Neural Networks and Deep Learning (ECE 326)

B. Tech (ECE) – VII Semester

**TOPIC: -**

**Face Mask Detection using simple CNN, VGG16, Resnet50, and YOLO**

**SUBMITTED BY: -**

Anurag Tripathi (7ECE2X, A12405117001)

**SUBMITTED TO: -**

**Dr. Partha Sarathi Mangipudi**



***DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING***

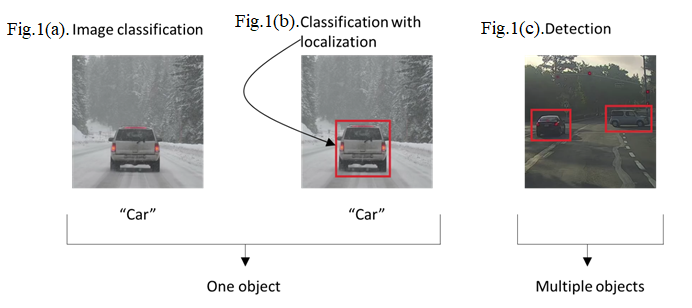
**Amity School of Engineering & Technology**

**Amity University, Uttar Pradesh**

**Table of Contents: -**

|  | Introduction and Motivation |
| --- | --- |
|  | Contribution by each member |
|  | Dataset and operations performed on it. |
|  | Block Diagram |
|  | Simple CNN model and its results |
|  | VGG 16 Architecture and its results |
|  | ResNet152 Architecture and its results |
|  | YOLO algorithm and its results |
|  | Conclusion |
|  | References |

1. **Introduction and Motivation**: -

****

We are already familiar with the image classification task where an algorithm looks at this picture and might be responsible for saying this is a car shown in fig.1.(a). This is what we mean by classification.

In fig.1.(b)., classification with localization means not only do we have to label this as say a car but the algorithm also is responsible for putting a bounding box or drawing a red rectangle around the position of the car in the image. The term localization refers to figuring out where in the picture the car, we've detective. The detection problem where now there might be multiple objects in the image as shown in fig.1.(c)., and we have to detect them all and localize them all.

So, the ideas we've learned about image classification will be helpful for classification with localization. And that the ideas we learn for localization will then turn out to be useful for detection.

Mandatory face mask rules are becoming more common in public settings around the world. There is growing scientific evidence supporting the effectiveness of face mask-wearing in reducing the spread of Covid-19.

Wearing a face mask will help prevent the spread of infection and prevent the individual from contracting any airborne infectious germs. When someone coughs, talks sneezes they could release germs into the air that may infect others nearby. Face masks are part of an infection control strategy to eliminate cross-contamination.

The objective of this work is to develop a deep-learning model which detects whether a person is wearing a face mask, or not.

For this we are going to use a multitude of algorithms, which will be performed by us as a team, our main aim being the YOLO algorithm.

As YOLO is performed under a DARKNET architecture which is written in C, there is not much work in implementing this algorithm as it only involves some basic imports and YAML files, therefore we have backed up by implementing other algorithms, which helped us code, they are: -

* Hard coding a simple CNN
* Implementing VGG16 and ResNet from importing them from the Keras library.
* Finally, implementing the YOLO algorithm.

All these algorithms are performed on the same dataset which we have downloaded from Kaggle.

1. **Dataset and Software Used: -**

* Dataset: -

Kaggle Face mask detection dataset. (License; cc0: Public domain).

NUMBER OF IMAGES: 3833 IMAGES

NUMBER OF CLASSES: 2 – with mask AND without mask.

Size: 399 mb.



**Fig.3.1.** Sample images of dataset used

* Software Used: -

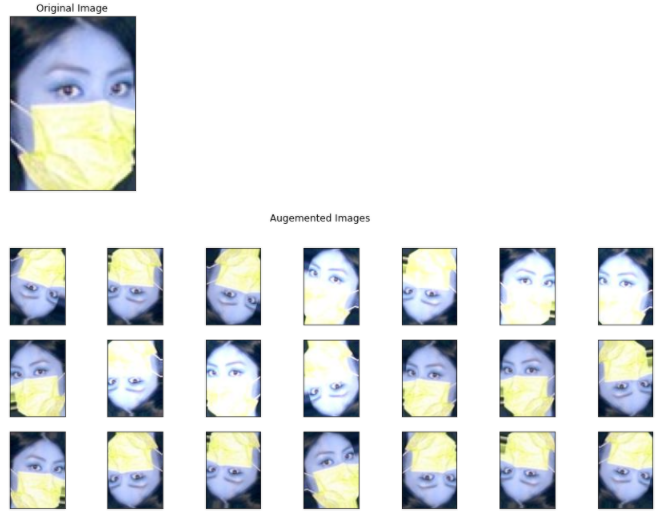
Most of the work was done on jupyter notebook (Anaconda) and we used the NumPy, keras and pytorch libraries in the process.

* Data Augmentation: -



**Fig.3.2.** Snap of code used for data augmentation

In the above ImageDataGenerator function, it augments our data using the following parameters to increase our dataset.

