INTRODUCTION

What is Deep Learning?

- Deep learning is a subset of machine learning, which is essentially a neural network with three or more layers. These neural networks attempt to simulate the behavior of the human brain—albeit far from matching its ability—allowing it to "learn" from large amounts of data. While a neural network with a single layer can still make approximate predictions, additional hidden layers can help to optimize and refine for accuracy.

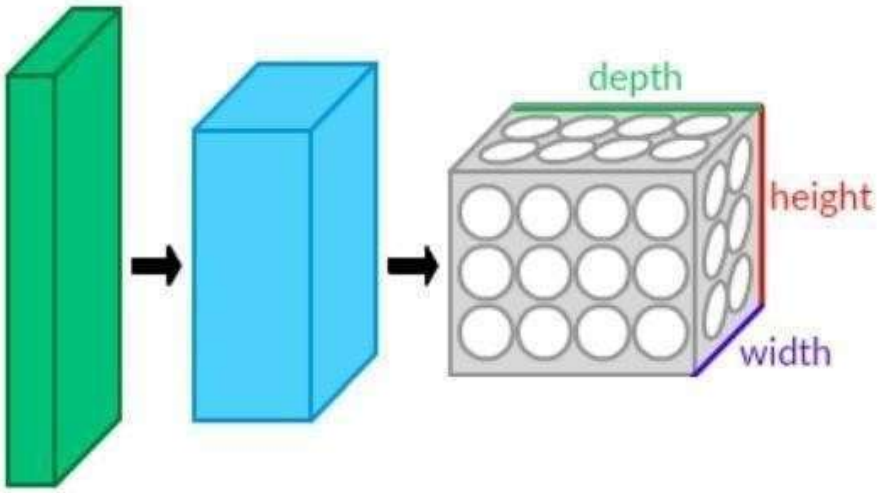
- Deep learning neural networks, or artificial neural networks, attempts to mimic the human brain through a combination of data inputs, weights, and bias. These elements work together to accurately recognize, classify, and describe objects within the data.

CNN(Convolutional Neural Network)

In neural networks, Convolutional neural networks (ConvNets or CNNs) is one of the main categories to do image recognition, images classifications. Objects detections, recognition face etc., are some of the areas where CNNs are widely used. CNN image classifications take an input image, process it and classify it under certain categories. Computers see an input image as an array of pixels and it depends on the image resolution. Based on the image resolution, it will see h x w x d(h = Height, w = Width, d = Dimension). Eg., An image of 6 x 6 x 3 array of matrix of RGB (3 refers to RGB values) and an image of 4 x 4 x 1 array of matrix of grayscale image. A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that convolve with a multiplication or other dot product. The activation function is commonly a RELU layer, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution.

Layers In Convolutional Neural Network

1. Convolutional layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.
2. The RELU layer will apply an element wise activation function, such as the max(0,x) thresholding at zero.
3. POOL layer will perform a down sampling operation along the spatial dimensions (width, height).
4. FULLY connected layer will compute the class scores, resulting in a volume of size.



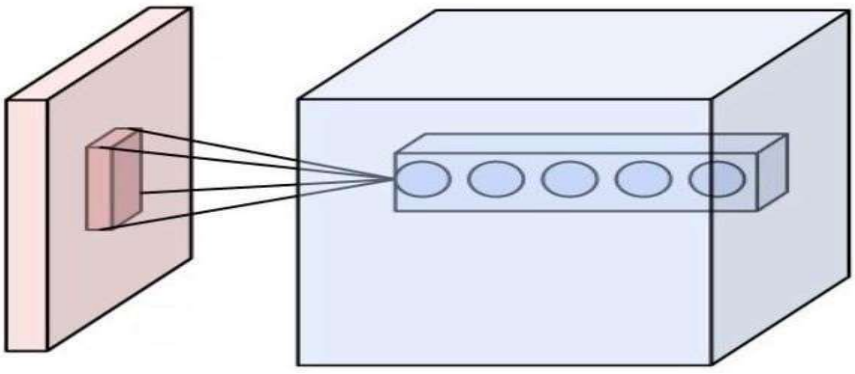
[1x1xX], where each of the X numbers correspond to a class score

The layers of a CNN have neurons arranged in 3 dimensions: width, height and depth

Convolutional Layer

When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through

the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input. When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume because such a network architecture does not take the spatial structure of the data into account. Convolutional networks exploit spatially local correlation by enforcing a sparse local connectivity pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume .The extent of this connectivity is a hyperparameter called the receptive field of the neuron. The connections are local in space (along width and height), but always extend along the entire depth of the input volume. Such an architecture ensures that the learnt filters produce the strongest response to a spatially local input pattern.



Neuron of a convolutional layer(blue), connected to their receptive field(red)

Convolutional layer within a neural network should have the following attributes:

- Convolutional kernels defined by a width and height (hyper-parameters).
- The number of input channels and output channels (hyper-parameter).
- The depth of the Convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map.

Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus.

SPATIAL ARRANGEMENT

The depth of the output volume controls the number of neurons in a layer that connect to the same region of the input volume. These neurons learn to activate for different features in the input. For example, if the first convolutional layer takes the raw image as input, then different neurons along the depth dimension may activate in the presence of various oriented edges, or blobs of color.

Stride controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1 then we move the filters one pixel at a time. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. When the stride is 2 then the filters jump 2 .

Pixels at a time as they slide around.Similarly, for any integer S>0,a stride of S causes the filter to be translated by S units at a time per output.In practice, stride length of S>=3 are rare.The receptive fields overlap less and the resulting output volume has smaller spatial dimensions when stride length is increased. Sometimes it is convenient to pad the input with zeros on the border of the input volume. The size of this padding is a third hyperparameter. Padding provides control of the output volume spatial size. In particular, sometimes it is desirable to exactly preserve the spatial size of the input volume.

PARAMETER SHARING

A parameter sharing scheme is used in convolutional layers to control the number of free parameters. It relies on the assumption that if a patch feature is useful to compute at some spatial position, then it should also be useful to compute at other positions. Denoting a single 2-dimensional slice of depth as a depth slice, the neurons in each depth slice are constrained to use the same weights and bias.

Since all neurons in a single depth slice share the same parameters, the forward pass in each depth slice of the convolutional layer can be computed as a convolution of the neuron's weights with the input volume. Therefore, it is common to refer to the sets of weights as a filter (or a kernel), which is convolved with the input. The result of this convolution is an activation map, and the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume. Parameter sharing contributes to the translation invariance of the CNN architecture.

