



Philadelphia University
Faculty of Engineering
Department of Computer Engineering

(Firearms detection using AI)

Project (3)

By:

(المنذر محمدمهدي خالد الدهني)

201610825

(عبدالقادر يوسف حسين حرب)

201520149

Supervisor

(Dr. Qadri Hamarsheh)

**Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Engineering / Faculty of Engineering,
Philadelphia University.**

May-2021

Abstract:

Firearms are the weapons of choice for people who decide to use them for protection. Unfortunately, they are also the weapon of choice for criminals.

According to National Crime Information Center (NCIC), there were 230k reported gun thefts in the USA during 2016¹.

Fast response from police departments could save lives and leads to the arrest of criminals.

This project studies the detection of firearms through CCTV systems using Deep Learning AI and (YOLO) you look only once an algorithm based on Convolutional Neural Network (CNN).

A dataset of gun images from IEEE-dataport website will be used.

A set of convolutional neural networks can be trained to detect firearms and then alarm the owners or directly report it to the police.

Convolutional neural networks are known for efficiency in the detection and identification of objects using image processing, in some cases, it's even more accurate and consistent than humans.

This project will study a simple AI algorithm that can be implemented to any pre-existing CCTV system that is connected to a PC to detect different types of guns.

Contents

(Firearms detection using AI)	1
<i>Abstract:</i>	2
1. Introduction.....	5
2. Literature review	8
2.1. Project features	9
2.2. Potential users.....	9
3. YOLO algorithm	10
3.1 YOLO versions.....	10
3.2 YOLOv3 implementations	11
3.3 YOLOv3 architecture	11
4. Dataset	16
4.1 Dataset requirements.....	16
4.2 Labeling	16
4.4 Images pre-processing	18
5. YOLO Training	18
5.1 Pre-trained model	18
5.2 Splitting dataset	18
5.3 Training parameters	19
5.4 Google Colab.....	23
5.5 Training process	23
5.6 When to stop training.....	27
6. Implementation	29
6.1 Implementation workflow	29
6.2 Dataset and labeling.....	29
6.3 Network Training Model	31
6.4 Evaluation and Testing	34
6.5 Live video detection	36
6.6 Results.....	39
7. Conclusions and Future work	39
References	40

List of figures

Figure 1 Crimes for which cctv recordings were useful in the investigation	5
figure 2 Main purposes of image processing	6
figure 3 Object detection models: speed comparison.....	7
figure 4 Object detection models: accuracy comparison.....	7
figure 5 Training time of with visdrone2018 on geforce rtx 2080 ti gpu.	7
figure 6 GSL system.....	8
figure 7 Project features.....	9
figure 8 mAP of different algorithms performed on kitti datasets.	10
figure 9 Comparison of mandelbrot set execution time	11
figure 10 Darknet 53 architecture	12
figure 11 Bounding boxes prediction example	14
figure 12 Computing to extract class probability.....	14
figure 13 Intersection over union	15
figure 14 Non max supression	16
figure 15 Training workflow.....	24
figure 16 Loss plotted against the batch number	27
figure 17 Project workflow	29
figure 18 Dataset images	29
figure 19 Dataset labels	30
figure 20 Supervisely tool.....	30
figure 21 Google colab initializing.....	31
figure 22 Make file.....	32
figure 23 Training parameters.....	32
figure 24 Weights file	33
figure 25 Manual labeled dataset loss	34
figure 26 Auto labeled dataset loss	34
figure27 mAP for manual labeled dataset	35
figure 28 mAP for auto labeled dataset	35

1. Introduction

In recent years the usage of Surveillance systems such as closed-circuit television (CCTV) has been more common.

The researches show the effectiveness of CCTV systems in crime detection, according to the British railway network between 2011 and 2015. CCTV was available to investigators in 45% of cases and judged to be useful in 29% (65% of cases in which it was available)².

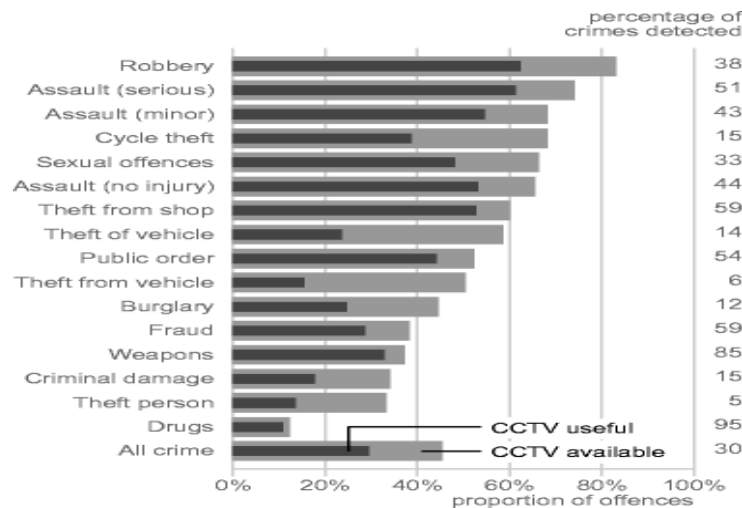


Figure 1 Crimes for which CCTV recordings were useful in the investigation¹

The figure shows a study on the percentage of crimes detected using CCTV systems.

But how to improve CCTVs to detect more crimes? Especially with crimes that include Firearms.

Machines can be taught to deal with images the same way our brains do and to analyze images much more deeply than we can.

When applied to image processing, Deep learning Artificial Intelligence (AI) can power authentication functionality to ensure security in public places, by detecting and recognizing objects in images and videos.

Deep learning AI can learn and extract features without human supervision, drawing from data that is either unstructured or labeled.

Image processing is manipulating an image to enhance it or extract information from it.

Today, image processing combined with AI is used in many fields such as face recognition, self-driving vehicles, surveillance, law enforcement, and many others.

Some of the main purposes of image processing are:

- Visualization: Represent processed data, giving visual form to objects that aren't visible.
- Image sharpening and restoration : Improve the quality of processed images
- Object measurement: Measure objects in an image
- Pattern recognition: **detect and classify objects** in an image, identify their positions.

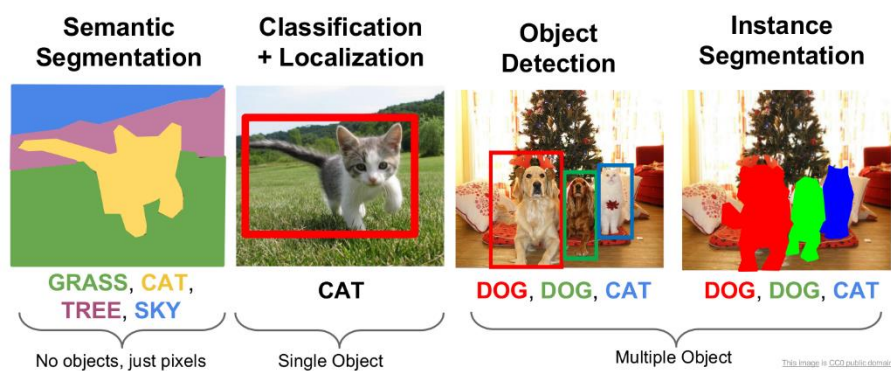


Figure 2 main purposes of image processing

This project studies the object detection part of image processing, using YOLO (You Only Look Once) algorithm.

YOLO algorithm is an algorithm based on CNN, it uses regression by predicting classes and bounding boxes for the whole image in one run of the Algorithm.

Although there are many other types of object detection models such as:

- Region-based Convolutional Neural Networks(R-CNN)
- Single Shot MultiBox Detector (SSD)
- Retina-Net

But all of these models fail at the most important criteria in this project which is Speed for real-time object detection.

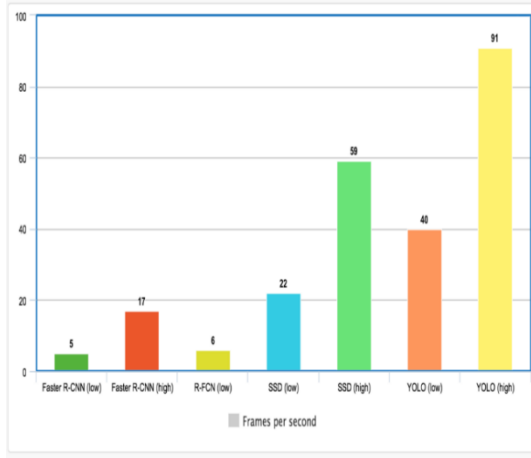


Figure 3 Object detection models: speed comparison

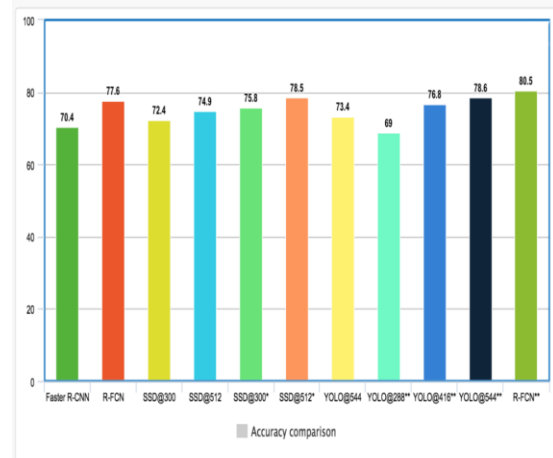


Figure 4 Object detection models: accuracy comparison³

Another major factor is training time, YOLO algorithm has one of the lowest training time (4.7 hours) between other object detection models.

