Training YOLO v3 for Object Detection with Custom Data – A Project

By-

Anirban Roy, 2020ITB001

Avik Hira, 2020ITB005

Madhuparna Pan, 2020ITB010

B. Tech (IT) – 2020-’24

4th Semester

IIEST, Shibpur

CONTENTS:

Introduction:

○ Background:

A few years ago, the creation of the software and hardware image processing systems was mainly limited to the development of the user interface, which most of the programmers of each firm were engaged in. The situation has been significantly changed with the advent of the Windows operating system when the majority of the developers switched to solving the problems of image processing itself. However, this has not yet led to the cardinal progress in solving typical tasks of recognizing faces, car numbers, road signs, analysing remote and medical images, etc. Each of these "eternal" problems is solved by trial and error by the efforts of numerous groups of the engineers and scientists. As modern technical solutions are turned out to be excessively expensive, the task of automating the creation of the software tools for solving intellectual problems is formulated and intensively solved abroad. In the field of image processing, the required tool kit should be supporting the analysis and recognition of images of previously unknown content and ensure the effective development of applications by ordinary programmers. Just as the Windows toolkit supports the creation of interfaces for solving various applied problems. Object recognition is to describe a collection of related computer vision tasks that involve activities like identifying objects in digital photographs. Image classification involves activities such as predicting the class of one object in an image. Object localization refers to identifying the location of one or more objects in an image and drawing an abounding box around their extent. Object detection combines these two tasks and localizes and classifies one or more objects in an image. When a user or practitioner refers to the term “object recognition”, they often mean “object detection “. It may be challenging for beginners to distinguish between different related computer vision tasks. So, we can distinguish between these three computer vision tasks with this example:

Image Classification: This is done by predicting the type or class of an object in an image.
Input: An image which consists of a single object, such as a photograph.
Output: A class label (e.g., one or more integers that are mapped to class labels).

Object Localization: This is done through locating the presence of objects in an image and indicating their location with a bounding box.
Input: An image which consists of one or more objects, such as a photograph.
Output: One or more bounding boxes (e.g., defined by a point, width, and height).

Object Detection: This is done through locating the presence of objects with a bounding box and types or classes of the located objects in an image.

Input: An image which consists of one or more objects, such as a photograph.
Output: One or more bounding boxes (e.g., defined by a point, width, and height), and a class label for each bounding box.

One of the further extensions to this breakdown of computer vision tasks is object segmentation, also called "object instance segmentation" or "semantic segmentation," where instances of recognized objects are indicated by highlighting the specific pixels of the object instead of a coarse bounding box. From this breakdown, we can understand that object recognition refers to a suite of challenging computer vision tasks. For example, image classification is simply straight forward, but the differences between object localization and object detection can be confusing, especially when all the three tasks may be just as equally referred to as object recognition. Humans can detect and identify objects present in an image. The human visual system is fast and accurate and can also perform complex tasks like identifying multiple objects and detecting obstacles with little conscious thought. With the availability of large sets of data, faster GPUs and better algorithms, we can now easily train computers to detect and classify multiple objects within an image with high accuracy. We need to understand terms such as object detection, object localization, loss function for object detection and localization, and finally explore an object detection algorithm known as "You only look once" (YOLO). Image classification also involves assigning a class label to an image, whereas object localization involves drawing a bounding box around one or more objects in an image. Object detection is always more challenging and combines these two tasks and draws a bounding box around each object of interest in the image and assigns them a class label. Together, all these problems are referred to as object recognition. Object recognition refers to a collection of related tasks for identifying objects in digital photographs.

YOLO is a method for real-time object identification and recognition in photographs, videos or in real-time cam. It is an acronym for **You Only Look Once**. Redmond et al. proposed the approach in a paper that was initially published in 2015 at the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). The OpenCV People's Choice Award was given to the paper. Unlike previous object identification methods, which repurposed classifiers to do detection, YOLO proposes the usage of an end-to-end neural network that predicts bounding boxes and class probabilities simultaneously. YOLO produces state-of-the-art results by taking a fundamentally new approach to object recognition, easily outperforming previous real-time object detection methods.

○ Motivation:

The main purpose of object detection is to identify and locate one or more effective targets from still image or video data. It comprehensively includes a variety of important techniques, such as image processing, pattern recognition, artificial intelligence and machine learning. It has broad application prospects in such areas such as road traffic accident prevention, warnings of dangerous goods in factories, military restricted area monitoring and advanced human–computer interaction. Since the application scenarios of multi-target detection in the real world are usually complex and variable, balancing the relationship between

accuracy and computing costs is a difficult task. The object detection process is traditionally established by manually extracting feature models, where the common features are represented by HOG (histogram of oriented gradient), SIFT (scale-invariant feature transform), Haar (Haar-like features) and other classic algorithms based on grayscale. Following feature extraction, the SVM (support vector machine) or Adaboost algorithms are used for classification in order to obtain target information. These traditional extracting feature models are only able to determine low-level feature information, such as contour information and texture information, and have limitations in detecting multiple targets under complex scenes due to their poor generalization performance. However, object detection models, such as the R-CNN (region-based convolutional neural networks) series and the YOLO (you only look once) or SSD (single shot multiBox detection) models based on the deep learning CNN (convolutional neural network) features are more well-known. Deep learning CNN models not only extract the detail texture features from pre-level convolution networks, but are also able to obtain higher-level information from the post-level convolution layer. Following the traditional CNN process, the R-CNN series uses an enumeration method to presuppose the target candidate region in the feature map, gradually fine-tuning the position information and optimizing the object position for classification and recognition. In contrast, other object detection models will simultaneously predict the bounding box and classification directly in the feature map by applying different convolution sets. The R-CNN model has two operation stages (candidate region proposal and further detection) that allow for higher detection accuracy, while SSD and YOLO are able to directly detect the classification and position information, improving the detection speed.

○ Project Scope and Deliverable:

In this project, we've trained **our own** Object Detector using YOLO v3-v4 algorithms.

1. As for beginning, we've **implemented already trained** YOLO v3-v4 on COCO dataset. We've detected objects **on image**, **video** and **in real time** by OpenCV deep learning library.
2. After that, we've **labelled individual dataset** as well as **created custom one** by extracting needed images from huge existing dataset.
3. Next, we've **converted Traffic Signs dataset** into YOLO format.
4. When datasets became ready, we've **trained and tested** YOLO v3-v4 detectors in Darknet framework.
5. Finally, we've **built graphical user interface** for Object Detection by YOLO and by the help of PyQt.

- **Outline of the Report:**

  The following sections of the report, firstly, go over the detailed methodology of the project. The report briefly describes the system design and architecture and then moves on to environment setup required to run the algorithm. After describing the core parts of the algorithm, the report goes on to mention the testing criteria and methods as well as the expected and achieved results. Next, the report highlights some of the applications of the algorithm, followed by a conclusion to wrap up the content.

Project Methodology:

Brief Introduction to Technical Concepts:

- **Computer Vision**: Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos and other visual inputs — and take actions or make recommendations based on that information. If AI enables computers to think, computer vision enables them to see, observe and understand. Computer vision works much the same as human vision, except humans have a head start. Human sight has the advantage of lifetimes of context to train how to tell objects apart, how far away they are, whether they are moving and whether there is something wrong in an image.

Computer Vision is perhaps the most important part of the AI Dream

Source: *Terminator 2: Judgement Day*

Machines that can understand what they see WILL change the world!

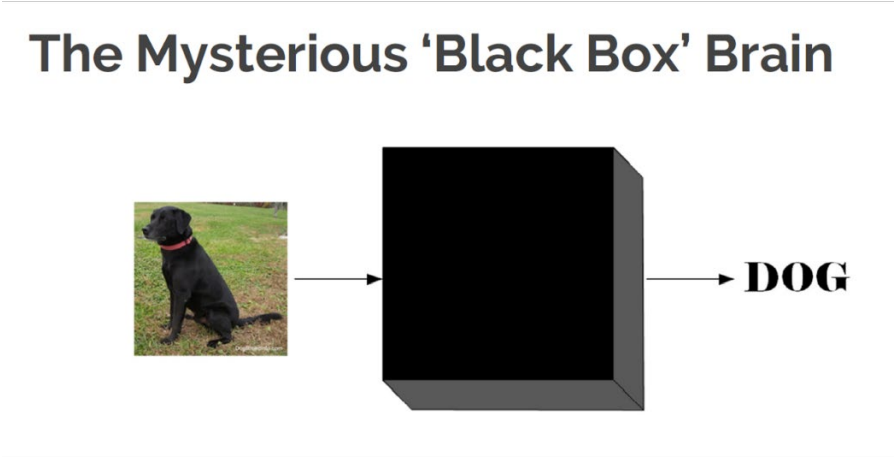
Robots or Machines that can see will be capable of...

- Autonomous Vehicles (self driving cars)
- Robotic Surgery & Medical Diagnostics
- Robots that can navigate our clutter world
  - Robotic chefs, farmers, assistants etc.
- Intelligent Surveillance and Drones
- Creation of Art
- Improved Image & Video Searching
- Social & Fun Applications
- Improve Photography

- **Neural Networks**: Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.

  Artificial neural networks (ANNs) are comprised of node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and

threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.



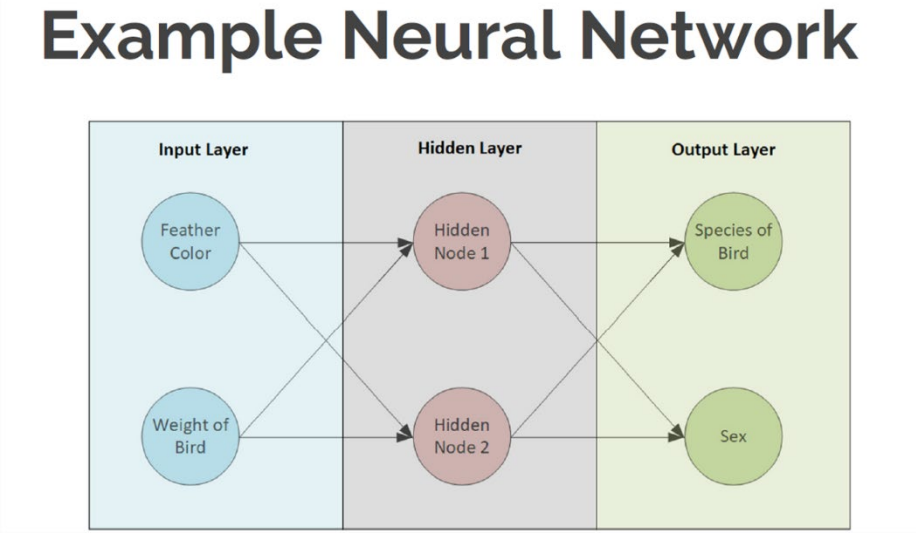
Networks act as a 'black box' brain that takes inputs and predicts an output.

Types of Neural Networks

There are different kinds of deep neural networks – and each has advantages and disadvantages, depending upon the use. Examples include:

- Convolutional neural networks (CNNs) contain five types of layers: input, convolution, pooling, fully connected and output. Each layer has a specific purpose, like summarizing, connecting or activating. Convolutional neural networks have popularized image classification and object detection. However, CNNs have also been applied to other areas, such as natural language processing and forecasting.
- Recurrent neural networks (RNNs) use sequential information such as time-stamped data from a sensor device or a spoken sentence, composed of a sequence of terms. Unlike traditional neural networks, all inputs to a recurrent neural network are not independent of each other, and the output for each element depends on the computations of its preceding elements. RNNs are used in forecasting and time series applications, sentiment analysis and other text applications.
- Feedforward neural networks, in which each perceptron in one layer is connected to every perceptron from the next layer. Information is fed forward from one layer to the next in the forward direction only. There are no feedback loops.
- Autoencoder neural networks are used to create abstractions called encoders, created from a given set of inputs. Although similar to more traditional neural networks, autoencoders seek to model the inputs themselves, and therefore the method is considered unsupervised. The premise of autoencoders is to desensitize the irrelevant and sensitize the relevant. As layers are added, further abstractions are formulated at higher

layers (layers closest to the point at which a decoder layer is introduced). These abstractions can then be used by linear or nonlinear classifiers.



How do Neural Networks work?

Think of each individual node as its own linear regression model, composed of input data, weights, a bias (or threshold), and an output. The formula would look something like this:

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + w_2 x_2 + w_3 x_3 + bias$$

$$\text{output} = f(x) = \begin{cases} 1 \text{ if } \sum w_1 x_1 + b \geq 0 \\ 0 \text{ if } \sum w_1 x_1 + b < 0 \end{cases}$$

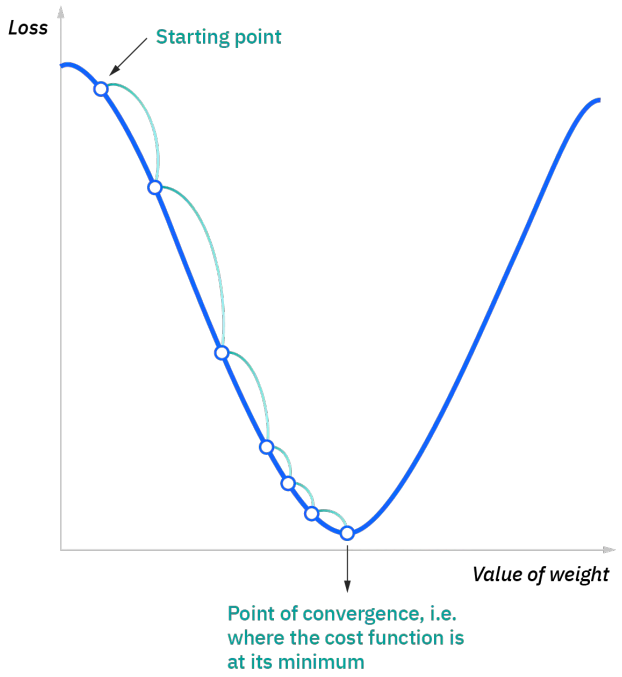
Once an input layer is determined, weights are assigned. These weights help determine the importance of any given variable, with larger ones contributing more significantly to the output compared to other inputs. All inputs are then multiplied by their respective weights and then summed. Afterward, the output is passed through an activation function, which determines the output. If that output exceeds a given threshold, it "fires" (or activates) the node, passing data to the next layer in the network. This results in the output of one node becoming in the input of the next node. This process of passing data from one layer to the next layer defines this neural network as a feedforward network.

