect that a training set with different sizes of the same objects could improve the performance of networks that isn't subsequently upscaled. In this way the advantages of training with large input images might also be revealed, as they in our experiments appears to offer no advantage over small input images.

Chapter 5

Conclusion

Upon reading this report we hope that you now have extensive knowledge as to how YOLO and YOLO 2 works. Our scripts and AnnoTool should enable you to easily prepare and train a network to perform object detection of any objects you desire.

Our tests showed that a network trained with random input sizes generally performs better than networks trained with specific sizes. This may depend on the usecase, if for instance you know that the distance to the object is constant it may make sense to use a specific input size. Once a network is trained, you can always scale the input image resolution, so feel free to experiment with different input sizes to find the balance between speed and accuracy that fits your project.

Chapter 6

Bibliography

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *CVPR 2016*, 2016, pp. 779–788.

[2] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *none*, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[3] J. Redmon. (2017, 4/4) YOLO: Real-Time Object Detection. [Online]. Available: https://pjreddie.com/darknet/yolo/

[4] Google Groups. (2017, 6/4) Darknet. [Online]. Available: https://groups.google.com/forum/#!forum/darknet

[5] G. Ning. (2017, 6/4) Recurrent YOLO for object tracking. [Online]. Available: http://guanghan.info/projects/ROLO/

[6] G. Taverriti, S. Lombini, L. Seidenari, M. Bertini, and A. Del Bimbo, "Real-time Wearable Computer Vision System for Improved Museum Experience," 2016. [Online]. Available: https://www.micc.unifi.it/wp-content/uploads/2016/10/p703-taverriti.pdf

[7] Redmon, Joseph. (2016, 4/4) YOLO slides. [Online]. Available: https://docs.google.com/presentation/d/1kAa7NOamBt4calBU9iHgT8a86RRHz9Yz2oh4-GTdX6M/edit#slide=id.p

[8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[9] P. Krähenbühl, C. Doersch, J. Donahue, and T. Darrell, "Data-dependent Initializations of Convolutional Neural Networks," *International Conference on Computer Vision*, pp. 1–12, 2015. [Online]. Available: http://arxiv.org/abs/1511.06856

[10] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds. San Francisco, CA: Morgan Kaufmann, 1992, pp. 950–957. [Online]. Available: ftp://ftp.ci.tuwien.ac.at/pub/texmf/bibtex/nips-4.bib

[11] ujjwalkarn. (2016) An intuitive explanation of convolutional neural networks. [Online]. Available: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

[12] B. Rohrer. (2016) How convolutional neural networks work. [Online]. Available: https://youtu.be/FmpDIaiMIeA

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[14] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." [Online]. Available: http://proceedings.mlr.press/v37/ioffe15.pdf

[15] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[16] Valve Corporation. (2017, 13/3) Valve Anti-Cheat System (VAC). [Online]. Available: https://support.steampowered.com/kb_article.php?ref=7849-Radz-6869&l=english#notvac

[17] D. Forsyth, "Object detection with discriminatively trained part-based models," *Computer*, vol. 47, no. 2, pp. 6–7, 2014.

[18] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders, "Selective search for object recognition," *International Journal of Computer Vision*, vol. 104, no. 2, pp. 154–171, 2013. [Online]. Available: https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013

[19] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, M. Najibi, M. Rastegari, L. S. Davis, W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. Reed, "SSD: Single Shot MultiBox Detector," *arXiv preprint*, pp. 1–15, 2016. [Online]. Available: http://arxiv.org/abs/1512.02325

Appendices

Appendix A

YOLO architecture

YOLO 1		YOLO 2	
Layer type	**Size**	**Layer type**	**Size**
Convolutional	7x7x64-s2	Convolutional	3x3x32
Maxpool	2x2-s2	Maxpool	2x2-s2
Convolutional	3x3x192	Convolutional	3x3x64
Maxpool	2x2-s2	Maxpool	2x2-s2
Convolutional	1x1x128	Convolutional	3x3x128
Convolutional	3x3x256	Convolutional	1x1x64
Convolutional	1x1x256	Convolutional	3x3x128
Convolutional	3x3x512	Maxpool	2x2-s2
Maxpool	2x2-s2	Convolutional	3x3x256
Convolutional	1x1x256	Convolutional	1x1x128
Convolutional	3x3x512	Convolutional	3x3x256
Convolutional	1x1x256	Maxpool	2x2-s2
Convolutional	3x3x512	Convolutional	3x3x512
Convolutional	1x1x256	Convolutional	1x1x256
Convolutional	3x3x512	Convolutional	3x3x512
Convolutional	1x1x256	Convolutional	1x1x256
Convolutional	3x3x512	Convolutional	3x3x512
Convolutional	1x1x512	Maxpool	2x2-s2
Convolutional	3x3x1024	Convolutional	3x3x1024
Maxpool	2x2-s2	Convolutional	1x1x512
Convolutional	1x1x512	Convolutional	3x3x1024
Convolutional	3x3x1024	Convolutional	1x1x512
Convolutional	1x1x512	Convolutional	3x3x1024
Convolutional	3x3x1024	Convolutional	1x1x1000
Convolutional	3x3x1024	Average pool	Global
Convolutional	3x3x1024-s2	Softmax	
Convolutional	3x3x1024		
Convolutional	3x3x1024		
Fully connected			
Fully connected			