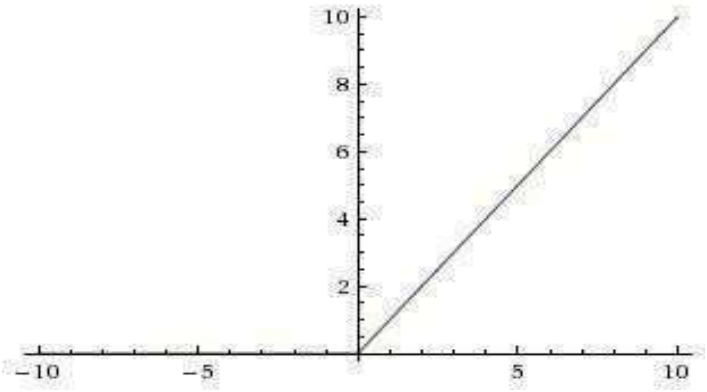
Sometimes, the parameter sharing assumption may not make sense. This is especially the case when the input images to a CNN have some specific centered structure; for which we expect completely different features to be learned on different spatial locations. One practical example is when the inputs are faces that have been centered in the image: we might expect different eye-specific or hair-specific features to be learned in different parts of the image. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a "locally connected layer".

ReLU Layer

ReLU means Rectified Linear Unit, ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning. As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

- Range: max (0 , z)
- But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

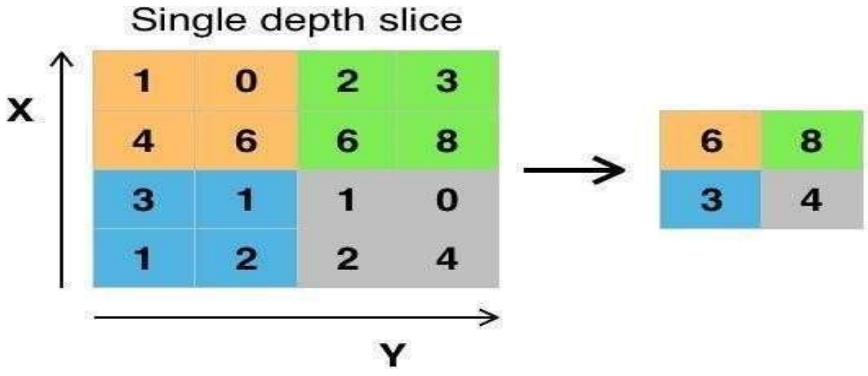
Graphically it looks like this



It's surprising that such a simple function (and one composed of two linear pieces) can allow your model to account for non-linearities and interactions so well. But the ReLU function works great in most applications, and it is very widely used as a result.

POOLING LAYER

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. There are several nonlinear functions to implement pooling among which max pooling is the most common. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. Intuitively, the exact location of a feature is less important than its rough location relative to other features. This is the idea behind the use of pooling in convolutional neural networks. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters, memory footprint and amount of computation in the network, and hence to also control overfitting. It is common to periodically insert a pooling layer between successive convolutional layers (each one typically followed by a ReLU layer) in a CNN architecture. The pooling operation can be used as another form of translation invariance.



The pooling layer operates independently on every depth slice of the input and resizes it spatially. More generally, the pooling layer:

Produces a volume of size W2×H2×D2 where:

- W2=(W1−F)/S+1
- H2=(H1−F)/S+1
- D2=D1
- Introduces zero parameters since it computes a fixed function of the input For Pooling layers, it is not common to pad the input using zero-padding.

FULLY CONNECTED

The objective of a fully connected layer is to take the results of the convolution/pooling process and use them to classify the image into a label . The fully connected part of the CNN network goes through its own back propagation process to determine the most accurate weights. Each neuron receives weights that prioritize the most appropriate label. Finally, the neurons "vote" on each of the labels, and the winner of that vote is the classification decision.

It is worth noting that the only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters.

- For any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing).
- Conversely, any FC layer can be converted to a CONV layer. For example, an FC layer with K=4096 that is looking at some input volume of size 7×7×512 can be equivalently expressed as a CONV layer with F=7,P=0,S=1,K=4096. In other words, we are setting the filter size to be

exactly the size of the input volume, and hence the output will simply be 1×1×4096 since only a single depth column “fits” across the input volume, giving identical results as the initial FC layer.

Darknet

Network Architecture

For understanding the network architecture on a high-level, let’s divide the entire architecture into two major components: Feature Extractor and Feature Detector (Multi-scale Detector). The image is first given to the Feature extractor which extracts feature embeddings and then is passed on to the feature detector part of the network that spits out the processed image with bounding boxes around the detected classes.

Feature Extractor

The previous YOLO versions have used Darknet-19 (a custom neural network architecture written in C and CUDA) as a feature extractor which was of 19 layers as the name suggests. YOLO v2 added 11 more layers to Darknet-19 making it a total 30-layer architecture. Still, the algorithm faced a challenge while detecting small objects due to downsampling the input image and losing fine-grained features.

YOLO V3 came up with a better architecture where the feature extractor used was a hybrid of YOLO v2, Darknet-53 (a network trained on the ImageNet), and Residual networks(ResNet). The network uses 53 convolution layers (hence the name Darknet-53) where the network is built with consecutive 3x3 and 1x1 convolution layers followed by a skip connection (introduced by ResNet to help the activations propagate through deeper layers without gradient diminishing).

The 53 layers of the darknet are further stacked with 53 more layers for the detection head, making YOLO v3 a total of a **106 layer fully convolutional underlying architecture**.

thus leading to a large architecture, though making it a bit slower as compared to YOLO v2, but enhancing the accuracy at the same time.

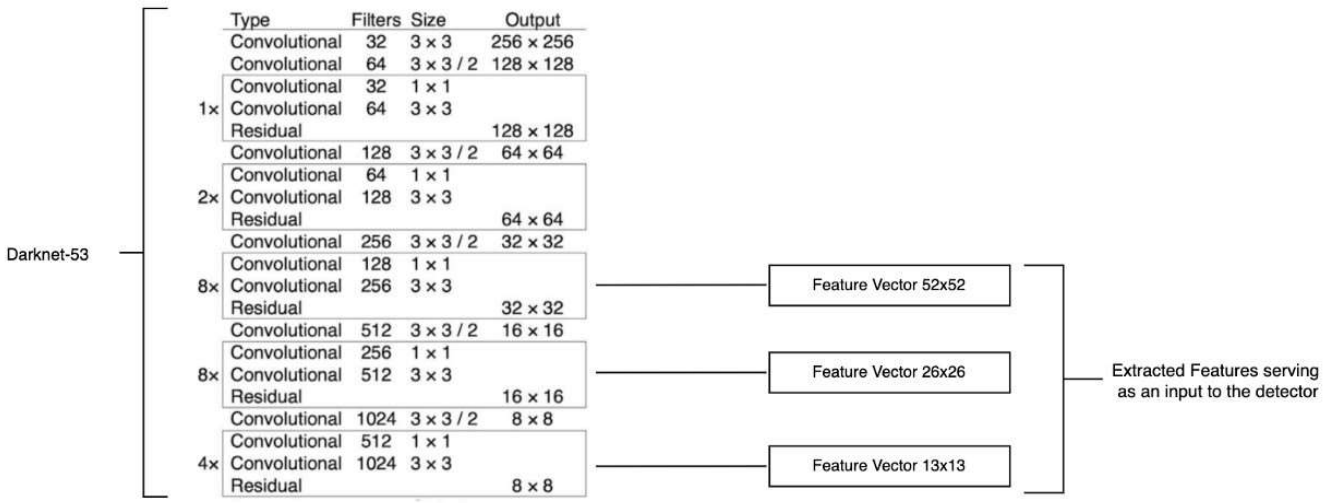
	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 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			

This image is the darknet-53 architecture

If the aim was to perform classification as in the ImageNet, then the Average pool layer, 1000 fully connected layers, and a SoftMax activation function would be added as shown in the image, but in our case, we would like to detect the classes along with the locations, so we would be appending a detection head to the extractor. The detection head is a multi-scale detection head hence, we would need to extract features at multiple scales as well.

