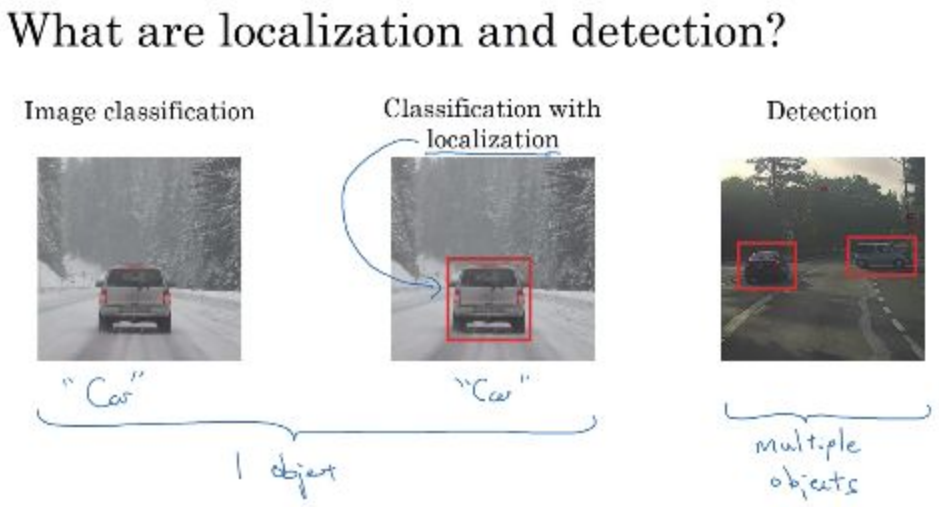
Andrew Ng's Convolutional Neural Networks

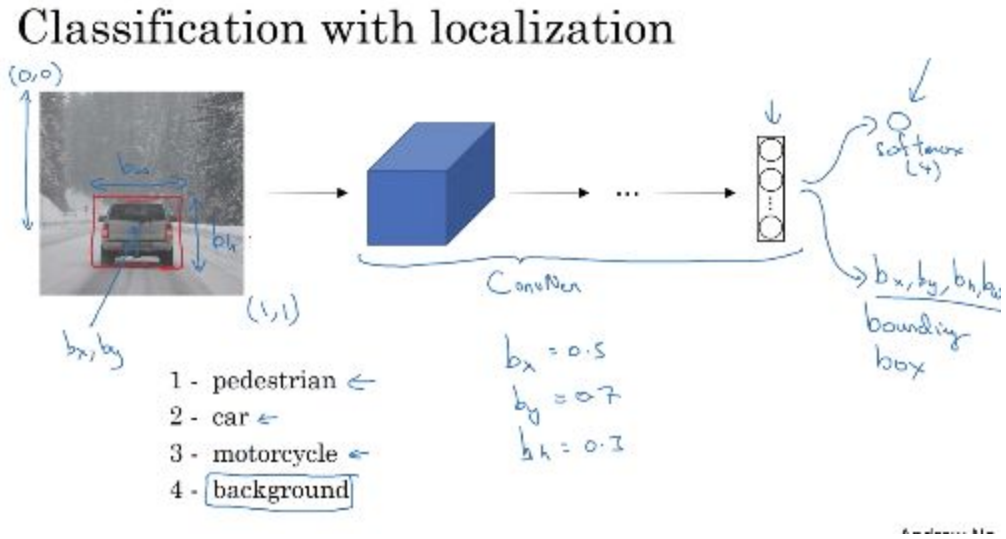
Week 3: Object Detection

Object Localization

- We start learning about object detection with object localization. Localization is shown on the left below (we are trying to classify an image, in this case a car). Classification with location means we not only identify the car but also add a boundary box around it. Detection means identifying and bounding multiple objects. In general, classification and classification with location deal with one object while detection deals with multiple objects.