As we start to think about practical use cases for neural networks, like image recognition or classification, we'll leverage supervised learning, or labelled datasets, to train the algorithm. As we train the model, we'll want to evaluate its accuracy using a cost (or loss) function. This is also commonly referred to as the mean squared error (MSE). In the equation below,

- *i* represents the index of the sample,
- y-hat is the predicted outcome,
- y is the actual value, and
- *m* is the number of samples.

$$\text{Cost Function} = MSE = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y} - y)^2$$

Ultimately, the goal is to minimize our cost function to ensure correctness of fit for any given observation. As the model adjusts its weights and bias, it uses the cost function and reinforcement learning to reach the point of convergence, or the local minimum. The process in which the algorithm adjusts its weights is through gradient descent, allowing the model to determine the direction to take to reduce errors (or minimize the cost function). With each training example, the parameters of the model adjust to gradually converge at the minimum.



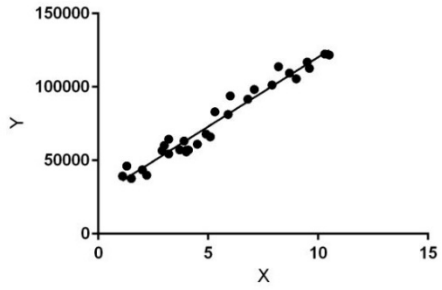
Most deep neural networks are feedforward, meaning they flow in one direction only, from input to output. However, you can also train your model through backpropagation; that is, move in the opposite direction from output to input. Backpropagation allows us to calculate and attribute the error associated with each neuron, allowing us to adjust and fit the parameters of the model(s) appropriately.

- **Neuron**: Each neuron is a mathematical operation that takes its input, multiplies it by its weights and then passes the sum through the activation function to the other neurons. A single node may take the input data and multiply it by an assigned weight value, then add a bias before passing the data to the next layer.

- **Weight**: Weight affects the amount of influence a change in the input will have upon the output. A low weight value will have no change on the input, and alternatively a larger weight value will more significantly change the output.

- **Bias**: Biases make up the difference between the function's output and its intended output. A low bias suggest that the network is making more assumptions about the form of the output, whereas a high bias value makes less assumptions about the form of the output.

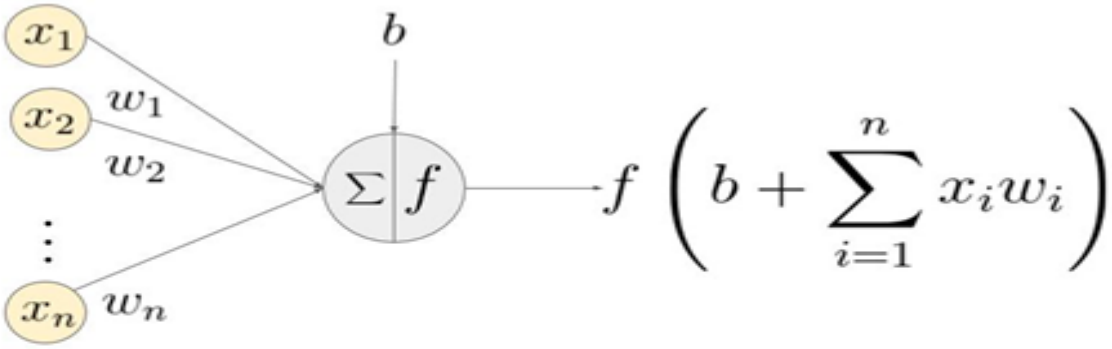
- **Linear Regression**: Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable to be predicted is called the dependent variable. The variable used to predict the other variable's value is called the independent variable.



- **Gradient Descent**: Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks.

- **Activation Function:** The activation function is the nonlinear transformation that we do over the input signal. This transformed output is then sent to the next layer of neurons as input. It helps the **network learn complex patterns in the data**. When comparing with a neuron-based model that is in our brains, the

activation function is at the end deciding **what is to be fired to the next neuron**. That is exactly what an activation function does. The activation is basically a mathematical "gate" that determines whether each neuron's input is relevant for the model's prediction. If it meets a certain threshold, it passes through the gate, if not, it is being disregarded. The function brings the relevant features on the image into focus.



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

Different types of Activation functions are-

1. Sigmoid function - The main reason why we use sigmoid function is because it exists between 0 to 1**.** Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1. The function is **differentiable**. That means, we can find the slope of the sigmoid curve at any two points.

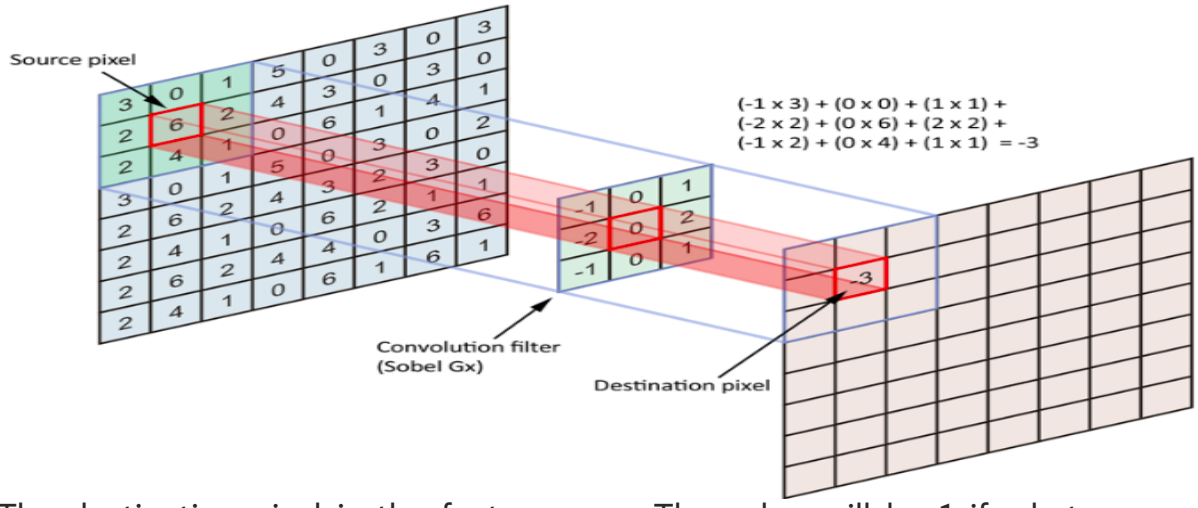
2. ReLU - Stands for *Rectified linear unit*. It is the most widely used activation function. Chiefly implemented in *hidden layers* of Neural network.

3. Softmax function - If we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function.

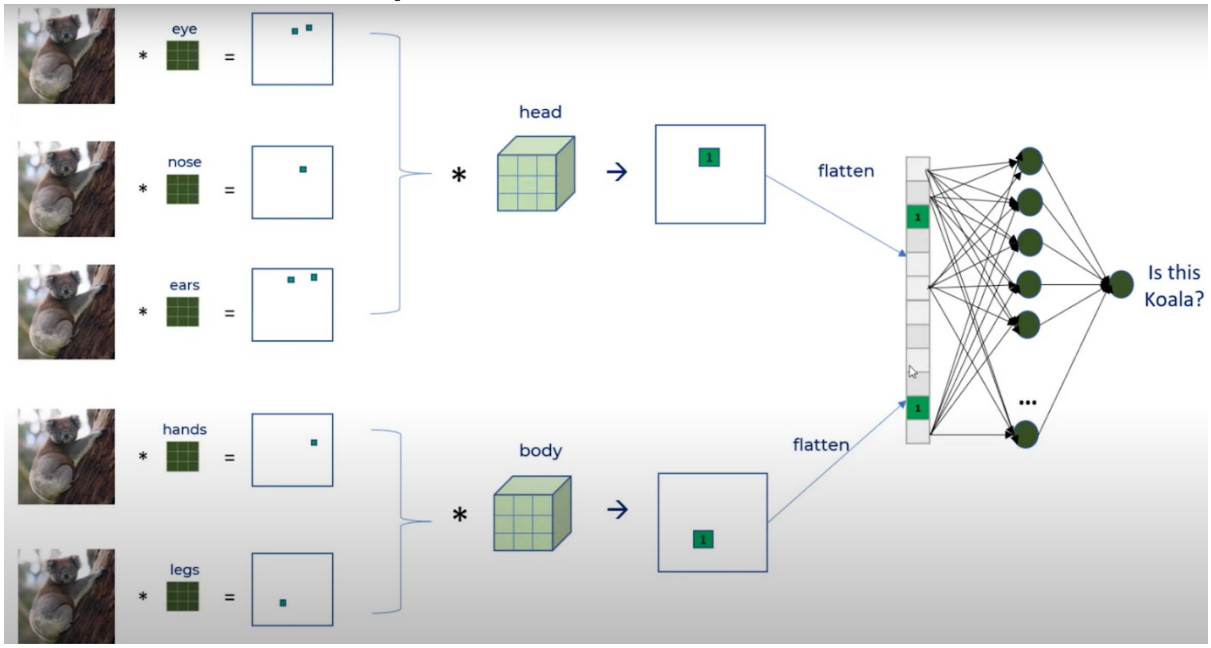
- **Convolutional layer:** Convolutional layers are the layers where filters (feature detectors) are applied to the original image, or to other **feature maps** in a deep CNN. This is where most of the user-specified parameters are in the network. The most important parameters are the number of **kernels** and the size of the

kernels. (A kernel is a matrix that moves over the input data, performs the dot product with the sub-region of input data, and gets the output as the matrix of dot products.) 3 x 3 denotes the 3 colour components RGB.

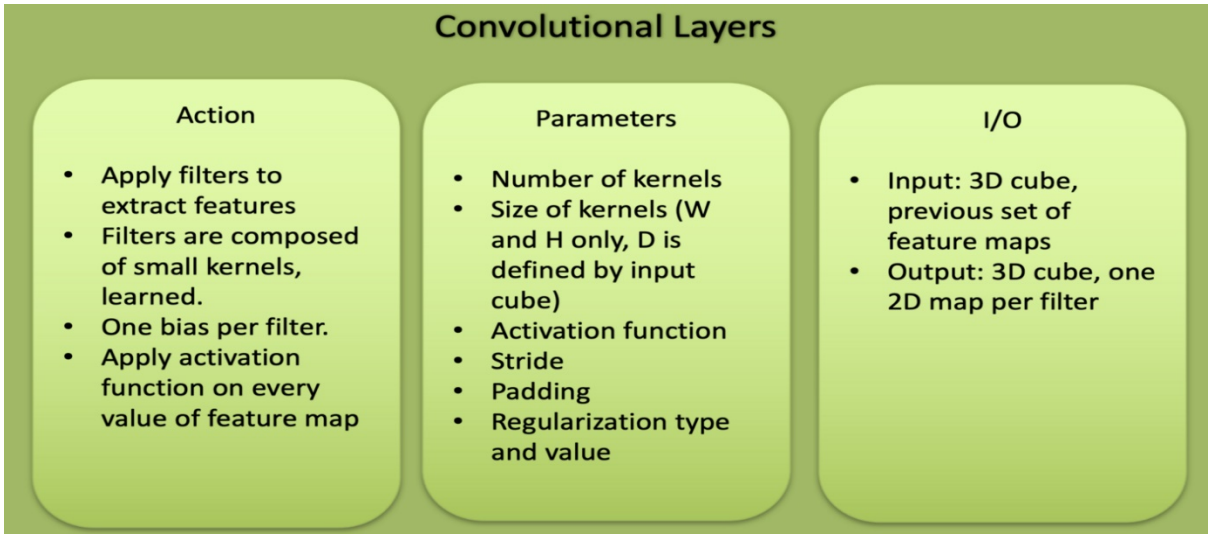
(Feature maps-The feature maps of a CNN capture the result of applying the filters to an input image, i.e., at each layer, the feature map is the output of that layer.)



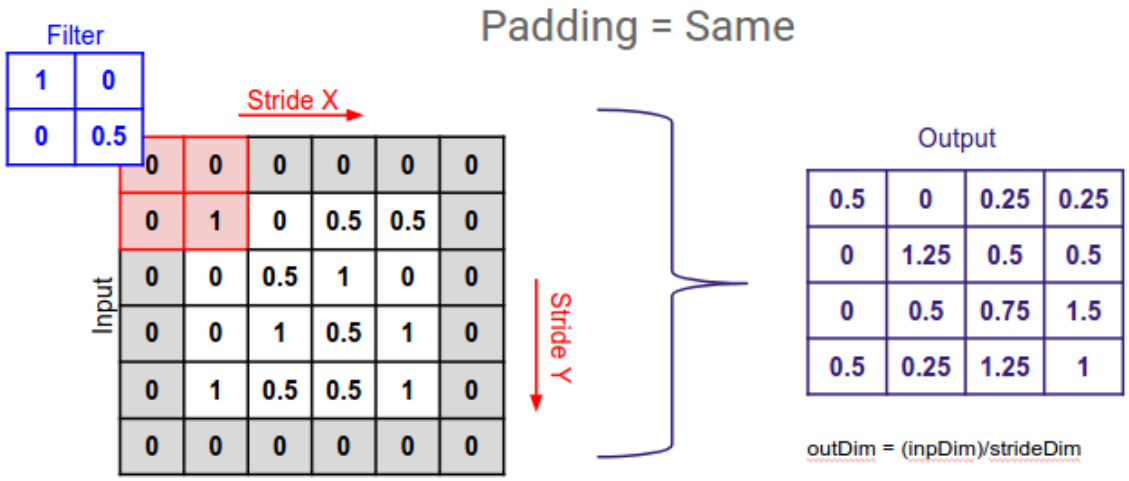
The destination pixel is the feature map. The value will be 1 if what we are detecting is found. The **no. of features to be detected = No. of convolutional filters = No. of feature maps.**



We can aggregate the outputs of different feature maps to get a single map.

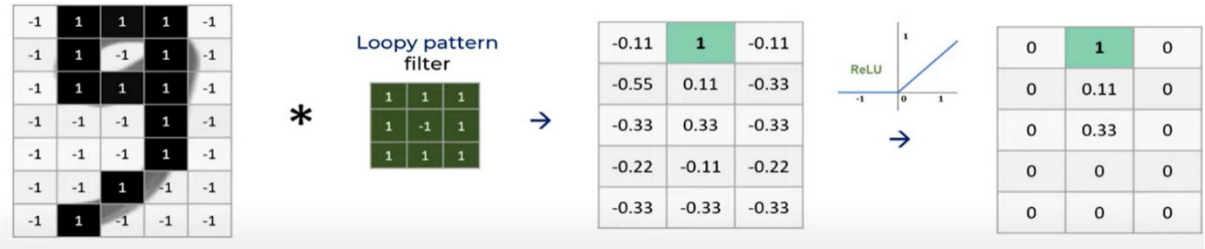


- **Padding:** Padding extends the area of an image in which a convolutional neural network process. Adding padding to an image processed by a CNN allows for a more accurate analysis of images.

