YOLO

Naïve learn about number of region of methods for recognizing and locating multiple objects in a scene.

Architecture like

1- Faster R-CNN are accurate, but the model itself is quite complex with multiple outputs that are each a potential source of error.

Once trained they `re still not fast enough to run in real time.

In this lesson we will be learning about YOLO which stands for You only look once and it`s a real time object detection algorithm that avoid spending too much time on generating **region proposals**.

Instead of locating objects perfectly it priorities speed and recognition.

Let`s see an example of YOLO in action.

Consider a self-driving car that sees this image of street.

It`s essential to self-driving car to be able to detect the location of objects all arounds it

Such as pedestrians cars, and traffic lights. On the top , the detection has to happen in near real time

So that the car can safely navigate the street.

The car doesn`t always need to know what all these objects are?

It mostly need to know not to crash into them but it does need to recognize traffic lights bikes and pedestrians to be able to correctly follow the rules of the road.

YOLO

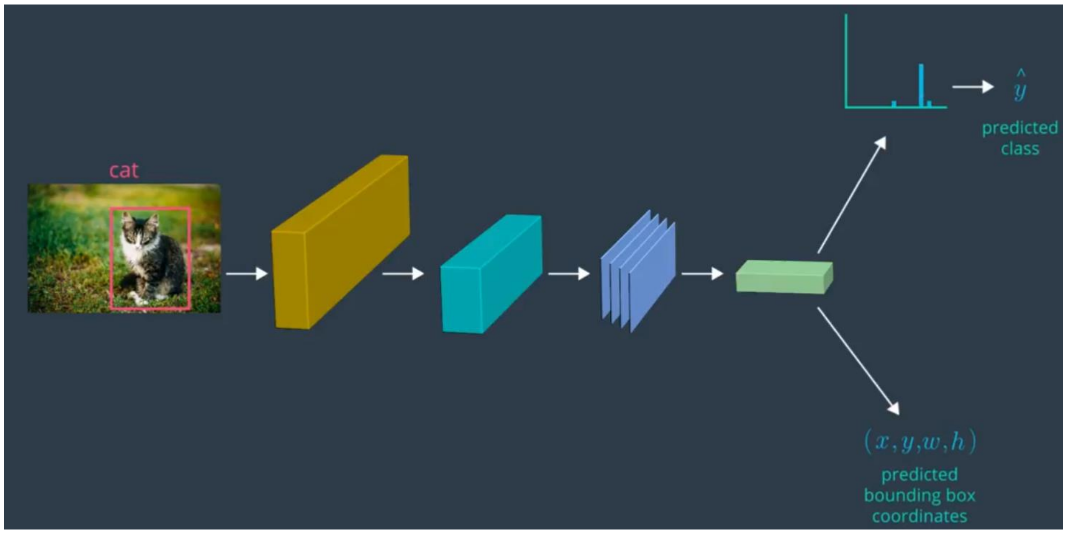
In this image we have used the YOLO algorithm to locate and classify different objects there`s a **bounding box** that locates each object and a corresponding class label.

In next we learn how the YOLO algorithms works.

When we took about localization in the image we talked about creating a CNN that could **output**

1- a Predicated class front object in an image
2- a Predicated bounding box for the object.

In the CNN examples that we have seen these outputs are analyzed separately in the network trains by using a weighted combination of **classification** and **regression losses**.



Another way to process these outputs is by merging them into a **single output vector which is what the YOLO algorithm does**

Let`s see this in an example.

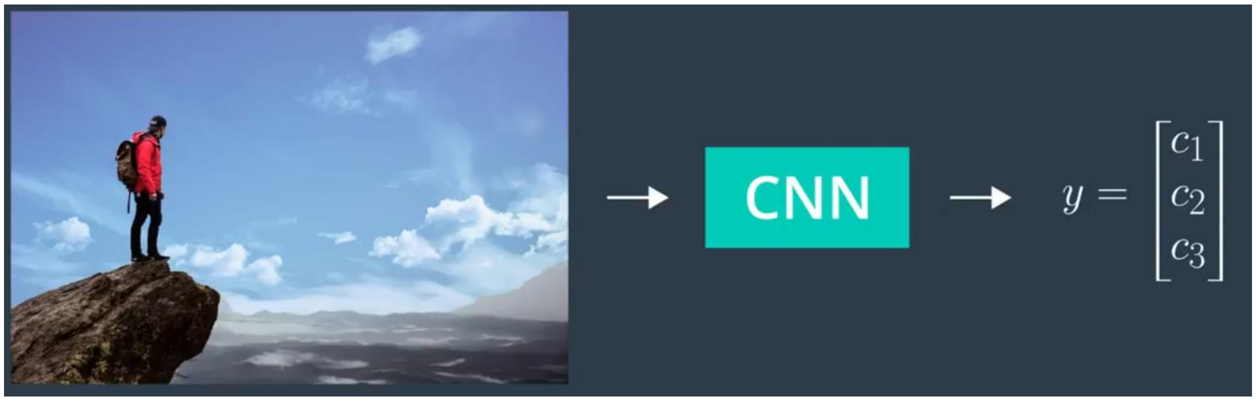
Let`s assume I want to train a CNN to be able to detect three classes,

1- Person
2- Cat
3- Dog

In this case, because we only have three classes,

1- The output vector Y will only have three elements,
   - **a. C1 = person**
   - **b. C2 = cat**
   - **c. C3 = dog**

Each of which is a class score or a probability that the image is of person or car or dog.



If you have more classes, this vector will get longer

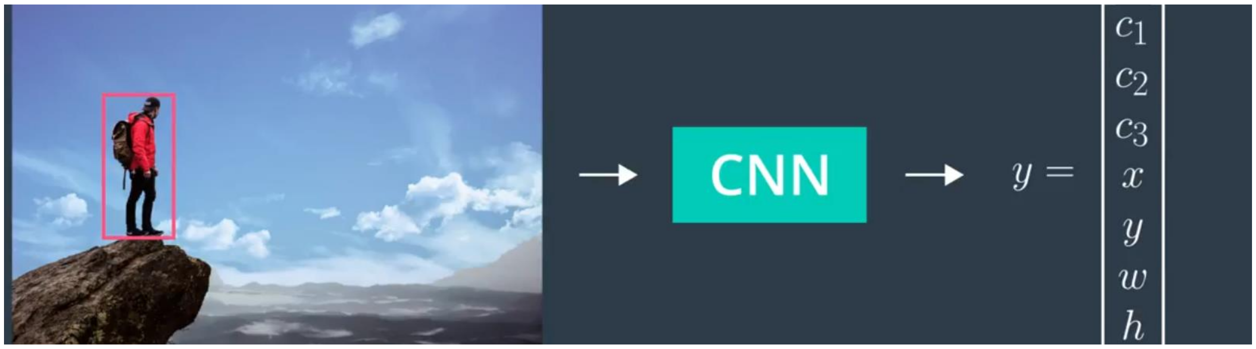
YOLO

For this image, we want to train the CNN so that it can identify the person in this image and look at that person within bounding box.

We can do this by adding **some bounding box parameters** to out output vector.

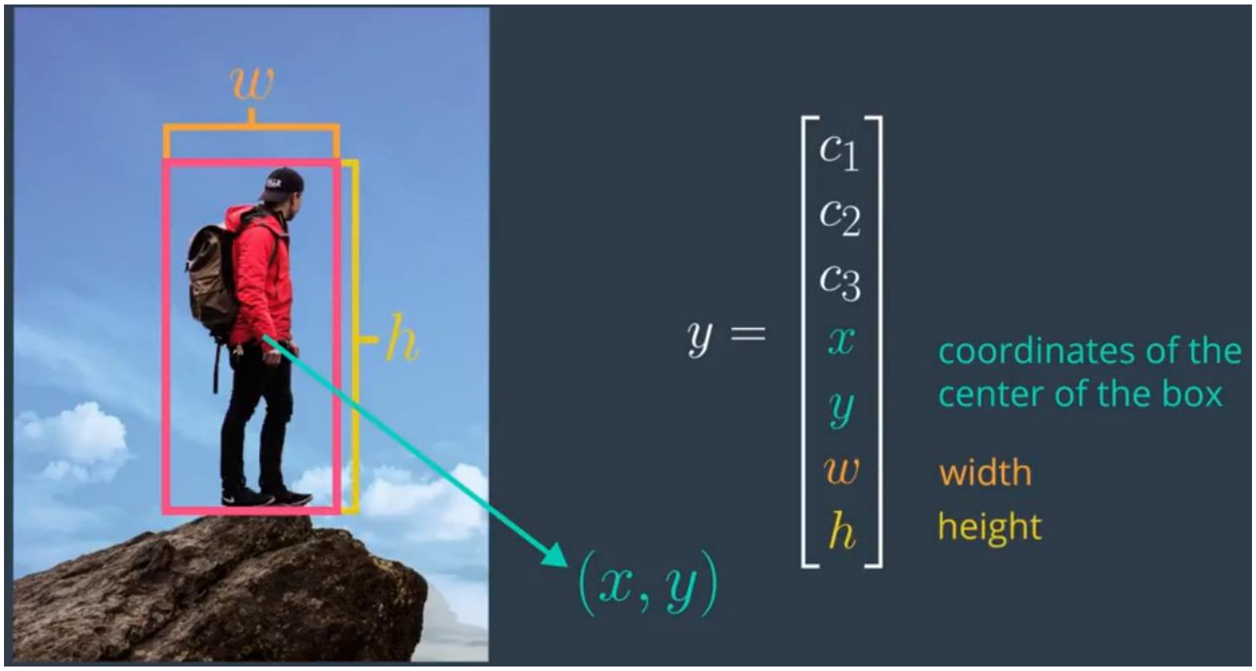
We can add four more numbers,

1- X
2- Y
3- W
4- H


CNN
$y = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix}$

That determine the position and size of the bounding box.

YOLO



w
h
(x, y)
$y = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix}$
coordinates of the center of the box
width
height

X and Y determine the coordinates of the center of the box, and w and H determine its width and height.

Once you have trained your CNN to output class probabilities and bounding box coordinates you `re **step closer to being able to detect objects** in any given image

Next we will go over the **sliding-windows** approach that you have seen before with our example output vector in mind.

Then you will see how YOLO improves upon sliding windows and breaks an image into grid for efficient object detection.

Since object, can be anywhere in given image you can make sure to detect all of them by

1- **sliding a small window over the entire image**
2- **checking for objects within each of the created window**

This is sliding window approach.

Let`s see how this works in detail.

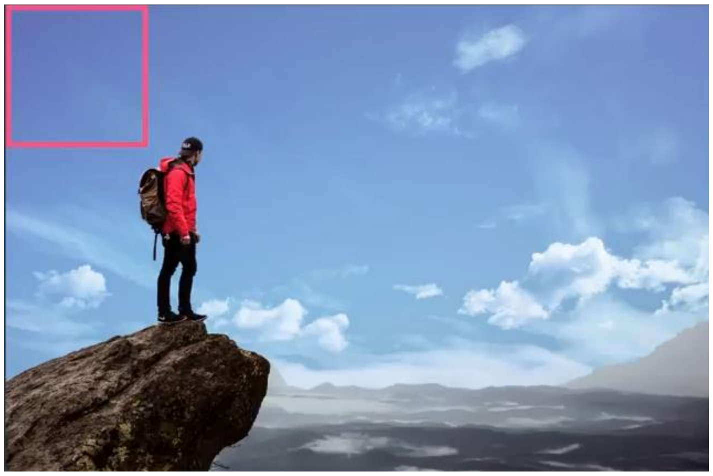
Suppose I have learned my CNN to detect my three classes.