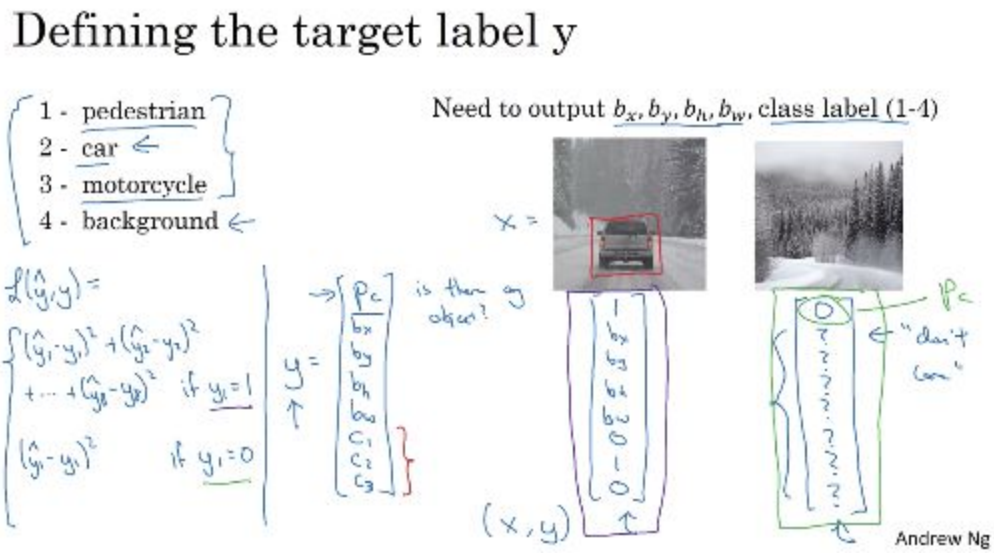


- Below is the standard pipeline for a classification task.



We input an image to a convnet which results in a vector of features that is fed to a softmax neuron which then outputs the clause. In this case we might have 4 outputs: pedestrian, car, motorcycle and background where background means none of the other 3. To add localization what we do is change the neural network to output a few more outputs: $bx,\ by,\ bh\ and\ bw$ where $(bx,\ by)$ is the central point of the boundary box, bh is the height and bw is the width of the box.

- Let's formalize this a little bit more. If we have a target label y that will have 4 values our y target label will be a vector of size 8. It will have $[p_c,\ bx,\ by,\ bh,\ bw,\ c_1,\ c_2,\ c_3]$ where $p_c = 1$ tells us there is an object and $p_c = 0$ tells us we don't have an object. The 2 examples below illustrate this. For the first one (the photo with a car) $y = [1,\ bx,\ by,\ bh,\ bw,\ 0,\ 1,\ 0]$. For the 2nd one $y = [0,\ ?,\ ?, ?, ?, ?, ?, ?]$ where ?= don't care because we know we don't have an object in this photo.