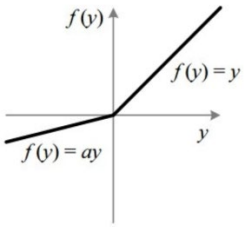


The grid in grey is the padding layer.

- **ReLU (Rectified Linear Unit):** Brings non-linearity to the feature map. This changes all the negative values to zero.

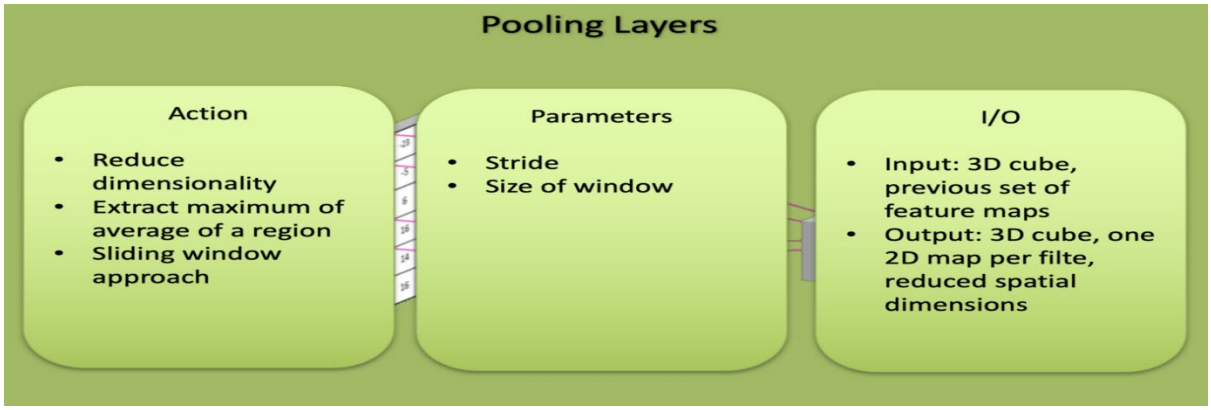


- **Leaky ReLU:** Leaky Rectified Linear Unit (Leaky ReLU) is a type of activation function based on a ReLU, but it has a small slope for negative values instead of a flat slope. The slope coefficient is determined before training, i.e., it is not learnt during training. This type of activation function is popular in tasks where we may suffer from sparse gradients, for example training generative adversarial networks.



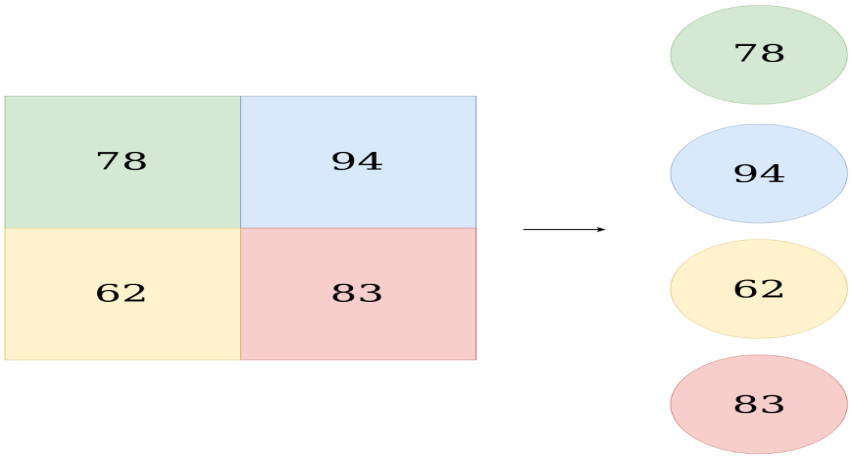
- **Pooling:** Pooling layers are similar to convolutional layers, but they perform a specific function such as max pooling, which takes the maximum value in a certain filter region, or average pooling, which takes the average value in a filter region. These are typically used to reduce the dimensionality of the network. Pooling is done on the feature map obtained after the convolutional layer. Benefits are - Reduces dimension and computation, reduce overfitting (explained later) as less parameters; Model is tolerant towards distortions.

The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features.



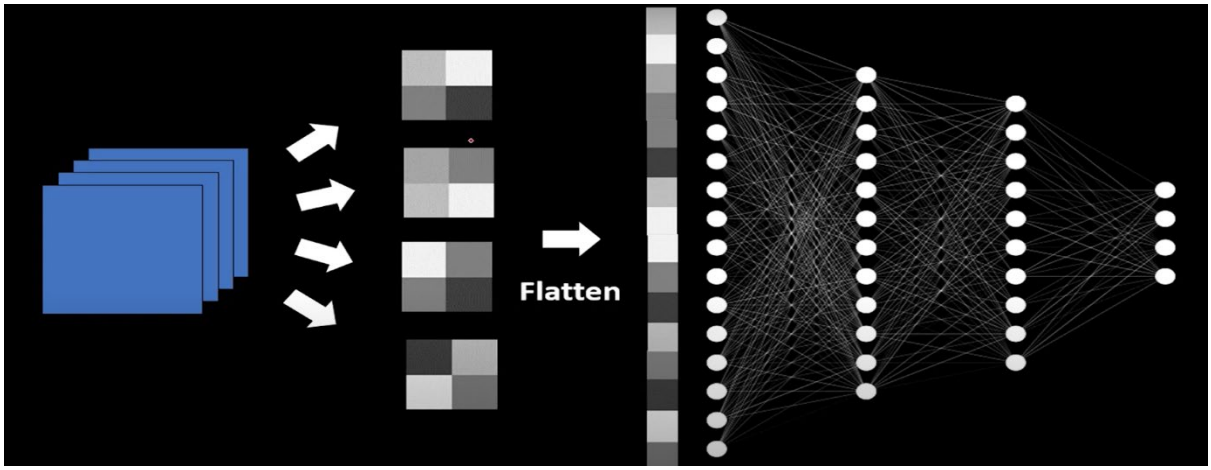
- **Connected layer:** Fully connected layers are placed before the classification output of a CNN and are used to flatten the results before classification. The input to the fully connected layer is the output from the *final* Pooling or Convolutional Layer, which is flattened and then fed into the fully connected layer.

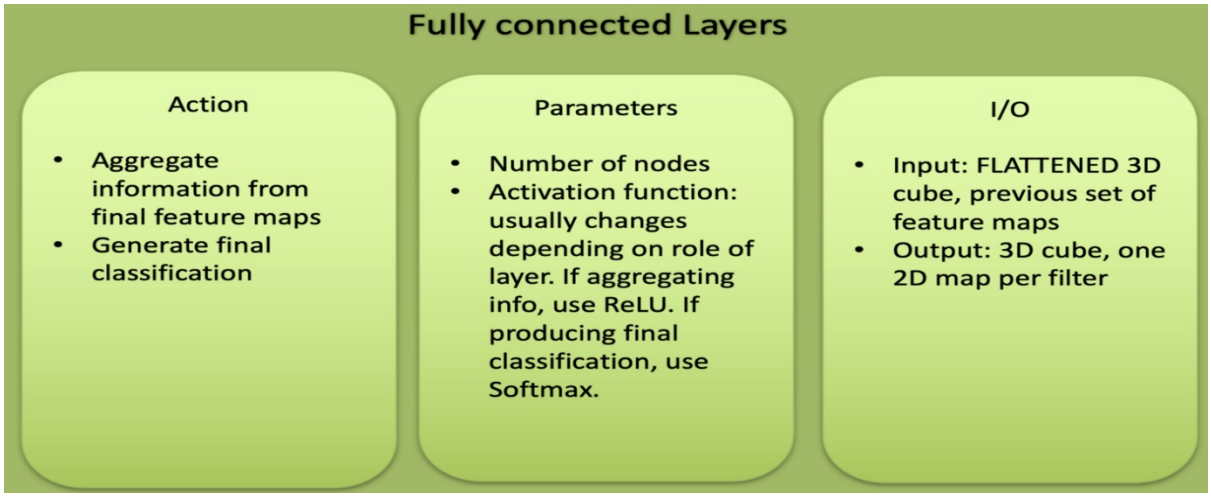
- **Flattened?:** The output from the final (and any) Pooling and Convolutional Layer is a 3-dimensional matrix, to flatten that is to unroll all its values into a vector or 1D array.



Connected layer is a dense network of neurons and connections between two neurons. It is used to classify an image to any specific category. It is also responsible for identifying images and giving it labels. E.g., if it finds a face it will be labelled as human.

In the figure below there are 3 fully connected layers, first layer being the input layer. Each node of a layer is connected to all the nodes of its previous and next layers. The no. of nodes in the final layer= no. of specific categories of images. [If its more than one then we use SoftMax activation function to classify images. For binary classification sigmoid activation function is used]





- **Under-fitting and Over-fitting:**

  In under-fitting we train the model by giving fewer identifying features therefore it gives outputs with many errors. Due to lack of enough attributes, model doesn't have enough knowledge and gives incoherent results.

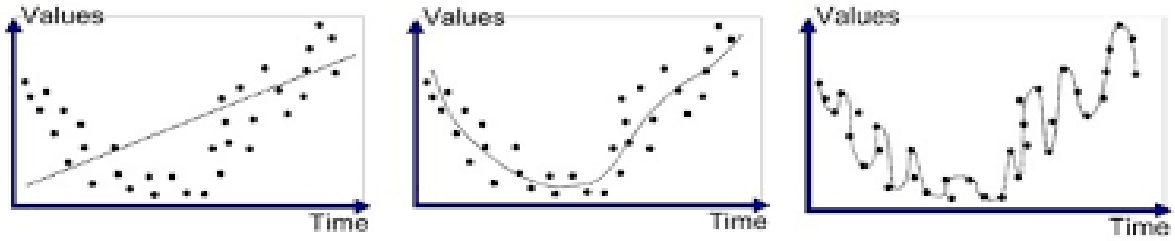
  In over-fitting we train the model by giving many identifying features which also produces error. Due to excess knowledge of attributes where model tries to use all attributes (even the least important ones);

  For example,

  If we train our model to identify a ball, if the only identifying feature we give is it should be a sphere, then it will identify an orange as a ball too. This is underfitting

  If the features we give to it are -

  1. Sphere, 2. It can be used to play, 3. It cannot be eaten, 4. Its radius should be 5cm, then it will not identify any object larger or smaller to be a ball. This is overfitting.



Underfitted Good Fit/Robust Overfitted

The black line represents the curve that is the best fit for the given data points. It can also be seen that curve fitting does not necessarily mean that the curve should pass over each and every data point. Instead, it is the most appropriate curve that represents all the data points adequately.

Detection of underfitting model -

1. Training and validation loss - It is important to check the loss that is generated by the model. If the model is underfitting, the loss for both training and validation will be significantly high.
2. Over Simplistic prediction graph - On visualization, it would clearly seem that a more complex curve can fit the data better.

Fixing an underfitted model -

1. **Train Longer:** Since underfitting means less model complexity, training longer can help in learning more complex patterns.
2. **Train a more complex model:** The main reason behind the model to underfit is using a model of lesser complexity than required for the data. Hence, the most obvious fix is to use a more complex model.
3. **Obtain more features:** If the data set lacks enough features to get a clear inference, then collecting more features will help fit the data better.
4. **Decrease Regularization:** Regularization is the process that helps Generalize the model by avoiding overfitting. However, if the model is learning less or underfitting, then it is better to decrease or completely remove Regularization techniques so that the model can learn better.

Detection of over-fitting model -

1. Training and validation loss - A very low training loss but a high validation loss would signify that the model is overfitting. If the training loss keeps on decreasing but the validation loss remains stagnant or starts to increase, it also signifies that the model is overfitting.
2. Too complex prediction graph- If the final "Best Fit" line crosses over every single data point by forming an unnecessarily complex curve, then the model is likely overfitting.

Fixing an over-fitted model –

1. **Early Stopping during Training:** Allowing the model to train for a high number of iterations may lead to overfitting. Hence it is necessary to stop the model from training when the model has started to overfit. This is done by monitoring the validation loss and stopping the model when the loss stops decreasing over a given number of iterations.

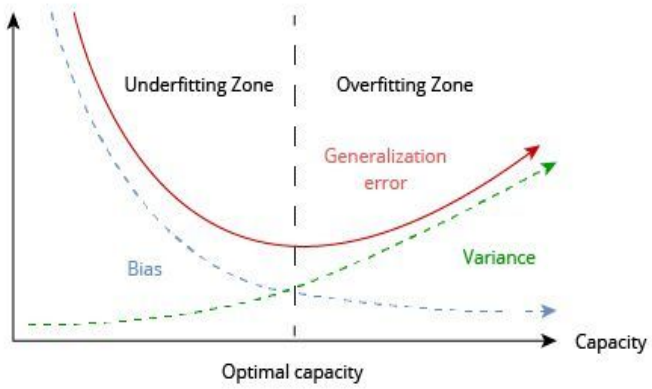
2. **Train with more data:** Often, the data available for training is less when compared to the model complexity. Hence, in order to get the model to fit appropriately, it is often advisable to increase the training dataset size.
3. **Train a less complex model:** The main reason behind overfitting is excessive model complexity for a relatively less complex dataset. Hence it is advisable to reduce the model complexity in order to avoid overfitting.

4. **Remove features:** Reducing the number of unnecessary or irrelevant features often leads to a better and more generalized.

- **Relationship between Overfitting and Underfitting with Bias-Variance Trade-off:**

Bias denotes the simplicity of the model. A high biased model will have a simpler architecture than that of a model with a lower bias. Similarly, complementing Bias, Variance denotes how complex the model is and how well it can fit the data with a high degree of diversity.