1- A person = c1
2- Cat = c2
3- Dog = c3

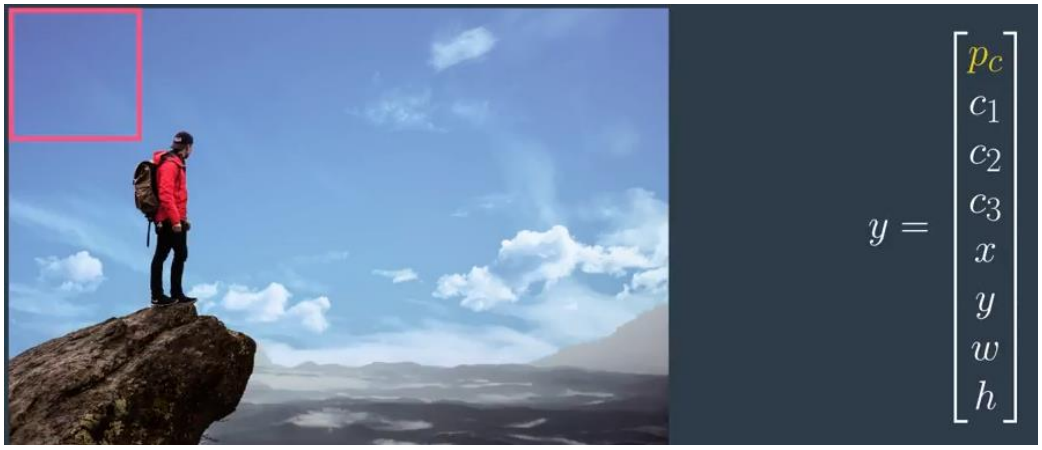
Now I want to use this train CNN to detect the person in this image.

a- **The first step in sliding windows is to choose the size of your window. We want it small enough to capture any small objects in an image.**

b- **Then we place our window at the beginning of the image and feed the region inside the window to the train CNN.**



c- **For each region, the CNN will output a prediction. Which is the output vector Y Notice the first element in this vector, PC, is different than what we have seen before.**


$y = \begin{bmatrix} p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix}$

PC is a probability between zero and one ,then an object exists within the window at all.

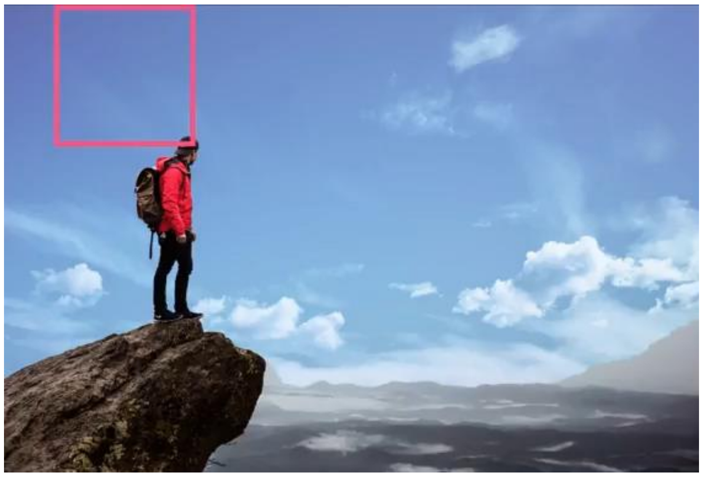
a- **If no object is detected we don`t have to proceed with trying to classify that particular region of the image.**

The next values in the vector are as usual. We have c1,c2 and c3 which correspond to the class of the object detect and the bounding box coordinates.

In this example, we see that first window region doesn't contain any of classes we are looking for no person , no cat and no dog.

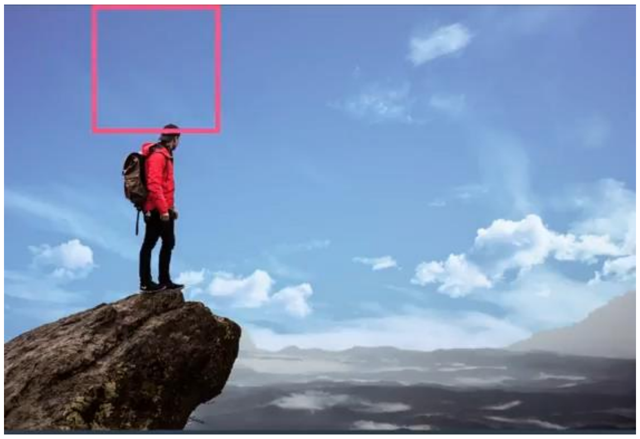
Therefore the CNN will output a vector with PC equal to zero, because not object were detect inside the window.

Then we move along and slide the window to the ray using some small stride and we repeat this process.



Small strides are used to make sure that we catch any object and to determine the location of objects within a few pixels of their location.

Now, since this region also doesn`t contain any objects, it will return PC equal to zero again.



We repeat this process until we cover the entire image.

You might also analyze the whole image again using windows of a different size.

One of detection windows might nicely capture small object in an image and another might capture large objects.

Here we see that when we feed the region inside this particular window to our CNN, it produce a y vector with a PC equal to one indicating that an object has been found and

$$y = \begin{bmatrix} 1 \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix}$$

It indicates that the objects is a person. With C1 equal to one.

$$y = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix}$$

In reality these probability values will typically be very close but not quit equal to one because of some Uncertainty.

Our model also give us the predicated bounding box coordinates which are determined by the window.

YOLO

The sliding window approach works well, but it`s a very computationally expensive because we have to scan the entire image with window of different size and each window has to be fed into the CNN.

You have seen that one way to get around this problem is to project regions of interest in the input image to a layer deeper in the CNN into a set of feature maps.

That way you can process an image through several convolutional and pooling layers just ones and use the resulting feature maps to analyze different regions of the input image.

But YOLO takes a different approach , and again look at each part of an image only once without overlapping windows.

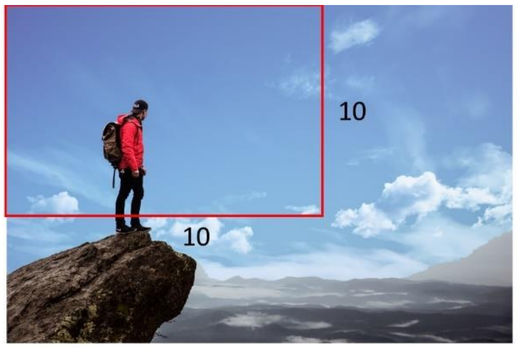
How do you think you might break up an entire image so that you could analyze it without looking at one region more than once.

A Convolutional Approach to Sliding Windows

Let’s assume we have a 16 x 16 x 3 image, like the one shown below. This means the image has a size of 16 by 16 pixels and has 3 channels, corresponding to RGB.

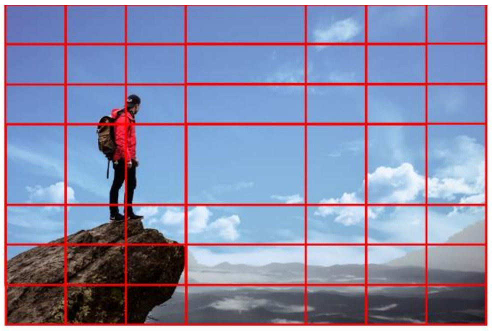
16 x 16 x 3

Let’s now select a window size of 10 x 10 pixels as shown below:



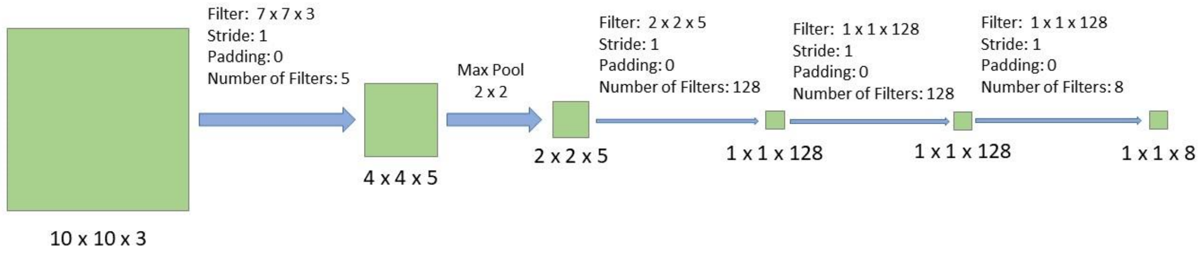
16 x 16 x 3

If we use a stride of 2 pixels, it will take 16 windows to cover the entire image, as we can see below.



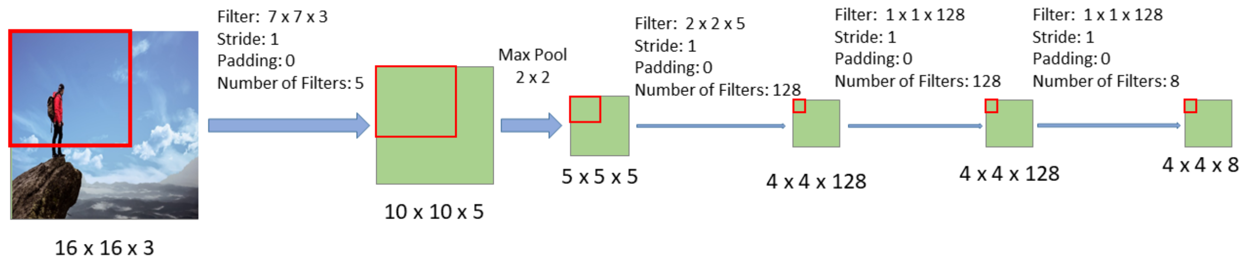
16 x 16 x 3

In the original Sliding Windows approach, each of these 16 windows will have to be passed **individually through a CNN**. Let's assume that CNN has the following architecture:



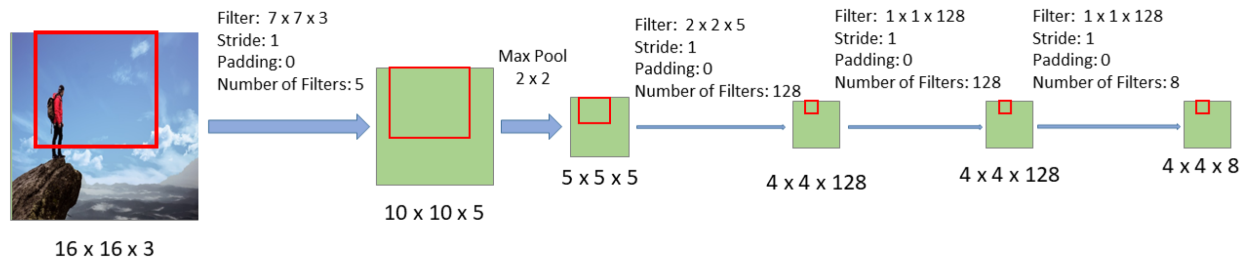
The CNN takes as input a 10 x 10 x 3 image, then it applies 5, 7 x 7 x 3 filters, then it uses a 2 x 2 Max pooling layer, then is has 128, 2 x 2 x 5 filters, then is has 128, 1 x 1 x 128 filters, and finally it has 8, 1 x 1 x 128 filters that represents a **softmax** output.