We'll take a deeper look at understanding how these layers work in the next section once we familiarize ourselves with the high-level architecture.

For visualizing how the multi-scale extractor would look like, I'm taking an example of a 416x416 image. A stride of a layer is defined as the ratio by which it downsamples the input, and hence the three scales in our case would be **52x52, 26x26,** and **13x13** where 13x13 would be used for larger objects and 26x26 and 52x52 would be used for medium and smaller objects.

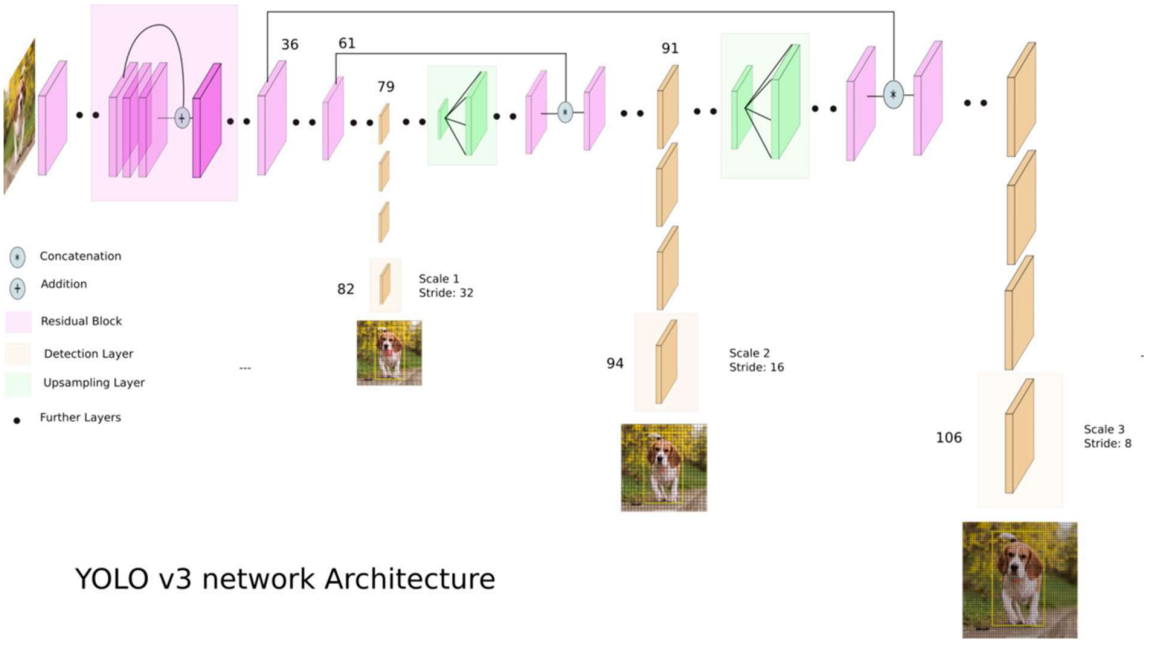


Multi-Scale Detector

An important feature of the YOLO v3 model is its multi-scale detector, which means that the detection for an eventual output of a fully convolutional network is done by applying 1x1 detection kernels on feature maps of three different sizes at three different places. The shape of the kernel is **1*x*1*x*(B*(5+C)).**

Complete Network Architecture

The following is a very elaborate diagram that beautifully explains the complete architecture of YOLO v3 (Combining both, the extractor and the detector).



YOLO v3 network Architecture

As seen in the above image where we take an example of a 416x416 image, the three scales where the detections are made are at the 82nd layer, 94th layer, and 106th layer.

For the first detection, the first 81 layers are downsampled such that the 81st layer has a stride of 32 (as mentioned earlier, a stride of a layer is defined as the ratio by which it downsamples the input) resulting in our first feature map of size 13x13 and the first detection is made with a 1x1 kernel, leading to our detection 3D tensor of size 13x13x255.

For the second detection, the 79th layer onwards is subjected to convolutional layers before upsampling to dimensions 26x26. This feature map is then depth concatenated with the feature map from layer 61 to form a new feature map which is further fused with the 61st layer with the help of 1x1 convolution layers. The second detection layer is at the 94th layer with a 3D tensor of size 26x26x255.

For the final(third) detection layer, the same process is followed as that of the second detection where the feature map of the 91st layer is subjected to convolution layers before being depth concatenated and fused with a feature map from 36th layer. The final detection is made at the 106th layer with a feature map of size 52x52x255.

The multi-scale detector is used to ensure that the small objects are also being detected unlike in YOLO v2, where there was constant criticism regarding the same. Upsampled layers concatenated with the previous layers end up preserving the fine-grained features which help in detecting small objects.

The details of how this kernel looks in our model is described below in the next section.

Working Of YOLO v3

YOLO is an algorithm that uses neural networks to provide real-time object detection. This algorithm is popular because of its speed and accuracy. It has been used in various applications to detect traffic signals, people, parking meters, and animals.

Object detection is a phenomenon in computer vision that involves the detection of various objects in digital images or videos. Some of the objects detected include people, cars, chairs, stones, buildings, and animals.

This phenomenon seeks to answer two basic questions:

1. *What is the object?* This question seeks to identify the object in a specific image.
2. *Where is it?* This question seeks to establish the exact location of the object within the image.

Object detection consists of various approaches such as fast R-CNN, Retina Net, and single-shot multi-box detector (SSD). Although these approaches have solved the challenges of data limitation and modelling in object detection, they are not able to detect objects in a single algorithm run. **YOLO algorithm** has gained popularity because of its superior performance over the aforementioned object detection techniques.

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

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.

ABOUT THE ALGORITHM

YOLO algorithm works using the following three techniques:

- Residual blocks
- Bounding box regression
- Intersection Over Union (IOU)

Residual blocks

First, the image is divided into various grids. Each grid has a dimension of S x S. The following image shows how an input image is divided into grids.

In the image above, there are many grid cells of equal dimension. Every grid cell will detect objects that appear within them. For example, if an object center appears within a certain grid cell, then this cell will be responsible for detecting it.