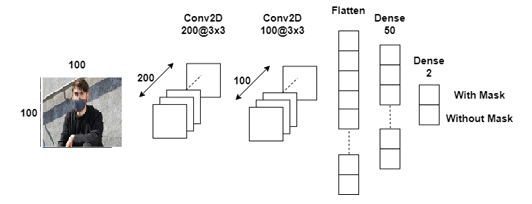


**Fig.3.3.** Snap of the augmented images for the original image

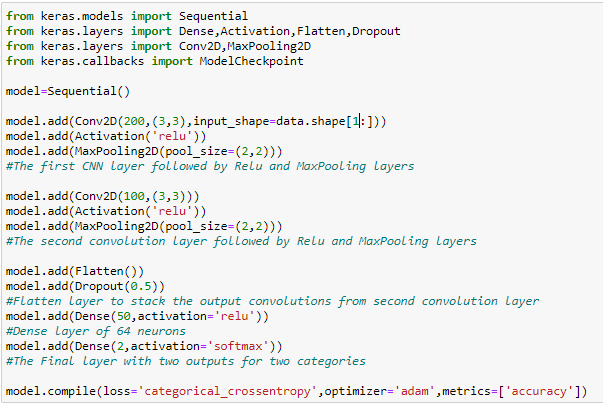
1. **Block Diagram:** **+**
2. **Simple CNN and its Results: -**

Here we have hardcoded a simple CNN model which consists of two convolution layers of filter 3X3 and two dense layers. We applied max pooling after each convolutional layer taking ReLU as an activation function. We have used categorical cross entropy as a loss function along with Adam as an optimizer for gradient descent. Finally, we have used the SoftMax layer to get out the predicted output.

The validation accuracy came out to be 94% calculated for 20 epochs.

****

**Fig.4.1.** Architecture of hardcoded simple CNN model



**Fig.4.2.** Snap of code used for building hardcoded simple CNN model

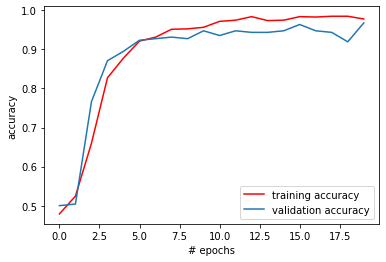
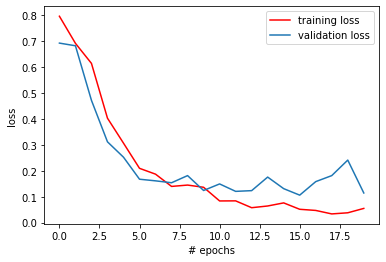
**Sequential()** – It implements our layers as a stack.

**Conv2D() -** This layer initializes our weights and filters. It can also resize our data.

**Activation()** – This specifies the activation function used.

**Model. Compile()** - compiles our defined model, also mentions the optimizer and loss function used,

**Results : -**

 **Fig.4.3.** Accuracy and Loss graph plotted for the basic CNN model

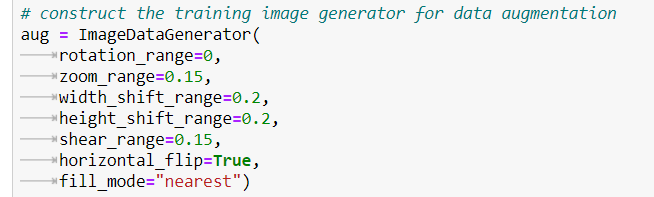
We have obtained a training accuracy of 98.1% and validation accuracy of 94.5%

1. **VGG 16 and it’s Results: -**

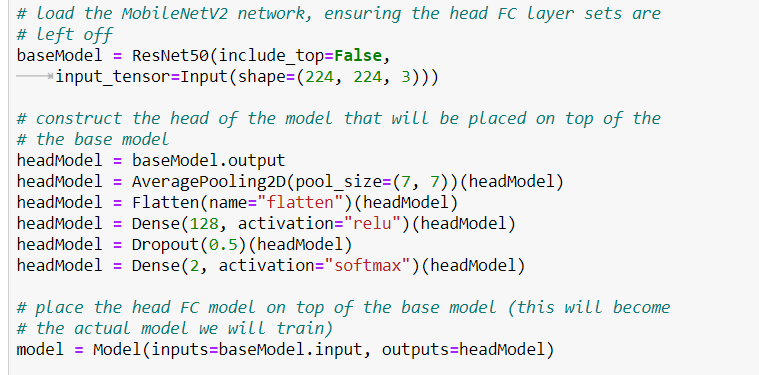
VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition ''. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous models submitted to [ILSVRC-2014](http://www.image-net.org/challenges/LSVRC/2014/results). It makes an improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using the NVIDIA Titan Black GPU.

We have applied transfer learning on the VGG16 algorithms by importing it from the Keras library and modifying its parameter to our requirements.