What will happen if we change the input of the above CNN from 10 x 10 x 3, to 16 x 16 x 3? The result is shown below:

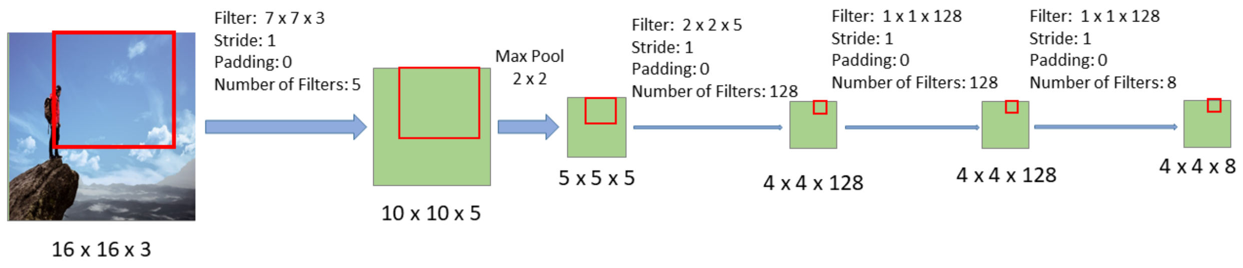


As we can see, this CNN architecture is the same as the one shown before except that it takes as input a 16 x 16 x 3 image. The sizes of each layer change because the input image is larger, but the same filters as before have been applied.

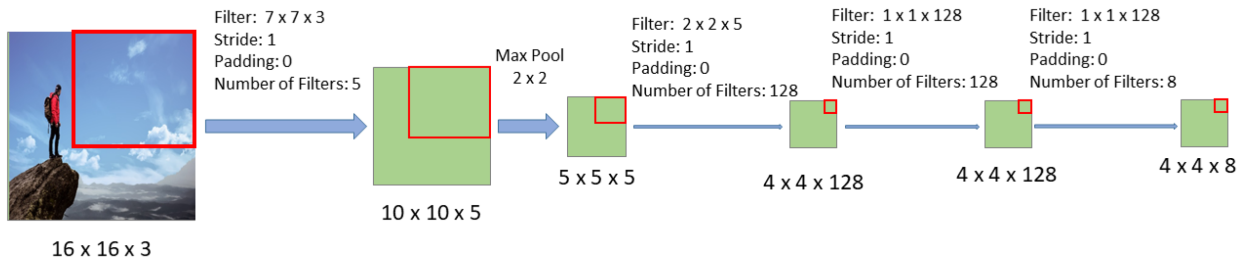
If we follow the region of the image that corresponds to the first window through this new CNN, we see that the result is the upper-left corner of the last layer (*see image above*). Similarly, if we follow the section of the image that corresponds to the second window through this new CNN, we see the corresponding result in the last layer:



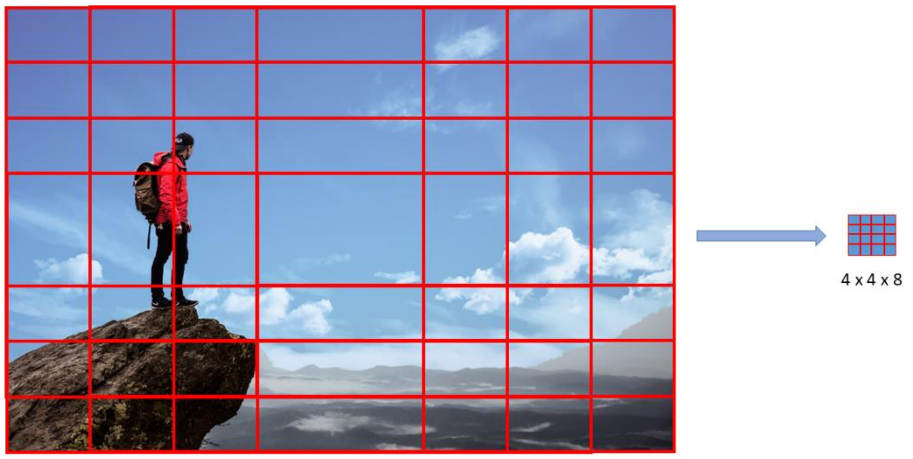
Likewise, if we follow the section of the image that corresponds to the third window through this new CNN, we see the corresponding result in the last layer, as shown in the image below:



Finally, if we follow the section of the image that corresponds to the fourth window through this new CNN, we see the corresponding result in the last layer, as shown in the image below:



In fact, if we follow all the windows through the CNN we see that all the 16 windows are contained within the last layer of this new CNN. Therefore, passing the 16 windows individually through the old CNN is exactly the same as passing the whole image only once through this new CNN.



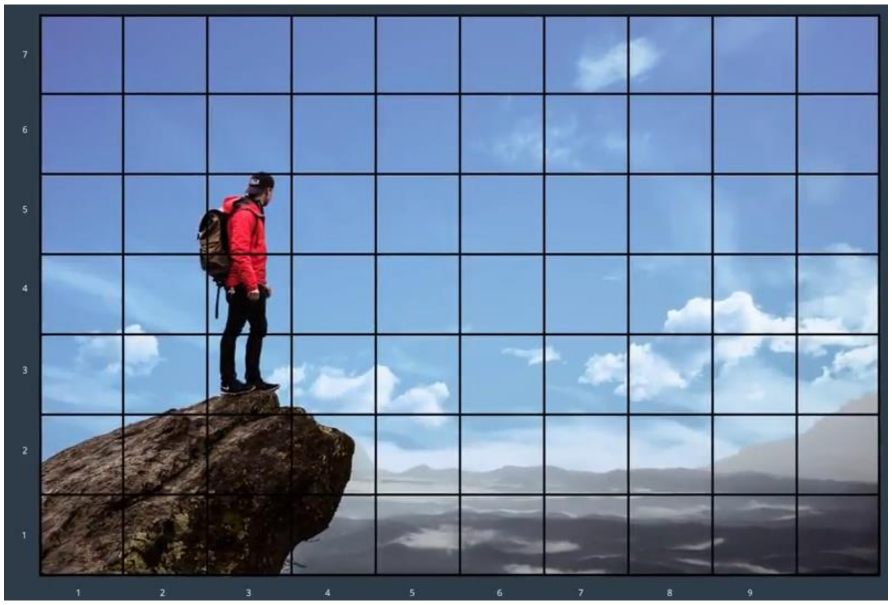
This is how you can apply **sliding windows** with a CNN. This technique makes the whole process much more efficient. However, this technique has a downside: the position of the bounding boxes is not going to be very accurate. The reason is that it is quite unlikely that a given size window and stride will be able to match the objects in the images perfectly. In order to increase the accuracy of the bounding boxes, **YOLO uses a grid instead of sliding windows**, in addition to two other techniques, known as Intersection **Over Union and Non-Maximal Suppression**.

The combination of the above techniques is part of the reason the YOLO algorithm works so well. Before diving into how **YOLO puts all these techniques together, we will look first at each technique individually**.

Using a Gird to improve Localizations

The implementation of sliding windows is very slow but it can be faster it you choose a stride so that Each window covers a new part of an image and there`s and there`s no overlap Inspired of this approach.

YOLO use a gird instead of sliding windows so let`s see how this works ?



In this example, we are using a seven by 10 gird in the YOLO algorithm, a much finer gird is used but **the overall process will be the same**. Now you might be wondering how we can get an accurate bounding box out of a gird.

This wan one of the challenges with sliding windows too.

How can you account for the fact that these gird cells are Well , the idea is that we can assign output vectors each gird cell , so each cell will have an associated vector that tells us one.

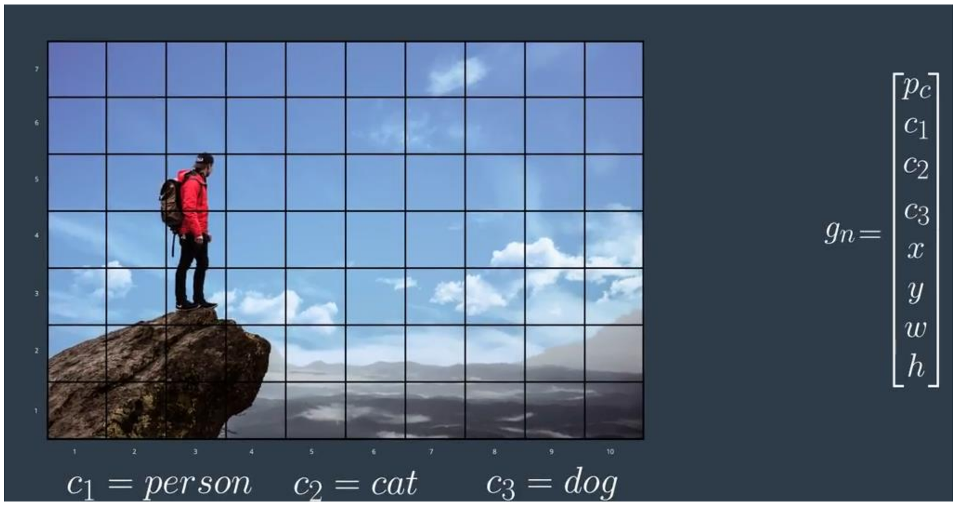
Associated Vectors

1- **If an object is in that cell.**
2- **The class of the object.**
3- **The predicated bounding box for that object**

Note in this way the bounding box coordinates do not have to be contained within a gird cell

Assuming we have an input image with two labels and bounding box, we can then train a CNN to produce the correct output vectors for each of these gird cells

We will call the output vector for each gird cell gn.

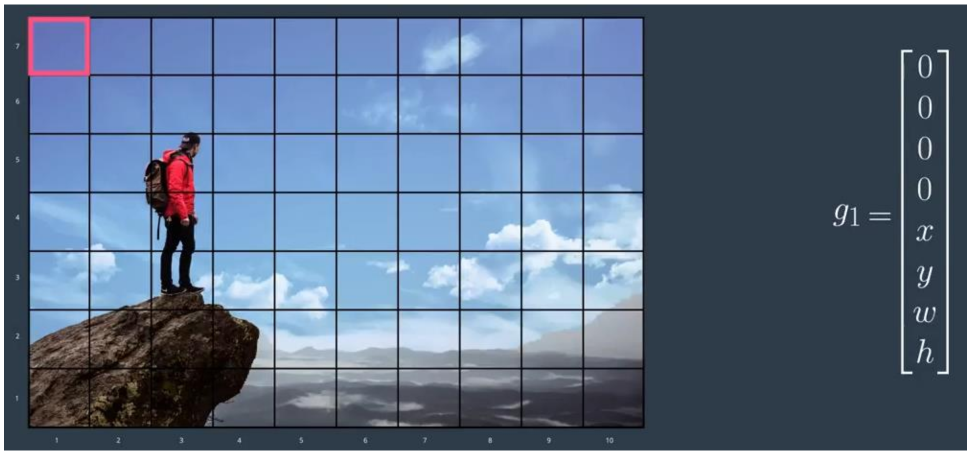


$g_n = \begin{bmatrix} p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix}$
$c_1 = person$
$c_2 = cat$
$c_3 = dog$

The output vector contains the same parameters as the output vector y we saw in previous lecture.

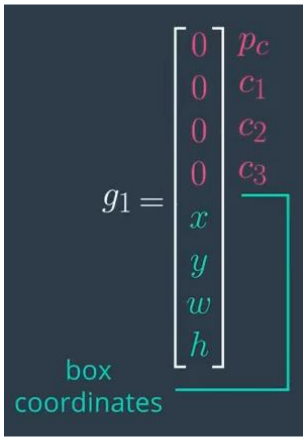
For the first cell the vector g1 will look like this. There is no objects in this gird cell,

So PC equals zero.

