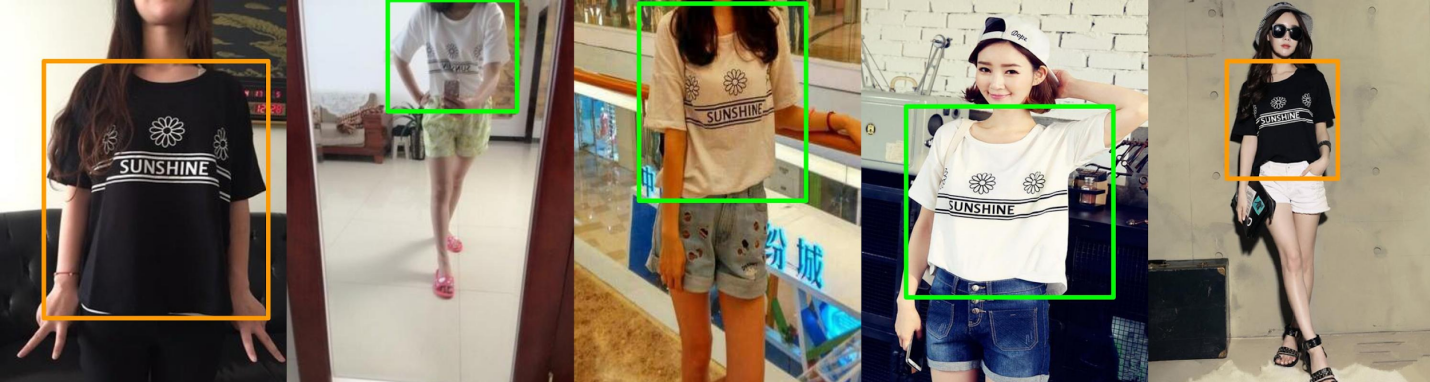
## horizontal line



Bounding Box detection using YOLO

09.05.2020

**─**

Roshni Koli

# Introduction

Object detection models are extremely powerful — from finding dogs in photos to improving healthcare, training computers to recognize which pixels constitute items unlocks near limitless potential. However, one of the biggest blockers keeping new applications from being built is adapting state-of-the-art, open source, and free resources to custom problems.

When it comes to deep learning-based object detection, there are three primary object detectors :

* R-CNN and their variants
* Single Shot Detector (SSDs)
* YOLO

R-CNNs are one of the first deep learning-based object detectors and are an example of a ***two-stage detector.***While R-CNNs tend to be very accurate, the biggest problem with the R-CNN family of networks is their speed — they were incredibly slow, obtaining only 5 FPS on a GPU.

To help increase the speed of deep learning-based object detectors, both Single Shot Detectors (SSDs) and YOLO use a ***one-stage detector strategy***. These algorithms treat object detection as a regression problem, taking a given input image and simultaneously learning bounding box coordinates and corresponding class label probabilities. So these tend to be generally less accurate than two-stage detectors, but are incredibly faster.

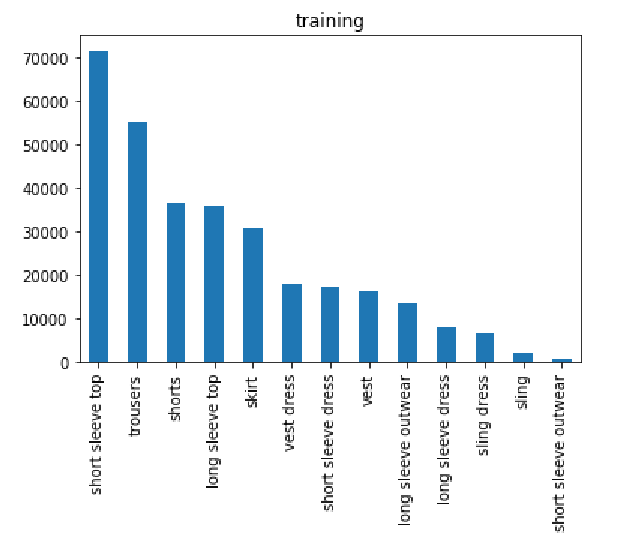
For this project I am using YOLO, i.e. You Look Only Once. First introduced in 2015 by Redmon et al., their paper,*[You Only Look Once: Unified, Real-Time Object Detection](https://arxiv.org/abs/1506.02640)*, details an object detector capable of super real-time object detection, obtaining **45 FPS** on a GPU.