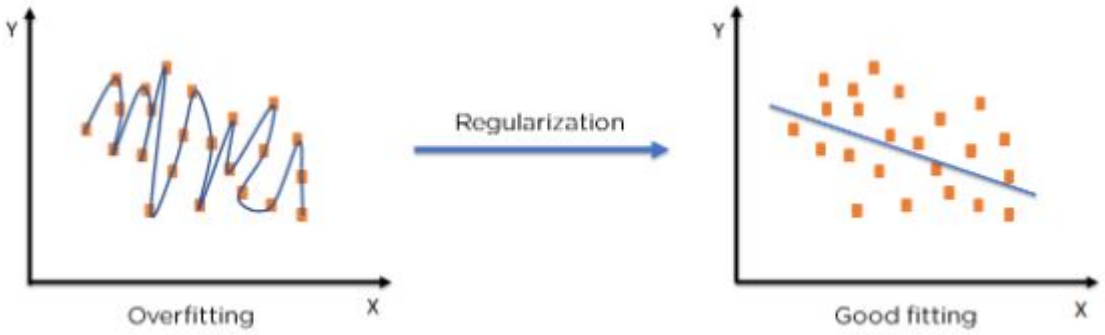
An ideal model should have Low Bias and Low Variance. However, when it comes to practical datasets and models, it is nearly impossible to achieve a "zero" Bias and Variance. These two are complementary of each other, if one decreases beyond a certain limit, then the other starts increasing. This is known as the Bias-Variance Trade-off.



A model with high Bias means the model is Underfitting the given data and a model with High Variance means the model is Overfitting the given data.

Hence, as it can be seen, at the optimal region of the Bias-Variance trade-off, the model is neither underfitting nor overfitting and is most generalised, as under these conditions the model is expected to perform equally well on Training and Validation Data. Thus, the graph depicts that the generalisation error is minimum at the optimal value of the degree of Bias and Variance.

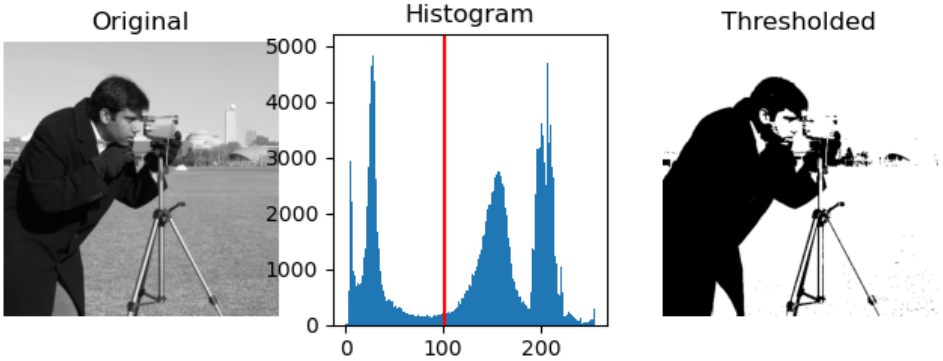
- **Regularization:** Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting. Using Regularization, we can fit our machine learning model appropriately on a given test set and hence reduce the errors in it.



- **Thresholding:** An image processing method that creates a bitonal (aka binary) image based on setting a threshold value on the pixel intensity of the original image. While most commonly applied to grayscale images, it can also be applied to colour images. The threshold of image intensity (relative image lightness) is set manually at a specific value or automatically set by an application. Pixels below that set threshold value are converted to black (bit value of zero), and pixels above the threshold value are converted to white (a bit value of one). The thresholding process is sometimes described as separating an image into foreground values (black) and background values (white).

  Simple thresholding operations establish a single global threshold value for all pixels in an image irrespective of any local variations in contrast. More sophisticated thresholding processes (adaptive thresholding) sample small and greater number of regions of the image and establish the threshold value accordingly.

  The quality of thresholding is of particular importance when performing OCR on images. Foxed, mottled, stained or irregularly faded materials can present a challenge in properly separating foreground (text) from background.

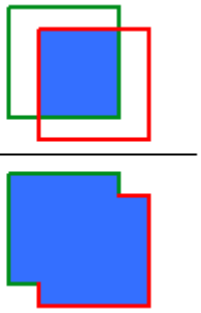


- **<u>Intersection over Union (IoU):</u>** Intersection over Union is a popular metric to measure localization accuracy and calculate localization errors in object detection models.

  To calculate the IoU with the predictions and the ground truth, we first take the intersecting area between the bounding boxes for a particular prediction and the ground truth bounding boxes of the same area. Following this, we calculate the total area covered by the two bounding boxes—also known as the Union.

  The intersection divided by the Union, gives us the ratio of the overlap to the total area, providing a good estimate of how close the bounding box is to the original prediction.

$$IOU = \frac{\text{area of overlap}}{\text{area of union}} =$$



- **<u>Average Precision (AP):</u>** Average Precision is calculated as the area under a precision vs recall curve for a set of predictions.

  Recall is calculated as the ratio of the total predictions made by the model under a class with a total of existing labels for the class.

  On the other hand, Precision refers to the ratio of true positives with respect to the total predictions made by the model.

  The area under the precision vs recall curve gives us the Average Precision per class for the model. The average of this value, taken over all classes, is termed as mean Average Precision (mAP).

- **YOLO:**

- **What is YOLO?:** YOLO is an abbreviation for the term 'You Only Look Once'. This is an algorithm that detects and recognizes various objects in a picture (in real-time). Object detection in YOLO is done as a regression problem and provides the class probabilities of the detected images.

  YOLO algorithm employs convolutional neural networks (CNN) to detect objects in real-time. As the name suggests, the algorithm requires only a single forward propagation through a neural network to detect objects.

  This means that prediction in the entire image is done in a single algorithm run. The CNN is used to predict various class probabilities and bounding boxes simultaneously.

  The YOLO algorithm consists of various variants. Some of the common ones include tiny YOLO and YOLOv3.

- **Why the YOLO algorithm is important?:** YOLO algorithm is important because of the following reasons:

  Speed: This algorithm improves the speed of detection because it can predict objects in real-time.

  High accuracy: YOLO is a predictive technique that provides accurate results with minimal background errors.

  Learning capabilities: The algorithm has excellent learning capabilities that enable it to learn the representations of objects and apply them in object detection.

Accurate object detection is slow!

	Pascal 2007 mAP	Speed	
DPM v5	33.7	.07 FPS	14 s/img
R-CNN	66.0	.05 FPS	20 s/img
Fast R-CNN	70.0	.5 FPS	2 s/img
Faster R-CNN	73.2	7 FPS	140 ms/img
YOLO	63.4	45 FPS	22 ms/img

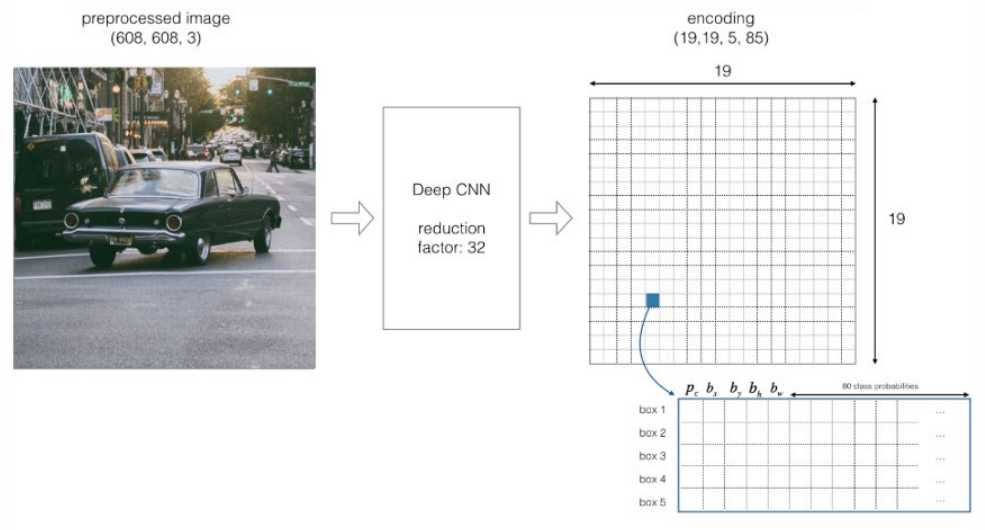
- **How does YOLO algorithm work?:** YOLO algorithm gives a much better performance on all the parameters we discussed along with a high fps for real-time usage. YOLO algorithm is an algorithm based on regression, instead of selecting the interesting part of an Image, it predicts classes and bounding boxes for the whole image in **one run of the Algorithm.**

  To understand the YOLO algorithm, first we need to understand what is actually being predicted. Ultimately, we aim to predict a class of an object and the bounding box specifying object location. Each bounding box can be described using four descriptors:

  1. Center of the box (**bx, by**)
  2. Width (**bw**)
  3. Height (**bh**)
  4. Value **c** corresponding to the class of an object

  Along with that we predict a real number **pc**, which is the probability that there is an object in the bounding box.

  YOLO doesn't search for interested regions in the input image that could contain an object, instead it splits the image into cells, typically 19x19 grid. Each cell is then responsible for predicting K bounding boxes.



  An Object is considered to lie in a specific cell only if the center co-ordinates of the anchor box lie in that cell. Due to this property the center co-ordinates are always calculated relative to the cell whereas the height and width are calculated relative to the whole Image size.

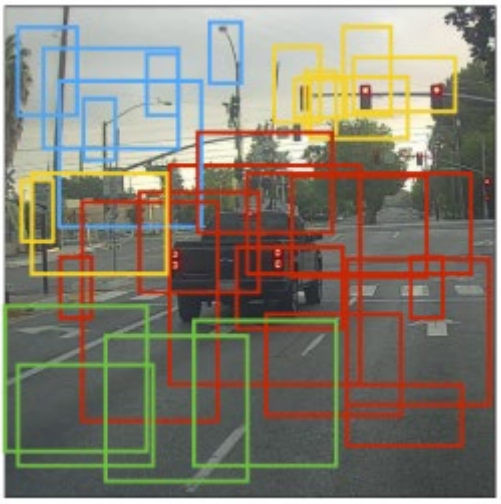
  During the one pass of forwards propagation, YOLO determines the probability that the cell contains a certain class. The equation for the same is:

$$score_{c,i} = p_c \times c_i$$

The class with the maximum probability is chosen and assigned to that particular grid cell. Similar process happens for all the grid cells present in the image.

After computing the above class probabilities, the image may look like this:

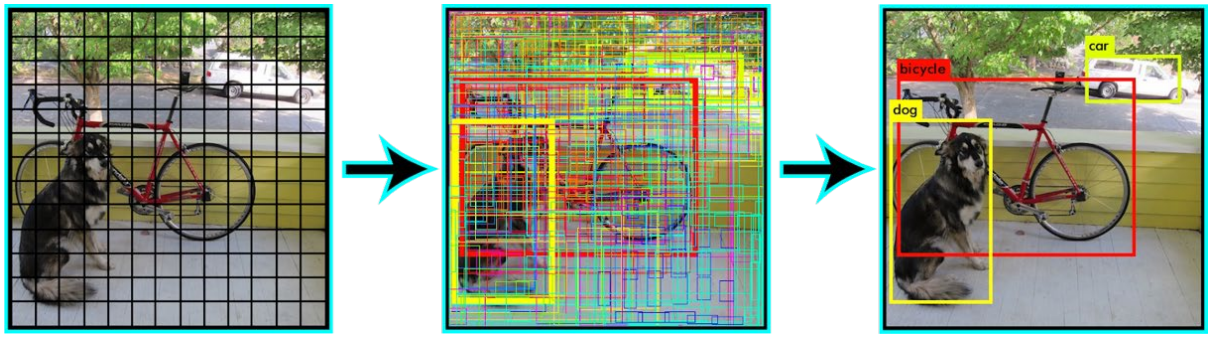
This shows the before and after of predicting the class probabilities for each grid cell. After predicting the class probabilities, the next step is non-max suppression, it helps the algorithm to get rid of the unnecessary anchor boxes, like you can see that in the figure below, there are numerous anchor boxes calculated based on the class probabilities.



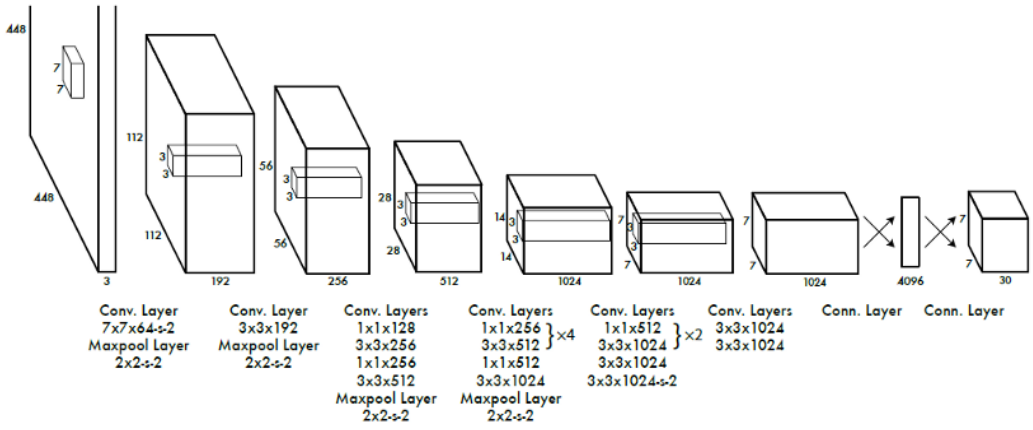
To resolve this problem non-max suppression eliminates the bounding boxes that are very close by preforming the IoU (Intersection over Union) with the one having the highest-class probability among them.

It calculates the value of IoU for all the bounding boxes respective to the one having the highest-class probability, it then rejects the bounding boxes whose value of IoU is greater than a threshold. It signifies that those two bounding boxes are covering the same object but the other one has a low probability for the same, thus it is eliminated.

Once done, algorithm finds the bunding box with next highest-class probabilities and does the same process, it is done until we are left with all the different bounding boxes.