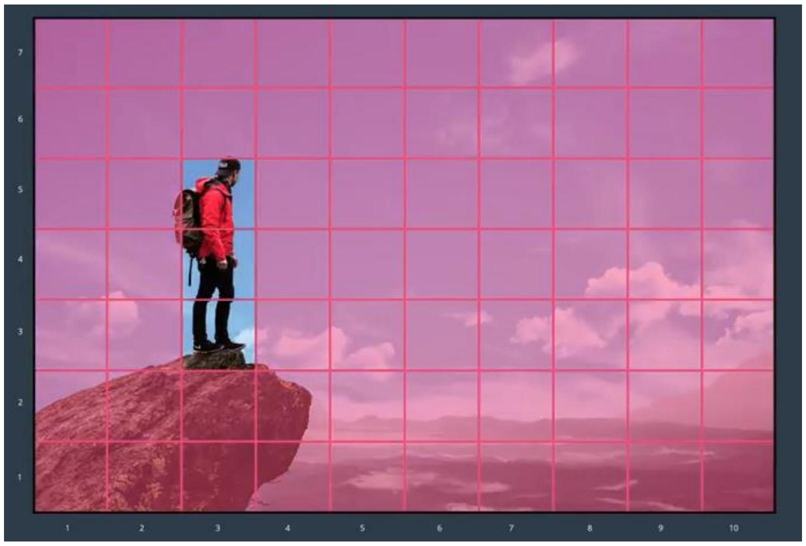


$g_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix}$

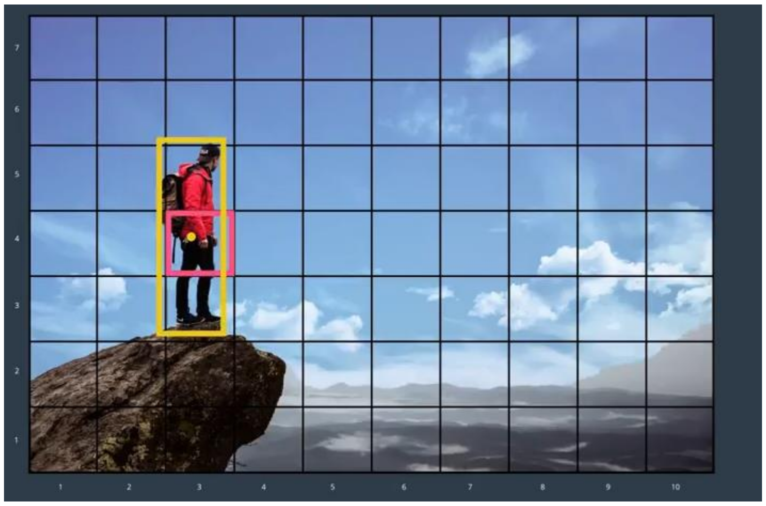
The vector will have zero for the class scores and some values for box coordinates.


$$g_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix} \begin{matrix} p_c \\ c_1 \\ c_2 \\ c_3 \\ \\ \\ \\ \\ \end{matrix}$$

The values don`t matter much because we will discard vectors with too low of a PC value we will get the same output vector for all the gird cells that have no object in them.



Now what about this cell on the top of the person in our image ?



This output vector which is numbered by gird cell looks like this.

$$g_{33} = \begin{array}{cl} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix} & \begin{array}{l} p_c \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \end{array}$$

Pc equals one because there is an object in the gird cell, and c1 equals one because the object s a person.

$$g_{33} = \begin{array}{cl} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix} & \begin{array}{l} p_c \\ c_1 \\ \\ \\ \\ \\ \\ \\ \end{array} \end{array}$$

The output vector will also hold the predicated bounding box coordinates and we will see how these are generated later in the lesson.

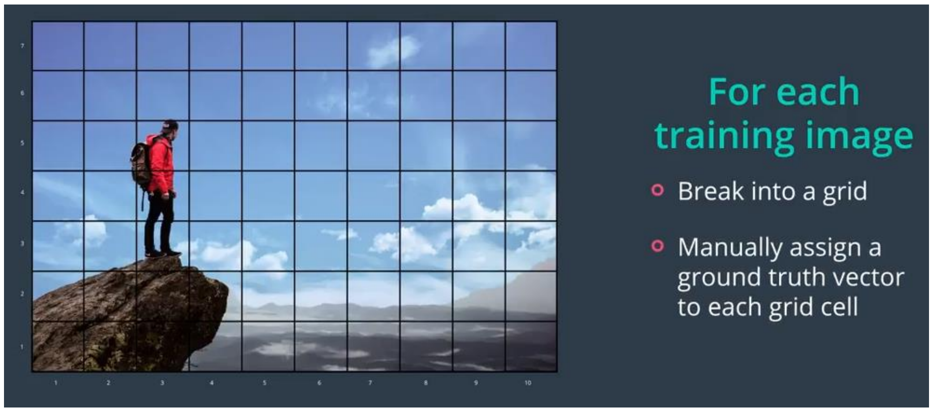
Know that we know what gn vector look like, let`s see how we can use them to train a CNN.

$$g_{33} = \begin{array}{cl} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix} & \begin{array}{l} p_c \\ c_1 \\ \\ \\ \\ \\ \\ \\ \end{array} \end{array}$$

predicted bounding box coordinates

Train on a gird require a specific kind train of data.

To train a network to output a **predicated vector** of class score and box coordinates for each cell we need to have a **true vector to compare** it to. So for each train image we have to break it into a gird, and **manually sign a ground truth vector to each gird cell.**

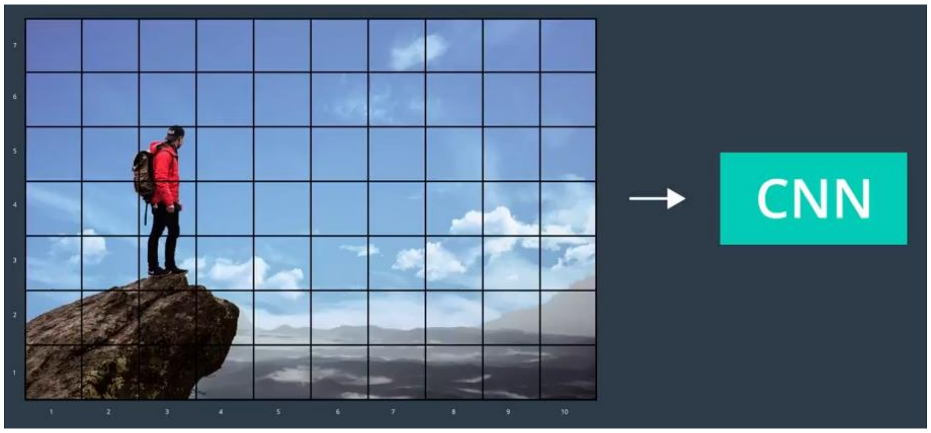


For each training image

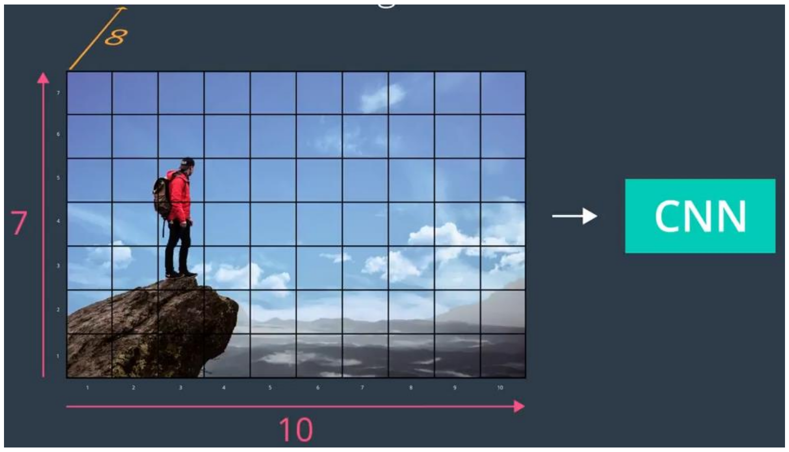
- Break into a grid
- Manually assign a ground truth vector to each grid cell

Once we have this gird cell labeled training data,

The second step to design a CNN that can be trained using vectors.



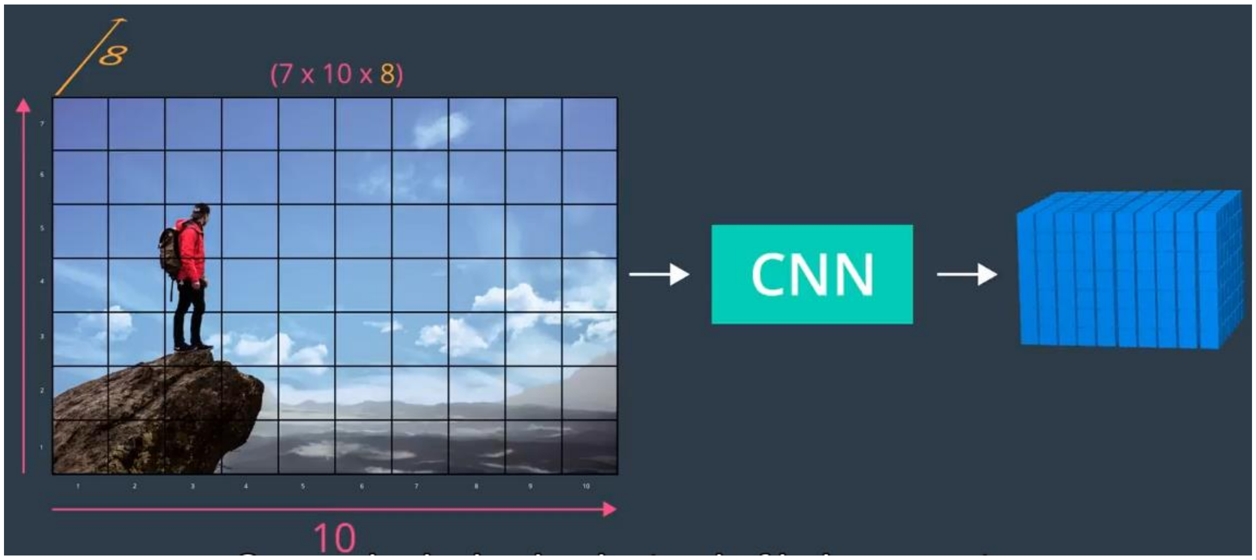
Since we have a seven by 10 gird in our example, and each cell has an associated eight-dimensional ground truth vector.



We have to design our CNN such that the output layer of the CNN is going to have a size seven by 10 by eight

(7 x 10 x 8)

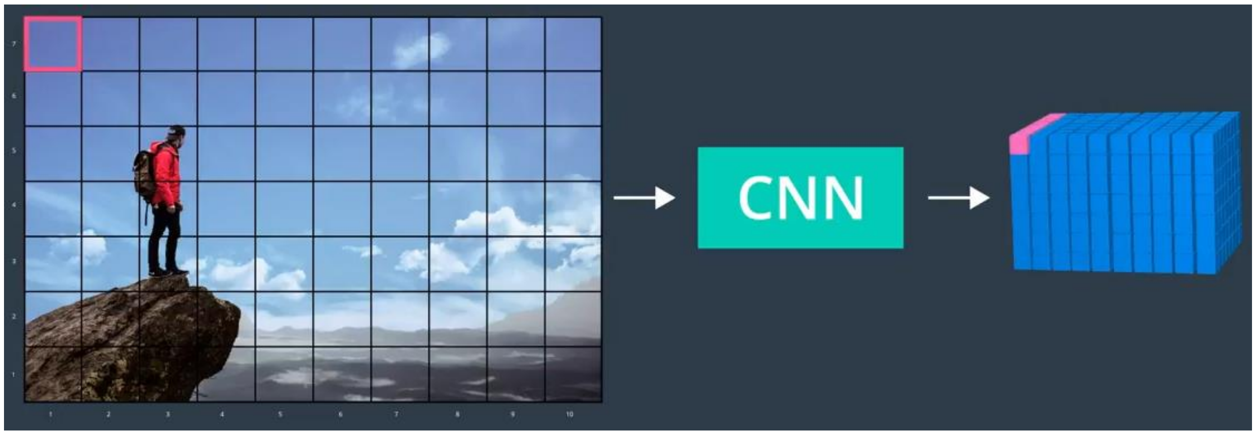
We can think of this as a seven by 10 image with a depth of eight.