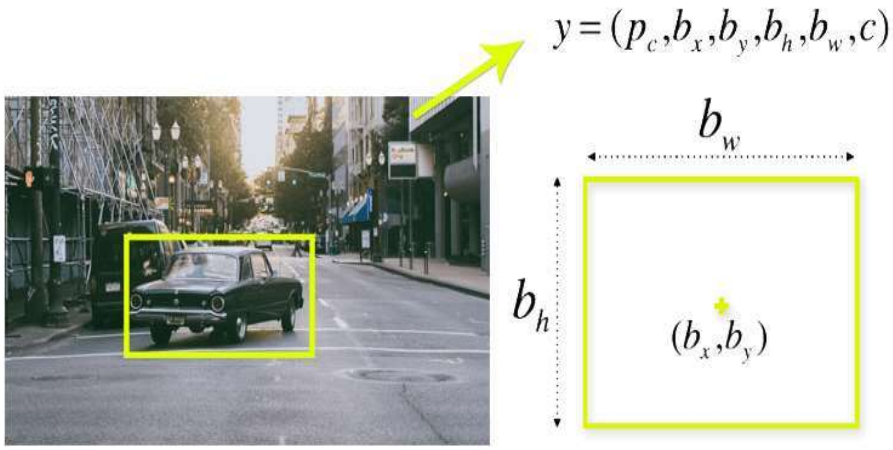
Bounding box regression

A bounding box is an outline that highlights an object in an image.

Every bounding box in the image consists of the following attributes:

- Width (bw)
- Height (bh)
- Class (for example, person, car, traffic light, etc.)- This is represented by the letter c.
- Bounding box center (bx,by)

The following image shows an example of a bounding box. The bounding box has been represented by a yellow outline



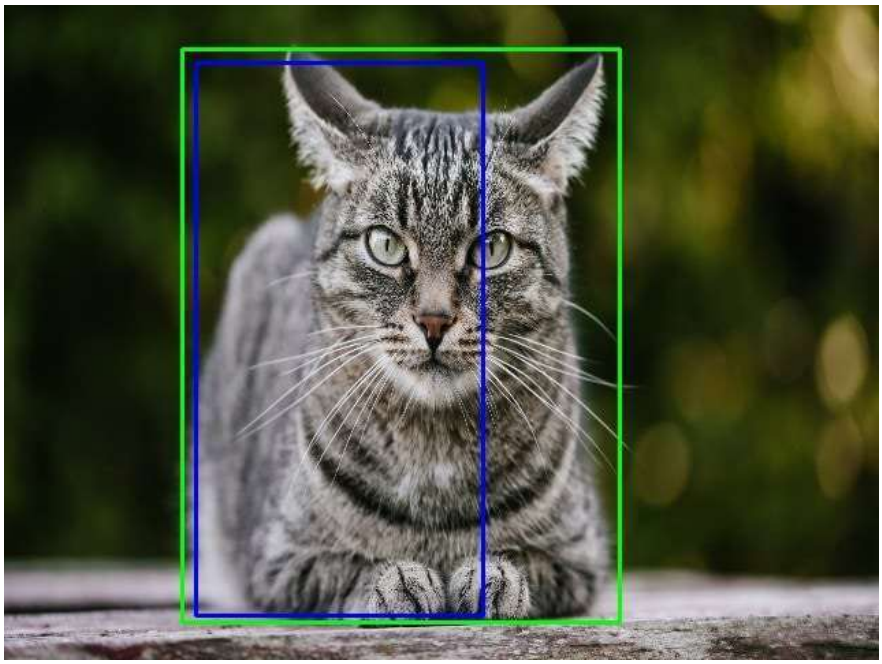
YOLO uses a single bounding box regression to predict the height, width, center, and class of objects. In the image above, represents the probability of an object appearing in the bounding box.

Intersection over union (IOU)

Intersection over union (IOU) is a phenomenon in object detection that describes how boxes overlap. YOLO uses IOU to provide an output box that surrounds the objects perfectly.

Each grid cell is responsible for predicting the bounding boxes and their confidence scores. The IOU is equal to 1 if the predicted bounding box is the same as the real box. This mechanism eliminates bounding boxes that are not equal to the real box.

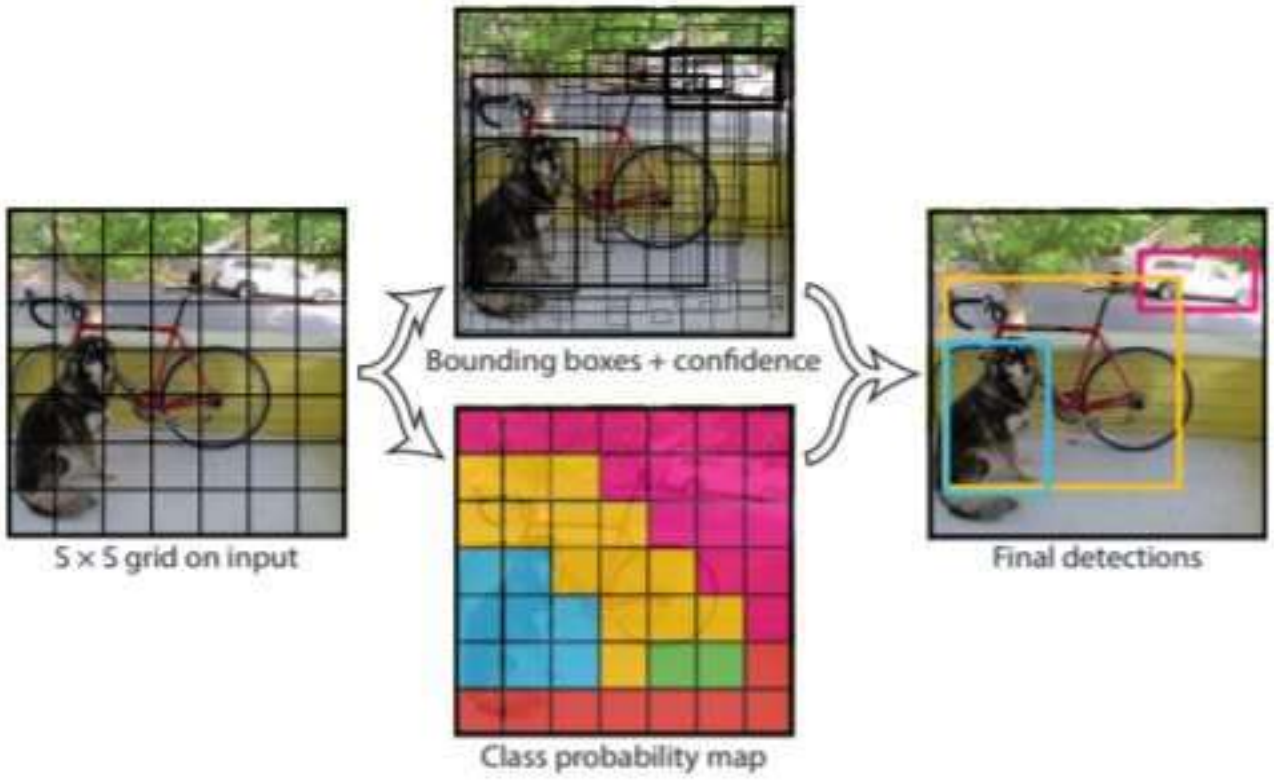
The following image provides a simple example of how IOU works.