Table A.1 – YOLO 1 and YOLO 2 structure

Appendix B

Other detection algorithms

Previous attempts at implementing object detection and classification is significantly slower compared to YOLO because of the way they are implemented. In these implementations the classifier is repurposed to also handle detection. Generally algorithms start by extracting a set of features from the image like SIFT, HOG or similar and then using a classifier to detect objects in feature space [1].

B.1 DPM

DPM (Deformable Parts Model) represents objects as mixtures of deformable parts models. This helps classifying objects where the shape can very greatly (called intraclass variability). Objects are represented as a collection of parts in a deformable configuration. The DPM classifier is also used to detect objects using a sliding window [17]. The window slides over the image in various sizes to perform the detection. This disjoint pipeline is very time consuming and can be difficult to optimize.

B.2 R-CNN

Unlike the sliding window used in DPM, R-CNN uses a selective search [18] algorithm to generate potential bounding boxes. Features are then extracted by a convolutional neural network and an SVM is then used to score these boxes. R-CNN is a complex pipeline with many components that all must be precisely tuned and the end result is a slow system that can take up to 40 seconds to search an image.

Faster variations such as Fast R-CNN and Faster R-CNN focuses on speeding up the network by replacing the selective search algorithm with a neural network, but they are still too slow for real time applications.

B.3 SSD

SSD (Single Shot MultiBox Detector) uses a somewhat similar approach to the problem as YOLO consisting of a single end-to-end neural network. SSD eliminates the need for bounding box proposals and pixel- or feature resampling stages, instead a small convolutional filter is applied to feature maps to predict category scores and box offsets [19]. This is where the fundamental improvement in speed compared to R-CNN and DPM comes from. SSD even outperforms YOLO 1.

Appendix C

Results for Precision and Recall

The following figures shows the avg. precision and recall for the networks at different iterations (number of batch) during training.

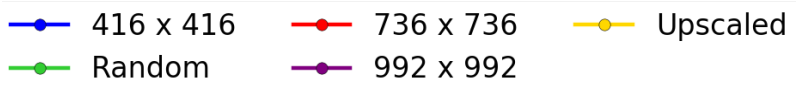
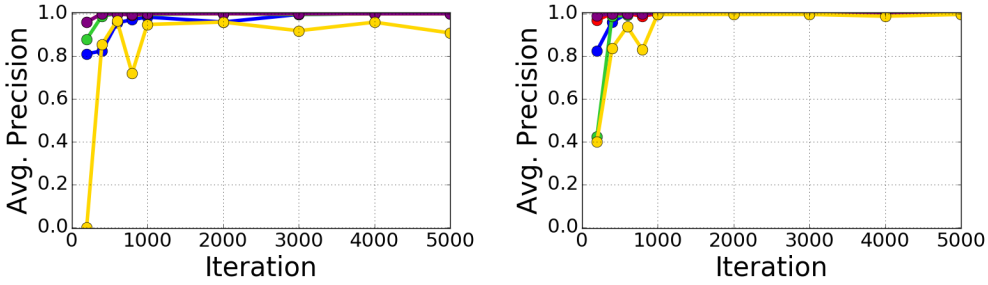
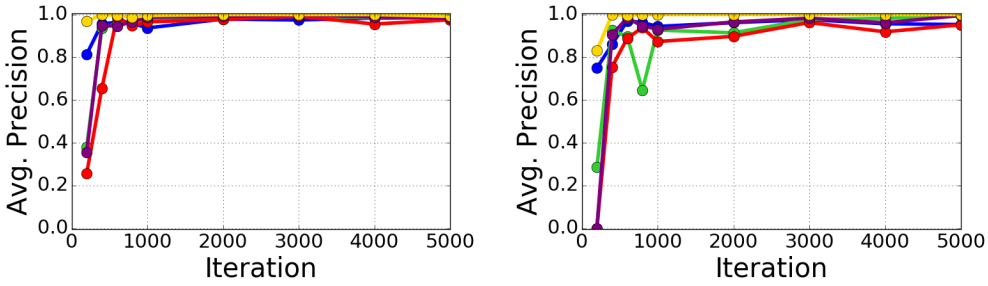


Figure C.1 – Common plot legend



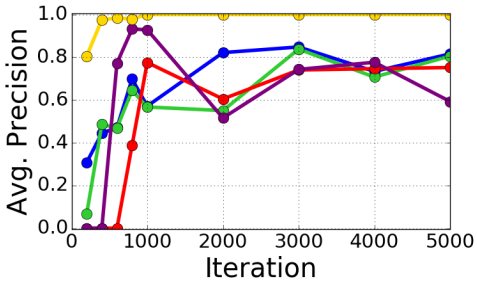
(a) counter-terrorist **(b)** terrorist

Figure C.2 – Avg. precision for the 150 unit validation set.