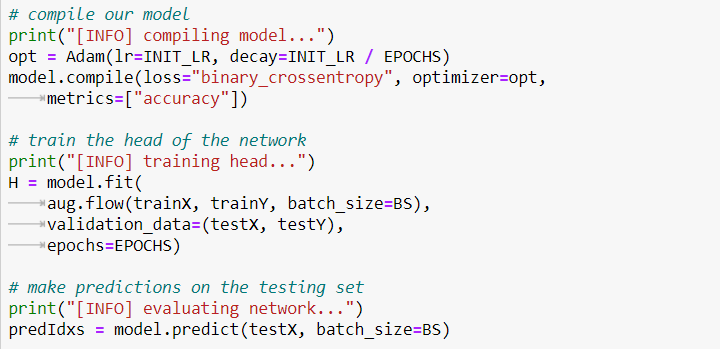
**Code: -**



**Fig.5.1.** Snap of code used for data augmentation



**Fig.5.2.** Snap of code used for building VGG16 model

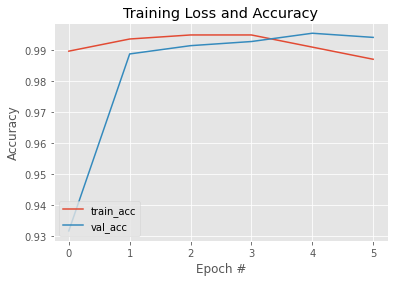
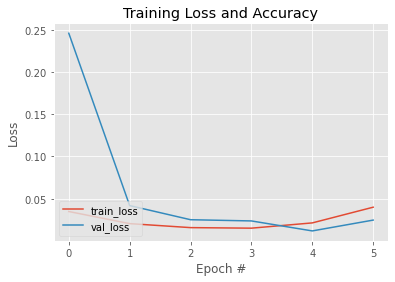


**Fig.5.3.** Snap of code used for compiling and fitting the model

**VGG16 (): -** This function, initializes our weights and also removes the last layer to be adjusted based on our requirements

**Flatten (): -** It flattens the last layer.

**model.fit\_generator ()** - it fits our model according to the model variable defined earlier, we can define the epochs etc…here.

**  Fig.5.4.** Accuracy and Loss graph plotted for the VGG16 CNN model

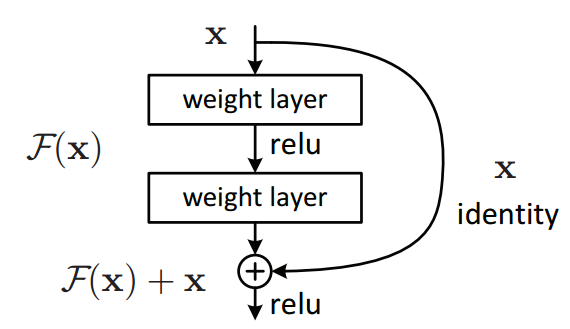
We have obtained a training accuracy of 99.3% and validation accuracy of 99.1%, at the third epoch. We can apply early dropout in this case, at the third epoch.

1. **ResNet 152 and its Results: -**

Here I have used the residual network with 152 layers. The basic key features that resnet bought with it were: -

1. Skip connections
2. Heavy batch normalization

This allowed us to reduce the problem of degradation of performance due to vanishing gradients in deeper layers. Ensuring many skip connections can lead to less degradation as if the expected output is not achieved, at least the input will be present at the output. Below is shown the architecture of resnet 152. Also, the dataset has been manually split into a 70-30 ratio and each training and testing consisted of two classes, namely with and without masks.

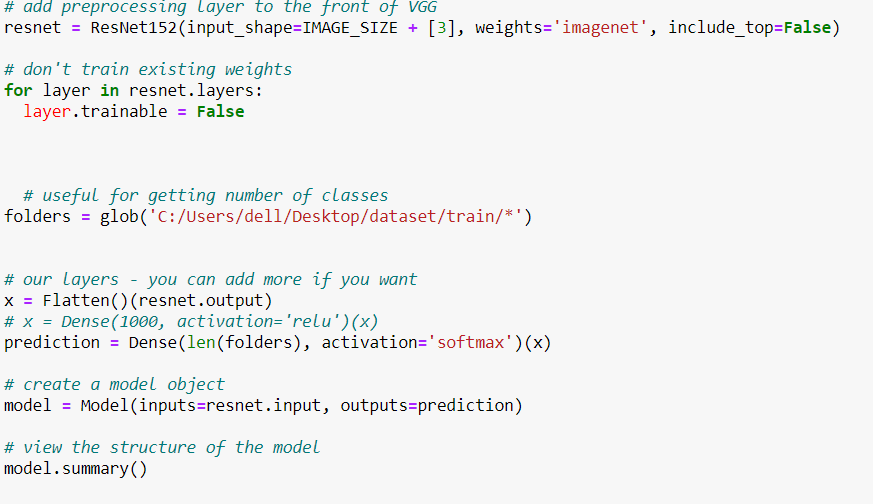


**Fig.6.1.** Basic architecture of ResNet 152

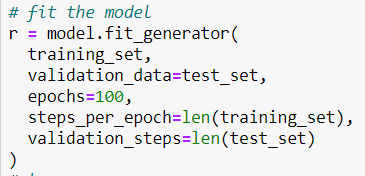
**Results: -**

I have used the Adam optimizer and the binary cross entropy as our loss function for implementing backpropagation. We have also played around with the learning rate and batch size to observe the changes that took place.