Architecture	Training Time (hours)
Faster R-CNN	8.20
RFCN	4.30
SNIPER	62.50
RetinaNet	10.61
CenterNet	5.3
YOLOv3	4.70
SSD	251.18

Figure 5 Training time of with VisDrone2018 on GeForce RTX 2080 Ti GPU.⁴

2. Literature review

With rapid technology development in the last years, companies started to use AI to produce security products that are more efficient and easy to implement.

V5systems Software Company in the US developed V5 Gun Shot locator (GSL).



Figure 6 GSL system

The system uses sound and video to detect guns and gunshots.

It's marketed as the World's First Wireless, Rapidly Deployable Gunshot Detection and Gunshot Location Solution. And some of the main features are:

- Artificial intelligent computing.
- Self-Powered using solar power
- Over 100 meters of localizable acoustic range
- Web-based user interface, android, and, IOS app
- Real-time acoustic alert/clips via email and text on any smart device
- Real-time video alert/clips via email and text on any smart device when deployed On V5 Portable Units with camera sensor
- Wireless Communications:
 - Outdoor Wi-Fi
 - Cellular
 - Radio frequency (RF)

But as the company's website describes: The current outdoor gunshot detection systems have an average cost of +/- \$250,000 per year, per square mile of coverage (not including trenching and deployment costs)⁵.

There are some other similar company solutions such as Shot Spotter Inc. and Safety Dynamics Inc. most of them have the same idea and the system is very expensive.

2.1. Project features

- Fully trained smart AI system to detect various types of guns.
- Fast real-time gun detection.
- The software will be compatible with any pre-existing CCTV system (no need to update the old camera system).
- Self-alarm SMS for the owner or call 911 in case of gun detection.

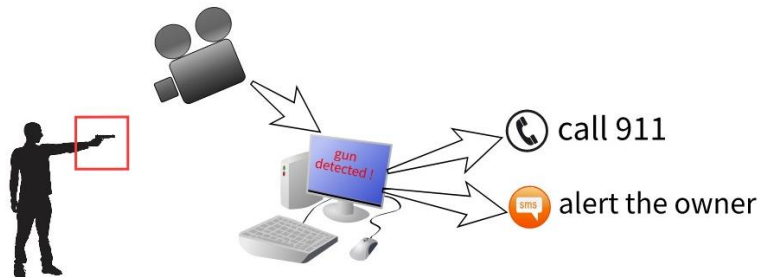


Figure 7 project features

2.2. Potential users

This project studies a simple AI algorithm that can be implemented to any pre-existing CCTV system that is connected to a PC.

- Easy, cheap and user-friendly security software for firearm detection.
- This software can be used by big or small business owners to keep their lives and customers live safe such as:

1. Hospitals.
 2. Banks.
 3. Malls.
 4. Money exchange companies.
 5. Schools.
 6. Gas stations.
- Etc.

3. YOLO algorithm

3.1 YOLO versions

YOLOv1 was designed first, it was fast but it suffered from Bad performance when there are groups of small objects because each grid can only detect one object.

The Main error of YOLOv1 is localization, because the ratio of bounding box is learned from data and YOLOv1 makes error on the unusual ratio bounding box.

In order to solve the 2 problems mentioned before and improve the performance, the authors make several modifications in YOLO V2.

Adding Batch Normalization helps to improve YOLOv2 mAP (mean Average Precision) by 2%.

YOLO V2 is trained on high resolution images for first 10 epochs. This improves the mAP by 4%

Direct location prediction: The values of box center coordinates, x and y, are between 0 and 1. But YOLO V1 can predict the values smaller than 0 or bigger than 1. The authors add logistic activation to force the prediction fall into this range⁶.

Compared to YOLO V1, YOLO V2 becomes better, faster and stronger. The authors of YOLO have tried many technics to improve the accuracy and performance.

Algorithm	mAP (%)
Fast RCNN [25]	49.87
Faster RCNN [26]	58.78
Sliding window & CNN [23]	68.98
SSD [28]	75.73
Context & RCNN [45]	79.26
Yolo v1 ($S \times S$) [27]	72.21
T-S Yolo v1 ($S \times 2S$)	74.67
Yolo v2 ($S \times S$) [29]	81.64
T-S Yolo v2 ($S \times 2S$)	83.16
Yolo v3 ($S \times S$) [31]	87.42
T-S Yolo v3 ($S \times 2S$)	88.39

Figure 8 mAP of different algorithms performed on KITTI datasets⁷.

YOLOv3 architecture is used for this project.

3.2 YOLOv3 implementations

There are a few different implementations of the YOLOv3 algorithm on the web.

Darknet is an open source neural network framework.

It was written in the C Language and CUDA technology (Compute Unified Device Architecture), which makes it really fast and provides for making computations on a GPU, which is essential for real-time predictions⁸.

There are some other implementations of the algorithm that is written in other languages such YOLO Darkflow that is written in python, however C language is much faster since it uses CUDA technology and according to the benchmark that is shown in figure 8 below.

<u>mandelbrot</u>								
source	secs	mem	gz	busy	cpu load			
<u>Python 3</u>	263.87	48,268	688	1,054.07	100%	100%	100%	100%
<u>C gcc</u>	1.64	27,024	1135	6.53	99%	99%	100%	100%

Figure 9 Comparison of Mandelbrot set execution time⁹

For this project Darknet implementation of YOLOv3 algorithm will be used.

3.3 YOLOv3 architecture

YOLO v3 is a classical object detection algorithm based on darknet-53 CNN architecture proposed by Joseph Redmon in 2018. It is currently a marketable object detection algorithm. Most importantly, it has ultra-fast detected speed than SSD, but almost as accurate as faster-RCNN.

YOLO makes use of only convolutional layers, making it a fully convolutional network (FCN). In YOLO v3 paper, the authors present new, deeper architecture of feature extractor called Darknet-53.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 10 Darknet 53 architecture

As its name suggests, it contains 53 convolutional layers, each followed by batch normalization layer and Leaky ReLU activation.

Average pooling is used, and a convolutional layer with stride 2 is used to down sample the feature maps. This helps in preventing the loss of low-level features often attributed to pooling.

YOLO is invariant to the size of the input image. However, for this project we will stick to a constant input size due to various problems that might occur when we are implementing the algorithm.

YOLOv3 algorithm considers object detection as a regression problem. It directly predicts class probabilities and bounding box offsets from full images with a single feed-forward convolution neural network.

YOLOv3 method divides the input image into $S \times S$ small grid cells. If the center of an object falls into a grid cell, the grid cell is responsible for detecting the object. Each grid cell predicts the position information of B bounding boxes and computes the objectness scores corresponding to these bounding boxes.

Each objectness score can be obtained as follows:

$$C_i^j = P_{i,j}(\text{Object}) * IOU_{\text{pred}}^{\text{truth}} \quad (1)$$

Where C_i^j the objectness score of the j the bounding box in the I the grid cell.

$P_{i,j}$ (Object) is a function of the object.

The $IOU_{\text{pred}}^{\text{truth}}$ represents the intersection over union (IOU) between the predicted box and ground truth box.

The position of each bounding box is based on four predictions: tx, ty, tw, th , on the assumption that (cx, cy) are the offset of the grid cell from the top left corner of the image. The center position of final predicted bounding boxes is offset from the top left corner of the image by (bx, by) . Those are computed as follows:

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \end{aligned} \quad (2)$$

Where $\sigma()$ is a sigmoid function.

The width and height of the predicted bounding box are Calculated like this:

$$\begin{aligned} b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \end{aligned} \quad (3)$$

Where p_w, p_h are the width and height of the bounding box prior.

The ground truth box consists of four parameters (gx, gy, gw and gh), which correspond to the predicted parameters bx, by, tw and th , respectively. Based on (2) and (3), the truth values of $\hat{tx}, \hat{ty}, \hat{tw}$ and \hat{th} can be obtained as follows:

$$\begin{aligned} \sigma(\hat{t}_x) &= g_x - c_x \\ \sigma(\hat{t}_y) &= g_y - c_y \\ w &= \log(g_w/p_w) \\ \hat{t}_h &= \log(g_h/p_h) \end{aligned} \quad (4)$$

The network down samples the image by a factor called the stride of the network. For example, if the stride of the network is 32, then an input image of size 416 x 416 will yield an output of size 13 x 13. Generally, stride of any layer in the network is equal to the factor by which the output of the layer is smaller than the input image to the network.