So each pixel value instead of being a vector of length three as in RGB images, is an eight dimensional output vector.

This way for each input grid cell, there`s an eight dimensional output vector and the output layer of the CNN.

For example when the network sees the first gird cell, it will produce an output vector in the upper left corner of the output layer.



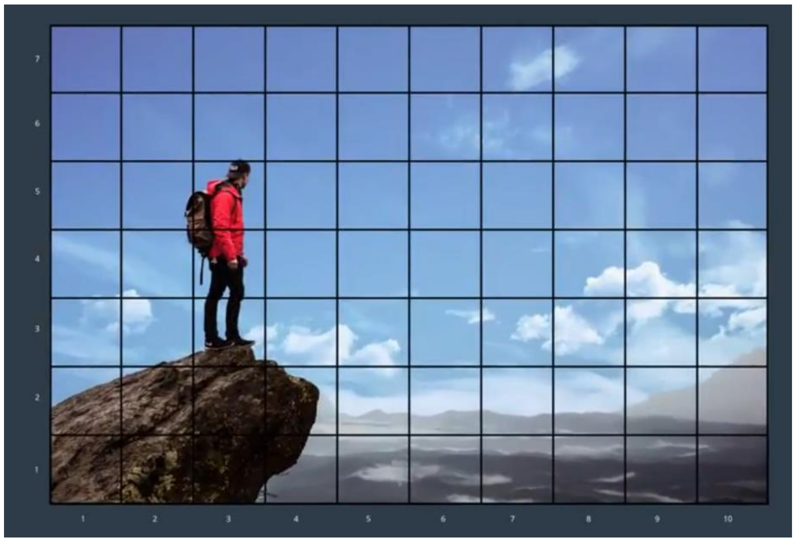
Having defined this output shape, we can train CNN

1- **using images and**
2- **their ground truth gird vector as input**.

Once the CNN has been trained, we can used to detect and localize objects in test images.

Next we let`s see how this method produces accurate bounding boxes

Generating the Bounding box



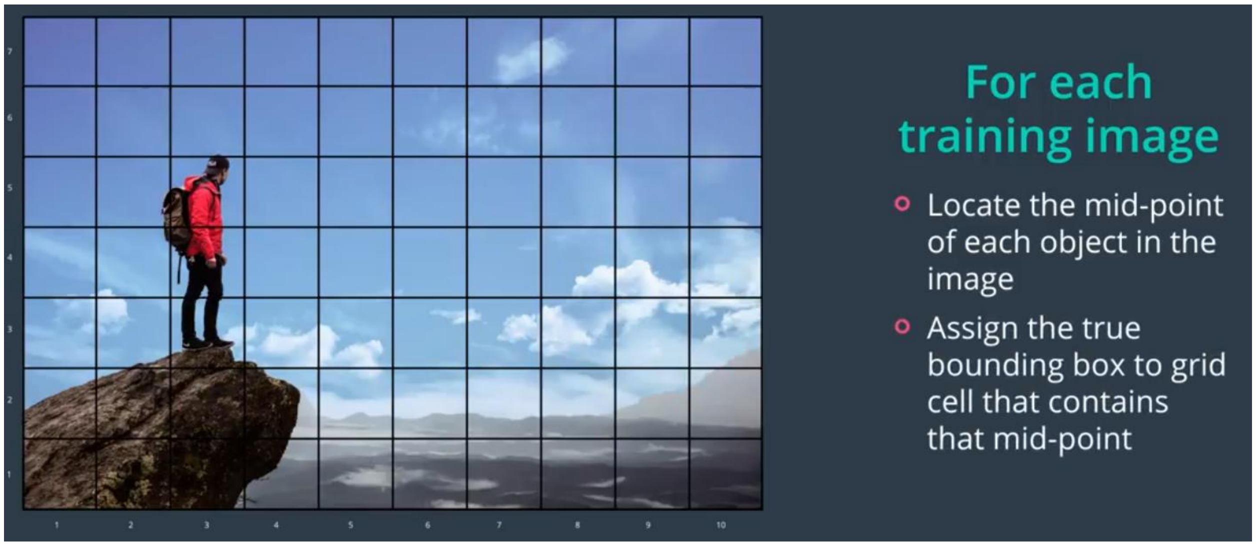
How do the YOLO finding correct a bounding box when look at the image broken by gird.

The trick is that it assigns **the ground-truth bounding box** for one object in an image to only grid cell in the training image. So **only one gird cell** is **meant to locate the object.**

Now how is this gird cell chosen ?

For each training image

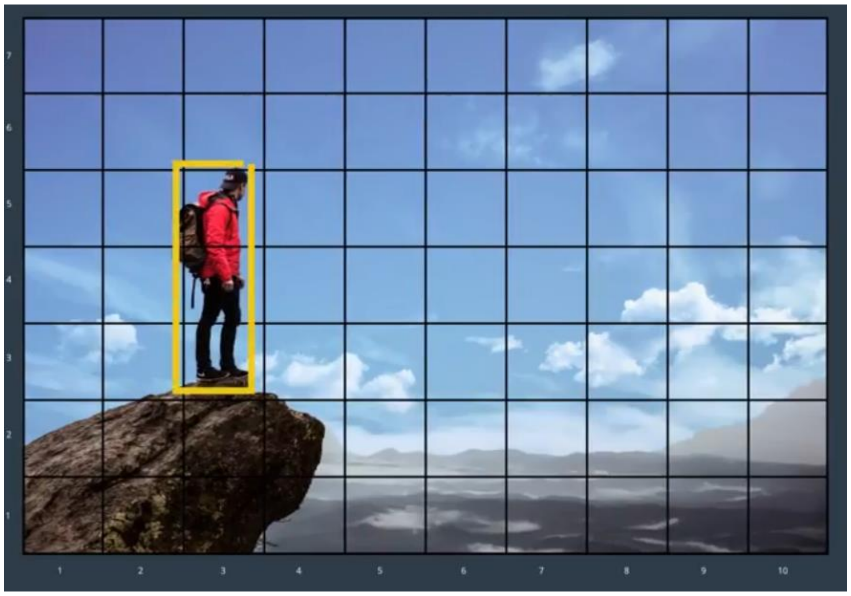
1- **we locate the midpoint of each object in the image then**
2- **we assign the true bounding box to the gird cell that contain that midpoint.**



For each training image

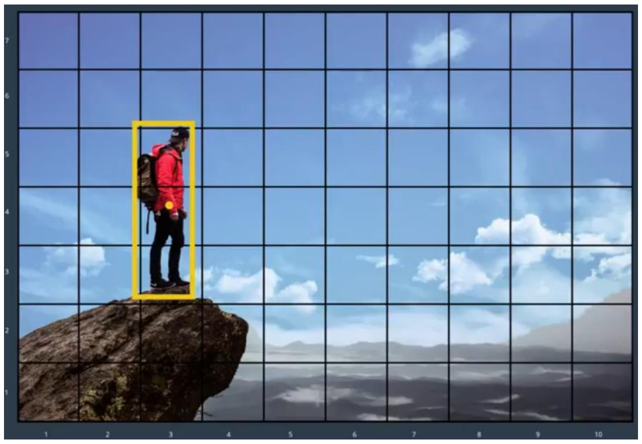
- Locate the mid-point of each object in the image
- Assign the true bounding box to grid cell that contains that mid-point

Let`s see an example.

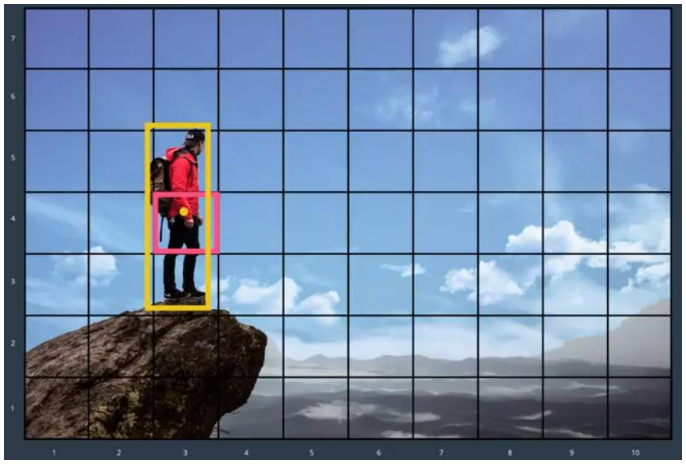


YOLO

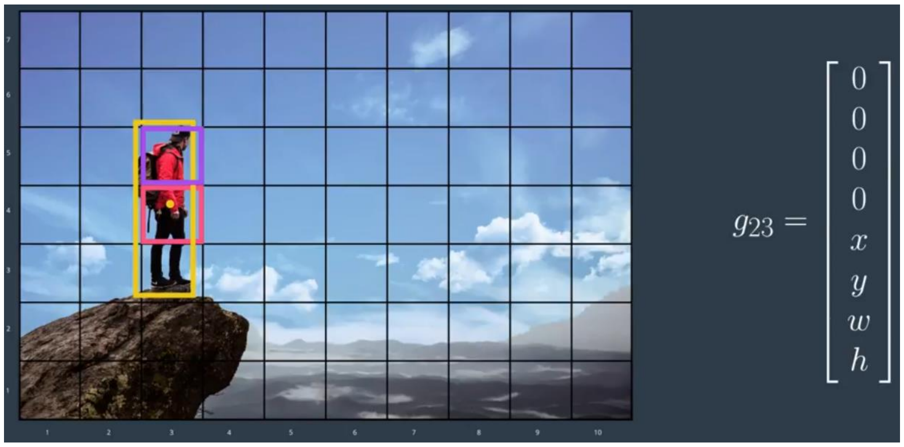
In this image of a person, we locate the midpoint of this person indicated by the yellow dot.