YOLO has gone through a number of different iterations, including*[YOLO9000: Better, Faster, Stronger](https://arxiv.org/abs/1612.08242)* (i.e., YOLOv2), capable of detecting over 9,000 object detectors and *[YOLOv3: An Incremental Improvement](https://arxiv.org/abs/1804.02767)*. For the purpose of this project we will be focusing more on YOLOv3.

# Dataset description

The dataset comprises images from both shops and user posted ones. The train data has 191961 images, validation data has 32123 images and the test data has 62629 images.

The dataset has images with objects from 13 classes and the distribution in test and validation sets is as shown in figure below.



# figure.1. Class-wise data distribution

# Dataset Source : https://github.com/switchablenorms/DeepFashion2

# Yolov3 description

YOLO stands for You Only Look Once. It's an object detector that uses features learned by a deep convolutional neural network to detect an object.

YOLO makes use of only convolutional layers, making it a fully convolutional network (FCN). It has 75 convolutional layers, with skip connections and upsampling layers. No form of pooling is used, and a convolutional layer with stride 2 is used to downsample the feature maps. This helps in preventing loss of low-level features often attributed to pooling.

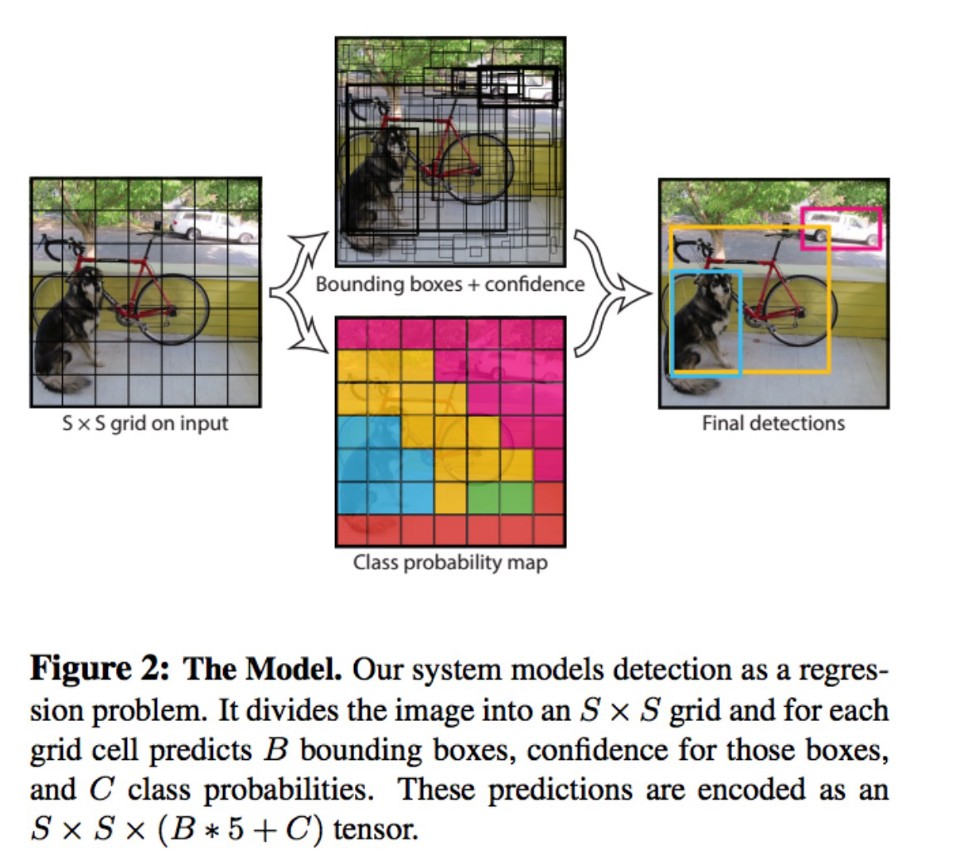
Being a FCN, YOLO is invariant to the size of the input image. So I kept the sizes as is since resizing such a dataset on CPU proved computationally expensive on my system.