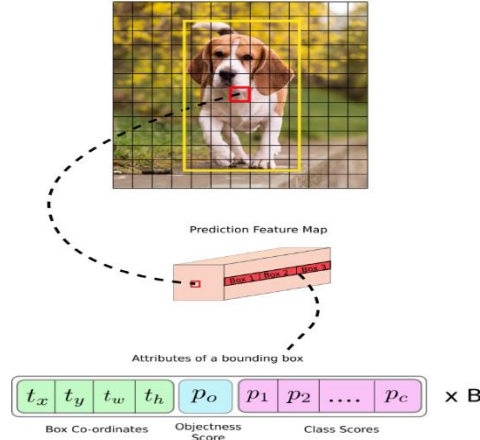
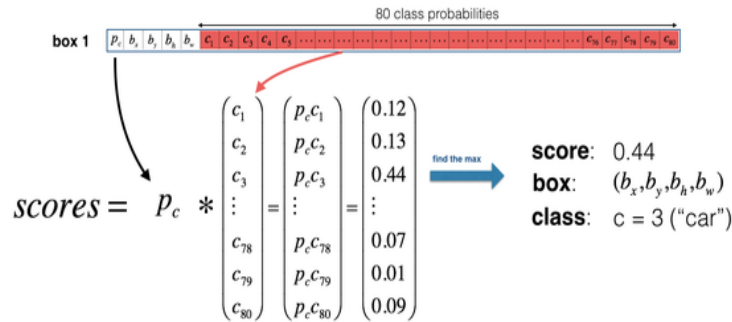


Figure 11 bounding boxes prediction example



the box (b_x, b_y, b_h, b_w) has detected $c = 3$ ("car") with probability score: 0.44

Figure 12 computing to extract class probability

For an image of size 416 x 416, YOLO predicts $((52 \times 52) + (26 \times 26) + (13 \times 13)) \times 3 = 10647$ bounding boxes. However, in case of the example image, there's only one object, a dog. So the network needs to reduce the detections from 10647 to 1. First, it filters boxes based on their objectness score. Generally, boxes having scores below a threshold (for example below 0.5) are ignored.

Next, Non-maximum Suppression (NMS) intends to cure the problem of multiple detections of the same image. For example, all the 3 bounding boxes of the red grid cell may detect a box or the adjacent cells may detect the same object, so NMS is used to remove multiple detections.

But Even after filtering by thresholding over the classes scores, the network still end up a lot of overlapping boxes.

A second filter for selecting the right boxes is called NMS. NMS uses a function called "Intersection over Union", or IoU.

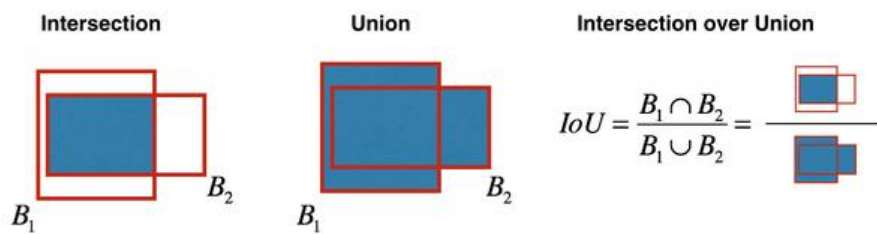


Figure 13 Intersection over Union

To calculate the area of a rectangle the network multiply its height ($y_2 - y_1$) by its width ($x_2 - x_1$)

Then it find the coordinates (xi_1, yi_1, xi_2, yi_2) of the intersection of two boxes:

xi_1 = maximum of the x_1 coordinates of the two boxes

yi_1 = maximum of the y_1 coordinates of the two boxes

xi_2 = minimum of the x_2 coordinates of the two boxes

yi_2 = minimum of the y_2 coordinates of the two boxes

```
xi1 = max(box1[0], box2[0])
yi1 = max(box1[1], box2[1])
xi2 = min(box1[2], box2[2])
yi2 = min(box1[3], box2[3])
inter_area = (xi2 - xi1)*(yi2 - yi1)

# Formula: Union(A,B) = A + B - Inter(A,B)
box1_area = (box1[3] - box1[1])*(box1[2]- box1[0])
box2_area = (box2[3] - box2[1])*(box2[2]- box2[0])
union_area = (box1_area + box2_area) - inter_area

# compute the IoU
IoU = inter_area / union_area
```



Figure 14 non max supression

4. Dataset

4.1 Dataset requirements

To train a custom model, a labelled dataset is needed.

Labelled datasets in object detection are: images with corresponding bounding box coordinates and labels. That is, the bottom left and top right (x, y) coordinates + the class.

The number of images is not the most thing considered here, instead, it is more important to properly understand in which scenarios the model will be used.

For example in a traffic sign detection model that will run in a car, it is important to use images taken under different weather, lighting and camera conditions in their appropriate context.

So if the model does not have enough data to learn general patterns, it won't perform well in production.

4.2 Labeling

Labeled data means the data is marked up, or annotated, to show the target, which is the answer you want your model to predict¹⁰.

During the labeling process, data features or properties get highlighted, later to be analyzed for patterns that help predict the target.

According to researches, 80% of AI project time is spent on gathering, organizing, and labeling data¹¹.

There are two main approaches for data labeling:

1. Manual labeling: this includes going through all images in the dataset and assign to each one a label with class (in this case a gun) and its coordinates.

This might be a long process, but it is accurate and it is still used. Today there are many companies that provide data labeling services for customers and big projects.

2. Auto labeling: or data programming, consists of writing labeling functions scripts that programmatically segment data, however the results of labeled images can be less accurate than those created by manual labeling.

Note that there is nothing such as fully automatic labeling but these tools helps to make the labeling process faster such as “Grab-Cut” and “Supervisely”.

We will be using both approaches for this project and compare the results.

Annotations are saved as TXT files. the format used by ImageNet. Besides, it also supports YOLO format.

Each row entry in a label file represents a single bounding box in the image and contains the following information about the box:

```
<object-class-id> <center-x> <center-y> <width> <height>
```

The first field object-class-id is an integer representing the class of the object. It ranges from 0 to (number of classes – 1).

In this case, it is only one class of a gun, so it is always set to 0.

The second and third entry, center-x and center-y are respectively the x and y coordinates of the center of the bounding box, normalized (divided) by the image width and height respectively.

The fourth and fifth entry, width and height are respectively the width and height of the bounding box, again normalized (divided) by the image width and height respectively.

4.4 Images pre-processing

As explained before the input image size needs to be 416x416, however there is no need to resize dataset images. YOLO architecture does it by itself keeping the aspect ratio safe (no information will miss) according to the resolution in configuration file.

For Example, if we have image size 1248 x 936, YOLO will resize it to 416 x 312 and then pad the extra space with black bars to fit into 416 x 416 network.

5. YOLO Training

5.1 Pre-trained model

When training a new object detector, it is a good idea to leverage existing models trained on very large datasets even though the large dataset may not contain the object you are trying to detect. This process is called **transfer learning**.

Instead of learning from scratch, we use a pre-trained model that contains convolutional weights trained on ImageNet. Using these weights as our starting weights, our network can learn faster.¹²

5.2 Splitting dataset

Any Deep learning training procedure involves first splitting the data randomly into two sets:

Training set: This is the part of the data on which we train the model. Depending on the amount of data, data can randomly be selected between 70% to 90% of the data for training.

Test set: This is the part of the data on which the model is tested. Typically, this is 10-30% of the data.

No image should be part of both the training and the test set.¹³

5.3 Training parameters

Along with the `darknet.data` and `classes.names` files, YOLO also needs a configuration file `darknet-yolov3.cfg`.

The training parameters are stored in the configuration file.

- Batch

```
[net]
# Training
batch=64
subdivisions=16
```

The batch parameter indicates the batch size used during training. Our training set contains a few thousands of images, but it is not uncommon to train on millions of images.

The training process involves iteratively updating the weights of the neural network based on how many mistakes it is making on the training dataset. It is impractical (and unnecessary) to use all images in the training set at once to update the weights. So, a small subset of images is used in one iteration, and this subset is called the batch size.

When the batch size is set to 64, it means 64 images are used in one iteration to update the parameters of the neural network.¹⁴

[batch size] is typically chosen between 1 and a few hundreds, e.g. [batch size] = 32 is a good default value¹⁵.

- Subdivisions

Even though we may want to use a batch size of 64 for training your neural network, we may not have a GPU with enough memory to use a batch size of 64.

Darknet allows specifying a variable called subdivisions that lets us process a fraction of the batch size at one time on the GPU.

The GPU will process (batch/subdivisions) number of images at any time, but the full batch or iteration would be complete only after all the 64 (as set above) images are processed.¹⁶

- Width, Height, Channels

```
width=416  
height=416  
channels=3
```

These configuration parameters specify the input image size and the number of channels.

The input training images are first resized to width x height before training.

The default values for YOLO are 416×416. The results might improve if we increase it to 608×608, but it would take longer to train too.

Channels=3 indicates that we would be processing 3-channel RGB input images.

- Momentum and Decay

```
momentum=0.9  
decay=0.0005
```

These parameters control how the weights are updated.

As explained before the weights of a neural network are updated based on a small batch of images and not the entire dataset. Because of this reason, the weight updates fluctuate quite a bit. Momentum is used to penalize large weight changes between iterations¹⁷.

A typical neural network has millions of weights and therefore they can easily over fit any training data.