This dot contained by one gird cell, so we assign the ground-truth bounding box this gird cell alone and the ground truth vector for this gird cell which will be used for training will look like this



And even thoughts this other gird cell also contains part of the person it`s ground-truth vector will look this.



$$g_{23} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix}$$

We treat it as

1- **if does not contain an object and its pc value equal zero.**

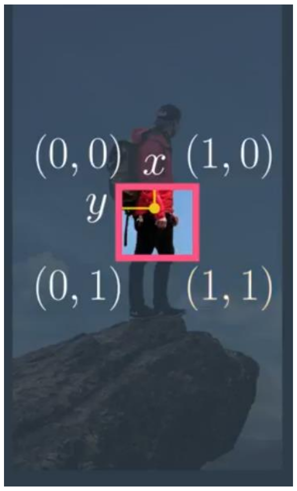
Now let`s see how we determine the numerical values of

1- **X**
2- **Y**
3- **H**
4- **W**

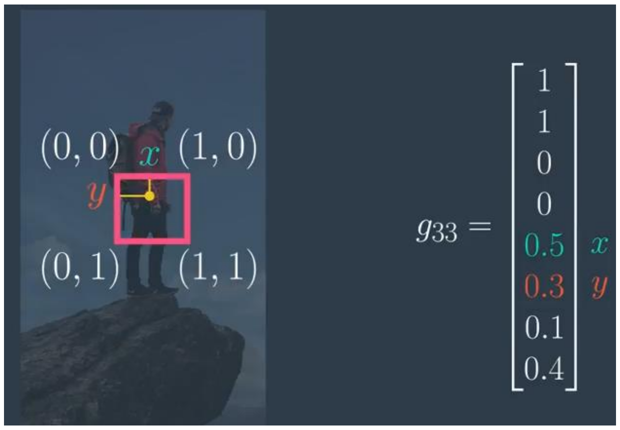
IN, the YOLO algorithm, X and Y determine the coordinates of center of the bounding box relative to the gird cell

And W and H determine the width and height of the box relative to the whole image,

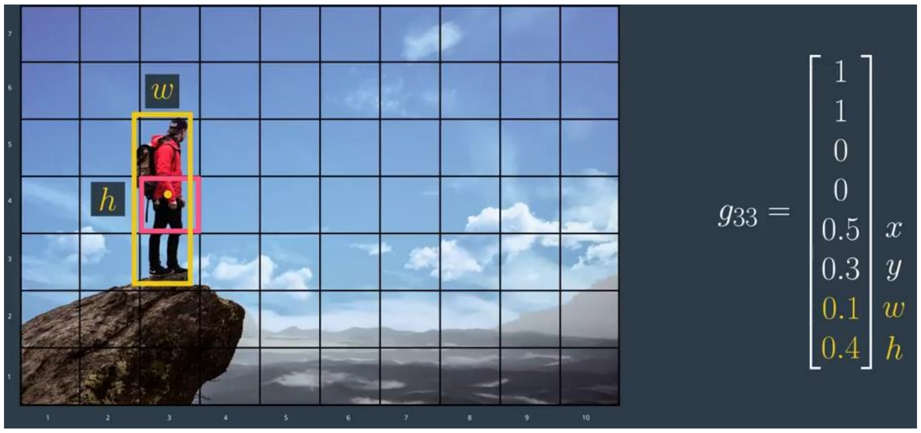
The convention is that the upper left corner of a gird cell has coordinates (0,0) or the bottom right hand corner has coordinates (1,1).



So in the example, the center point relative to the gird cell coordinate system is X is equal to about 0.5 and Y equal to 0.3.


$(0,0)$
x
$(1,0)$
y
$(0,1)$
$(1,1)$
$g_{33} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0.5 \\ 0.3 \\ 0.1 \\ 0.4 \end{bmatrix}$
x
y

Now the width of the predicated bounding box W is 0.1 because it width is about 10 percent the width of the entire image and the height H is 0.4 because its height is about 40 percent the height of the entire image.



w
h
$g_{33} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0.5 \\ 0.3 \\ 0.1 \\ 0.4 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ x \\ y \\ w \\ h \end{matrix}$

Notice that in this system, all bounding box coordinates values fall between zero and one and the width and height of the bounding box can be bigger than the size of the grid cell.

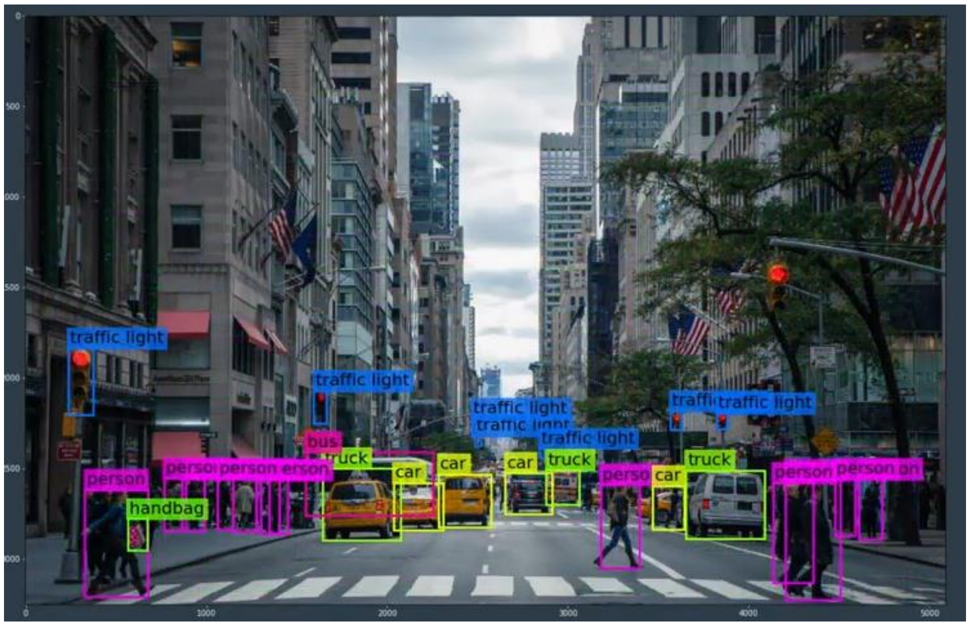
This technique is very similar to normalization.

By standardizing the range of these values, this algorithm become easier to train and converge to a smaller error. But there`s one problem with this model imagine that a network has trained on a fine grid and only one small grid cell has a true the bounding box for an object in an image what do you think will happen a network like this sees a new test image with an object in it

Too Many box

One of the problem with the grid-based method for object detection is that a trained a CNN when faced with a new test image will often produce multiple grid cell vectors that are all trying to detect the small object.

This means lots of object output vectors that all contain slightly different bounding boxes for the same object To account for this, we use a technique called non-maximal suppression.



non-maximal suppression which tries to find the bounding box that best matches an object in an image.

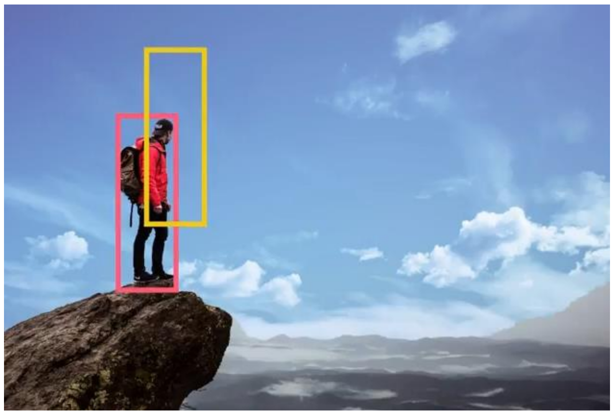
Next, we will learn how it finds the best box in a group.

Intersection over Union IOU

Before learning non-maximal suppression we will need to learn about intersection over union or IOU, which is a technique used in non-maximal suppression to compare how good two different bounding box are a given object.

It`s easiest to see how to calculate IOU in an example.

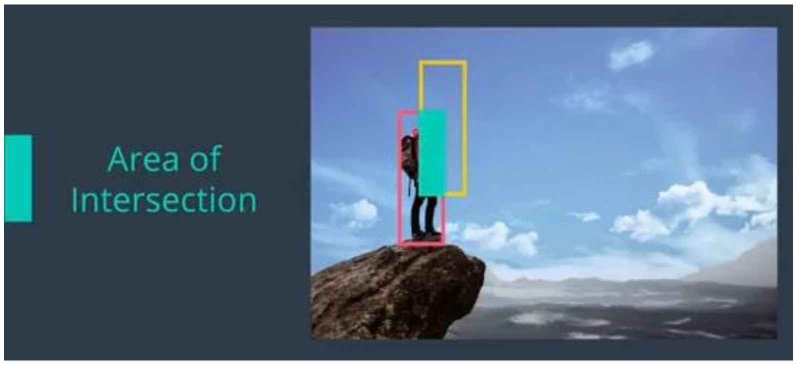
Take these two bounding boxes.



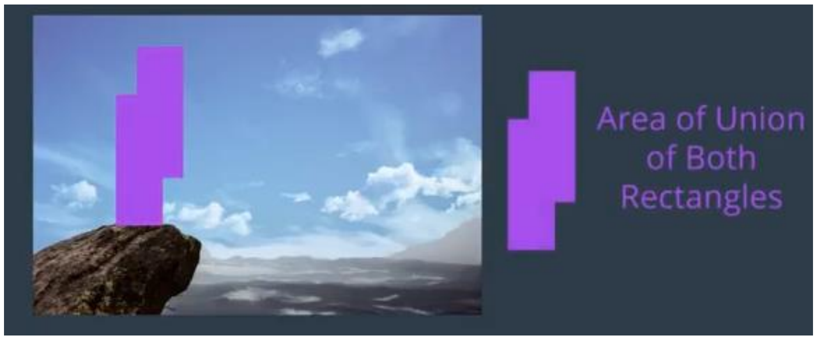
We define the intersection over union for the two boxes to be the ratio of the area of intersection to the area of union.

$$IOU = \frac{\text{Area of Intersection}}{\text{Area of Union of Both Rectangles}}$$

The area of intersection between the two boxes is marked by the green rectangle and it`s just the area where boxes overlap.



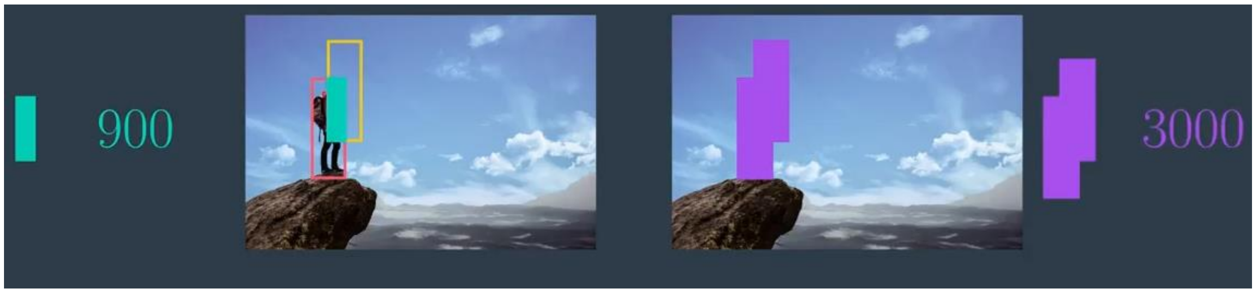
The area of union is denoted by the purple area.



And the total area of the boxes

1- If they were smooched into one bigger unified shape

The area of intersection is 900 square pixels while the union of the boxes is 3000 square pixel.



This is mean the IOU of the boxes is 900 over 3000 or 0.3

Now imagine you are comparing a ground-truth box to predicated box.

What IOU value would indicate a good match?