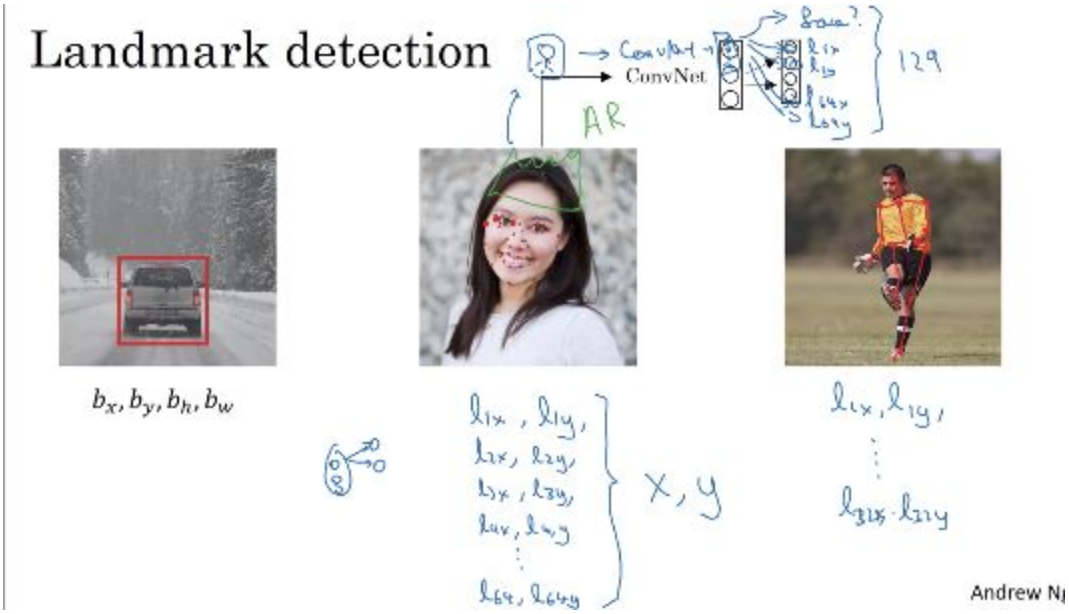


Finally, let's take a look at the loss function. If $y_1 = 0$ then the loss function $L(yhat,\ y) = (y_1hat - y_1)^2 + (y_2hat - y_2)^2 + \ldots + (y_8hat - y_8)^2$. $If\ y_1 = 0\ then\ L(yhat,\ y) = (y_1hat - y_1)^2$.

Landmark Detection

- Last time around we looked at how we can get a neural net to output four numbers that specify a bounding box. This time we are going to see how we can do the same but for landmarks, which are just important points in an image. Let's say we have a face recognition app and we want to recognize the eyes and/or mouth of a person. What we can do is have a conv net that outputs 2 additional numbers, (l_x, l_y), that show where the corner of one eye starts. We can generalize this by saying we want 2 numbers for each corner of each eye so we end up with 8 additional numbers. We can further generalize

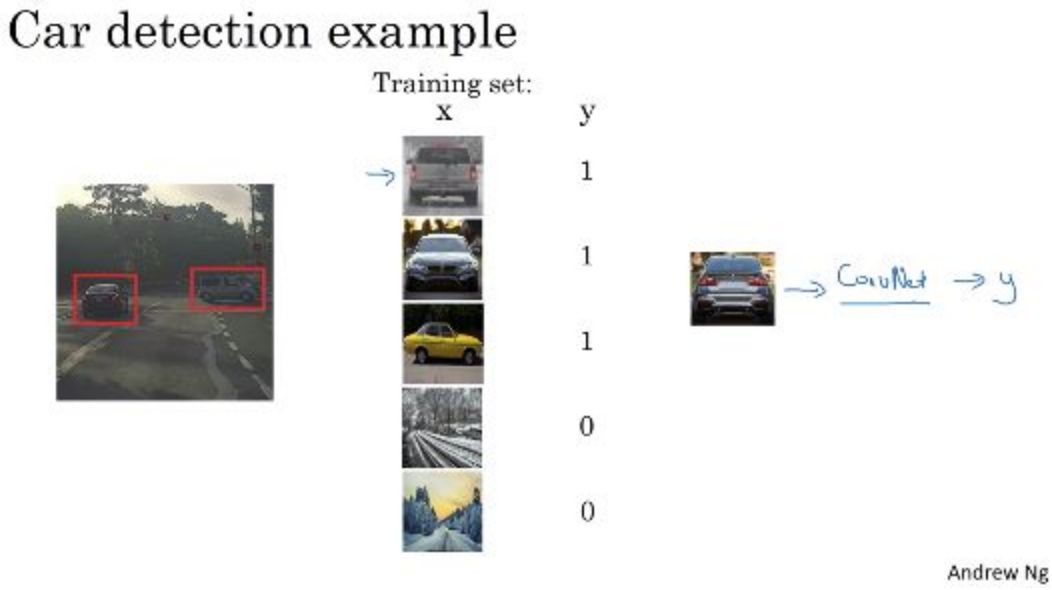
this by saying we need to recognize more ‘landmarks’ (the edge of the face, for example) and we end up with more outputs.



Another example is detecting someone’s pose as shown in the 3rd image above. In that case we need more landmarks. Basically, the idea is just to add more output units that show the x and y coordinates of all the landmarks we are interested in. Note that the landmarks must be consistent across images (i.e. landmark 1 is always the corner of the first eye). Also note we need a labeled dataset so we can train our neural net.