In the image above, there are two bounding boxes, one in green and the other one in blue. The blue box is the predicted box while the green box is the real box. YOLO ensures that the two bounding boxes are equal.

Combination of the three techniques

The following image shows how the three techniques are applied to produce the final detection results.



First, the image is divided into grid cells. Each grid cell forecasts B bounding boxes and provides their confidence scores. The cells predict the class probabilities to establish the class of each object.

For example, we can notice at least three classes of objects: a car, a dog, and a bicycle. All the predictions are made simultaneously using a single convolutional neural network.

Intersection over union ensures that the predicted bounding boxes are equal to the real boxes of the objects. This phenomenon eliminates unnecessary bounding boxes that do not meet the characteristics of the objects (like height and width). The final detection will consist of unique bounding boxes that fit the objects perfectly.

For example, the car is surrounded by the pink bounding box while the bicycle is surrounded by the yellow bounding box. The dog has been highlighted using the blue bounding

SYSTEM REQUIREMENTS

HARDWARE REQUIREMENTS

- **Processor -**Intel Core i5
- **System with CPU & GPU**
- **RAM -** 8GB or above

SOFTWARE REQUIREMENTS

- **Operating System -** Windows, Linux, Mac
- **Coding Language -** Python
- **Software -** Python IDEs, Google Colab, LabelImg software

LIBRARIES AND FRAMEWORK USED

NumPy - NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

CV2 - OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

Glob - Glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order.

Darknet - Darknet is an open source neural network framework. It is a fast and highly accurate (accuracy for custom trained model depends on training data, epochs, batch size and some other factors) **framework for real time object detection**.

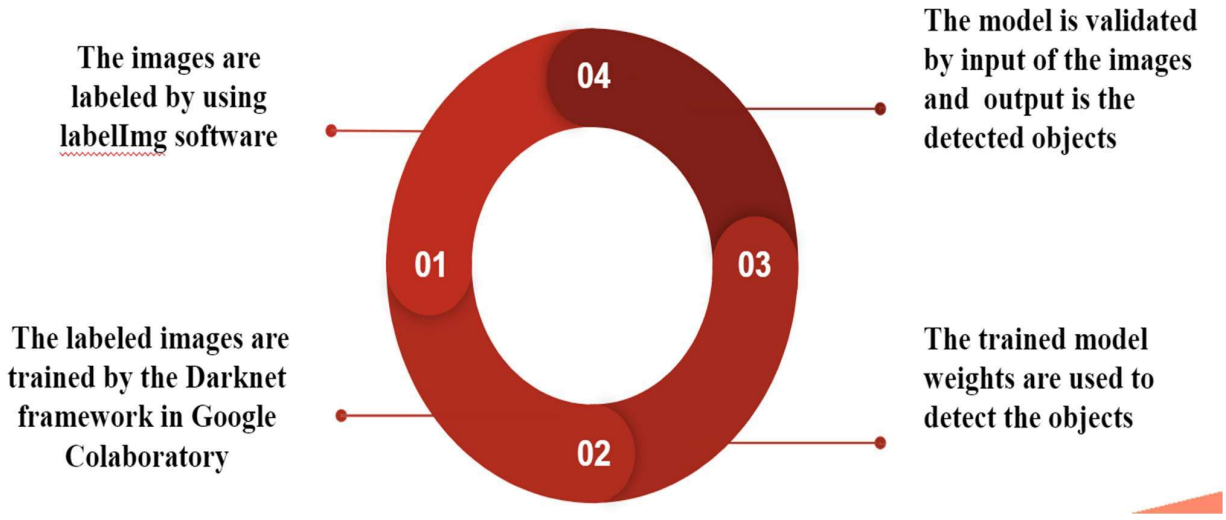
PROPOSED METHOD

Step-1: The images are labelled by using labelImg software.

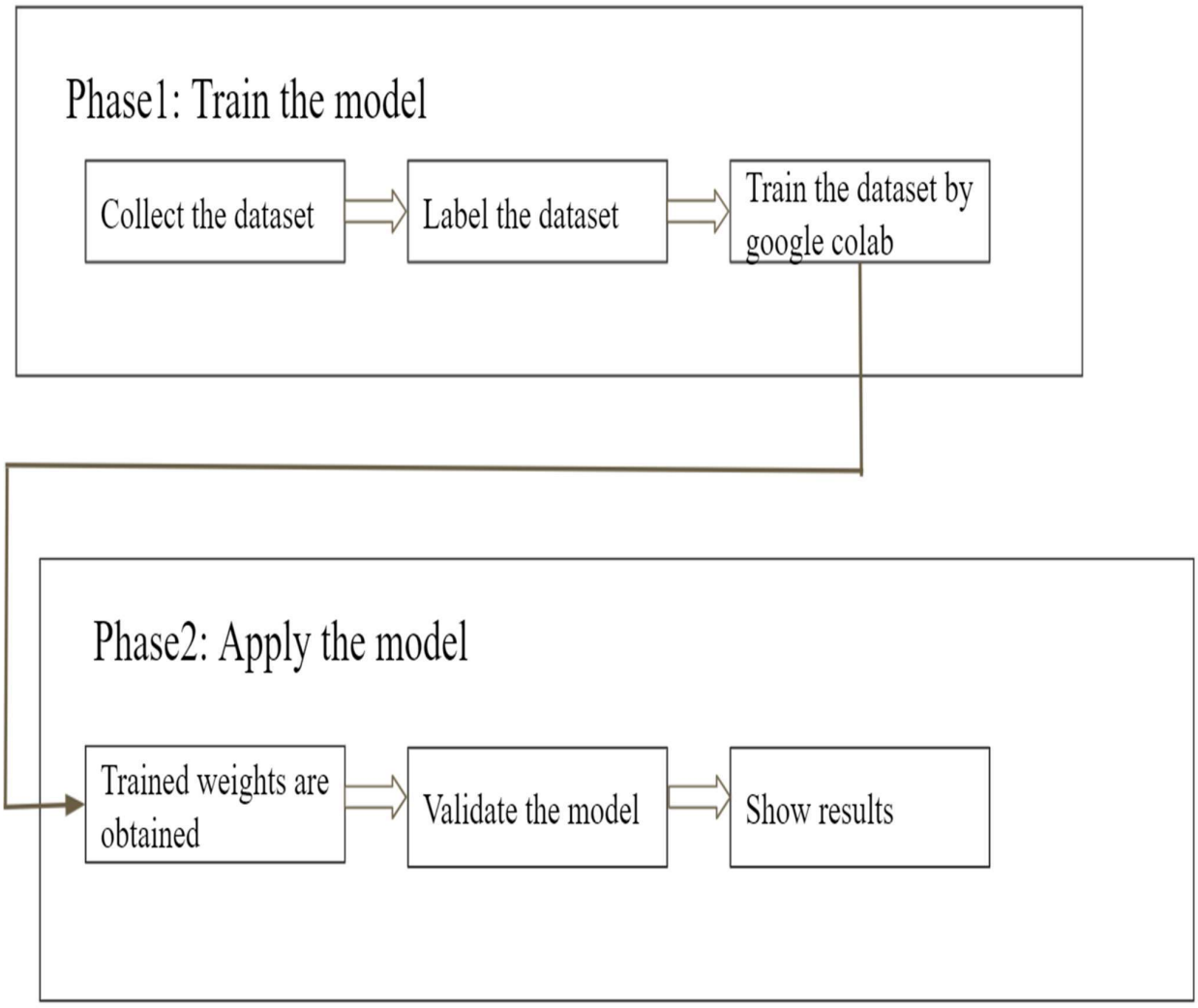
Step-2: The labelled images are trained by the Darknet framework in Google Colabor.