Over fitting means it will do very well on training data and poorly on test data¹⁸. It is almost like the neural network has memorized the answer to all images in the training set, but really not learned the underlying concept. One of the ways to mitigate this problem is to penalize large value for weights.

The parameter decay controls this penalty term. The default value works just fine, but we can tweak this if we notice over fitting.

- Learning Rate, Steps, Scales, Burn In

```
learning_rate=0.001
policy=steps
steps=3800
scales=.1
burn_in=400
```

The parameter learning rate controls how fast we should learn based on the current batch of data. Typically this is a number between 0.01 and 0.0001.

The beginning of the training process starts with zero information so the learning rate needs to be high. But as the neural network sees a lot of data, the weights need to change less aggressively.

In other words, the learning rate needs to be decreased over time. In the configuration file, this decrease in learning rate is accomplished by first specifying that our learning rate decreasing policy is steps.

In our case, the learning rate will start from 0.001 and remain constant for 3800 iterations, and then it will multiply by scales to get the new learning rate¹⁹.

As explained above, the learning rate needs to be high in the beginning and low later on. While that statement is largely true, it has been empirically found that the training speed tends to increase if we have a lower learning rate for a short period of time at the very beginning. This is controlled by the burn-in parameter²⁰.

Sometimes this burn-in period is also called warm-up period.

- Data augmentation

```
angle=0  
saturation = 1.5  
exposure = 1.5  
hue=.1
```

To make maximum use of our dataset it is possible to create new data from the existing dataset.

This process is called data augmentation. For example, an image of the weapon rotated by 5 degrees is still an image of a weapon, but it can help in better results for training.

The angle parameter in the configuration file allows to randomly rotate the given image by \pm angle.

Similarly, transforming the colors of the entire picture can be done using saturation, exposure, and hue.

The values above are the default values for training.

- Number of iterations

It is important to specify how many iterations the training process should be run for.

```
max_batches=2000
```

For multi-class object detectors, the max_batches number is higher, i.e. we need to run for more batches (e.g. in yolov3.cfg).

For an n-classes object detector, it is advisable to run the training for at least $2000 \cdot n$ batches. In our case with only 1 class, 2000 seemed like a safe number for max_batches²¹.

5.4 Google Colab

Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs²².

Why?

Google colab provides computing power of the Google servers instead of our own machine which is slow and could lose all data if the connection disconnects during the training process.

Because running python scripts requires often a lot of computing power and can take time.

Also by running scripts in the cloud it is possible to automatically save checkpoints of your generated weight files so that we can continue from that point and also test which weight file gives the best results.

The modified data earlier will be uploaded to Google colab to be trained.

5.5 Training process

During training, convolutional neural networks learn features from the image dataset.

In the earlier layers, the model learns to detect simple patterns such as edges and curves. However, as we move deeper into the network, it begins to learn more high level features with complex patterns or textures, perhaps patterns like faces.

Neural networks themselves are differentiable with respect to their inputs, making feature visualization an optimization problem. To discover what parts of the input image helped the network produce certain predictions, through activating certain neurons, we can use backpropagation on a trained network with fixed weights learned during training.

The training workflow goes like this:

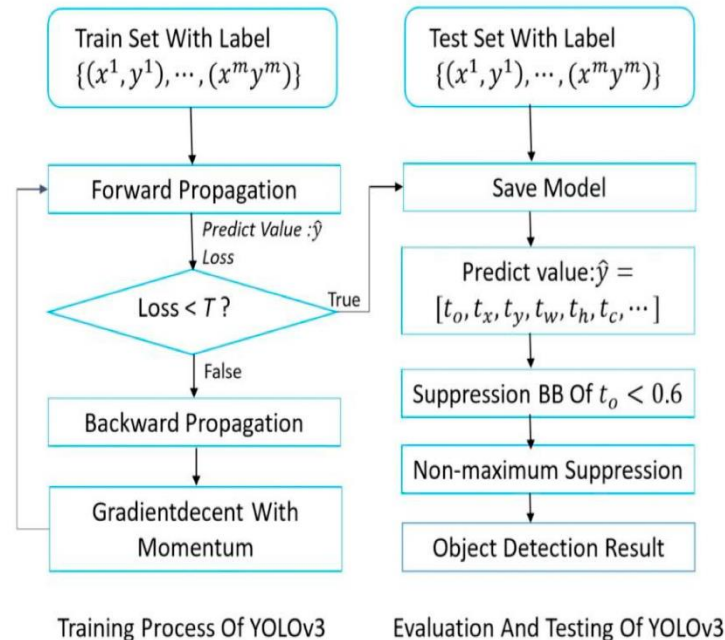


Figure 15 training workflow

• Loss function

YOLO predicts multiple bounding boxes per grid cell. To compute the loss for the true positive, it selects the one with the highest IoU (intersection over union) with the ground truth. This strategy leads to specialization among the bounding box predictions. Each prediction gets better at predicting certain sizes and aspect ratios.

YOLO uses sum-squared error between the predictions and the ground truth to calculate loss. The loss function composes of:

- The classification loss.
- The localization loss (errors between the predicted boundary box and the ground truth).
- The confidence loss (the objectness of the box) ²³.

$$\begin{aligned}
& \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
& + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \quad (5) \\
& + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \\
& + \sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

- **Backpropagation algorithm**

The algorithm is used to effectively train a neural network through a method called chain rule. In simple terms, after each forward pass through a network, backpropagation performs a backward pass while adjusting the model's parameters.

Backpropagation is an algorithm of artificial neural networks using gradient descent. Given an artificial neural network and an error function, the method calculates the gradient of the error function with respect to the neural network's weights. It is a generalization of the delta rule for perceptron to multilayer feed-forward neural networks.

The "backwards" part of the name stems from the fact that calculation of the gradient proceeds backwards through the network, with the gradient of the final layer of weights being calculated first and the gradient of the first layer of weights being calculated last.

Partial computations of the gradient from one layer are reused in the computation of the gradient for the previous layer.

This backwards flow of the error information allows for efficient computation of the gradient at each layer versus the naive approach of calculating the gradient of each layer separately²⁴.

For the partial derivatives:

$$\frac{\partial E_d}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} \quad (6)$$

For the hidden layers' error terms,

$$\delta_j^k = g'(a_j^k) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1} \quad (7)$$

For the final layer's error term:

$$\delta_1^m = g'_o(a_1^m)(\hat{y}_d - y_d) \quad (8)$$

For combining the partial derivatives for each input-output pair:

$$\frac{\partial E(X, \theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^N \frac{\partial}{\partial w_{ij}^k} \left(\frac{1}{2} (\hat{y}_d - y_d)^2 \right) = \frac{1}{N} \sum_{d=1}^N \frac{\partial E_d}{\partial w_{ij}^k} \quad (9)$$

For updating the weights:

$$\Delta w_{ij}^k = -\alpha \frac{\partial E(X, \theta)}{\partial w_{ij}^k} \quad (10)$$

5.6 When to stop training

As the training goes on, the log file contains the loss in each batch. It is possible to stop training after the loss has reached below some threshold. (figure 22) below is an example of the loss plotted against the batch number.

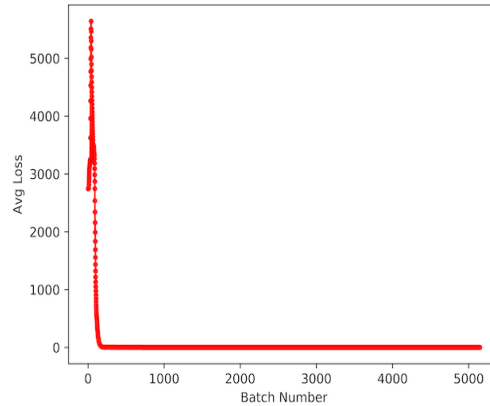


Figure 16 loss plotted against the batch number

But the actual test should be seeing the mean Average Precision (mAP) using the learned weights.

For object detection, we use the concept of Intersection over Union (IoU). IoU measures the overlap between 2 boundaries. We use that to measure how much our predicted boundary overlaps with the ground truth (the real object boundary).

- If $\text{IoU} \geq 0.5$, classify the object detection as True Positive(TP);
- If $\text{IoU} < 0.5$, then it is a wrong detection and classifies it as False Positive(FP);
- When ground truth is present in the image and the model failed to detect the object, we classify it as False Negative(FN);
- True Negative (TN): TN is every part of the image where we did not predict an object. This metrics is not useful for object detection, hence we ignore TN.

$$\begin{aligned} \text{Precision} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \\ \text{Recall} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \end{aligned} \tag{11}$$

Precision measures how accurate is your predictions. i.e. the percentage of your predictions are correct.

Recall measures the model's ability to detect Positive samples. The higher the recall, the more positive samples detected.

$$AP = \sum_{k=0}^{k=n-1} [\text{Recalls}(k) - \text{Recalls}(k + 1)] * \text{Precisions}(k) \quad (12)$$

$$\text{Recalls}(n) = 0, \text{Precisions}(n) = 1$$

n = Number of thresholds.