Object Detection

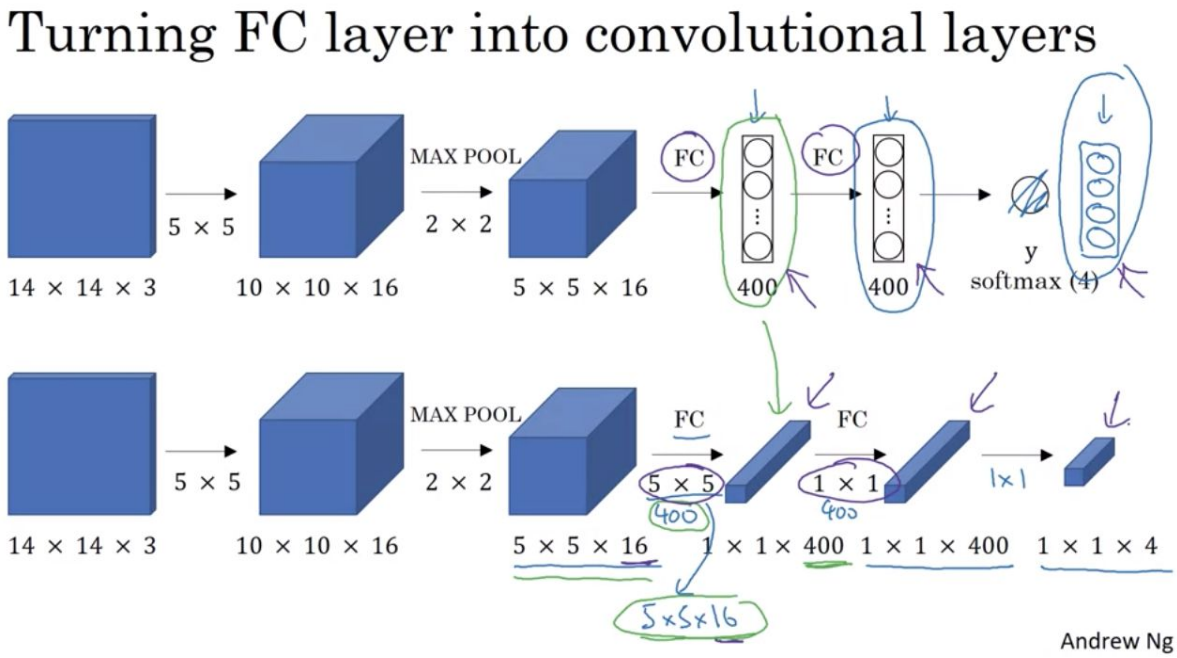
- Now we are going to learn how to use a convnet to perform object detection using an algorithm called the sliding windows detection algorithm. We start by having a training set that inputs whether the image has a car or not. We then take the close cropped images of the training set to end up with a training set that has images of a car in the center. We can then take that training set to train a convnet to tell us if there is a car or not in the image.