Step-3: The trained model weights are used to detect the objects.

Step-4: The model is validated by input of the images and output is the detected objects



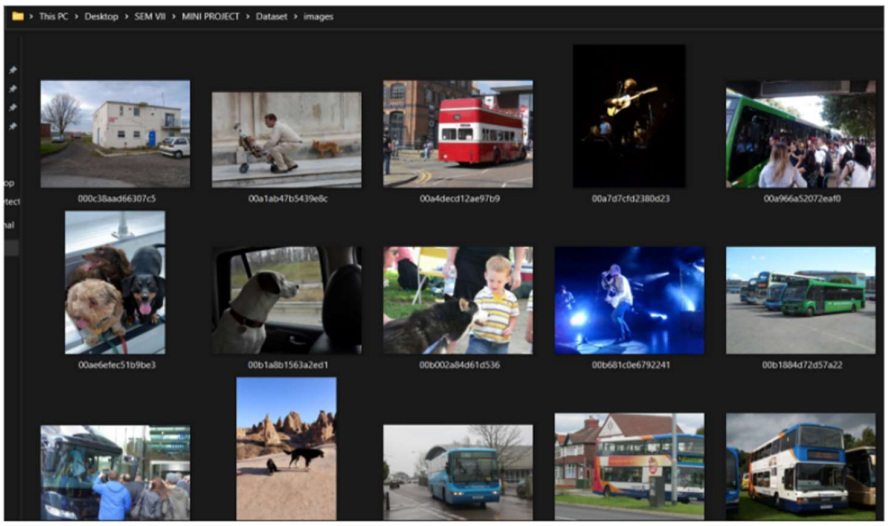
WORKFLOW PROCESS



IMPLEMENTATION

Prepare the Image Dataset

- An image dataset is a folder containing a lot of images, where there is the custom object you want to detect. For this project we collected image of 4 objects i.e; person, dog, car, bus.



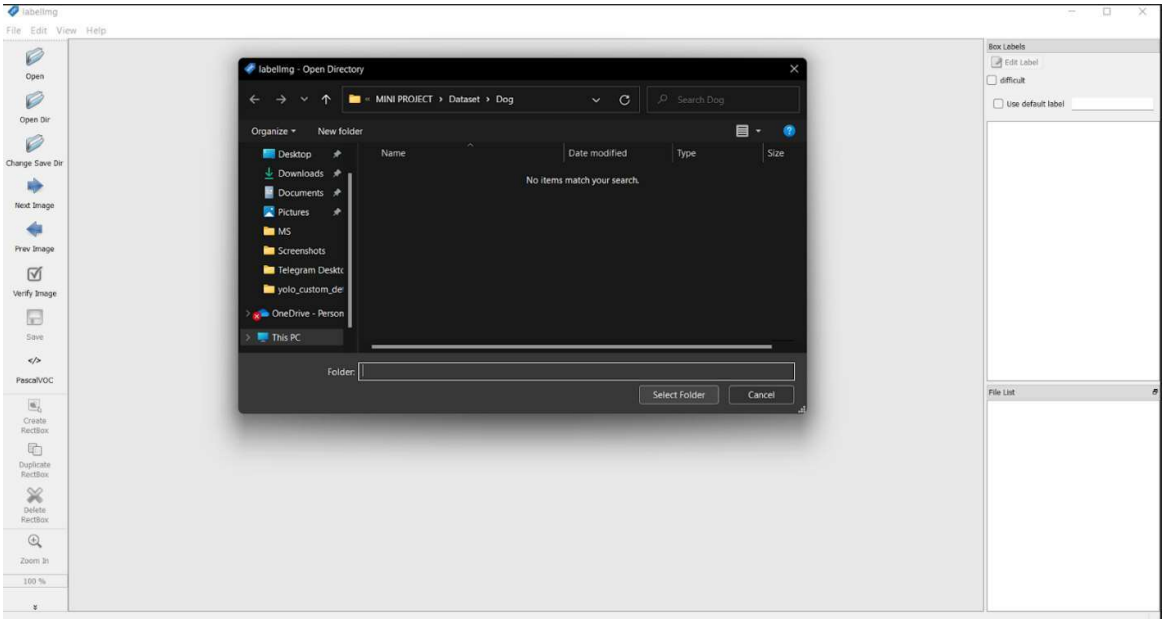
- Having the images is not enough, but we also need to specify where the custom object is located on the specific image.

For this operation we will need an external software: **LabelImg.**

Let's set labelImg for our dataset:

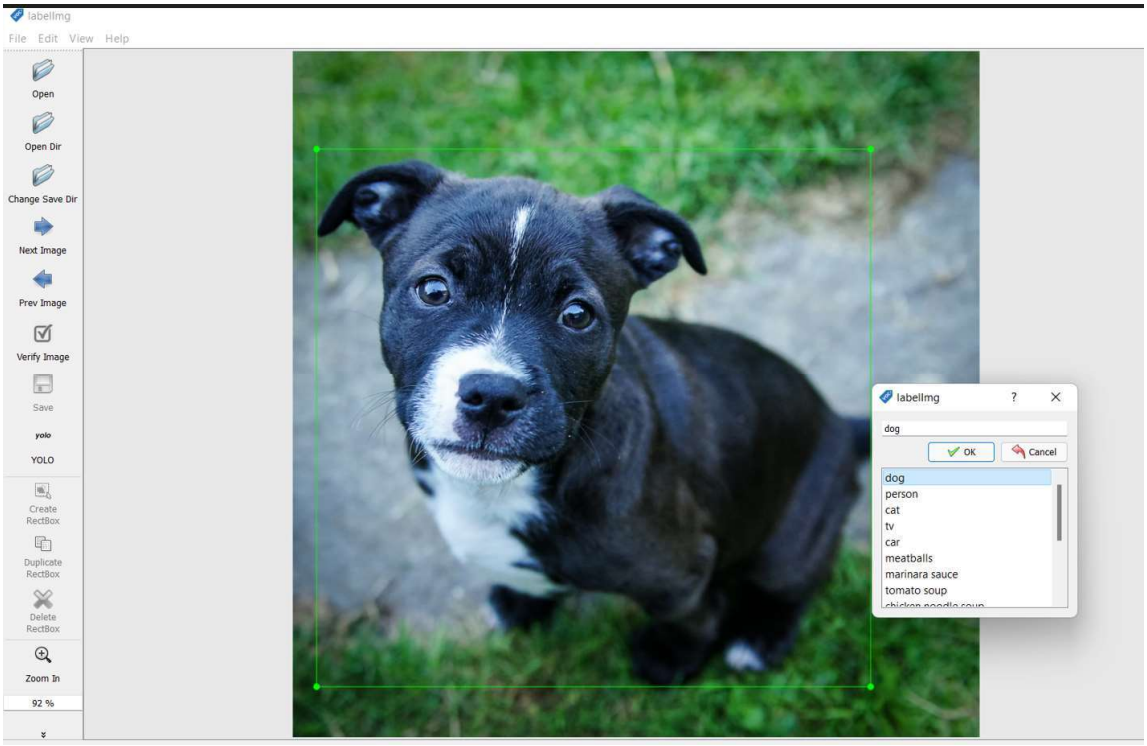
1. Once we run LabelImg let's click on "Open Dir".
2. We choose the folder where the images are located
3. Then we click on "Select folder"
4. We then click on "Change save dir".
5. We select the folder where the images are located (same folder we selected on step 2).