**Code: -**



**Fig.6.2.** Snap of code used for building ResNet152 model



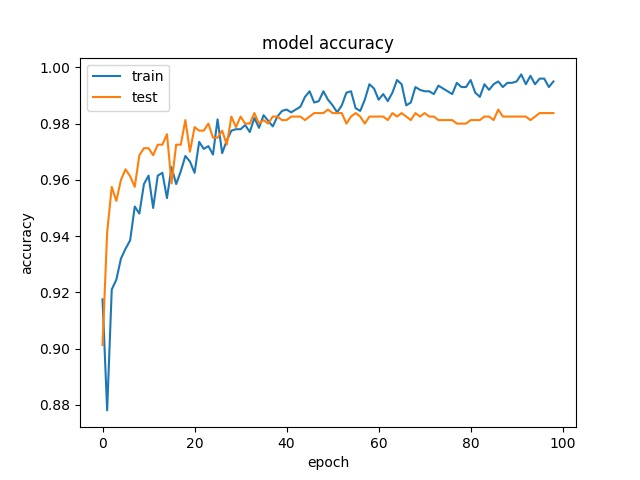
**Fig.6.3.** Snap of code used for fitting the model and plotting the results

**Resnet152 (): -** This function, initializes our weights and also removes the last layer to be adjusted based on our requirements

**Flatten (): -** It flattens the last layer.

**model.fit\_generator ()** - it fits our model according to the model variable defined earlier; we can define the epochs, etc…here.

**Result: -**



**Fig.6.4.** Accuracy graph plotted for the ResNet152 CNN model

We have obtained a training accuracy of 99.8% and validation accuracy of 98.1 %.

1. **YOLO (You Only Look Once) Algorithm: -**

All of the YOLO models are [object detection models](https://models.roboflow.com/object-detection). Object detection models are trained to look at an image and search for a subset of object classes. When found, these object classes are enclosed in a bounding box and their class is identified. Object detection models are typically trained and evaluated on the [COCO](https://roboflow.com/formats/coco-json) dataset which contains a broad range of 80 object classes. From there, it is assumed that object detection models will generalize to new object detection tasks if they are exposed to new training data.

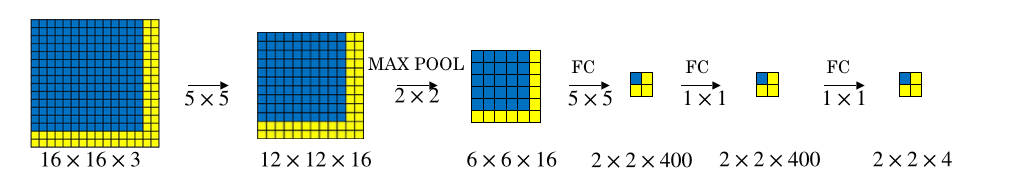
The original YOLO (You Only Look Once) was written by Joseph Redmon (now retired from CV) in a custom framework called Darknet. Darknet is a very flexible research framework written in low-level languages and has produced a series of the best real-time object detectors in computer vision: YOLO, YOLOv2, YOLOv3, YOLOv4, and YOLOv5

The Original YOLO - YOLO was the first object detection network to combine the problem of drawing bounding boxes and identifying. For our project, we have used the YOLOv5, which is the latest state-of-the-art in object detection.

Before YOLO even came into existence, there were object detection algorithms that were not as efficient. The most popular approach is sliding windows.

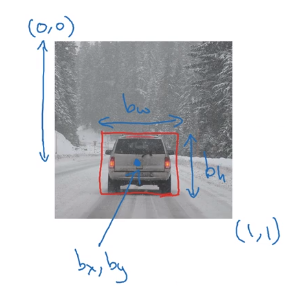
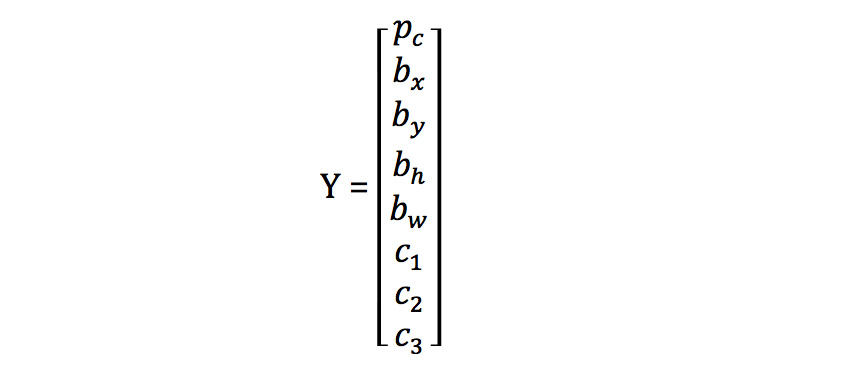
The sliding window algorithm, came with flaws, in that it was computationally expensive and very time taking. Here, each and every pixel was given into the pre-trained convolutional network to give the particular class output, therefore the kernel takes in all the respective one by one, hence taking a lot of time.

YOLO (You only look once), as the name suggests takes in the entire image at once and performs what is called a convolutional sliding window, and the entire image is compressed into a small matrix, and the depth of that matrix gives the class information, probability, parameters of the bounding box, etc.