Before v3, YOLO used to softmax the class scores. However, that design choice has been dropped in v3, and authors have opted for using sigmoid instead. The reason is that Softmaxing class scores assume that the classes are mutually exclusive which is rarely the case in a real-time object detection scenario.

# Working of Yolo

Yolo splits each image into a SxS grid where each grid predicts only one object. If the center of an object falls into the grid then that particular grid is responsible for detecting that object(class).

Each grid now predicts bounding boxes for object of that grid with **confidence scores** associated with each predicted bounding box.



Confidence score is calculated by using **IoU** between the predicted box and the ground truth. So effectively, each class will have multiple bounding boxes with different probabilities. This is where **Non-Max Suppression** comes into play.

Using **non-max Suppression,** YOLO considers all bounding boxes with confidence scores>0.5. Then it picks only the value with the highest IoU and suppresses the other bounding boxes for identifying the same object.For e.g., if we have three rectangles with the 0.6 and the 0.7 and 0.9. For YOLO to identify the object, Non-Max Suppression will keep the bounding box with IoU 0.9 and will suppress the remaining bounding boxes of 0.6 and 0.7 IoU.

# Properties of Yolo

# Strengths

* YOLO is extremely fast at test time since it only requires a single network evaluation, unlike classifier-based methods
* Performs feature extraction, bounding box prediction, non-max suppression, and contextual reasoning all concurrently.
* YOLO network trains the features in-line and optimizes them for the detection task. Unified architecture leads to a faster, more accurate model

# Limitations

* YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class and this limits the number of nearby objects that the model can predict.
* YOLO struggles with small objects that appear in groups, such as flocks of birds
* Struggles to generalize to objects in new or unusual aspect ratios or configurations

# EDA to convert the given data into YOLO format

The image annotations in the dataset are in COCO format. YOLO requires the data to be in YOLO specified format, which is, for each bounding box and category pair in the image a corresponding text file with category id, and bounding box parameters normalized by image dimensions is required. The format is as follows -

*class\_id x y width height*

I have written a few functions to convert the bounding box parameters from COCO to YOLO format, obtain category ids and create txt files for each image.

Eventually we have 191962 images and corresponding text files for training, 32153 images and corresponding text files for validation and 62629 images in test data.

Besides these folders, the input to darknet for training YOLO needs -

1. classes.names file containing the name of each category
2. train.txt having full path to each image in the training data
3. test.txt having full path to each image of validation data
4. labelled\_data.data having the following 5 lines in order -
   1. classes = *number of classes*
   2. train = *full path to train.txt*
   3. test = *full path to test.txt*
   4. names = *full path to classes.names*
   5. backup = *location to store the weights file.*

These files have been created using corresponding python functions.

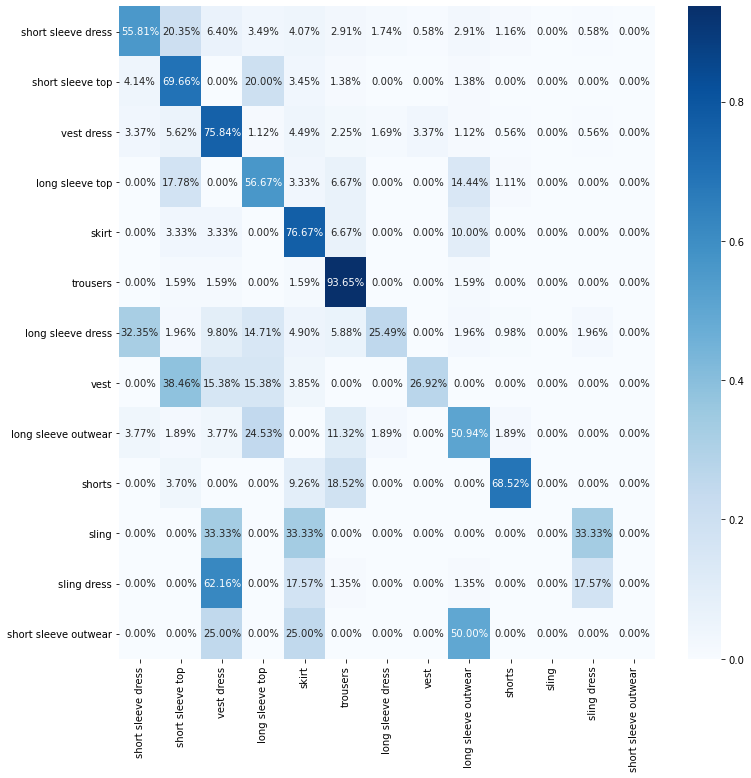
Preprocessing file list:

1. CV fashion data analysis.ipynb
2. Creating train.txt and test.txt.ipynb
3. S3 data transfer.ipynb

**GUI and results**

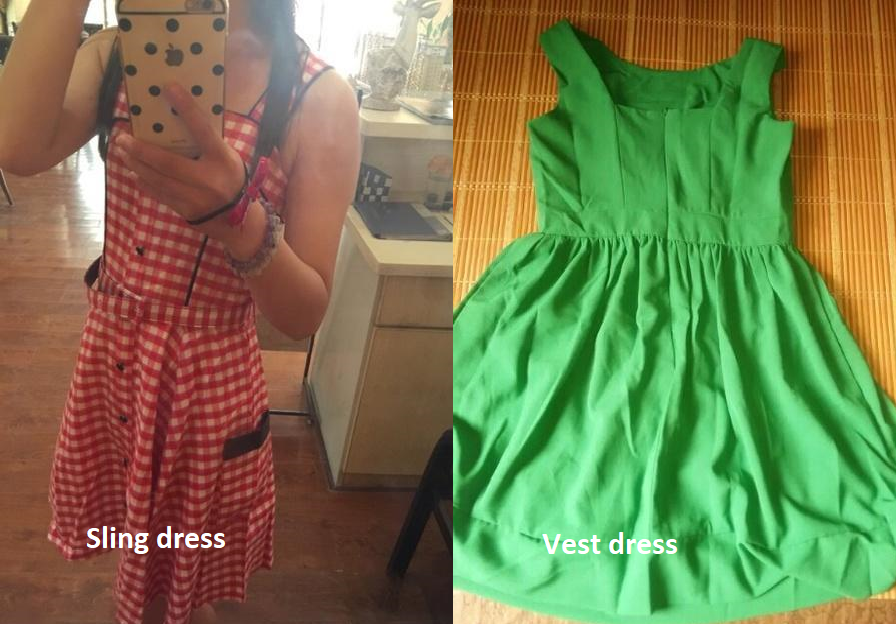
Filtering out about 1412 training images with only a single label per image and comparing them with the input produces the below confusion matrix.

Comparing this result with the distribution of the training data, the 6 classes with the highest available train images - short sleeve top, trousers, skirt, shorts, vest dress and long sleeve top show a better probability of detection as compared to 7 remaining classes that comprise less than 36% of the total train data.



# figure.2. Confusion matrix on train data

The highly misclassified category-sling dress has been classified to vest dress. This is acceptable because vest dresses and sling dresses look quite similar in many examples (see sample image below) and the training samples of vest dresses are 6 times that of sling dresses.



This analysis suggests that the algorithm has not fully converged yet since it picks out high level differences but fails to understand the low level differences. I had executed the algorithm for 2 complete epochs. I believe a higher number of training epochs would have produced better results.