6. Then we click on “Select folder”.
7. Finally make sure that we’re using the settings for YOLO. If pascalVOC is written, then let’s click and we will see YOLO.



Now we're ready to label the images.

1. Let's click on "Create RectBox"
2. Let's select the area where our object is located.
3. We add the label with the name of our object. In my case typed Koala and press Ok.
4. We click on "Save".



Train the Image dataset online

To train the image dataset we're going to use the free server offered by google colab.

Google colab is a free service offered by google where you can run python scripts and use machine learning libraries taking advantage of their powerful hardware.
It's for free with the only disadvantage the you can use it for 12 hours in a row, after that you'll be disconnected and your files will be deleted. you can restart it again but doing everything from scratch.

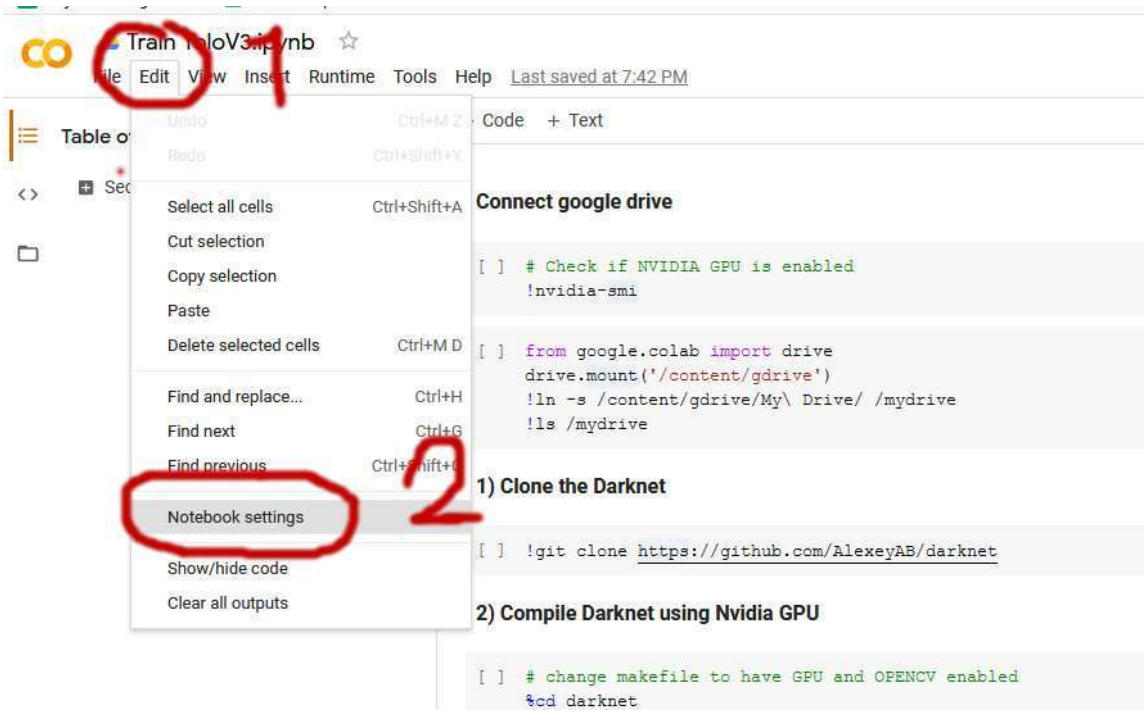
We can solve this problem by connecting google colab with google drive, so we won't lose the files in case of disconnection.

Set up google drive:

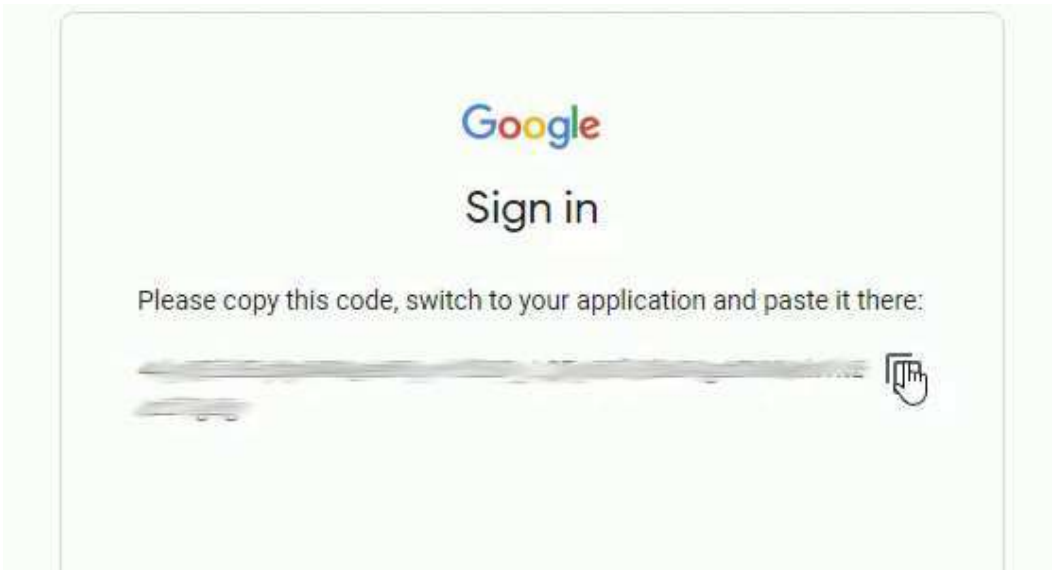
1. Go on google drive and log in. If you' don't have an account, create one and log in.
2. Create a new folder called "yolov3".
3. Then upload the file "images.zip" you created before inside the yolov3 folder.