**Fig.7.1.** Working of YOLO model

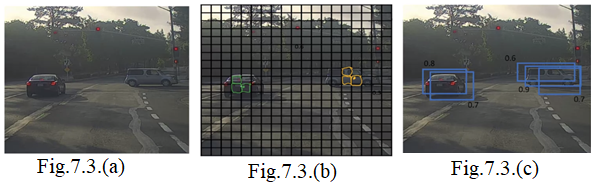
Also, in YOLO, the output of the SoftMax layer must include a few other parameters as shown: -

**Fig.7.2.** Parameters used by the output of the SoftMax layer in YOLO model

Here, pc denotes whether the object is present or not, if (pc =1, the object is present; pc = 0, if object is not present), also bx, by, bh denotes the parameters of the bounding box and c1, c2 and c3 denotes the classes of interest (here we consider only 2 classes as we have only two classes). Also, the yolo algorithm does not use any max pooling layers and uses simple convolution to scale down the dimension, which avoids the loss of information.

Each bounding box of the grid cells of the image will consist of many bounding boxes and by the principles of non – suppression and IoU (intersection over union), the irrelevant boxes will be removed or suppressed.

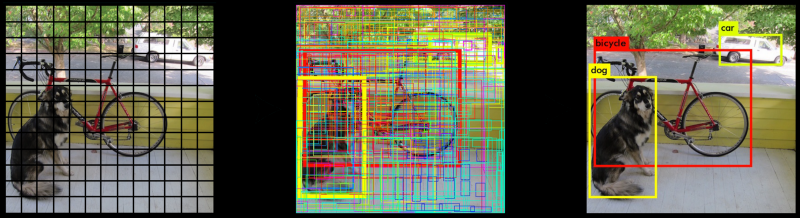


**Fig.7.3.** Non-max suppression working with car example

Let's say we want to detect cars in the fig.7.3.(a). We might place a grid over this (19 by 19 grid) as shown in fig.7.3.(b). Now, while technically the car shown in the fig.7.3.(b). has just one midpoint, so it should be assigned just one grid cell. And the car on the left also has just one midpoint, so technically only one of those grid cells should predict that there is a car. In practice, we're running an object classification and localization algorithm for every one of these split cells. So, it's quite possible that this split cell might think that the center of a car is in it, and for the car on the left as well. Maybe not only this box, if this is a test image you've seen before, not only that box might decide things that's on the car, maybe this box, and this box and maybe others as well will also think that they've found the car. Let's step through an example of how non-max suppression will work.

So, because you're running the image classification and localization algorithm on every grid cell. So, when we run your algorithm, we might end up with multiple detections of each object. So, what non-max suppression does, is it cleans up these detections. So, they end up with just one detection per car, rather than multiple detections per car. So concretely, what it does, is it first looks at the probabilities associated with each of these detections. But for now, let's just say it is Pc with the probability of a detection. And it first takes the largest one, which in this case is 0.9 shown in fig.7.3.(c). Having done that, the non-max suppression part then looks at all of the remaining rectangles and all the ones with a high overlap, with a high IOU, with this one that you've just output will get suppressed. So those two rectangles with the 0.6 and the 0.7. Both of those overlap a lot with the light blue rectangle. So, we are going to suppress and darken them to show that they are being suppressed. Next, we then go through the remaining rectangles and find the one with the highest probability, the highest Pc, which in this case is the one with 0.8. So, let's commit to that, and then, the non-max suppression part is to then get rid of any other ones with a high IOU. So now, every rectangle has been either highlighted or darkened. And if you just get rid of the darkened rectangles, you are left with just the highlighted ones, and these are your two final predictions. So, this is non-max suppression. And non-max means that you're going to output your maximal probabilities classifications but suppress the close-by ones that are non-maximal. Hence the name, non-max suppression.

The principle of non-max suppression will keep the highest probability box and suppress the remaining boxes based on the IoU ratio, and finally, the desired object will be enclosed in the box.



**Fig.7.4.** YOLO detection of cycle and dog using bounding box

The yolov5 algorithm consists of the following components: -

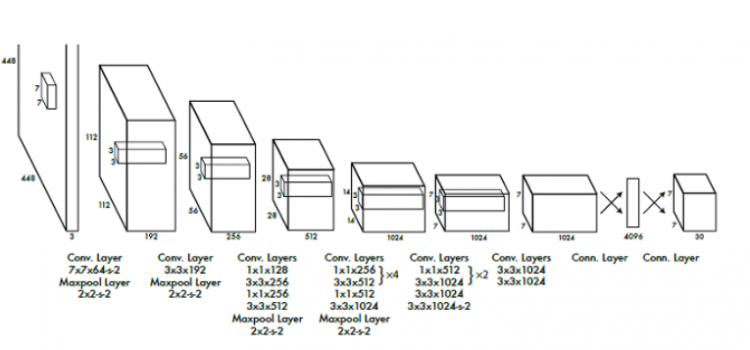
* Backbone
* Neck
* Head

Basically, all object detectors take an image in for **input** and compress features down through a convolutional neural network [backbone](https://blog.roboflow.com/glossary/#:~:text=backbone). In [image classification](https://models.roboflow.com/classification), these backbones are the end of the network and predictions can be made off of them. In [object detection](https://models.roboflow.com/object-detection), multiple bounding boxes need to be drawn around images along with classification, so the feature layers of the convolutional backbone need to be mixed and held up in light of one another. The combination of backbone feature layers happens in **the neck.**

It is also useful to split object detectors into two categories: one-stage detectors and two stage detectors. Detection happens in the **head**. Two-stage detectors decouple the task of object localization and classification for each bounding box. One-stage detectors make the predictions for object localization and classification at the same time. YOLO is a one-stage detector, hence, You Only Look Once.