After this, almost all of our work is done, the algorithm finally outputs the required vector showing the details of the bounding box of the respective class. The overall architecture of the algorithm can be viewed below:



Also, the most important parameter of the Algorithm, its Loss function is shown below. YOLO simultaneously learns about all the four parameters it predicts:

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

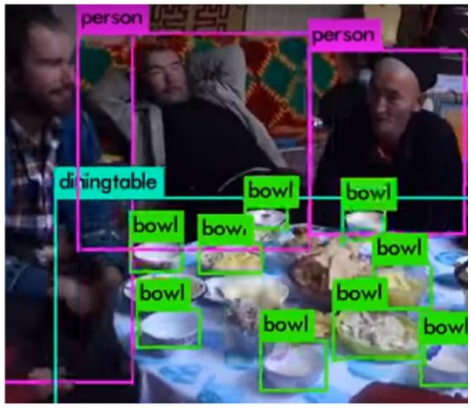
$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

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

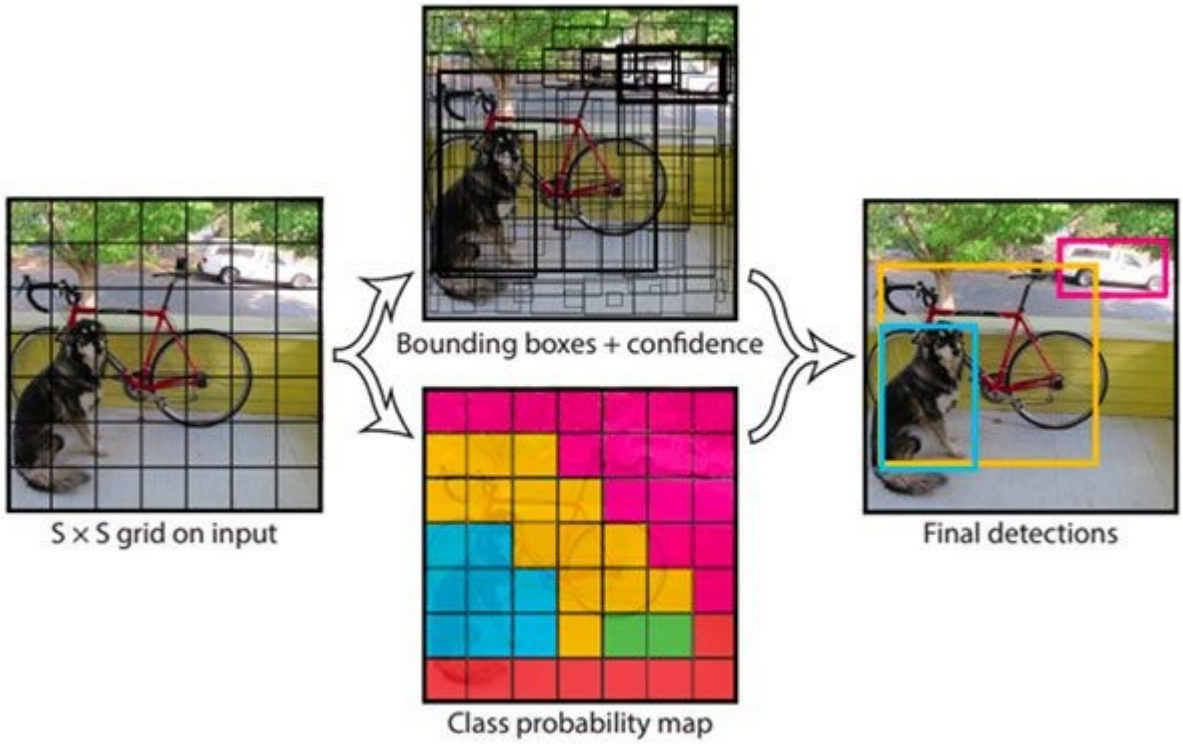
$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

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

- The idea behind YOLO is that a single neural network is applied to full image. This allows YOLO to reason globally about the image when generating predictions
- It is a direct development of MultiBox, but it turns MultiBox from region proposal in to an objection recognition method by adding a softmax layer in parallel with a box regressor and box classifier layer.
- It divides the image into regions and predicts bounding boxes and probabilities for each region.
- YOLO uses a Fully Convolution Neural Network allowing for input of various image sizes.



- **YOLO Model:**



- **YOLO v3:** There are major differences between YOLOv3 and older versions occur in terms of speed, precision, and specificity of classes. YOLOv2 and YOLOv3 are worlds apart in terms of accuracy, speed, and architecture. YOLOv2 came out in 2016, two years before YOLO v3.

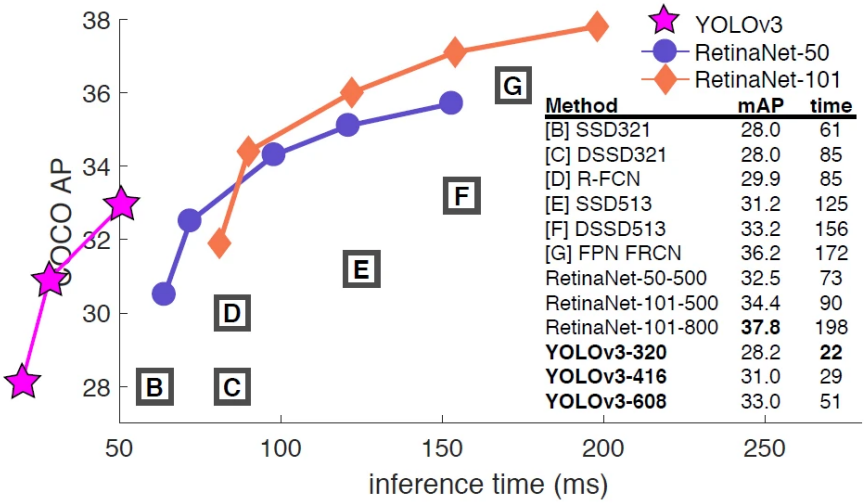
Speed: YOLOv2 was using Darknet-19 as its backbone feature extractor, while YOLOv3 now uses Darknet-53. Darknet-53 is a backbone also made by the YOLO creators Joseph Redmon and Ali Farhadi.

Darknet-53 has 53 convolutional layers instead of the previous 19, making it more powerful than Darknet-19 and more efficient than competing backbones (ResNet-101 or ResNet-152).

Backbone	Top-1	Top-5	Ops	BFLOP/s	FPS
Darknet-19	74.1	91.8	7.29	1246	**171**
ResNet-101	77.1	93.7	19.7	1039	53
ResNet-152	**77.6**	**93.8**	29.4	1090	37
Darknet-53	77.2	**93.8**	18.7	**1457**	78

YOLOv3 is fast and accurate in terms of mean average precision (mAP) and intersection over union (IOU) values as well. It runs significantly faster than other detection methods with comparable performance (hence the name – You only look once).

Moreover, we can easily trade-off between speed and accuracy simply by changing the model's size, without the need for model retraining.



Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

Precision for Small Objects: The chart below shows the average precision (AP) of detecting small, medium, and large images with various algorithms and backbones. The higher the AP, the more accurate it is for that variable.

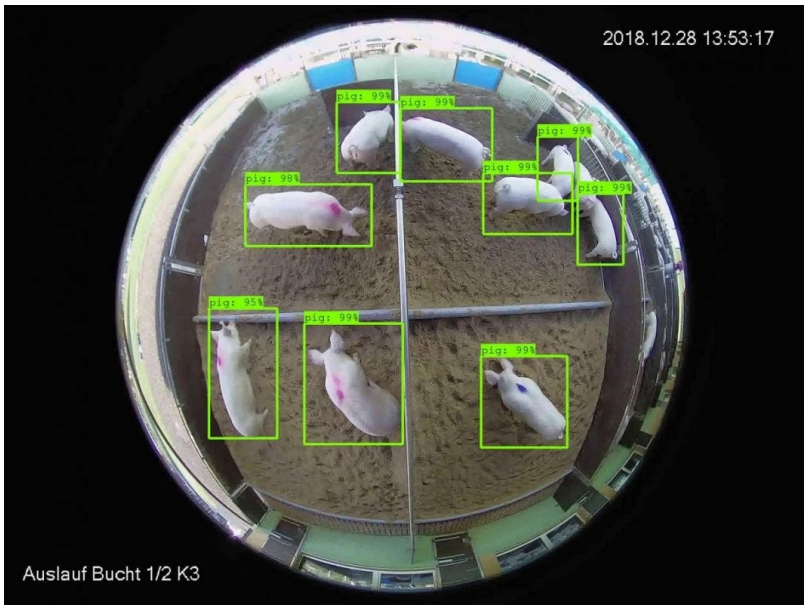
The precision for small objects in YOLOv2 was incomparable to other algorithms because of how inaccurate YOLO was at detecting small objects. With an AP of 5.0, it paled compared to other algorithms like RetinaNet (21.8) or SSD513 (10.2), which had the second-lowest AP for small objects.

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI	Inception-ResNet-v2	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2	DarkNet-19	21.6	44.0	19.2	5.0	22.4	35.5
SSD513	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

Specificity of Classes: The new YOLOv3 uses independent logistic classifiers and binary cross-entropy loss for the class predictions during training. These edits make it possible to use complex datasets such as Microsoft's Open Images Dataset (OID) for YOLOv3 model training. OID contains dozens of overlapping labels, such as "man" and "person" for images in the dataset.

YOLO v3 uses a multilabel approach which allows classes to be more specific and be multiple for individual bounding boxes. Meanwhile, YOLOv2 used a softmax, which is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector.

Using a softmax makes it so that each bounding box can only belong to one class, which is sometimes not the case, especially with datasets like OID.



Installing Dependencies:

For this project, we need the following dependencies installed in our PC:

- Python v3
- Miniconda
- IDE (VS Code)
- OpenCV
- OIDv4 toolkit
- LabelImg
- Darknet Framework
- PyQT

Object Detection on Image with YOLOv3:

When **YOLO works** it **predicts classes' labels** and **detects locations of objects** at the same time. That is why, YOLO can detect multiple objects in one image. The name of the algorithm means that a single network just once is applied to whole image. YOLO **divides image into regions**, **predicts bounding boxes** and **probabilities** for every such region. YOLO also predicts **confidence for every bounding box** showing information that this particular bounding box actually includes object, and **probability of included object** in bounding box being a particular class. Then, bounding boxes are filtered with technique called **non-maximum suppression** that excludes some of them if confidence is low or there is another bounding box for this region with higher confidence.

YOLO-3 is the latest version that uses successive *3x3* and *1x1* convolutional layers. In total it has **53** convolutional layers with architecture as shown on the figure below. Every layer is followed by batch normalization and *Leaky ReLU* activation.

	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			

- **What is inside configuration file?**

Inside *yolov3.cfg* we have parameters that are used for training and testing. Some of them are described below.

Section:

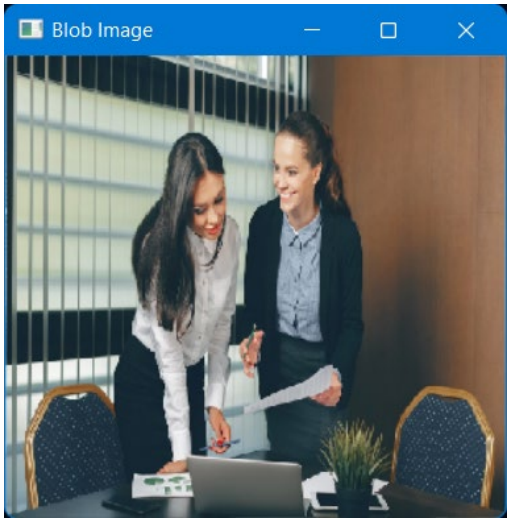
- batch=64–number of samples that will be processed in one batch.
- subdivisions=16–number of *mini batches* in one batch; GPU processes *mini batchsamples at once*; the weights will be updated for batch samples, that is 1 iteration processes batch images.
- width=608–every image will be resized during training and testing to this number.
- height=608 – every image will be resized during training and testing to this number.
- channels=3 – every image will be converted during training and testing to this number.

Optimization:

- momentum=0.9 – hyperparameter for optimizer that defines how much history will influence further updating of weights.
- decay=0.0005 – decay the learning rate over the period of the training.
- learning_rate=0.001 – initial learning rate for training.

Training:

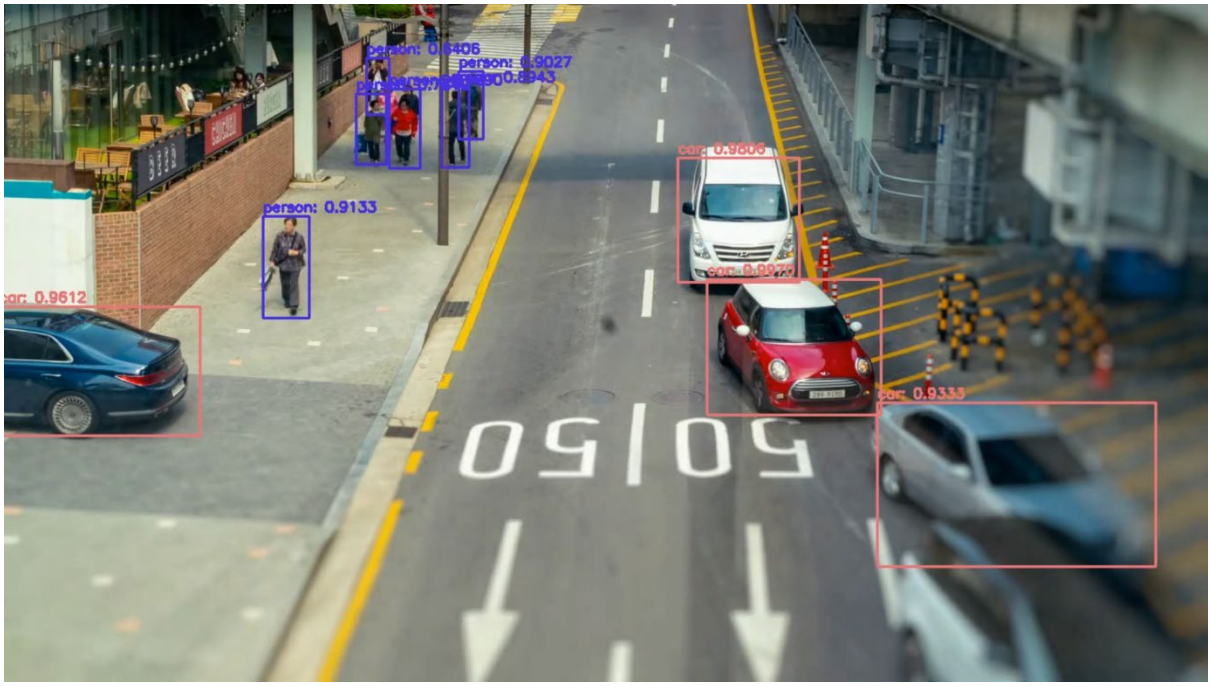
- angle=0 – parameter that randomly *rotates* images during training.
- saturation=1.5 – parameter that randomly *changes saturation* of images during training.
- exposure=1.5 – parameter that randomly *changes brightness* of images during training.
- hue=.1 – parameter that randomly *changes hue* of images during training.