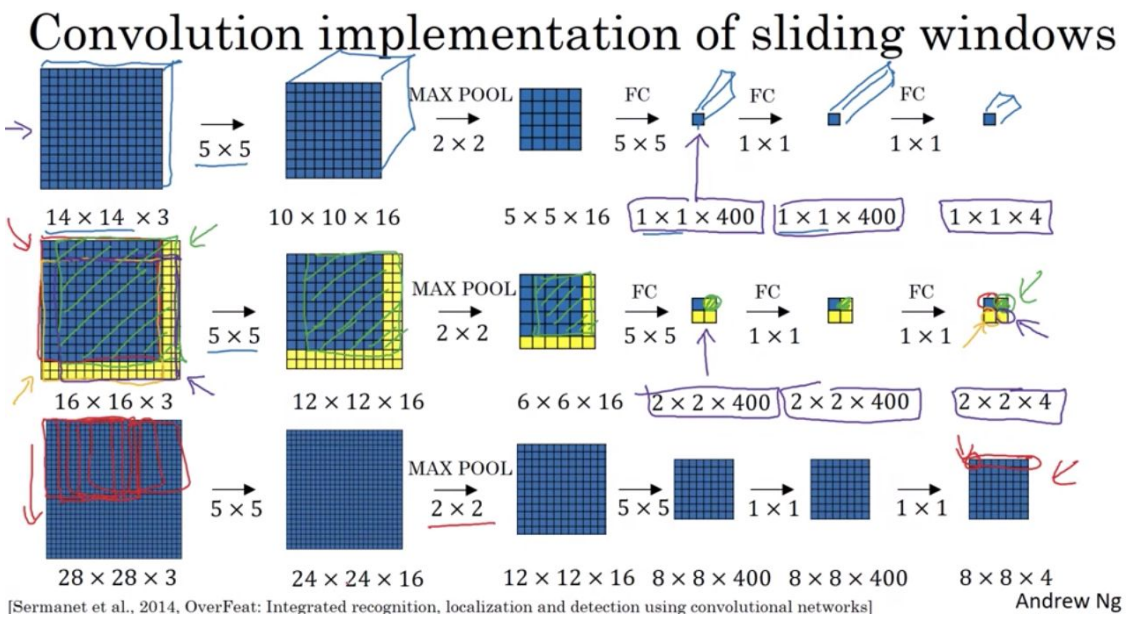
- Once the convnet is trained we are ready to use it in sliding window detection. Sliding windows works by taking a ‘window’ as shown below and ‘sliding’ it across the image. Each piece or window of the image is then fed to the convnet which then outputs 0 (no car) or 1 (car detected). We start with a small window and then use larger size windows. A disadvantage of this algorithm is its computational cost but we have a convolutionally oriented solution to go around it. Its cost is a problem because we might have a large number of image pieces/windows to run through the convnet if we have a small stride. If we have a big stride we then have less windows to run through the convnet and that might hurt the convnet accuracy.

Convolutional Implementation of Sliding Windows

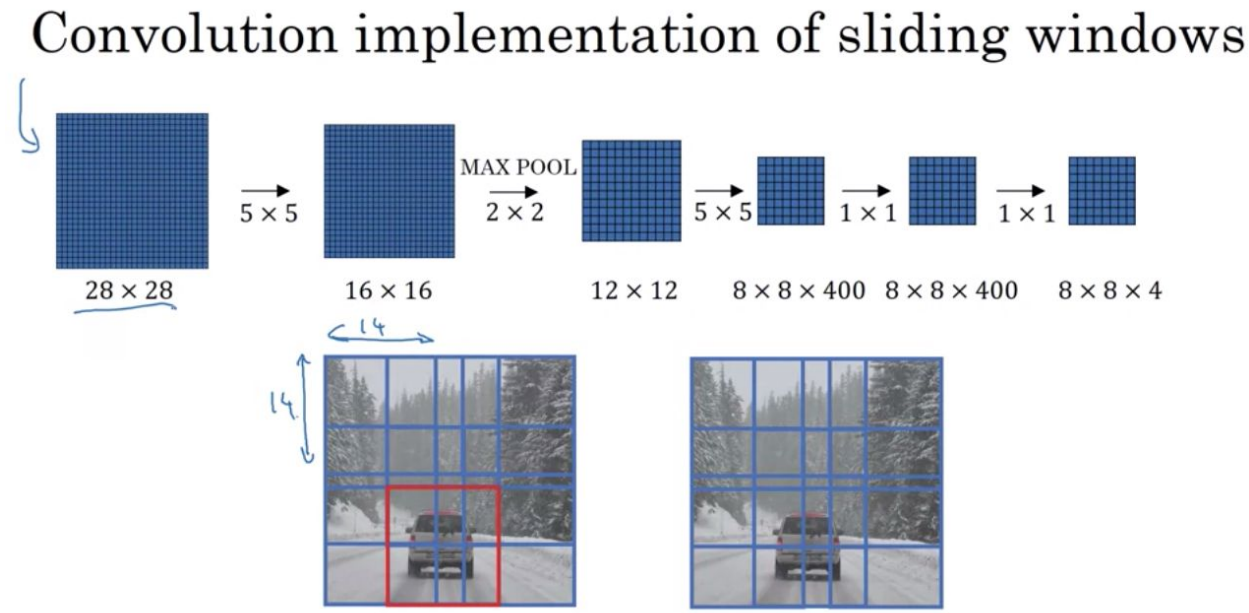
- Before looking at the convolutional implementation of sliding windows we need to know how to turn a fully connected layer of a convnet into a convolutional layer. Below up top we have a common covnet with 2 fully connected layers at the end, just before a softmax activation layer. What we do to convert those 2 FC layers to convolutional layers is the following: we take the output of the previous layer, a 5x5x16 and apply 400 5x5x16 filters; this will give us as output a 1x1x400 layer. We then apply a 1x1 filter followed by a softmax activation to finally get a 1x1x4 output.