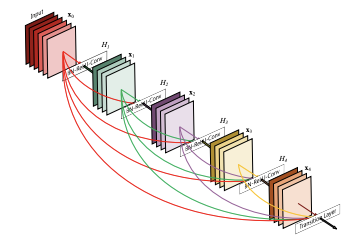
The backbone network for an object detector is typically pretrained on ImageNet classification. Pretraining means that the network's weights have already been adapted to identify relevant features in an image, though they will be tweaked in the new task of object detection.

The authors considered the CSPDarknet53 backbone for the YOLOv5 object detector. The backbone architecture is shown below: -



**Fig.7.5.** Architecture ofYOLO model

The CSPResNext50 and the CSPDarknet53 are both based on Dense Net. Dense Net was designed to connect layers in convolutional neural networks with the following motivations: to alleviate the vanishing gradient problem (it is hard to backprop loss signals through a very deep network), to bolster feature propagation, encourage the network to reuse features, and reduce the number of network parameters.



**Fig.7.6.** Architecture ofCSPResNext50 model

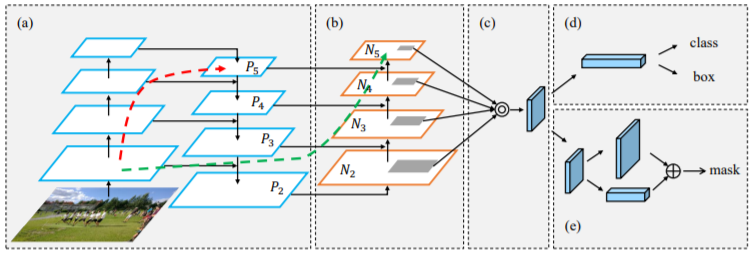
The next step in object detection is to mix and combine the features formed in the ConvNet backbone to prepare for the detection step. YOLOv5 considers PANet.

The components of the neck typically flow up and down among layers and connect only the few layers at the end of the convolutional network.

Each one of the P(i) above represents a feature layer in the CSPDarknet53 backbone.

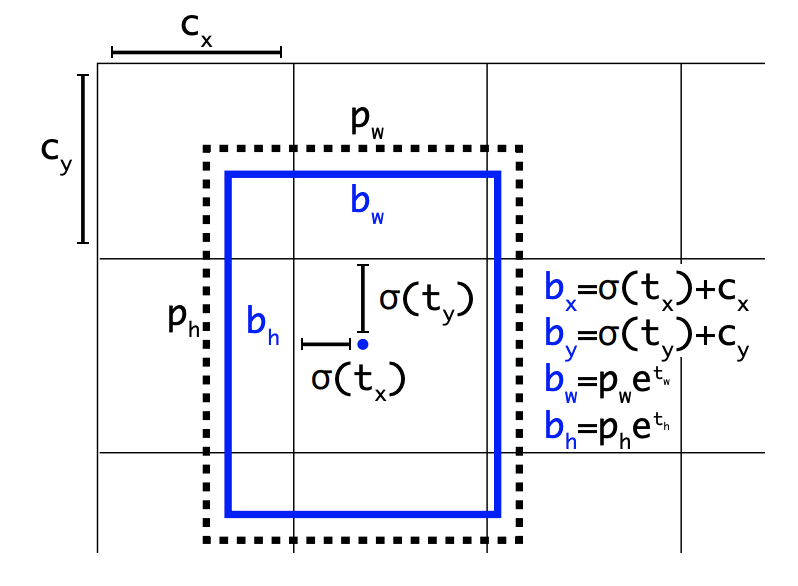
The image above comes from YOLOv4's predecessor, EfficientDet. Written by Google Brain, EfficientDet uses neural architecture search to find the best form of blocks in the neck portion of the network, arriving at NAS-FPN. The EfficientDet authors then tweak it slightly to make the architecture more intuitive (and probably perform better on their development sets).

Additionally, YOLOv5 adds an SPP block after CSPDarknet53 to increase the receptive field and separate out the most important features from the backbone.



**Fig.7.7.** Architecture ofPaNet model

YOLOv5 deploys the same YOLO head as YOLOv3 for detection with the anchor-based detection steps, and three levels of detection granularity.