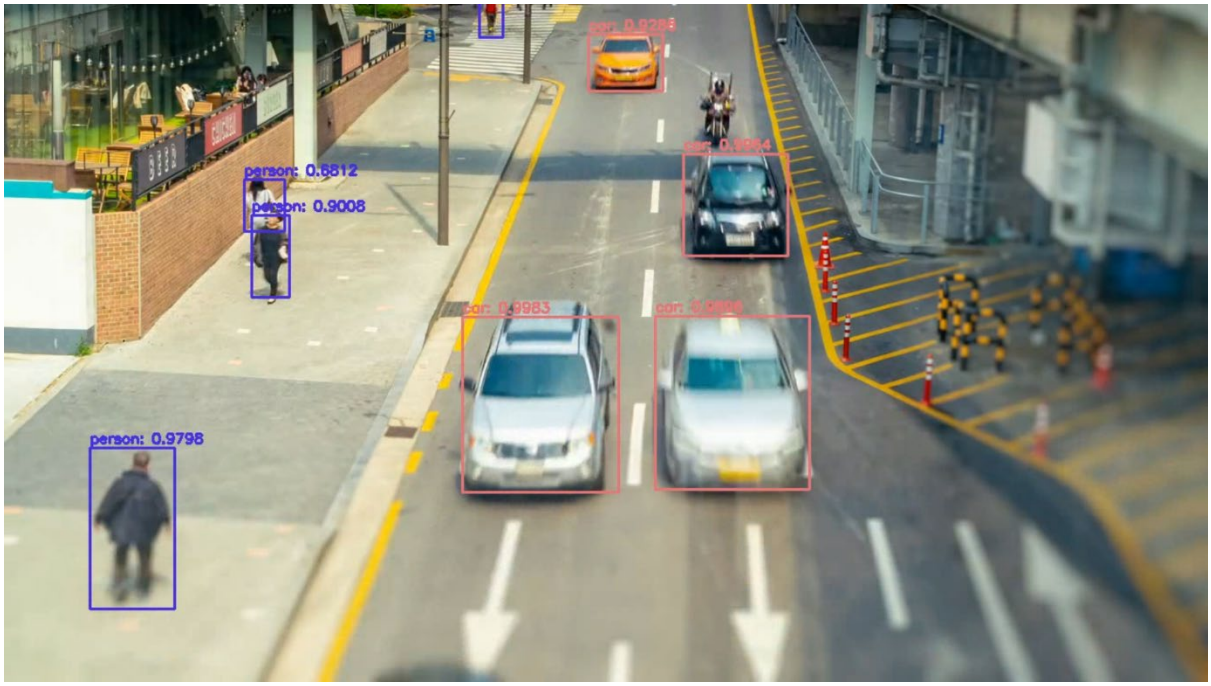


Object Detection on Video with YOLOv3:

In case of a video file, we've to convert it to separate images (frames), implement the same algorithm to each and every frame to detect objects in them and finally merge the resulting frames to create a video file with detected objects. We use ffmpeg command line tool to convert the video to set of images.

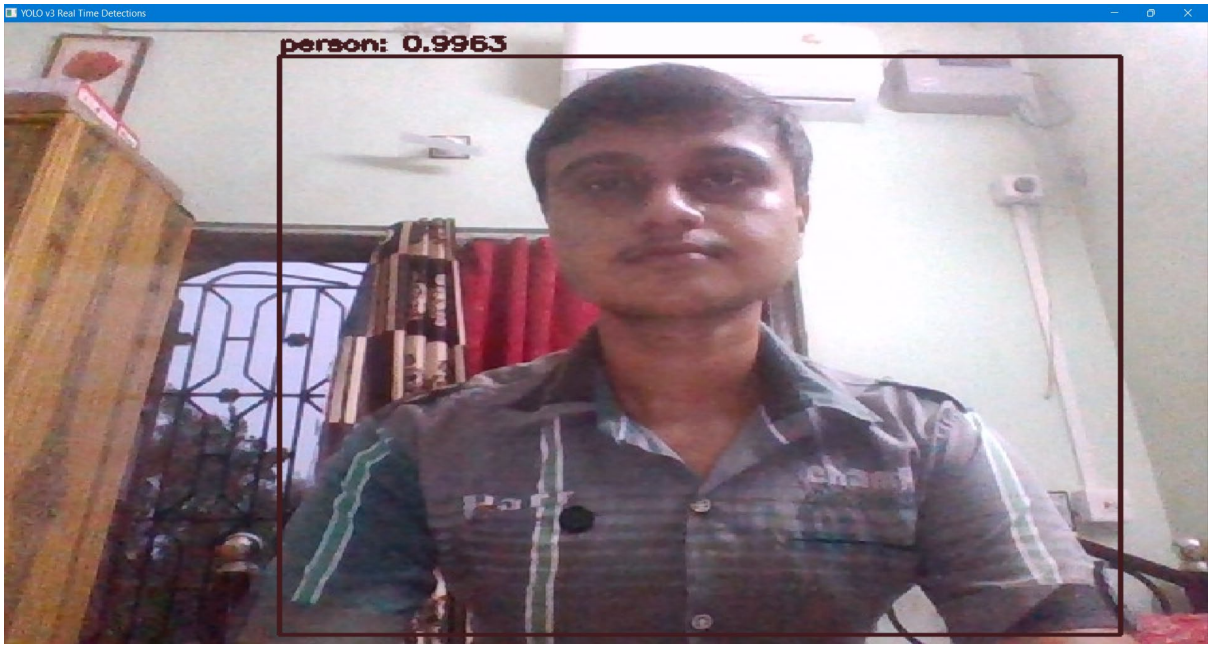
Command:
ffmpeg -i filename -vf fps=4 image-%d.jpeg





Object Detection in Real-Time with YOLOv3:

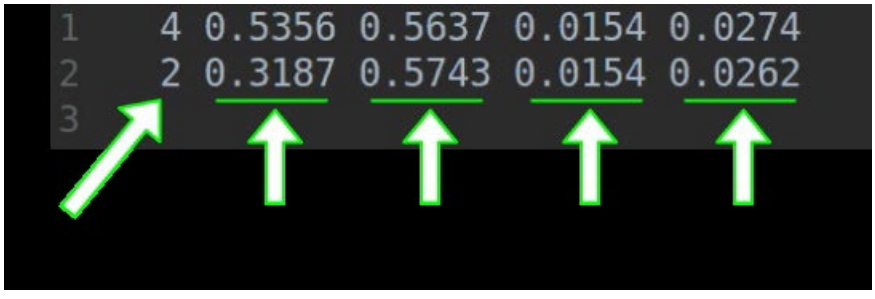
The procedure is exactly the same as that for a video file, the only difference is, instead of taking input as the path of a video file, we've to take the real-time input from camera, convert the real-time to frames and implement our algorithm to it.



Labelling new dataset in YOLO format:

Labelled Image in YOLO Format:

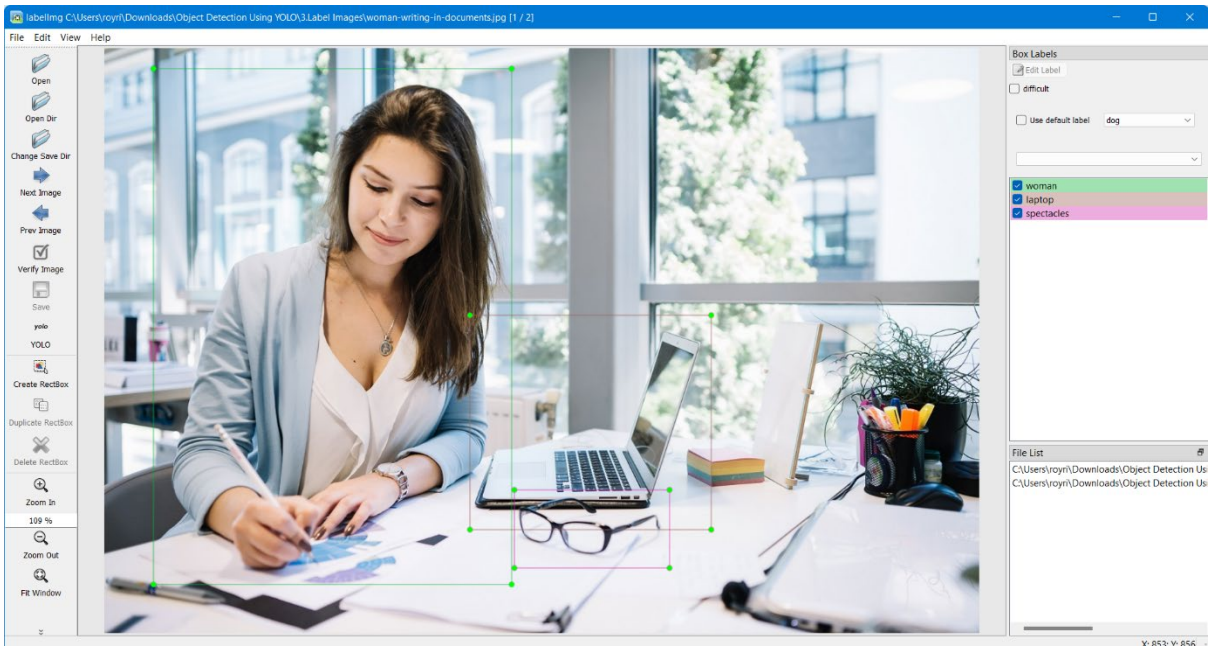
Every image has to have **text file** with **the same name** as image file has:
own-dataset/
image001.jpg
image001.txt
Inside text file **every line** describes **only one object** and consists of *class number*, *object centre in x*, *object centre in y*, *object width* and *object height*.



Class number	Object centre in x	Object centre in y	Object width	Object height

Numbers for *centre point in x*, *centre point in y*, *object width* and *object height* need to be **in range from 0 to 1**. They are **normalized by** real **image width** and real **image height** consequently.

We use LabelImg tool to label our images in YOLO format.



Preparing Files for Training:

After all images were labelled, it's time to prepare other files needed for training in *Darknet framework*.

These files are:
-labelled_data.data
-classes.names
-train.txt
-test.txt

Five lines inside *labelled_data.data* are:
•classes = 2
•train = /home/my_name/**train.txt**
•valid = /home/my_name/**test.txt**
•names = /home/my_name/**classes.names**
•backup = backup

First line specifies number of classes, namely, number of labelled objects that *YOLO v3* will be trained on, and that will be used for detection after training.

Second line specifies full path to the file *train.txt* that in turn consists of full paths to the images for training. The same is true for **third line** with difference that images are used for validation during training.

Fourth line specifies full path to the file *classes.names* that has names of labelled objects.

Fifth line specifies folder where trained weights will be saved.

Files *train.txt* and *test.txt* look like following (every path is in a new line):

- /home/my_name/labelled-images/image001.jpg
- /home/my_name/labelled-images/image002.jpg
- /home/my_name/labelled-images/image003.jpg
- ...
- /home/my_name/labelled-images/image799.jpg
- /home/my_name/labelled-images/image800.jpg

File *classes.names* looks like following (classes' names and their number can be different):

- Motorbike
- Car

Creating Custom dataset in YOLO format:

We use OIDv4 toolkit to download images from huge dataset.

- **Download images:**

To start download images, run following command in *Terminal* (or *Anaconda Prompt*):

```
python3 main.py downloader --classes Car Bicycle_wheel Bus --type_csv train --multiclasses 1 --limit 800
```

or:

```
python main.py downloader --classes Car Bicycle_wheel Bus --type_csv train --multiclasses 1 --limit 800
```

Used arguments in the example above:

- ***--classes Car Bicycle_wheel Bus***

names of the classes (pay attention, here we have class name that consists of two words and we need to use bottom dash character to connect them instead of using the space).

• ***--type_csv train***
specifying the type of dataset (train, validation and test; or all).

• ***--multiclasses 1***
specifying that all classes will be downloaded together in one folder.

• ***--limit 800***
specifying the number of images that will be downloaded for every class.

- **Verify by visualizer:**

In order to verify annotations, we simply launch ***visualizer*** that will show images and bounding boxes, by following command in *Terminal* (or *Anaconda Prompt*):
python3 main.py visualizer
or:
python main.py visualizer

We follow the prompts and type needed folder which is ***train*** in the example above and needed class to visualize which is the name of the folder that contains all three classes ***Car_Bicycle wheel_Bus***. By using ***d*** and ***a*** go next and previous between images. By using ***q*** exit from the visualizer.

- **Converting downloaded files into YOLO format:**

After downloading images and annotations from *Open Images Dataset*, it is needed to convert given annotations into YOLO format. Annotations of bounding boxes' coordinates in *csv* file are as following:

XMin **XMax** **YMin** **YMax**

but YOLO needs following:

[centre in x] **[centre in y]** **[width]** **[height]**

- **Preparing files for training:**

After all images were downloaded and annotations were converted, it's time to prepare other files needed for training in *Darknet framework*.

Converting Traffic Signs Dataset in YOLO Format:

Here, we use German Traffic Signs benchmark datasets and results for YOLO problems.

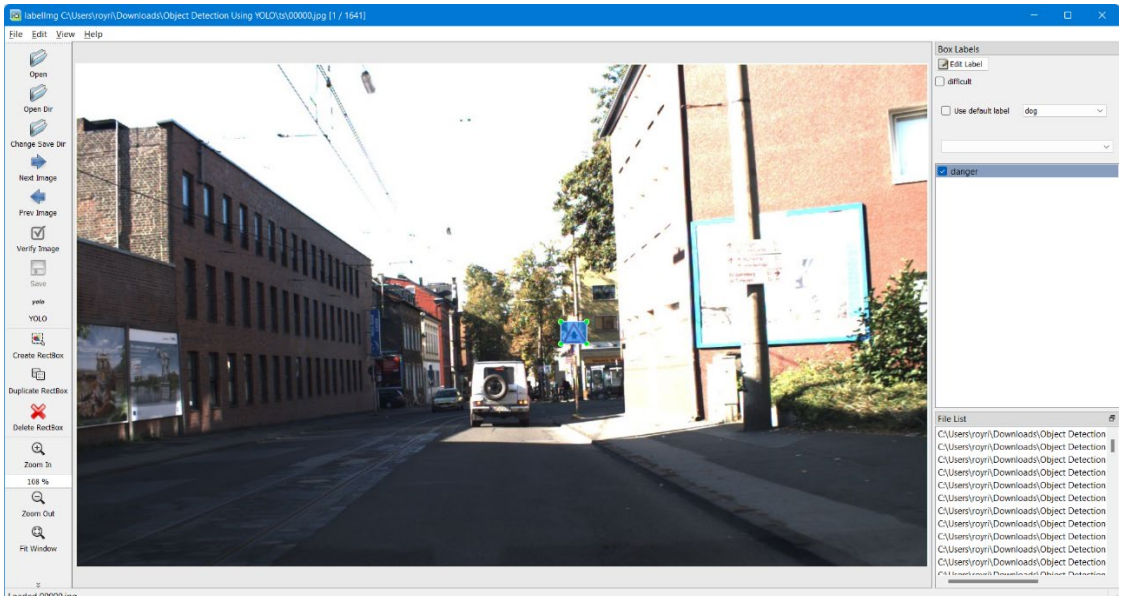
Recognition of traffic signs is a challenging real-world problem relevant for intelligent transportation systems. It is a multi-category classification problem with unbalanced class frequencies. Traffic signs show a wide range of variations between classes in terms of colour, shape, and the presence of pictograms or text. However, there exist subsets of classes (e.g., speed limit signs) that are very similar to each other. Further, the classifier has to cope with large variations in visual appearances due to illumination changes, partial occlusions, rotations, weather conditions etc.