The **average precision** (AP) is a way to summarize the precision-recall curve into a single value representing the average of all precisions.

$$mAP = \frac{1}{n} \sum_{k=1}^{k=n} AP_k \quad (13)$$

AP_k = the AP of class k

n = the number of classes

Along with the loss and mAP, we should test our weights file on new data and see the results visually to make sure we are happy with the results.

6. Implementation

6.1 Implementation workflow

First, the dataset will be collected, then it will be labeled and splitted into training set and testing set, after that we will start training the model to predict guns.

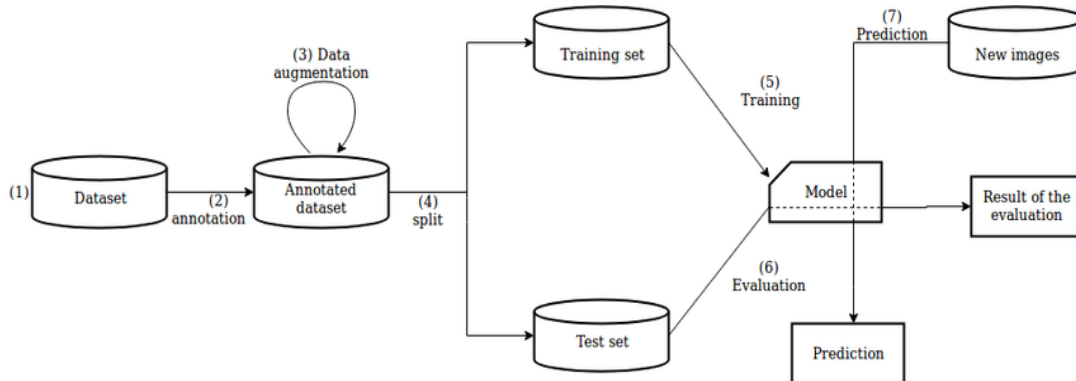


Figure 17 project workflow

6.2 Dataset and labeling

As explained before we will be using a pre-labeled dataset from IEEE-dataport with 3000 images.

The data looks like this

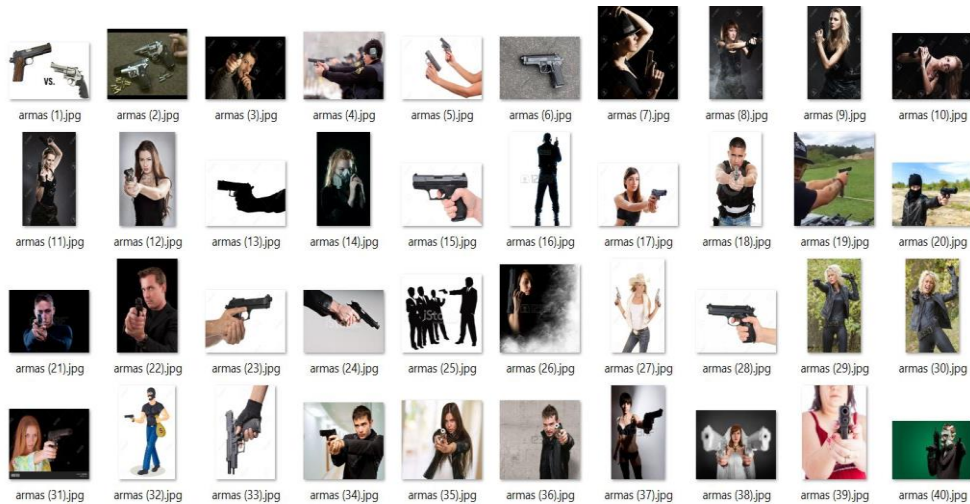


Figure 18 dataset images

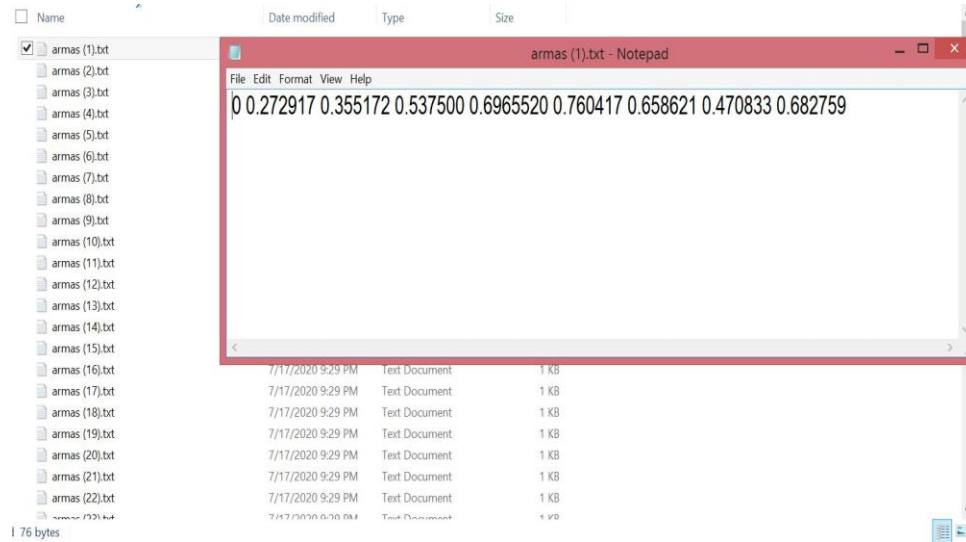


Figure 19 dataset labels

The same images are labeled using auto labeling tool “Supervisely” for comparison purposes.

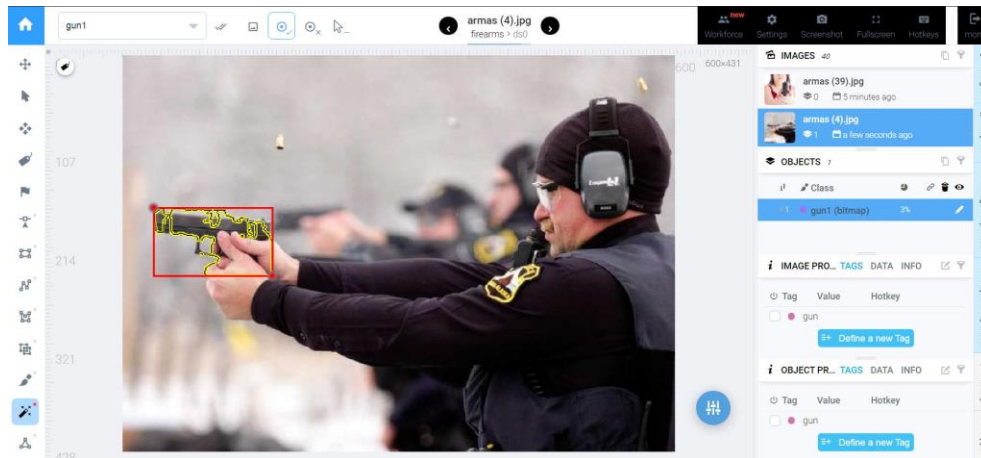


Figure 20 Supervisely tool

After this step we ended up with 2 separate datasets 3000 images each. These 2 data set will be trained separately later on.

But before that the images are split into training set (80%) and testing set (20%) using simple python script

```
data = os.listdir(image_directory)

from sklearn.model_selection import train_test_split
train, valid = train_test_split(data, test_size=0.2, random_state=1)
```

6.3 Network Training Model

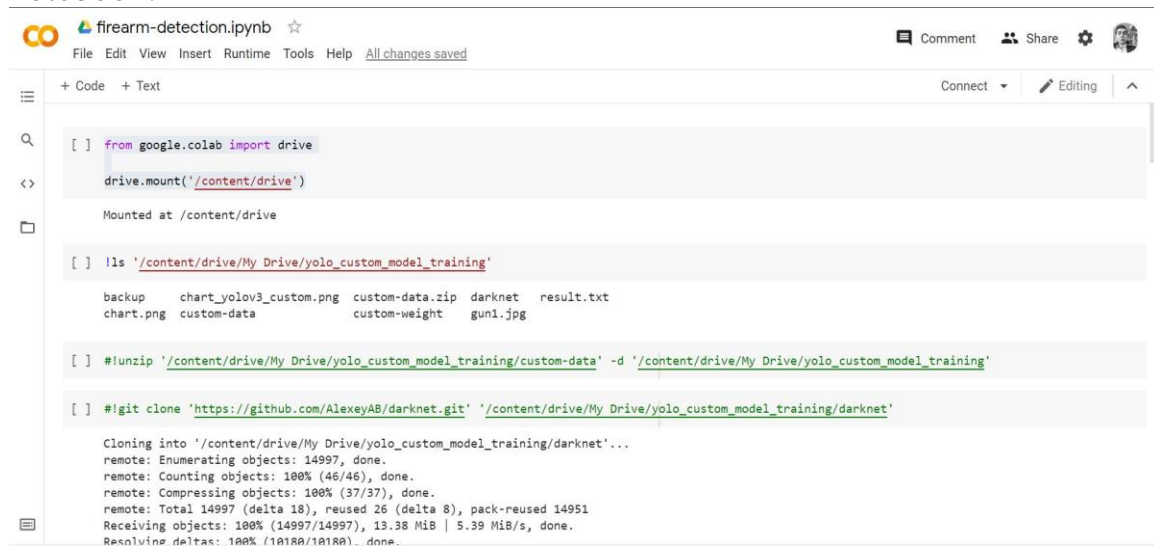
As explained before we will be using darknet53 network, code is provided in appendix.