- Now let's see the convolutional implementation of sliding windows. Let's say we have as an input a 14x14x3 image and the convnet architecture shown below up top. It's the same architecture we had in the previous slide. But now let's say we have as an input a 16x16x3 image. If we were going to do it the old way, we would need to take 4 windows of the image (using our 14x14x3 window) and run each piece through the whole convnet so we end up with having to run through the whole convnet 4 times for only 1 image. This is highly inefficient. A better way would be to take our full original 16x16x3 image and run it through our convnet image and we end up with a 2x2x4 output where each part of the output corresponds to one of the 4 pieces/windows of the input image. This is shown in the 2nd row below. Finally, let's say we have a bigger input image, 28x28x3, so we do the same thing (run the full image through the convnet) and we end up with an 8x8x4 output where each output unit corresponds to one image/piece of the original image. Notice this is done with a stride of 2, determined by the 2x2 max pool layer.



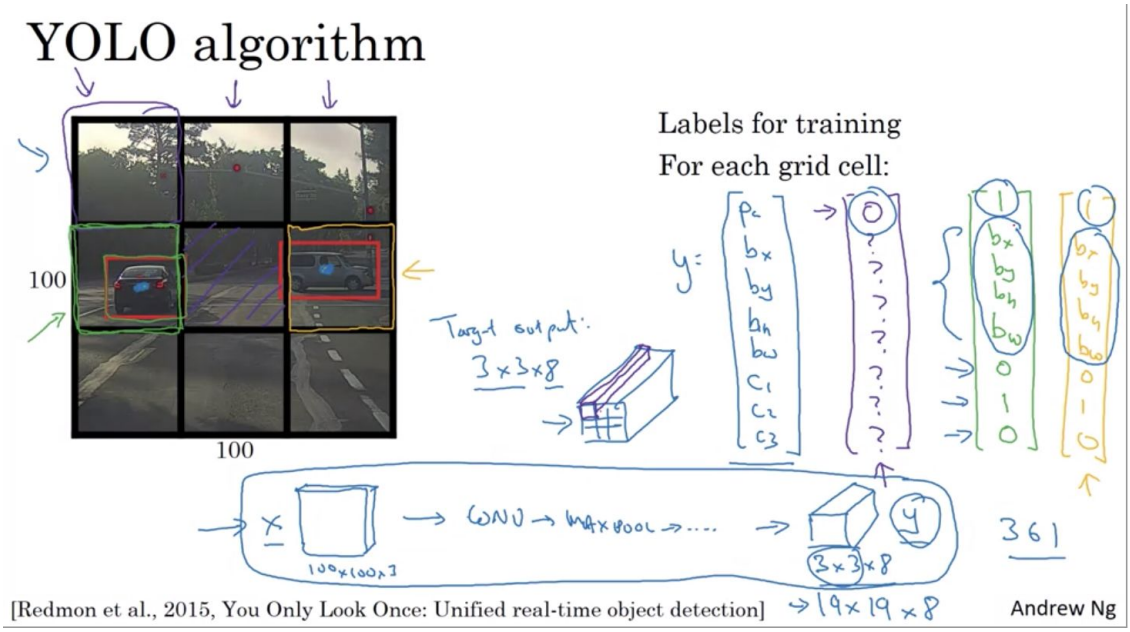
- Summarizing, instead of doing a sequential implementation of sliding windows like we saw at the beginning we can now take the whole image and do all the predictions at the same time by just doing forward propagation once. We then go from many runs through the convnet to just once and that makes the process much more efficient.