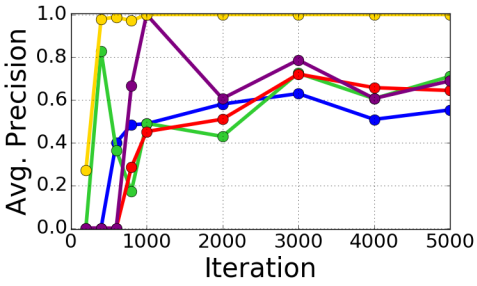


(a) counter-terrorist **(b)** terrorist

Figure C.3 – Avg. precision for the 300 unit validation set.

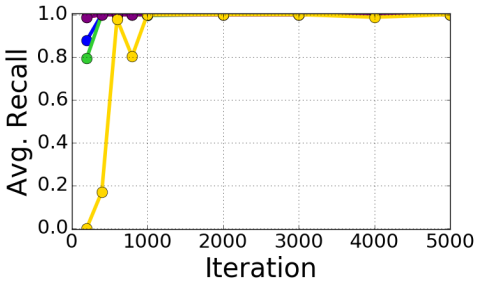


(a) counter-terrorist

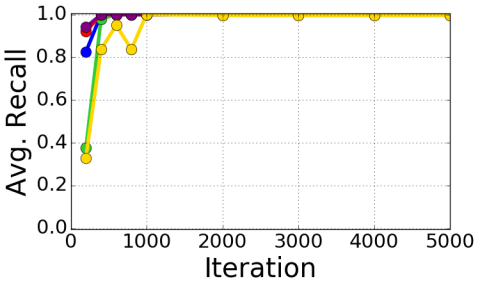


(b) terrorist

Figure C.4 – Avg. precision for the 450 unit validation set.

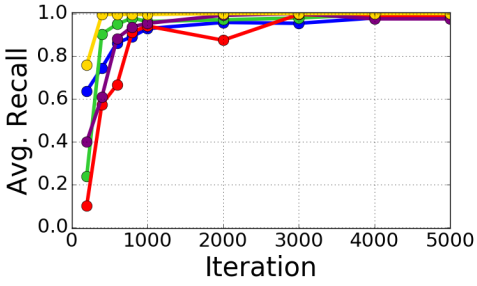


(a) counter-terrorist

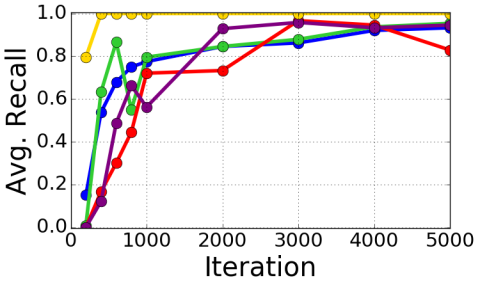


(b) terrorist

Figure C.5 – Avg. recall for the 150 unit validation set.

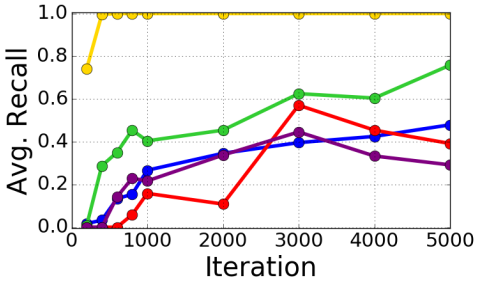


(a) counter-terrorist

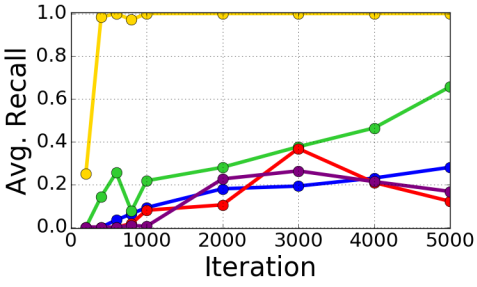


(b) terrorist

Figure C.6 – Avg. recall for the 300 unit validation set.



(a) counter-terrorist



(b) terrorist

Figure C.7 – Avg. recall for the 450 unit validation set.