Non-Maximal Supersession

For test image broken down into a grid, how can we handle the case in which our grid produces

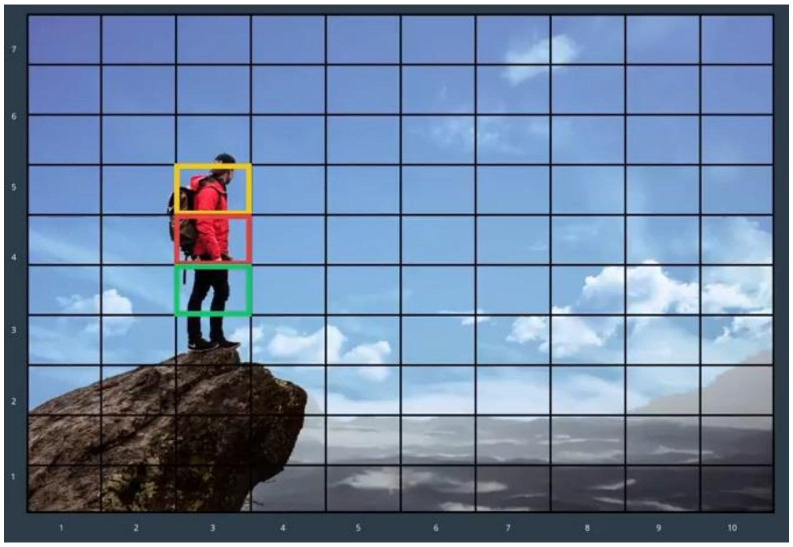
1- **multiple grid cell vectors**
2- **multiple bounding boxes**

for the same object ?

To account for this, we use a technique called non-maximal suppression.

This is uses the IOU between two predicated bounding boxes to select the **best bonding box.**

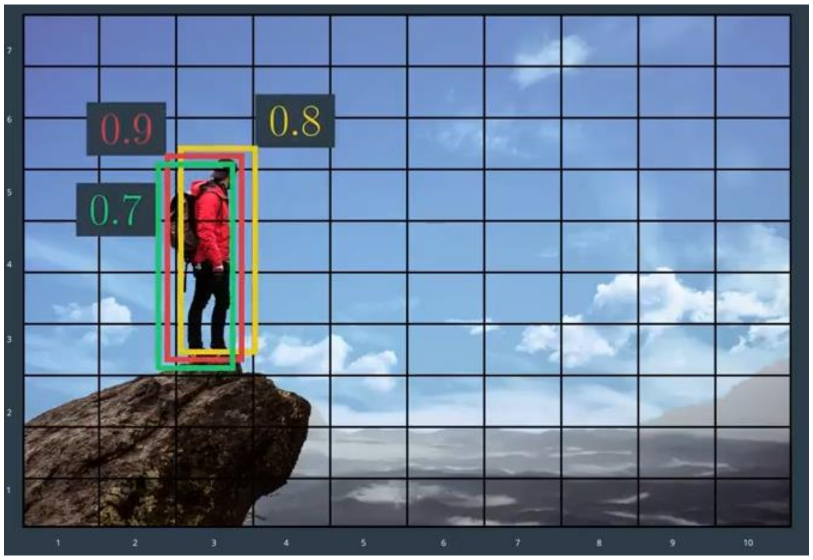
Let`s see an example .In this example



These three grid cells have a non-zero PC value, meaning they have all detecting an object and bounding box for that object.

If we plot the bounding boxes predicated by each grid cell along with the value of a PC, we get this.

Three bounding boxes for the same object.



But the PC value associated with each box is different.

1- **We see that the first bounding box has a PC value of 0.8**
2- **The second has a value of 0.9**
3- **And last has a value of zero 0.7**

PC is a measure of confidence that there has been an object detected, and so high PC mean a high confidence of object detection.

Non-Maximal Suppression

High p_c = high confidence of object detection

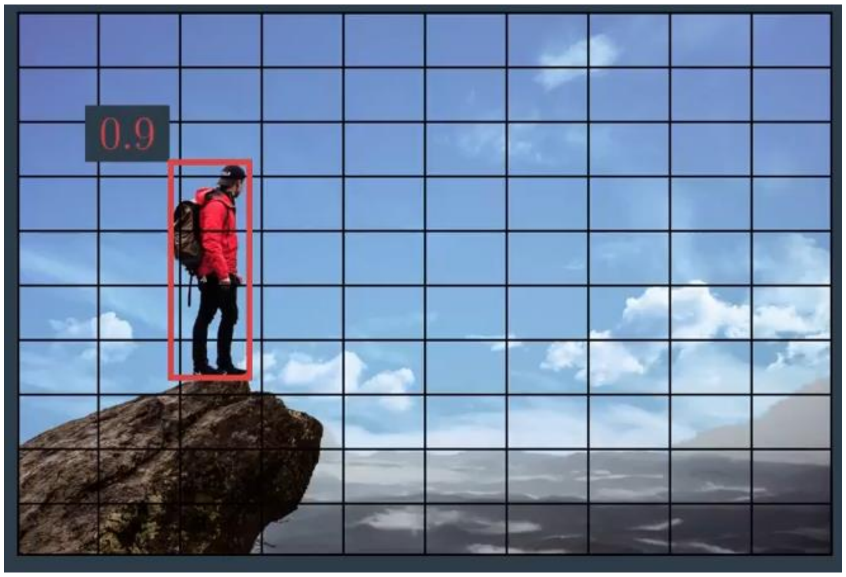
Now non-maximal suppression selects only the bounding box with the highest PC value.

In this case highest PC value is the red bounding box with a value of 0.9 .

It then removes all of the bounding boxes that have a high IOU value when compared to this best bounding box that I just selected.

In this way it gets rid of the boxes that are too similar to the red bounding box.

Then you are left with just one bounding box, which should correspond to the best prediction.



This is called non-maximal suppression because you suppress overlapping bounding boxes that do not have the maximum probability for object detection.

Non-Maximal Suppression

Suppress overlapping bounding boxes that do not have the maximum probability for object detection

So far, we have been going through an example with only one object in the image when there are multiple object in an image, we have to apply non-maximal suppression to each class in dependability so if we have three classes.

We have to apply non-maximal suppression three times.

Non-Maximal Suppression Steps

In practice Non-maximal Suppression is implemented in a few steps.

1. **Look at the output vector of each grid cell. Recall that each grid cell will have an output vector with a Pc value and bounding box coordinates.**
2. **Remove all bounding boxes that have a Pc value less than or equal to some threshold, say 0.5. Therefore, we will only keep a bounding box, if there is more than a 50% chance of an object being inside of it.**
3. **Select the bounding box with the highest Pc value.**
4. **Remove all the bounding boxes that have a high IoU* with the box selected in the last step.**

* A high IoU usually means a that the IoU is greater than or equal to 0.5.

Next we will see another technique that YOLO used to detect objects even if they overlap

Anchor boxes

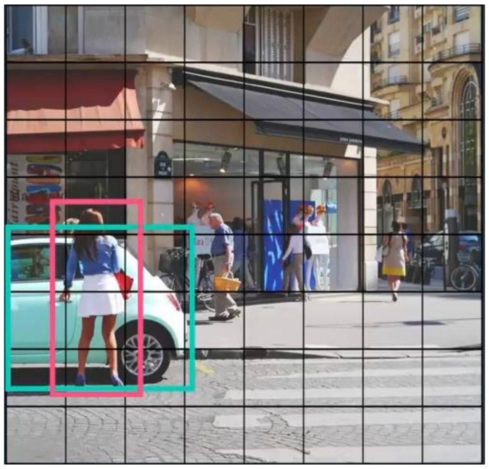
YOLO can work will from multiple objects where each object associated with one grid cell.

But what about in the case of overlap, in which one grid cell actually contains the center points of two different objects?

We can use something called Anchor boxes to allowed one grid cell to detect multiple objects.

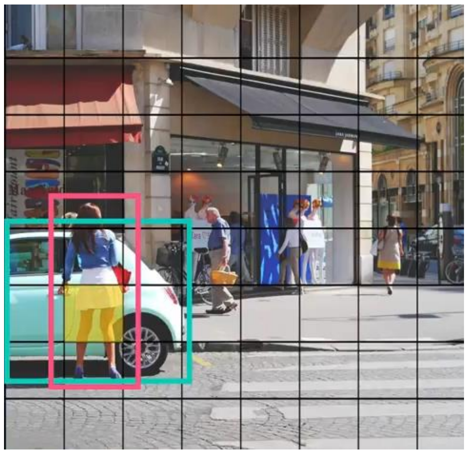
YOLO

Example in this image, we see that we have a person and a car overlapping in the image.



So part of the car is obscured

We can see that the centers of both bounding boxes, the car, and the pedestrian fall in the same grid cell.



Since the output vector of each grid cell can only have one class, then it will be forced to pick either the car or the person.

$$y = \begin{bmatrix} p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix} \qquad \begin{aligned} c_1 &= person \\ c_2 &= car \\ c_3 &= dog \end{aligned}$$

But by defining **anchor boxes**, we can cerate a longer grid cell vector and associated multiple classes with each gird cell.

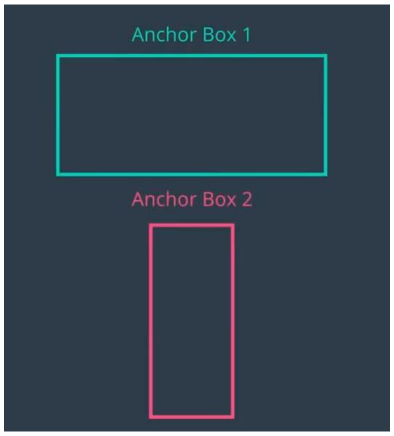
So let`s define two **anchor boxes** with some initial width and height.

In practice, you can define as many anchor boxes as you want.

You can imagine that these boxes are two different shapes for gift boxes.

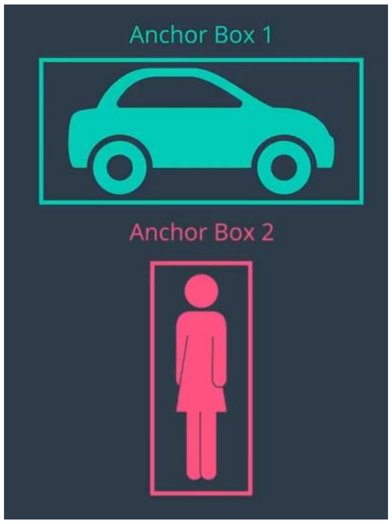
You typically choose gift boxes so that the nicely fit any object you put in them, and we you want to find anchor boxes so that they span the variety of objects shapes we want to detect.

Anchor boxes have a defined aspect ratio, and they tried to detect objects that nicely fit into a box with that ratio. For example



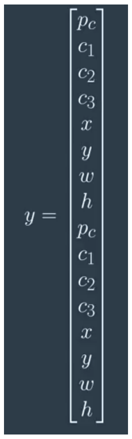
Since we are detecting a wide car and a standing person, we will defined one anchor box that is roughly the shape of a car this box will be wider that it is tall.

And we will defined another anchor box that can fit a standing person inside of it, which will be taller that it is wide.