There is one more problem: the position of the bounding boxes is not going to be very accurate. Next session we will see how to fix that.

Bounding Box Predictions

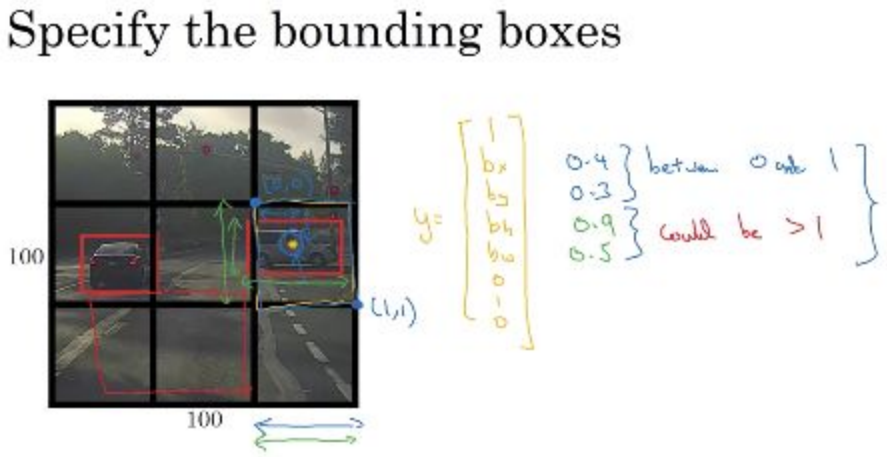
- We use the YOLO (You Only Look Once) algorithm to fix the bounding box accuracy. What we do is the following: we take the input image and over impose a grid like the one below. In this case, we are using a 3x3 grid but in practice you will use finer grids, say 19x19. Next we apply the image classification algorithm to each grid to detect if there is an object of interest. So for each grid we have an output vector that has $y = [P_{c,}\ b_x,\ b_y,\ b_h,\ b_w,\ C_1,\ C_2,\ C_3]$ where P_c tells us if there is an object of interest, the b's tell us the bounding box coordinates and the C's tell us what kind of object it is (pedestrian, car or say, sign). So for the first grid, top left, we would get an output vector like this one: $[0,\ ?,\ ?,\ ?\ ,?,\ ?\ ,\ ?,\ ?\ ]$ because there is no object of interest. For the 2 grids with cars on them we get vectors like this one: $[1,\ b_x,\ b_y,\ b_h,\ b_w\ 0,\ 1,\ 0\ ]$. So we have a 3x3 grid and we have an 8-dimensional vector as the output for each grid we end up with an output volume of 3x3x8. One clarification: the cars were assigned to that particular cell because the YOLO algorithm assigns them to the cell where the object's mid point resides.



Notice this is a convolutional implementation so we only run the whole image through the convnet only once so it is a very efficient algorithm and runs very fast.