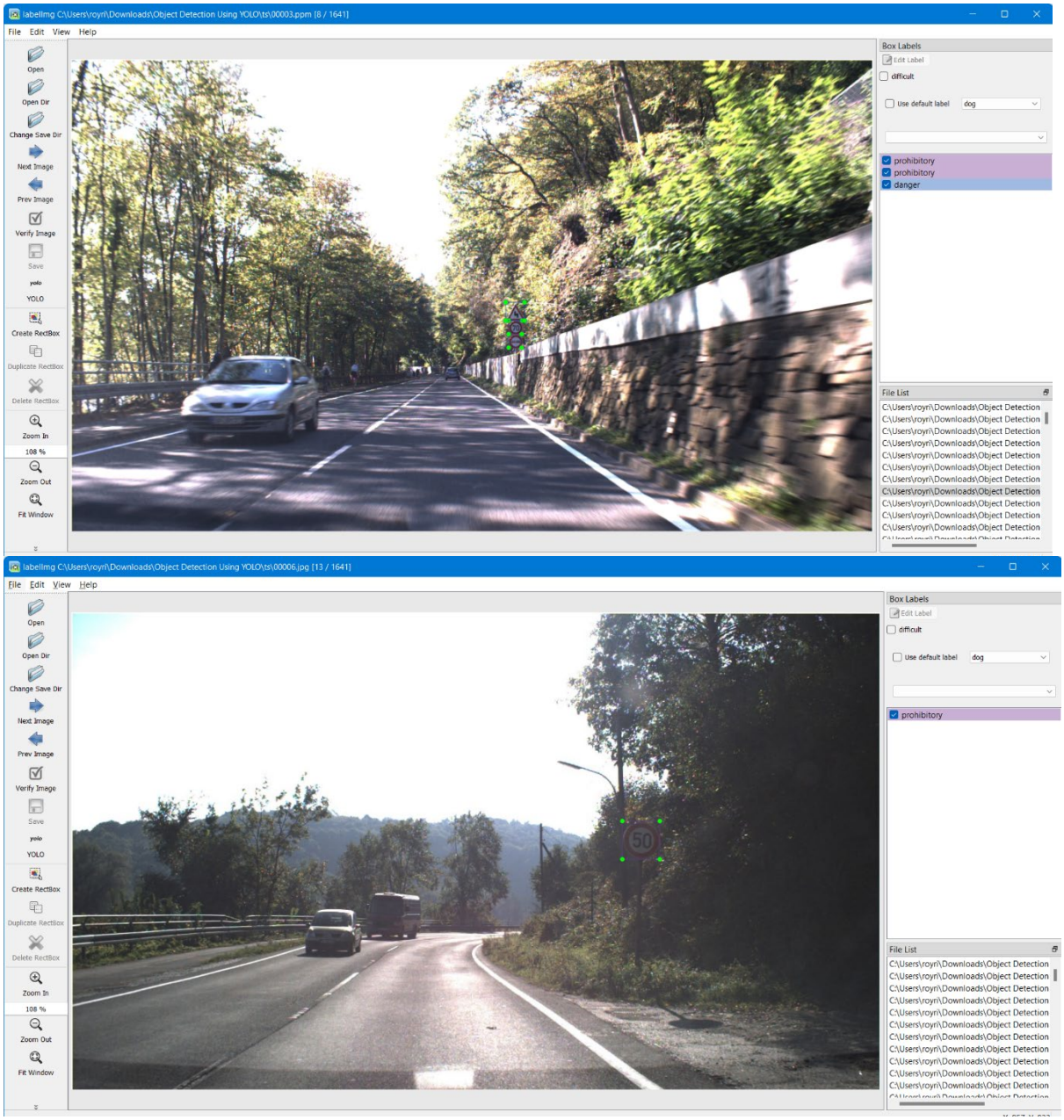
We have to categorize the downloaded Traffic Signs into 4 groups:

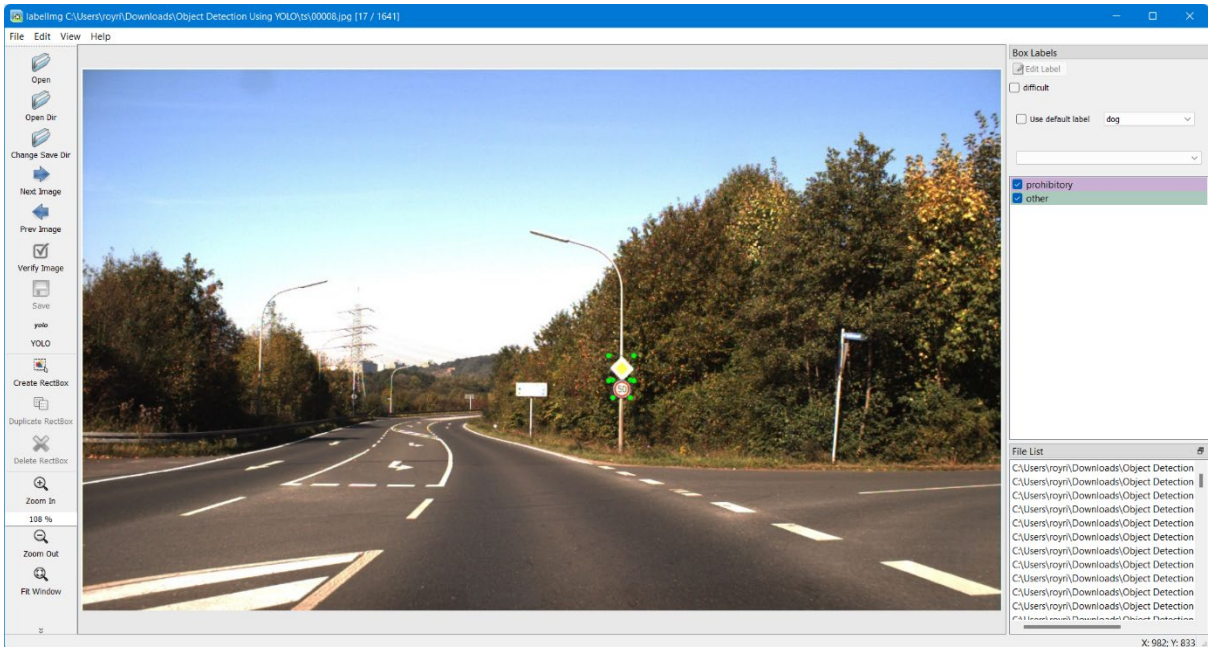
- Prohibitory
- Danger
- Mandatory
- Other

After downloading images of Traffic Signs and annotations, it is needed to convert given annotations into YOLO format.

When images of *Traffic Signs* were downloaded and annotations were converted, it's time to prepare certain files needed for training in *Darknet framework*.




labelImg C:\Users\royri\Downloads\Object Detection Using YOLO\ts\00003.ppm [8 / 1641]
File Edit View Help
Open
Open Dir
Change Save Dir
Next Image
Prev Image
Verify Image
Save
YOLO
Create RectBox
Duplicate RectBox
Delete RectBox
Zoom In
108 %
Zoom Out
Fit Window
Box Labels
Edit Label
difficult
Use default label
dog
prohibitory
prohibitory
danger
File List
C:\Users\royri\Downloads\Object Detection
labelImg C:\Users\royri\Downloads\Object Detection Using YOLO\ts\00006.jpg [13 / 1641]
File Edit View Help
Open
Open Dir
Change Save Dir
Next Image
Prev Image
Verify Image
Save
YOLO
Create RectBox
Duplicate RectBox
Delete RectBox
Zoom In
108 %
Zoom Out
Fit Window
Box Labels
Edit Label
difficult
Use default label
dog
prohibitory
File List
C:\Users\royri\Downloads\Object Detection



Training YOLO v3 in Darknet Framework:

Darknet is an open-source neural network framework written in C and CUDA. It is fast, easy to install, and supports CPU and GPU computation. Darknet is mainly for Object Detection, and have different architecture, features than other deep learning frameworks. It is faster than many other NN architectures and approaches like FasterRCNN etc. We have to be in C if you need speed, and most of the deep NN frameworks are written in c. Tensorflow has a broader scope, but Darknet architecture & YOLO is a specialized framework, and they are on top of their game in speed and accuracy. YOLO can run on CPU but we get 500 times more speed on GPU as it leverages CUDA and cuDNN.

Commands to detect objects on image:
General detection of objects on image:

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector test cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights data/test-image.jpg

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector test cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights data\test-image.jpg

Thresholding while detecting:
In order to **threshold objects** with weak predictions, add *thresholding argument* at the end of command *before image path* and specify *thresholding rate*: -thresh 0.85

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector test cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights -thresh 0.85 data/test-image.jpg

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector test cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights -thresh 0.85 data\test-image.jpg

Coordinates of detected objects:
In order to **print coordinates** of bounding boxes around detected objects, add following *argument* at the end of command *before image path*: -ext_output

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector test cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights -ext_output data/test-image.jpg

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector test cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights -ext_output data\test-image.jpg

Commands to detect objects on video:
Resulted and processed video will be saved in root directory of *Darknet framework*: result.avi

General detection of objects on video:

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector demo cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights data/test-video.mp4 -out_filename result.avi

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector demo cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights data\test-video.mp4 -out_filename result.avi

Thresholding while detecting:

In order to **threshold objects** with weak predictions, add *thresholding argument* at the end of command *before path to video* and specify *thresholding rate*: -thresh 0.85

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector demo cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights -thresh 0.85 data/test-video.mp4 -out_filename result.avi

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector demo cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights -thresh 0.85 data\test-video.mp4 -out_filename result.avi

Switch off window while processing video:
In order to **switch off window** with processed frames while detecting, add *argument* at the end of command *before path to video*: -dont_show

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector demo cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights -dont_show data/test-video.mp4 -out_filename result.avi

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector demo cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights -dont_show data\test-video.mp4 -out_filename result.avi

Commands to detect objects in real time by camera:
General detection of objects in real time:

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector demo cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights -c 0

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector demo cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights -c 0

Thresholding while detecting:
In order to **threshold objects** with weak predictions, add *thresholding argument* at the end of command *before camera* and specify *thresholding rate*: -thresh 0.85

• **For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:

./darknet detector demo cfg/coco.data cfg/yolov3.cfg weights/yolov3.weights -thresh 0.85 -c 0

• **For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector demo cfg\coco.data cfg\yolov3.cfg weights\yolov3.weights -thresh 0.85 -c 0

- max_batches is updated in the cfg file according to the number of classes. General equation is as following:
  max_batches = classes * 2000
  (But not less than 4000)
- steps are calculated as 80% and 90% from max_batches.
- General equation that represents how to calculate proper number of *filters* in three *[convolutional]* layers right before every of three *[yolo]* layers is as following: filters = (classes + coordinates + 1) * masks

Training on Traffic Signs dataset:

•**For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector train cfg/ts_data.data cfg/yolov3_ts_train.cfg weights/darknet53.conv.74

•**For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector train cfg\ts_data.data cfg\yolov3_ts_train.cfg weights\darknet53.conv.74

Continue training with saved weights after 1000 iterations:

It is possible to stop training, for example, after 1000 iteration and continue later by using already saved weights. In order to continue training just specify at the end of command location of needed weights to continue training from.

•**For Linux and MacOS** navigate to root directory where *Darknet framework* was installed and type in following command:
./darknet detector train cfg/ts_data.data cfg/yolov3_ts_train.cfg backup/yolo-obj_1000.weights

•**For Windows** navigate to *darknet\build\darknet\x64* and type in following command:
darknet.exe detector train cfg\ts_data.data cfg\yolov3_ts_train.cfg backup\yolo-obj_1000.weights

When do we stop training?

start checking saved and trained weights from the end one by one calculating *mean average precision (mAP)*. The goal is to find *weights* that have the biggest *mAP*.