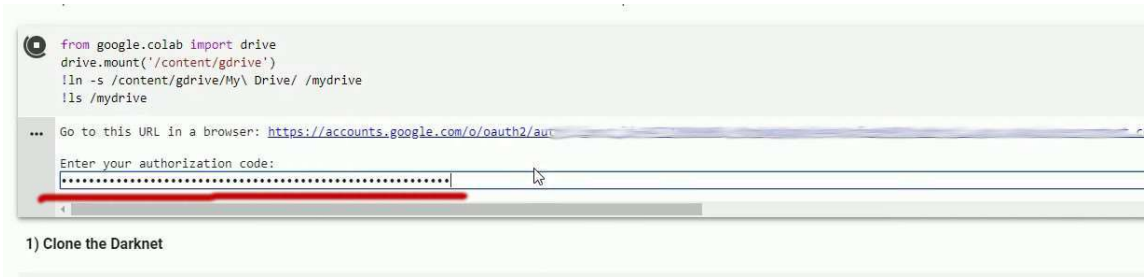
Set up google colab:

1. Go on google colab and log in with the same account you used to log in on google drive.
2. Upload this file "Training.ipynb"
   n.b. You can get this file by clicking on "Click here to download the Source code" at the beginning of the post.
3. Then we need to enable the GPU. So click on "Edit".


Edit
Runtime
Tools
Help
Last saved at 7:42 PM
Code
+ Text
Select all cells Ctrl+Shift+A
Cut selection
Copy selection
Paste
Delete selected cells Ctrl+M D
Find and replace... Ctrl+H
Find next Ctrl+G
Notebook settings
Show/hide code
Clear all outputs
Connect google drive
Check if NVIDIA GPU is enabled
!nvidia-smi
from google.colab import drive
drive.mount('/content/gdrive')
!ln -s /content/gdrive/My\ Drive/ /mydrive
!ls /mydrive
1) Clone the Darknet
!git clone https://github.com/AlexeyAB/darknet
2) Compile Darknet using Nvidia GPU
change makefile to have GPU and OPENCV enabled
%cd darknet


Notebook settings
Runtime type
Python 3
Hardware accelerator
GPU
To get the most out of Colab, avoid using
a GPU unless you need one. Learn more
Omit code cell output when saving this notebook
CANCEL
SAVE





1) Clone the Darknet

Test the model we created

Each 100 iterations, our custom object detector is going to be updated and saved on our Google drive, inside the folder “yolov3”.

The **file that we need is “yolov3_training_last.weights”.**
You might find that other files are also saved on your drive, “yolov3_training__1000.weights”, “yolov3_training_2000.weights” and so on because the darknet makes a backup of the model each 1000 iterations.

I created a python project to test your model with Opencv.
The project is on the folder yolo_custom_detection, which contains 2 files (yolo_object_detection.py and yolov3_testing.cfg).
n.b. You can get this file by clicking on “Click here to download the Source code” at the beginning of the post.

You need to download the file **yolov3_training_last.weights** from Google Drive and place in on the same folder with yolo_object_detection.py and yolov3_testing.cfg.

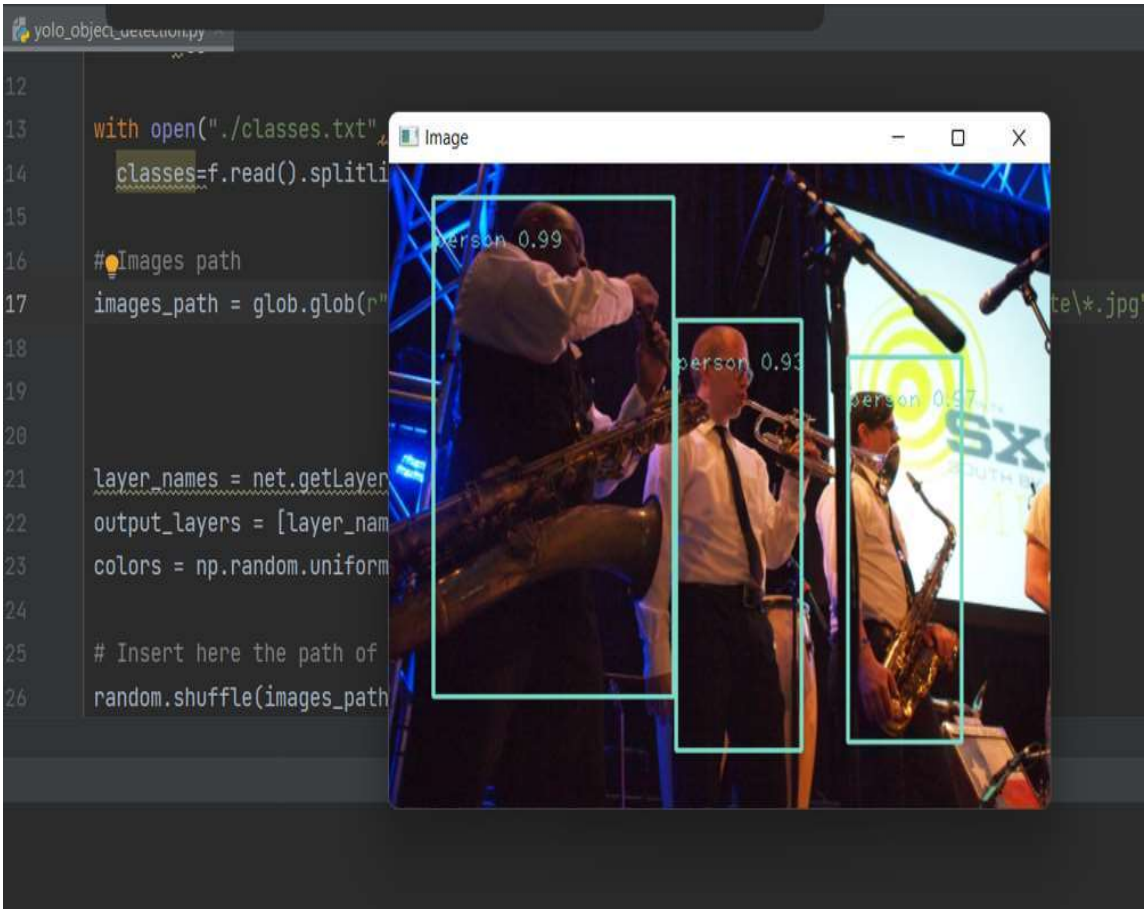
```
import cv2
import numpy as np
import glob
import random


# Load Yolo
net = cv2.dnn.readNet("yolov3_training_final.weights", "yolov3_testing.cfg")

# Name custom object
classes=[]

with open("./classes.txt", 'r') as f:
  classes=f.read().splitlines()

# Images path
images_path = glob.glob(r"C:\Users\kodur\Desktop\SEM VII\MINI PROJECT\Dataset\validate\*.jpg")
```


yolo_object_detection.py
12
13 with open("./classes.txt"
14 classes=f.read().splitli
15
16 #Images path
17 images_path = glob.glob(r"
18
19
20
21 layer_names = net.getLayer
22 output_layers = [layer_nam
23 colors = np.random.uniform
24
25 # Insert here the path of
26 random.shuffle(images_path
Image
person 0.99
person 0.93
te*.jpg