**Fig.7.8.** Calculation of bounding box at the head layer.

**Code: -**

The imported yaml file for yolov5 from github, implemented by python: -

# parameters

nc: {num\_classes}  # number of classes

depth\_multiple: 0.33  # model depth multiple

width\_multiple: 0.50  # layer channel multiple

# anchors

anchors:

  - [10,13, 16,30, 33,23]  # P3/8

  - [30,61, 62,45, 59,119]  # P4/16

  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 backbone

backbone:

  # [from, number, module, args]

  [[-1, 1, Focus, [64, 3]],  # 0-P1/2

   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4

   [-1, 3, BottleneckCSP, [128]],

   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8

   [-1, 9, BottleneckCSP, [256]],

   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16

   [-1, 9, BottleneckCSP, [512]],

   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32

   [-1, 1, SPP, [1024, [5, 9, 13]]],

   [-1, 3, BottleneckCSP, [1024, False]],  # 9

  ]

# YOLOv5 head

head:

  [[-1, 1, Conv, [512, 1, 1]],

   [-1, 1, nn.Upsample, [None, 2, 'nearest']],

   [[-1, 6], 1, Concat, [1]],  # cat backbone P4

   [-1, 3, BottleneckCSP, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],

   [-1, 1, nn.Upsample, [None, 2, 'nearest']],

   [[-1, 4], 1, Concat, [1]],  # cat backbone P3

   [-1, 3, BottleneckCSP, [256, False]],  # 17 (P3/8-small)

   [-1, 1, Conv, [256, 3, 2]],

   [[-1, 14], 1, Concat, [1]],  # cat head P4

   [-1, 3, BottleneckCSP, [512, False]],  # 20 (P4/16-medium)

   [-1, 1, Conv, [512, 3, 2]],

   [[-1, 10], 1, Concat, [1]],  # cat head P5

   [-1, 3, BottleneckCSP, [1024, False]],  # 23 (P5/32-large)

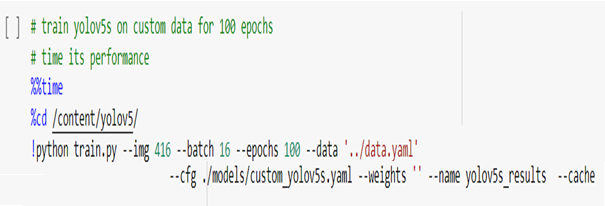
   [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)

  ]

**{num\_classes}** – defines the number of classes, in our cases there are two classes namely “with mask” and without mask”

**anchors –** these boxes are used for multiple object detection and the number of anchor boxes are given by the user.

**conv and bottleneckCSP –** these two commands refer to the convolutional layers and the 1x1 convolutions implemented by the CSPDarknet53 architecture (the layer size is mentioned near each line).

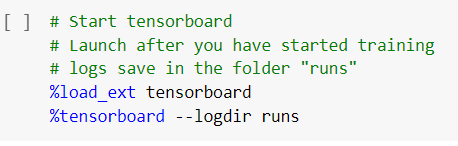
 **Fig.7.9.** Training YOLO on custom data for 100 epochs

Here, we are able to pass a number of arguments:

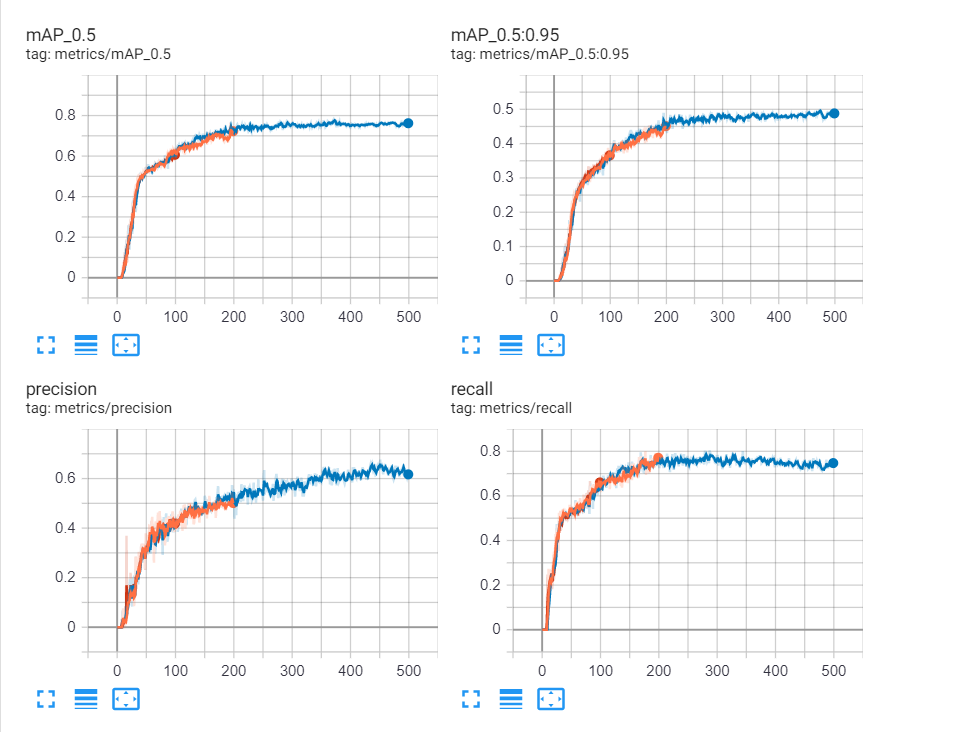
* **img:** define input image size
* **batch:** determine batch size
* **epochs:** define the number of training epochs. (Note: often, 3000+ are common here!)
* **data:** set the path to our YAML file
* **cfg:** specify our model configuration
* **weights:** specify a custom path to weights.
* **name:** result names
* **no save:** only save the final checkpoint
* **cache:** cache images for faster training