Find the biggest mAP

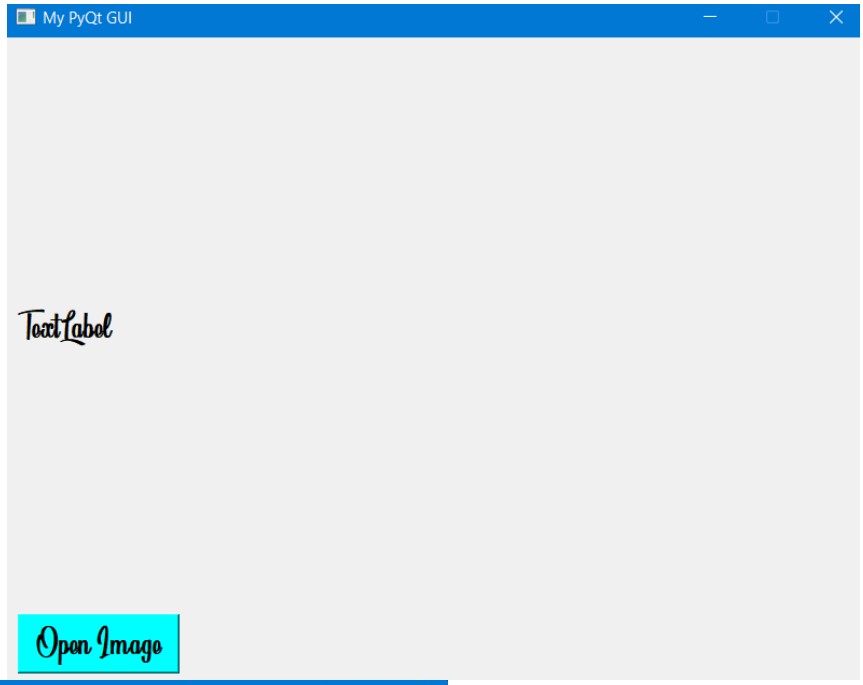
yolo-obj_8000.weights
yolo-obj_7000.weights
yolo-obj_6000.weights

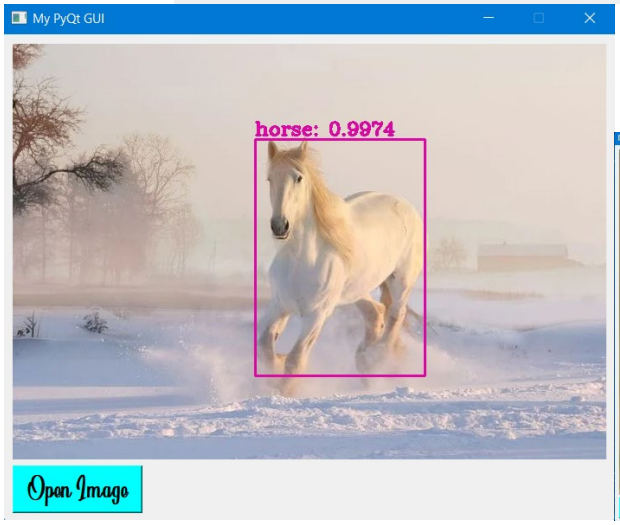
For example, if we consider *Traffic Signs* dataset, and if *mAP* for *7000 iterations* is bigger than for *8000*, then it is needed to check *weights* for *6000 iterations*. Next, if *mAP* for *6000 iterations* is already less than for *7000 iterations*, then you can stop checking and use weights for *7000 iterations* in detection tasks. Also, it is possible to continue checking *weights* between *6000* and *7000 iterations*, trying to find *weights* even with bigger *mAP*.

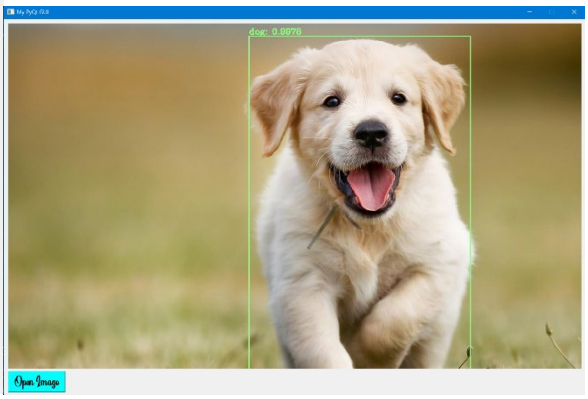
Building PyQT UI for Object Detection with YOLOv3:

PyQt is the Python bindings for *Qt* application development framework. **Qt**, in turn, is a free and open-source widget toolkit for creating graphical user interfaces.

We've created the PyQT interface and integrated YOLOv3 into it.







Sample Code:

```
"""
Objects Detection on Image with YOLO v3 and OpenCV
File: yolo-3-image.py
"""


# Detecting Objects on Image with OpenCV deep learning library
#
# Algorithm:
# Reading RGB image --> Getting Blob --> Loading YOLO v3 Network -->
# --> Implementing Forward Pass --> Getting Bounding Boxes -->
# --> Non-maximum Suppression --> Drawing Bounding Boxes with Labels
#
# Result:
# Window with Detected Objects, Bounding Boxes and Labels


# Importing needed libraries
import numpy as np
import cv2
import time


"""
Start of:
Reading input image
"""

# Reading image with OpenCV library
# In this way image is opened already as numpy array
# WARNING! OpenCV by default reads images in BGR format
image_BGR = cv2.imread('C:/Users/royri/Downloads/Object Detection Using YOLO/YOLO-
3-OpenCV/images/woman-working-in-the-office.jpg')

# Showing Original Image
# Giving name to the window with Original Image
# And specifying that window is resizable
```

```
cv2.namedWindow('Original Image', cv2.WINDOW_NORMAL)
# Pay attention! 'cv2.imshow' takes images in BGR format
cv2.imshow('Original Image', image_BGR)
# Waiting for any key being pressed
cv2.waitKey(0)
# Destroying opened window with name 'Original Image'
cv2.destroyWindow('Original Image')

# # Check point
# # Showing image shape
print('Image shape:', image_BGR.shape)  # tuple of (511, 767, 3)

# Getting spatial dimension of input image
h, w = image_BGR.shape[:2]  # Slicing from tuple only first two elements

# # Check point
# # Showing height an width of image
print('Image height={0} and width={1}'.format(h, w))  # 511 767

"""
End of:
Reading input image
"""


"""
Start of:
Getting blob from input image
"""

# Getting blob from input image
# The 'cv2.dnn.blobFromImage' function returns 4-dimensional blob
# from input image after mean subtraction, normalizing, and RB channels swapping
# Resulted shape has number of images, number of channels, width and height
# E.G.:
# blob = cv2.dnn.blobFromImage(image, scalefactor=1.0, size, mean, swapRB=True)
blob = cv2.dnn.blobFromImage(image_BGR, 1 / 255.0, (416, 416),
                             swapRB=True, crop=False)
```

```
# Check point
# print('Image shape:', image_BGR.shape)  # (511, 767, 3)
# print('Blob shape:', blob.shape)  # (1, 3, 416, 416)

# Check point
# Showing blob image in OpenCV window
# Slicing blob image and transposing to make channels come at the end
blob_to_show = blob[0, :, :, :].transpose(1, 2, 0)
print(blob_to_show.shape)  # (416, 416, 3)

# Showing Blob Image
# Giving name to the window with Blob Image
# And specifying that window is resizable
cv2.namedWindow('Blob Image', cv2.WINDOW_NORMAL)
# Pay attention! 'cv2.imshow' takes images in BGR format
# Consequently, we DO need to convert image from RGB to BGR firstly
# Because we have our blob in RGB format
cv2.imshow('Blob Image', cv2.cvtColor(blob_to_show, cv2.COLOR_RGB2BGR))
# Waiting for any key being pressed
cv2.waitKey(0)
# Destroying opened window with name 'Blob Image'
cv2.destroyWindow('Blob Image')

"""
End of:
Getting blob from input image
"""


"""
Start of:
Loading YOLO v3 network
"""

# Loading COCO class labels from file
# Opening file
with open('C:/Users/royri/Downloads/Object Detection Using YOLO/YOLO-3-
OpenCV/yolo-coco-data/coco.names') as f:
    # Getting labels reading every line
```

```
    # and putting them into the list
    labels = [line.strip() for line in f]


# # Check point
# print('List with labels names:')
# print(labels)

# Loading trained YOLO v3 Objects Detector
# with the help of 'dnn' library from OpenCV
network = cv2.dnn.readNetFromDarknet('C:/Users/royri/Downloads/Object Detection
Using YOLO/YOLO-3-OpenCV/yolo-coco-data/yolov3.cfg',
                                     'C:/Users/royri/Downloads/Object Detection
Using YOLO/YOLO-3-OpenCV/yolo-coco-data/yolov3.weights')

# Getting list with names of all layers from YOLO v3 network
layers_names_all = network.getLayerNames()

# # Check point
# print()
# print(layers_names_all)

# Getting only output layers' names that we need from YOLO v3 algorithm
# with function that returns indexes of layers with unconnected outputs
layers_names_output = \
    [layers_names_all[i - 1] for i in network.getUnconnectedOutLayers()]

# # Check point
# print()
# print(layers_names_output)  # ['yolo_82', 'yolo_94', 'yolo_106']

# Setting minimum probability to eliminate weak predictions
probability_minimum = 0.5

# Setting threshold for filtering weak bounding boxes
# with non-maximum suppression
threshold = 0.3

# Generating colours for representing every detected object
```

```
# with function randint(low, high=None, size=None, dtype='l')
colours = np.random.randint(0, 255, size=(len(labels), 3), dtype='uint8')

# # Check point
# print()
# print(type(colours))  # <class 'numpy.ndarray'>
# print(colours.shape)  # (80, 3)
# print(colours[0])  # [172  10 127]

"""
End of:
Loading YOLO v3 network
"""


"""
Start of:
Implementing Forward pass
"""

# Implementing forward pass with our blob and only through output layers
# Calculating at the same time, needed time for forward pass
network.setInput(blob)  # setting blob as input to the network
start = time.time()
output_from_network = network.forward(layers_names_output)
end = time.time()

# Showing spent time for forward pass
print('Objects Detection took {:.5f} seconds'.format(end - start))

"""
End of:
Implementing Forward pass
"""


"""
Start of:
Getting bounding boxes
```

```
"""

# Preparing lists for detected bounding boxes,
# obtained confidences and class's number
bounding_boxes = []
confidences = []
class_numbers = []


# Going through all output layers after feed forward pass
for result in output_from_network:
    # Going through all detections from current output layer
    for detected_objects in result:
        # Getting 80 classes' probabilities for current detected object
        scores = detected_objects[5:]
        # Getting index of the class with the maximum value of probability
        class_current = np.argmax(scores)
        # Getting value of probability for defined class
        confidence_current = scores[class_current]

        # # Check point
        # # Every 'detected_objects' numpy array has first 4 numbers with
        # # bounding box coordinates and rest 80 with probabilities for every
class
        # print(detected_objects.shape)  # (85,)

        # Eliminating weak predictions with minimum probability
        if confidence_current > probability_minimum:
            # Scaling bounding box coordinates to the initial image size
            # YOLO data format keeps coordinates for center of bounding box
            # and its current width and height
            # That is why we can just multiply them elementwise
            # to the width and height
            # of the original image and in this way get coordinates for center
            # of bounding box, its width and height for original image
            box_current = detected_objects[0:4] * np.array([w, h, w, h])

            # Now, from YOLO data format, we can get top left corner coordinates
            # that are x_min and y_min
```

```
            x_center, y_center, box_width, box_height = box_current
            x_min = int(x_center - (box_width / 2))
            y_min = int(y_center - (box_height / 2))

            # Adding results into prepared lists
            bounding_boxes.append([x_min, y_min, int(box_width), int(box_height)])
            confidences.append(float(confidence_current))
            class_numbers.append(class_current)

"""
End of:
Getting bounding boxes
"""


"""
Start of:
Non-maximum suppression
"""

# Implementing non-maximum suppression of given bounding boxes
# With this technique we exclude some of bounding boxes if their
# corresponding confidences are low or there is another
# bounding box for this region with higher confidence

# It is needed to make sure that data type of the boxes is 'int'
# and data type of the confidences is 'float'
results = cv2.dnn.NMSBoxes(bounding_boxes, confidences,
                           probability_minimum, threshold)

"""
End of:
Non-maximum suppression
"""


"""
Start of:
Drawing bounding boxes and labels
```

```
"""

# Defining counter for detected objects
counter = 1

# Checking if there is at least one detected object after non-maximum suppression
if len(results) > 0:
    # Going through indexes of results
    for i in results.flatten():
        # Showing labels of the detected objects
        print('Object {0}: {1}'.format(counter, labels[int(class_numbers[i])]))

        # Incrementing counter
        counter += 1

        # Getting current bounding box coordinates,
        # its width and height
        x_min, y_min = bounding_boxes[i][0], bounding_boxes[i][1]
        box_width, box_height = bounding_boxes[i][2], bounding_boxes[i][3]

        # Preparing colour for current bounding box
        # and converting from numpy array to list
        colour_box_current = colours[class_numbers[i]].tolist()

        # # # Check point
        # print(type(colour_box_current))  # <class 'list'>
        # print(colour_box_current)  # [172 , 10, 127]

        # Drawing bounding box on the original image
        cv2.rectangle(image_BGR, (x_min, y_min),
                      (x_min + box_width, y_min + box_height),
                      colour_box_current, 2)

        # Preparing text with label and confidence for current bounding box
        text_box_current = '{}: {:.4f}'.format(labels[int(class_numbers[i])],
                                               confidences[i])

        # Putting text with label and confidence on the original image
```

```
        cv2.putText(image_BGR, text_box_current, (x_min, y_min - 5),
                    cv2.FONT_HERSHEY_COMPLEX, 0.7, colour_box_current, 2)


# Comparing how many objects where before non-maximum suppression
# and left after
print()
print('Total objects been detected:', len(bounding_boxes))
print('Number of objects left after non-maximum suppression:', counter - 1)

"""
End of:
Drawing bounding boxes and labels
"""


# Showing Original Image with Detected Objects
# Giving name to the window with Original Image
# And specifying that window is resizable
cv2.namedWindow('Detections', cv2.WINDOW_NORMAL)
# Pay attention! 'cv2.imshow' takes images in BGR format
cv2.imshow('Detections', image_BGR)
# Waiting for any key being pressed
cv2.waitKey(0)
# Destroying opened window with name 'Detections'
cv2.destroyWindow('Detections')
```

Conclusion:

To summarize:

- We have gained an overview of object detection and the YOLO algorithm.
- We have gone through the main reasons why the YOLO algorithm is important.
- We have learned how the YOLO algorithm works. We have also gained an understanding of the main techniques used by YOLO to detect objects.
- We have learned the real-life applications of YOLO.

Acknowledgement:

We would like to thank our supervisor, Professor Arindam Biswas Sir for his continuous help, guidance and mentorship over the course of this project's timeline.

We would also like to thank our friends for their help with formatting and designing the report.

References:

1. You Only Look Once: Unified, Real-Time Object Detection: Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi.

2. YOLOv3: An Incremental Improvement: Joseph Redmon, Ali Farhadi.

3. Alexe, B., Deselaers, T., and Ferrari, V. (2010). "What is an object?" in ComputerVision and Pattern Recognition (CVPR), 2010 IEEE Conference on (San Francisco, CA: IEEE), 73–80. doi:10.1109/CVPR.2010.5540226.

4. Dr. Rachna Verma, 2017, A Review of Object Detection and Tracking Methods, International Journal of Advanced Engineering and Research Development, Volume 4, Issue 10.

5. Afzal Godil, Roger Bostelman, Will Shackleford, Tsai Hong, Michael Shneier, 2014, Performance Metrics for Evaluating Object and Human Detection and Tracking Systems, National Institute of Standards and Technology, US Department of Commerce.

6. YOLO-Real Time Object Detection, PJ Reddie, Official Website https://pjreddie.com/darknet/yolo/