Tensorflow framework work is used to construct the network with help of other libraries such as openCV and keras.

The network is uploaded into Google Colab with the training set and testing set to start the training online using Google Colab free GPU service.

First the labeled dataset with classes is zipped and uploaded into a file to Google drive, then a new Google Colab notebook is created and connected to Google drive.

Then the dataset is unzipped and darknet53 model is cloned into the Colab notebook.



```
[ ] from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

[ ] ls '/content/drive/My Drive/yolo_custom_model_training'

backup  chart_yolov3_custom.png  custom-data.zip  darknet  result.txt
chart.png  custom-data  custom-weight  gun1.jpg

[ ] !unzip '/content/drive/My Drive/yolo_custom_model_training/custom-data' -d '/content/drive/My Drive/yolo_custom_model_training'

[ ] !git clone 'https://github.com/AlexeyAB/darknet.git' '/content/drive/My Drive/yolo_custom_model_training/darknet'

Cloning into '/content/drive/My Drive/yolo_custom_model_training/darknet'...
remote: Enumerating objects: 14997, done.
remote: Counting objects: 100% (46/46), done.
remote: Compressing objects: 100% (37/37), done.
remote: Total 14997 (delta 18), reused 26 (delta 8), pack-reused 14951
Receiving objects: 100% (14997/14997), 13.38 MiB | 5.39 MiB/s, done.
Resolving deltas: 100% (18188/18188), done.
```

Figure 21 google colab initializing

After that we modified the make file in Darknet to support GPU usage, Compute Unified Device Architecture (CUDA), and CUDA Deep Neural Network Library (cuDNN). To enhance system performance²⁵.

```
GPU=1
CUDNN=1
CUDNN_HALF=0
OPENCV=1
AVX=0
OPENMP=0
LIBSO=0
ZED_CAMERA=0
ZED_CAMERA_v2_8=0
```

Figure 22 Make File

Back to Google Colab notebook the make file is built using the command (!make).

As explained before a pertained model will be used to initialize the training process (Transfer learning).

A darknet53.conv.74 file is a weights file with 80 pertained classes provided by YoloV3 developers will be uploaded into the drive to be used. This will help our network to learn faster.

The last step before starting the training is setting up training parameters In the configuration file.

```
batch=64
subdivisions=16
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
learning_rate=0.001
burn_in=1000
max_batches = 2000
policy=steps
steps=1800,2200
scales=.1,.1
```

Figure 23 training parameters

The parameters were selected on recommendation of YOLO creators.

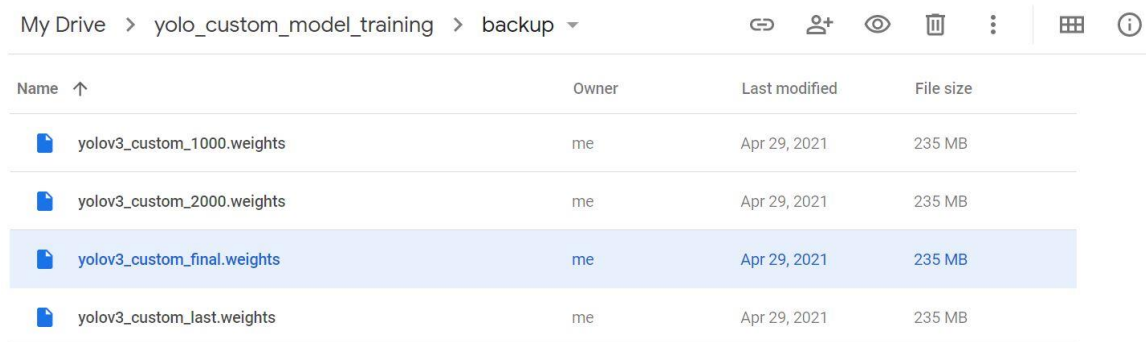
Now the model is ready for training the process can be started using the Following code line:

```
!darknet/darknet detector train custom-data/labelled_data.data  
darknet/cfg/yolov3_custom.cfg custom-weight/darknet53.conv.74
```

Note: Same method with same parameters and settings is exactly used for both manual and auto labeled datasets.

The manual labeled dataset finished training after about 5:30 hours while the auto labeled data set took about 6:00 hours.

We ended up with 2 trained weights files one for each dataset.



The screenshot shows a Google Drive interface with the path 'My Drive > yolo_custom_model_training > backup'. It displays a table of four files, all owned by 'me' and last modified on 'Apr 29, 2021', each with a size of '235 MB'. The file 'yolov3_custom_final.weights' is highlighted in blue.

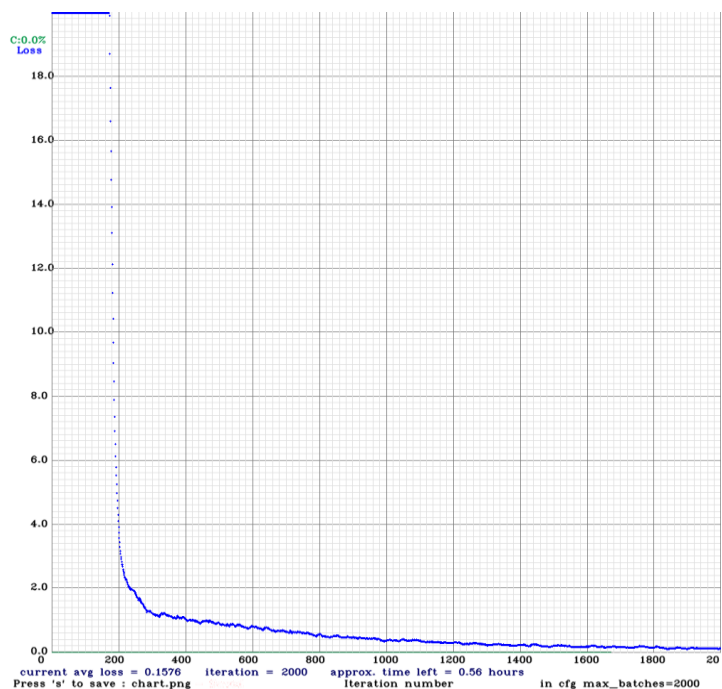
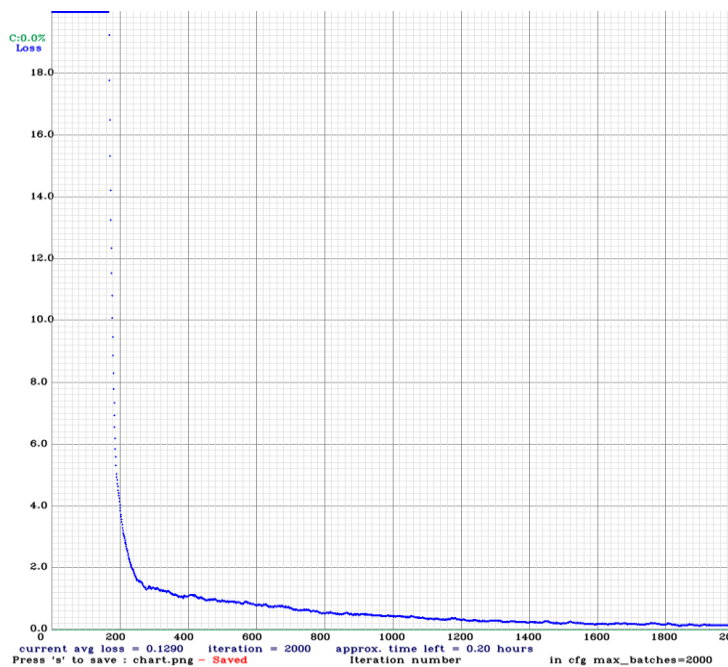
My Drive > yolo_custom_model_training > backup										
Name	Owner	Last modified	File size							
yolov3_custom_1000.weights	me	Apr 29, 2021	235 MB							
yolov3_custom_2000.weights	me	Apr 29, 2021	235 MB							
yolov3_custom_final.weights	me	Apr 29, 2021	235 MB							
yolov3_custom_last.weights	me	Apr 29, 2021	235 MB							

Figure 24 Weights file

6.4 Evaluation and Testing

As explained before the loss function for YOLO is sum of loss functions for bounding box, objectness score and class predictions.

By implementing (Equation 5) in python and plotting the output after each iteration.



The manual labeled dataset loss was 0.1290 after 2000 iterations, and the auto labeled dataset loss was 0.1576 after same number of iterations.

They are both acceptable values, this gives us an idea that our model is learning and minimizing the error overtime.

During the training the loss kept decreasing continuously which tells us that the model did well and didn't suffer from overfitting.

However the real test here should be testing the model accuracy, this can be done using mean average precision (mAP).