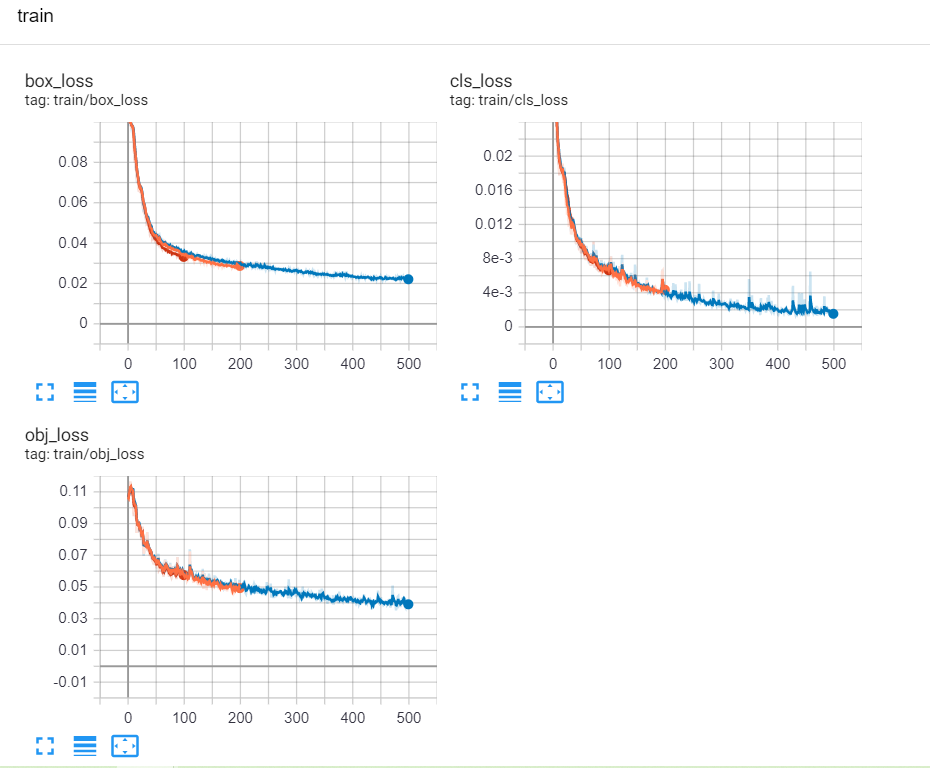
**Results obtained from YOLOv5** :-

From tensor board command, we get the following results:-



**Fig.7.9.** This command gives continuous callbacks and traces the data, in the tensor board.





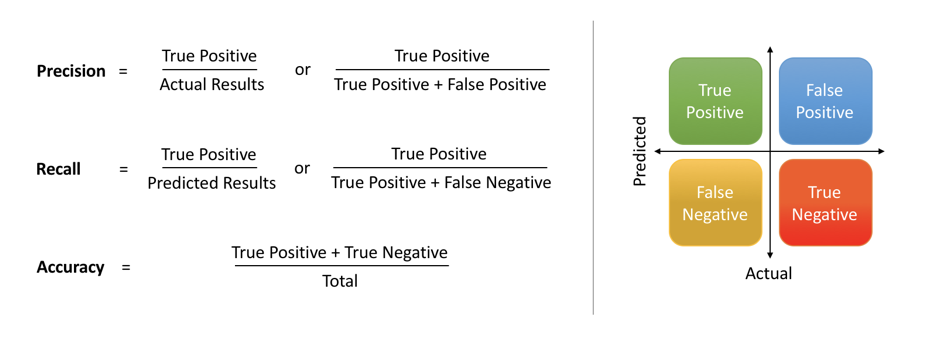
**Fig.7.10.** Results obtained from YOLOv5

mAP (mean average precision) is the average of AP. In some contexts, we compute the AP for each class and average them. But in some contexts, they mean the same thing. mAP[0.5 to 0.95], gives the average precision of (average of IoU’s), of IoU > 0.5 to 0.95 with steps of 0.05, which basically gives us 10 levels.

If the IoU >= 0.5, then the images are predicted as true positives.

If the IoU < 0.5, then the images are labeled True negative.

Also, if the ground truth image is present and our prediction is negative, then it is regarded as a false negative.



**Fig.7.11.** Formulas used to calculate Precision, Recall, and accuracy

1. **Predictions Obtained:-**

**Fig.7.12.** Detection of mask found





**Fig.7.13.** Detection of the mask is not found

1. **References :-**

[1]. C. Liu, Y. Tao, J. Liang, K. Li and Y. Chen, "Object Detection Based on YOLO Network," *2018 IEEE 4th Information Technology and Mechatronics Engineering Conference (ITOEC)*, Chongqing, China, 2018, pp. 799-803, doi: 10.1109/ITOEC.2018.8740604.

[2] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.

[3] H. Qassim, A. Verma and D. Feinzimer, "Compressed residual-VGG16 CNN model for big data places image recognition," *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, Las Vegas, NV, 2018, pp. 169-175, doi: 10.1109/CCWC.2018.8301729.

[4] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.