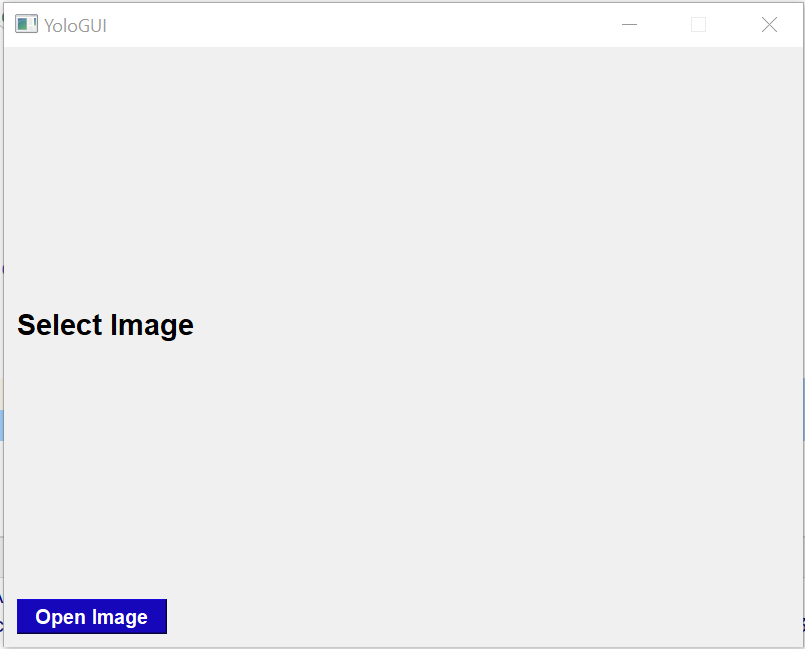
A similar analysis on the test set with 727 single class input images yields the below output. This is quite similar to the training set result.

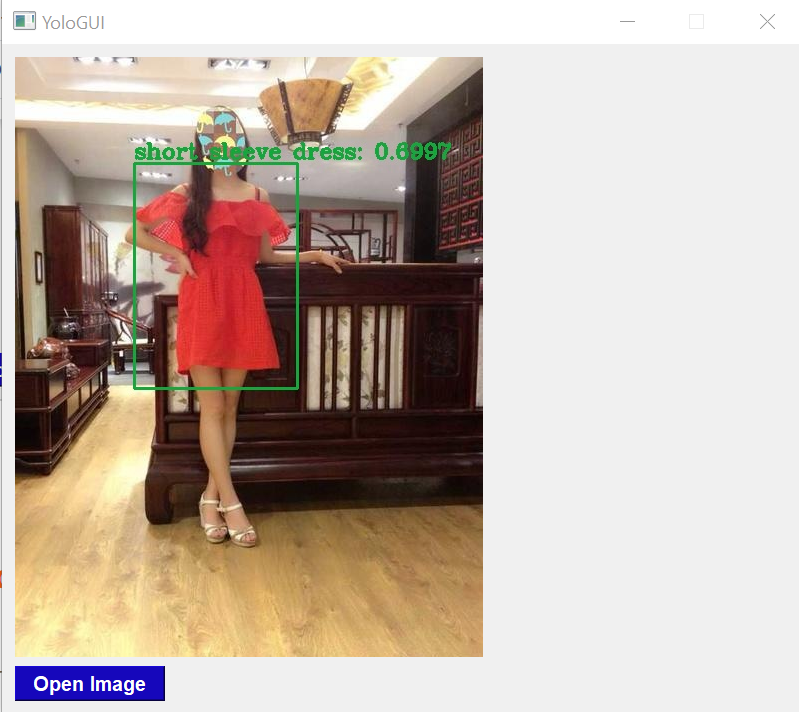


# figure.2. Confusion matrix on test data

Nonetheless, when checked visually the output is quite acceptable and since YOLO is best known for giving quick visual output for object detection, I created a GUI that allows the user to select the input image and uses the calculated weights to predict the object.

I created the gui using PyQt GUI designer module and integrated it with darknet using opencv-python. The console output shows the time taken to detect and the number of objects detected. The app and result are as follows -

I



# figure.3. GUI landing page and output

**System Requirements**

Amazon AWS EC2 GPU

Softwares - cmake, opencv, Yolo\_mark, darknet, anaconda for Python.

The next step is darknet installation on aws GPU. I used the steps given by <https://pjreddie.com/darknet/install/> for this.

**Training Parameters**

Total training time - 48hrs on AWS GPU