YOLO

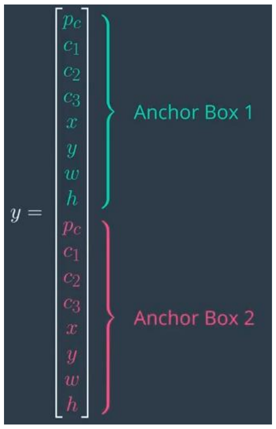
We can modify the output vector of each grid cell to contain both anchor boxes,



$$y = \begin{bmatrix} p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \\ p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix}$$

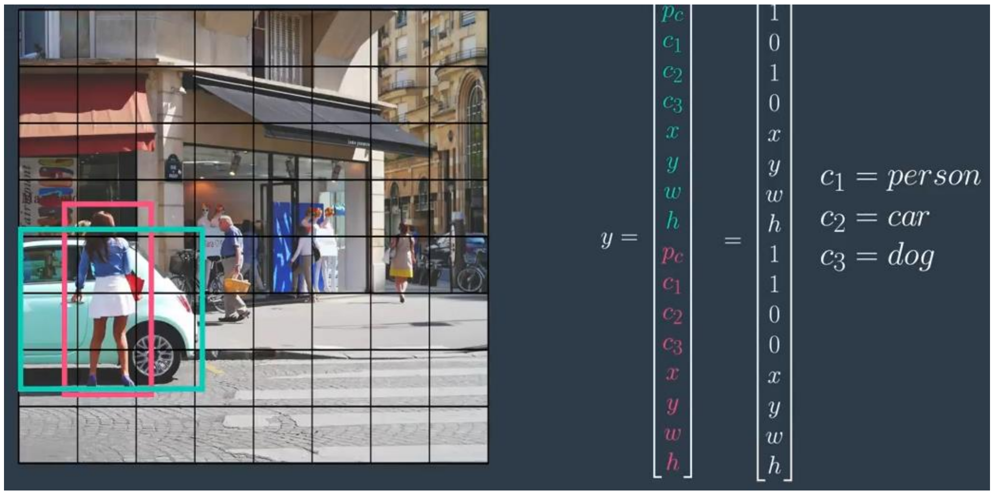
Which have coordinates and class scores. So the output vector will now contain 16 elements.

The first eight elements will correspond to anchor box one and the last eight will correspond to anchor box two.



$$y = \begin{bmatrix} p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \\ p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix}$$

Now ,because the bounding box around the car has a higher IOU with anchor box one, the anchor one elements of our output vector will include the classification and box parameters of the car.



$$y = \begin{bmatrix} p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \\ p_c \\ c_1 \\ c_2 \\ c_3 \\ x \\ y \\ w \\ h \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ x \\ y \\ w \\ h \\ 1 \\ 1 \\ 0 \\ 0 \\ x \\ y \\ w \\ h \end{bmatrix} \quad \begin{matrix} c_1 = person \\ c_2 = car \\ c_3 = dog \end{matrix}$$

Similarly, since the bounding box around the person has higher IOU with anchor box two, the anchor two element will include the parameters of the person.

It`s very challenging even for human to identify overlapping objects in a scene and anchor boxes give us a way to do this, which is pretty cool but as with any object detection method, it has a limitation.

Consider the case of two overlapping people this algorithm check which present fits with anchor box one and which one fits with anchor box two.

It only associate one object with each type of anchor box so it doesn't work very well if you have two overlapping objects that are roughly the same shape.

Similarly, if you only define two anchor boxes but have three overlapping objects then the algorithm fails because it will be forced to identify only two of the objects.

The good news is that that above cases are somewhat rare

In the next video, we will see how YOLO puts all these techniques together and performs objects detection for a complex image.

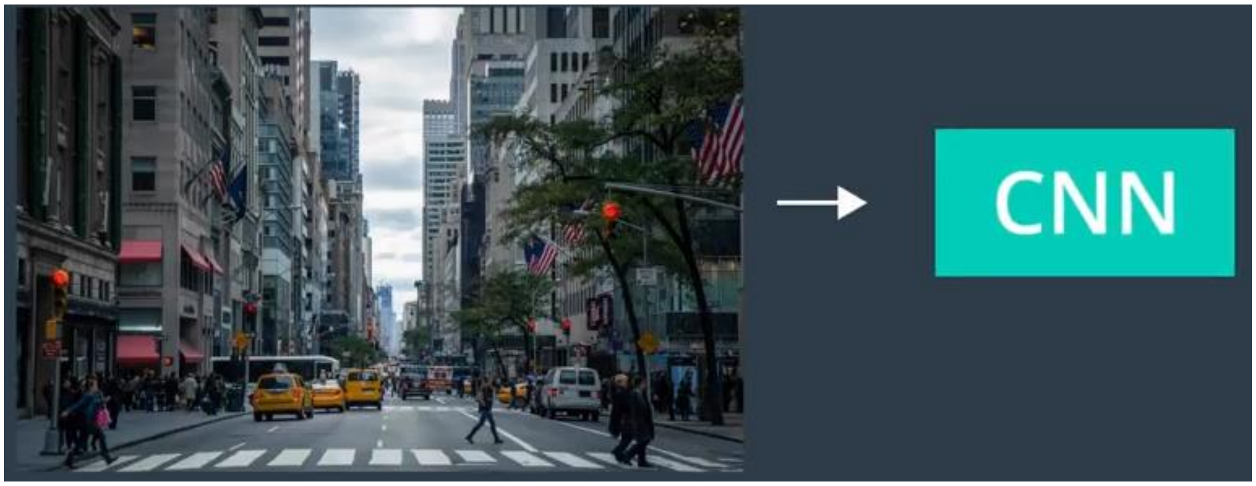
YOLO Algorithm

Let`s see how to takes an image and take a most possible objects Say we have a CNN that`s been trained to recognize several classes, including

1- **Traffic light**
2- **Car**
3- **Person**
4- **Truck**

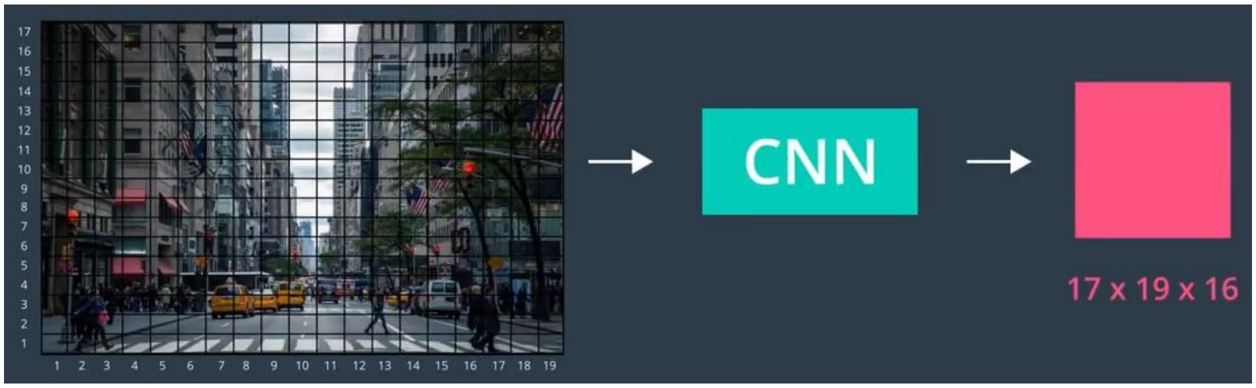
We give it two types of anchor boxes a tall one and a wide one so that it can handle overlapping objects of different shapes.

Once the CNN has been trained, we can now detect objects in images by feeding at new test images

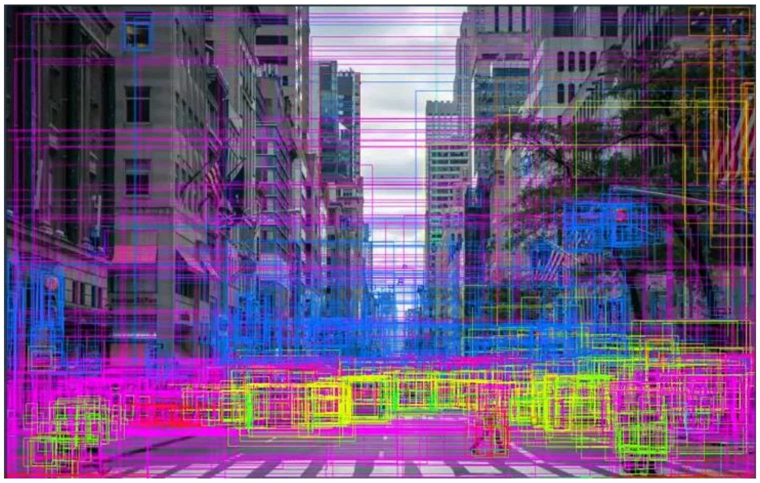


YOLO

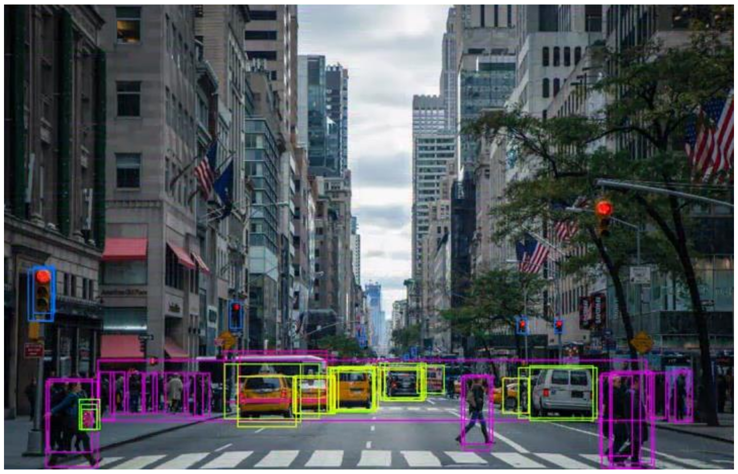
The test images is first broken up into a grid and the network then produces output vectors, one of each grid cell.



These vectors tell us if a cell has an object in it, what class the object is, and the bounding boxes for the object. Since we are using two anchor boxes we will get two predicated anchor boxes for each grid cell.



Some in fact most of the predicated anchor boxes will have a very low PC value. After producing these output vectors, we use non-maximal suppression to get rid of unlikely bounding boxes.

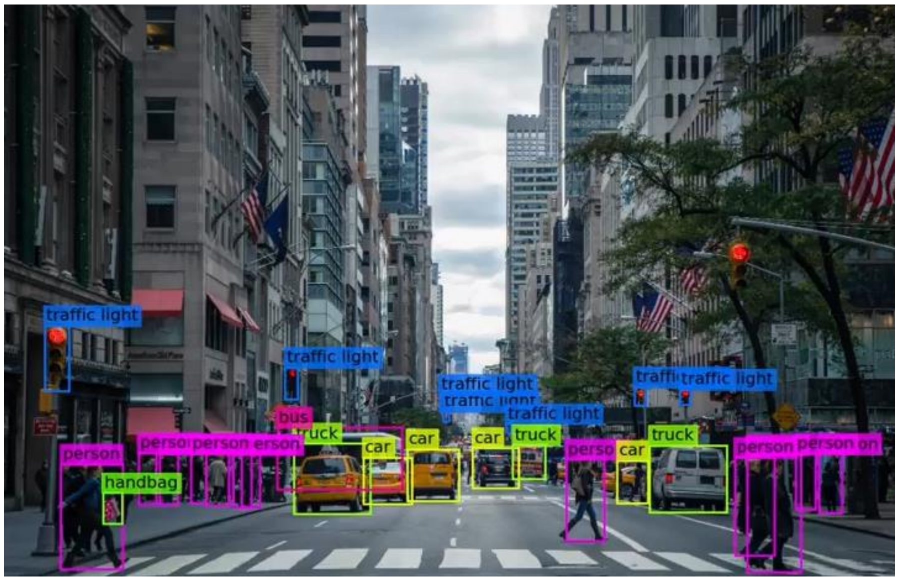


For each class, non-maximal suppression get rid of the bounding boxes that have a PC value lower than some given threshold.

It then selects the bounding boxes with the highest PC value, and remove bounding boxes that are too similar to this.

It will repeat this until all of the non-maximal bounding boxes had been removed for every class.

The end result will look at like this, we can see that yellow has effectively detect many objects in the image such as a cars and people.



Now that you know how YOLO works, you can see why it`s one of most widely used object detection algorithms today.

Next you will get work with the code implementation of the YOLO algorithms, and really see how it detects objects in different scenes and with varying levels of confidences.