The mAP for both models was calculated using (ml_metrics) library in python.

```
class_id = 0, name = gun, ap = 81.97% (TP = 40, FP = 7)

for conf_thresh = 0.25, precision = 0.85, recall = 0.80, F1-score = 0.82
for conf_thresh = 0.25, TP = 40, FP = 7, FN = 10, average IoU = 63.72 %

IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
mean average precision (mAP@0.50) = 0.819656, or 81.97 %
```

Figure27 mAP for manual labeled dataset

```
class_id = 0, name = gun, ap = 73.07% (TP = 39, FP = 5)

for conf_thresh = 0.25, precision = 0.89, recall = 0.71, F1-score = 0.79
for conf_thresh = 0.25, TP = 39, FP = 5, FN = 16, average IoU = 67.38 %

IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
mean average precision (mAP@0.50) = 0.730747, or 73.07 %
```

Figure 28 mAP for auto labeled dataset

The mAP for the manual labeled dataset was 81.97% and it was 73.07% for the auto labeled dataset.

Both loss and mAP are tested and calculated using our provided test dataset, but the main purpose of the model is the real life application.

So next step is testing it with new data live video, to make sure we are satisfied with the results.

6.5 Live video detection

After preparing the custom dataset and training it for custom detection using darknet53 model, we ended up with a weights file contains all the trained weights.

Now for live detection of (guns) we need to read this weights file and inserts new data –live video in our case- to check our model capability to perform in real life.

First step is passing Yolo weights to the network along with the configuration file (yolo.cfg) that contains information about the network.

```
net = cv2.dnn.readNet(r"C:\Users\Monzer\Desktop\project 3\yolov3_4000.weights",  
                    r"C:\Users\Monzer\Desktop\project 3\yolov3_testing.cfg")
```

Then we extract the object names (classes) to a list, in our case one class of a gun. And start the video capturing and reading the video frame by frame.

```
classes = ["gun"]  
cap = cv2.VideoCapture(0)  
_, img = cap.read()
```

After a frame is read from the input video stream, it is passed through (blobFromImage) function to convert it to an input blob for the neural network.

A blob is a 4D numpy array object (images, channels, width, and height). It has the following parameters:

- the image to transform
- the scale factor (1/255 to scale the pixel values to [0..1])
- the size, here a 416x416 square image
- the mean value (default=0)
- the option swapRB=True (since OpenCV uses BGR)

```
blob = cv.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)
```

The output blob is then passed in to the network as its input and a forward pass is run to get a list of predicted bounding boxes as the network's output.

```
net.setInput(blob)
```

The forward function in OpenCV's Net class needs the ending layer till which it should run in the network. Since we want to run through the whole network, we need to identify the last layer of the network. We do that by using the function `getUnconnectedOutLayers()` that gives the names of the unconnected output layers, which are essentially the last layers of the network.

Then we run the forward pass of the network to get output from the output.

```
output_layers_name = net.getUnconnectedOutLayersNames()

layerOutputs = net.forward(output_layers_name)

boxes = []
confidences = []
class_ids = []
```

Next we scan through all the bounding boxes output from the network and keep only the ones with high confidence scores. Assign the box's class label as the class with the highest score.

```
for output in layerOutputs:
    for detection in output:
        score = detection[5:]
        class_id = np.argmax(score)
        confidence = score[class_id]
        if confidence > 0.7:
            center_x = int(detection[0] * width)
            center_y = int(detection[1] * hight)
            w = int(detection[2] * width)
            h = int(detection[3] * hight)
            x = int(center_x - w / 2)
            y = int(center_y - h / 2)
            boxes.append([x, y, w, h])
            confidences.append((float(confidence)))
            class_ids.append(class_id)

indexes = cv2.dnn.NMSBoxes(boxes, confidences, .5, .4)
```

Finally, we draw the boxes that were filtered through the non maximum suppression, on the input frame with their assigned class label and confidence scores.

```
font = cv2.FONT_HERSHEY_PLAIN
colors = np.random.uniform(0, 255, size=(len(boxes), 3))
if len(indexes) > 0:
    for i in indexes.flatten():
        x, y, w, h = boxes[i]
        label = str(classes[class_ids[i]])
        confidence = str(round(confidences[i], 2))
        color = colors[i]
        cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
        cv2.putText(img, label + " " + confidence, (x, y + 400), font, 2, color, 2)

cv2.imshow('img', img)
if cv2.waitKey(1) == ord('q'):
    break

cap.release()
cv2.destroyAllWindows()
```

For making a phone call in case of a gun detection we can use one of many python API for making phone calls like twilio.

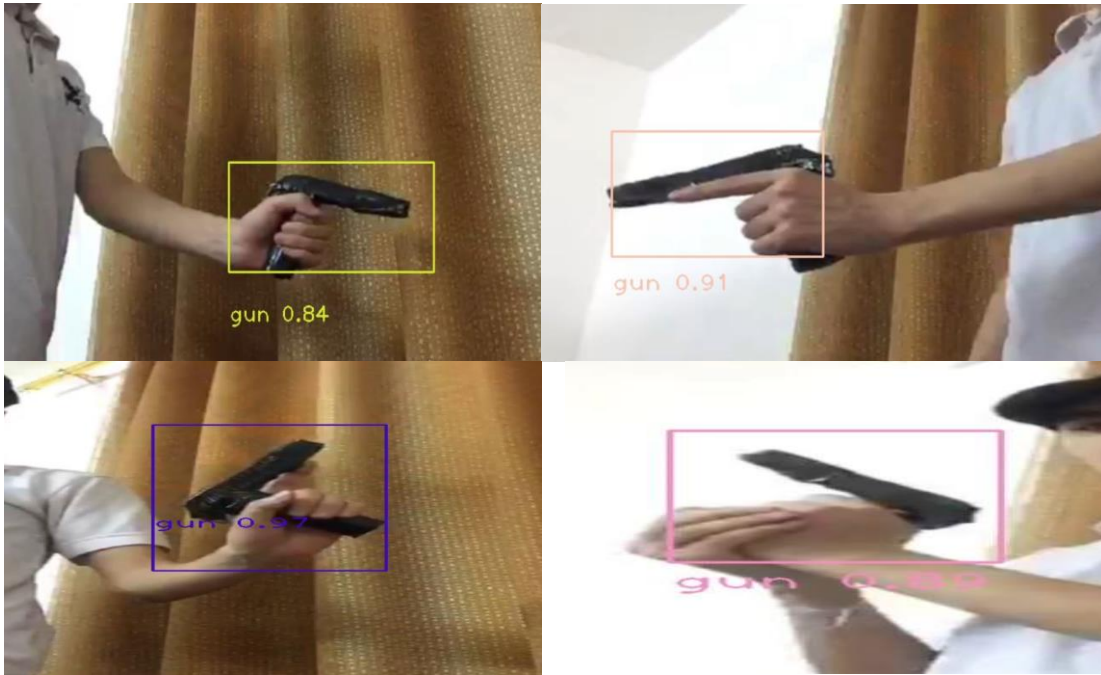
```
import os
from twilio.rest import Client

account_sid = os.environ['TWILIO_ACCOUNT_SID']
auth_token = os.environ['TWILIO_AUTH_TOKEN']
client = Client(account_sid, auth_token)

call = client.calls.create(
    twiml='<Response><Say>Ahoy,
World!</Say></Response>',
    to='+14155551212',
    from_='+15017122661'
)

print(call.sid)
```

6.6 Results



7. Conclusions and Future work

In this project, we considered the problem of detecting of weapons within video frames.

We suggested a solution based on artificial intelligence techniques and the principle of deep learning.

Instead of designing a full system that contains a special computer, specific security cameras, and hard insulation/maintaining process that can be very expensive, a simple trained algorithm based software can be run on any pre-existing security system.

We also compared manual and auto labeling techniques in identical conditions and find proved that manual labeling is more accurate. However the auto labeling still gives acceptable results when the accuracy is not important in the systems.

For future developments, we can improve our dataset to make it larger or even more specific to cover only types of guns in the country that the system is installed in.

We could also add new classes to detect more objects such as knives.

Using AI in safety and critical situations raises a lot of questions about the accuracy of the detections, our aim was to develop an effective system that may be used to help in detecting threatening situations in civil society and to help more in preventing loss of lives.

References

-
- [1] Brian Freskos. (Nov 20, 2017) Missing Pieces Gun theft from legal owners is on the rise, quietly fueling violent crime across America.
 - [2] Matthew P. J. Ashby. (21 April 2017) The Value of CCTV Surveillance Cameras as an Investigative Tool: An Empirical Analysis.
 - [3] Jifeng Dai. (21 Jun 2016) R-FCN: Object Detection via Region-based Fully Convolutional Networks.
 - [4] Khang Nguyen. (Mar 2020) Detecting Objects from Space: An Evaluation of Deep-Learning Modern Approaches.
 - [5] v5systems <https://v5systems.us/products/v5psu-ptz/>
 - [6] Joseph Redmon. (25 Dec 2016) YOLO9000: Better, Faster, Stronger.
 - [7] Xiang Zhang. (6 December 2018) A Fast Learning Method for Accurate and Robust Lane Detection Using Two-Stage Feature Extraction with YOLO v3.
 - [8] Redmon, J. (2013–2016). Darknet: Open source neural networks in c.

[9]<https://benchmarksgameteam.pages.debian.net/benchmarksgame/performance/mandelbrot.html>

[10] Tijl De Bie (22-24 April 2009) Machine Learning with Labeled and Unlabeled Data.

[11] Cognilytica. (Jan. 31, 2019) Data Engineering, Preparation, and Labeling for AI.

[12] Jason Brownlee on (May 15, 2019) Transfer Learning in Keras with Computer Vision Models.

[13] Jason Brownlee on (July 24, 2020) Train-Test Split for Evaluating Machine Learning Algorithms.

[14] Jason Brownlee on (January 21, 2019) Control the Stability of Training Neural Networks With the Batch Size.

[15]Yoshua Bengio. Version 2, (Sept. 16th, 2012) Practical Recommendations for Gradient-Based Training of Deep Architectures.

[16] Ceren Gulra Melek. (July 2019) Object Detection in Shelf Images with YOLO.

[17] John Chen. (October 8, 2020) Demon: Momentum Decay For Improved Neural Network Training.

[18] Jason Brownlee. on (November 11, 2020) How to Identify Overfitting Machine Learning Models in Scikit-Learn.

[19] Jason Brownlee. on (January 23, 2019) Configure the Learning Rate When Training Deep Learning Neural Networks.

[20] Oliver Struckmeier. (28 May 2019) Improving object detector performance and flexibility through automatically generated training data and domain randomization.

[21] GaneshRaj V. (Jul 29, 2020) Face Mask Detection using Yolo V3 .

[22] Google research

<https://research.google.com/colaboratory/faq.html#:~:text=Colaboratory%2C%20or%20%E2%80%9CColab%E2%80%9D%20for,learning%2C%20data%20analysis%20and%20education.>

[23] Danyang Cao. (11 April 2020) An improved object detection algorithm based on multi-scaled and deformable convolutional neural networks.

[24] R. Rojas. (1996) Neural Networks, Springer-Verlag, Berlin.

[²⁵] Luis Barba-Guaman. (2020) Deep Learning Framework for Vehicle and Pedestrian Detection in Rural Roads on an Embedded GPU.

Appendices

Darknet53 network model:

```
import tensorflow as tf
import numpy as np
from tf import Sequential
from tf import Conv2D
from tf import AveragePooling2D
from tf import Flatten
from tf import Dense

#darknet model
tf.reset_default_graph()
inputX = tf.reshape(x, shape=[-1, 256,256 , 1])
conv1 = tf.layers.conv2d(inputX, 32, 3, activation=tf.nn.relu)
conv2 = tf.layers.conv2d(conv1, 64, 3, activation=tf.nn.relu)
pool1 = tf.layers.AveragePooling2D(conv2, 3, 2)
#block1
conv3 = tf.layers.conv2d(pool1, 32, 1, activation=tf.nn.relu)
conv4 = tf.layers.conv2d(conv3, 64, 3, activation=tf.nn.relu)
conv5 = tf.layers.conv2d(conv4, 128, 3, activation=tf.nn.relu)
pool2 = tf.layers.AveragePooling2D(conv5, 3, 2)
#block2
conv6 = tf.layers.conv2d(pool2, 64, 1, activation=tf.nn.relu)
conv7 = tf.layers.conv2d(conv6, 128, 3, activation=tf.nn.relu)
conv8 = tf.layers.conv2d(conv7, 64, 1, activation=tf.nn.relu)
conv9 = tf.layers.conv2d(conv8, 128, 3, activation=tf.nn.relu)
conv10 = tf.layers.conv2d(conv9, 256, 3, activation=tf.nn.relu)
pool3 = tf.layers.AveragePooling2D(conv10, 3, 2)
#block3
conv11 = tf.layers.conv2d(pool3, 128, 1, activation=tf.nn.relu)
conv12 = tf.layers.conv2d(conv11, 256, 3, activation=tf.nn.relu)
conv13 = tf.layers.conv2d(conv12, 128, 1, activation=tf.nn.relu)
conv14 = tf.layers.conv2d(conv13, 256, 3, activation=tf.nn.relu)
conv15 = tf.layers.conv2d(conv14, 128, 1, activation=tf.nn.relu)
conv16 = tf.layers.conv2d(conv15, 256, 3, activation=tf.nn.relu)
conv17 = tf.layers.conv2d(conv16, 128, 1, activation=tf.nn.relu)
conv18 = tf.layers.conv2d(conv17, 256, 3, activation=tf.nn.relu)
conv19 = tf.layers.conv2d(conv18, 128, 1, activation=tf.nn.relu)
conv20 = tf.layers.conv2d(conv19, 256, 3, activation=tf.nn.relu)
conv21 = tf.layers.conv2d(conv20, 128, 1, activation=tf.nn.relu)
conv22 = tf.layers.conv2d(conv21, 256, 3, activation=tf.nn.relu)
conv23 = tf.layers.conv2d(conv22, 128, 1, activation=tf.nn.relu)
conv24 = tf.layers.conv2d(conv23, 256, 3, activation=tf.nn.relu)
conv25 = tf.layers.conv2d(conv24, 128, 1, activation=tf.nn.relu)
conv26 = tf.layers.conv2d(conv25, 256, 3, activation=tf.nn.relu)
conv27 = tf.layers.conv2d(conv26, 512, 3, activation=tf.nn.relu)
pool4 = tf.layers.AveragePooling2D(conv27, 3, 2)
#block4
conv28 = tf.layers.conv2d(pool4, 256, 1, activation=tf.nn.relu)
conv29 = tf.layers.conv2d(conv28, 512, 3, activation=tf.nn.relu)
conv30 = tf.layers.conv2d(conv29, 256, 1, activation=tf.nn.relu)
conv31 = tf.layers.conv2d(conv30, 512, 3, activation=tf.nn.relu)
```

```

conv32 = tf.layers.conv2d(conv31, 256, 1, activation=tf.nn.relu)
conv33 = tf.layers.conv2d(conv32, 512, 3, activation=tf.nn.relu)
conv34 = tf.layers.conv2d(conv33, 256, 1, activation=tf.nn.relu)
conv35 = tf.layers.conv2d(conv34, 512, 3, activation=tf.nn.relu)
conv36 = tf.layers.conv2d(conv35, 256, 1, activation=tf.nn.relu)
conv37 = tf.layers.conv2d(conv36, 512, 3, activation=tf.nn.relu)
conv38 = tf.layers.conv2d(conv37, 256, 1, activation=tf.nn.relu)
conv39 = tf.layers.conv2d(conv38, 512, 3, activation=tf.nn.relu)
conv40 = tf.layers.conv2d(conv39, 256, 1, activation=tf.nn.relu)
conv41 = tf.layers.conv2d(conv40, 512, 3, activation=tf.nn.relu)
conv42 = tf.layers.conv2d(conv41, 256, 1, activation=tf.nn.relu)
conv43 = tf.layers.conv2d(conv42, 512, 3, activation=tf.nn.relu)
conv44 = tf.layers.conv2d(conv43, 1024, 3, activation=tf.nn.relu)
pool5 = tf.layers.AveragePooling2D(conv44, 3, 2)
#block5
conv45 = tf.layers.conv2d(pool5, 512, 1, activation=tf.nn.relu)
conv46 = tf.layers.conv2d(conv45, 1024, 3, activation=tf.nn.relu)
conv47 = tf.layers.conv2d(conv46, 512, 1, activation=tf.nn.relu)
conv48 = tf.layers.conv2d(conv47, 1024, 3, activation=tf.nn.relu)
conv49 = tf.layers.conv2d(conv48, 512, 1, activation=tf.nn.relu)
conv50 = tf.layers.conv2d(conv49, 1024, 3, activation=tf.nn.relu)
conv51 = tf.layers.conv2d(conv50, 512, 1, activation=tf.nn.relu)
conv52 = tf.layers.conv2d(conv51, 1024, 3, activation=tf.nn.relu)
# Flatten the data to a 1-D vector for the fully connected layer
layers_flat = tf.contrib.layers.flatten(conv52)
# Fully connected layer
fc1 = tf.layers.dense(layers_flat, batch_size)

```

Loss function:

```
from keras import backend as K
#coordinates loss
#xy_loss
xy_loss = K.sum(K.sum(K.square(y_true_xy -
y_pred_xy)+k.square(x_true_xy - x_truexy))*y_true_obj

#wh_loss
wh_loss = K.sum(K.sum(K.square(K.sqrt(y_true_wh) -
K.sqrt(y_pred_wh)))*y_true_obj

#class loss
cls_loss = K.sum(K.square(y_true_class -
y_pred_class)*y_true_obj

#confidence loss
intersect_wh = K.maximum(K.zeros_like(y_pred_wh), (y_pred_wh +
y_true_wh)/2 - K.abs(y_pred_xy - y_true_xy) )
intersect_area = intersect_wh[0] * intersect_wh[1]
true_area = y_true_wh[0] * y_true_wh[1]
pred_area = y_pred_wh[0] * y_pred_wh[1]
union_area = pred_area + true_area - intersect_area
iou = intersect_area / union_area
conf_loss = K.sum(K.square(y_true_conf*iou - y_pred_conf) *
y_true_conf

loss = cls_loss + xy_loss + wh_loss + conf_loss
```