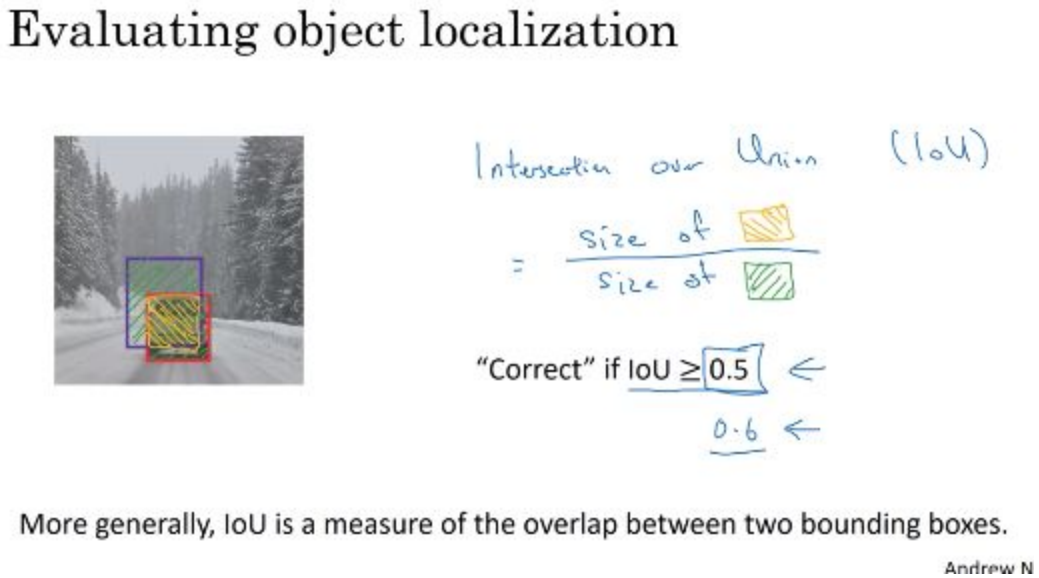
- How do you specify the bounding boxes? Let's go through an example: take the car on the right in the image below. The cell will have an output of $y = [1,\ b_x,\ b_y,\ b_h,\ b_w\ 0,\ 1,\ 0\ ]$. For the car on the right, the upper left corner is (0,0) and the bottom right corner is (1,1). So the midpoint with coordinates $b_x,\ b_y$ would approximately be (0.4, 0.3). These 2 coordinates have to be between 0 and 1. b_h *and* b_w are more like 0.9, 0.5 because the width of the bounding box is about 90% of the overall cell width and the height of the box is about half of the overall cell height. b_h *and* b_w could be >= 1 because a car can cover multiple cells like the red box shown at the

bottom. This is just one example of how to specify bounding boxes. There are other ways of specifying bounding boxes but this is a common one that works well.



Intersection over Union

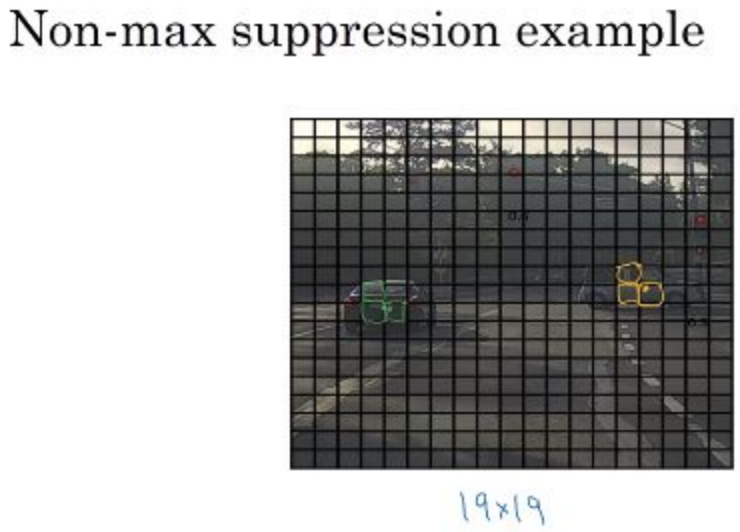
- How do we know if our object detection algorithm is working? We use a function called Intersection over Union (IoU). Let's see how it works with an example: in the image below we have the red box which is the best possible outcome for the bounding box and let's assume the purple box is what our object detection algorithm outputs. How do we know this is a good answer? What we do is we compute the size of the union (the total area of both boxes) and the size of the intersection (the area where both boxes intersect) and we divide intersection over union. If $IoU \geq 0.5$ we say the algorithm's output is correct.). 0.5 is a human-chosen value so we can also choose a higher threshold. Professor Ng mentions he sees people using values between 0.5 and 0.7 and rarely sees the threshold being dropped below 0.5.



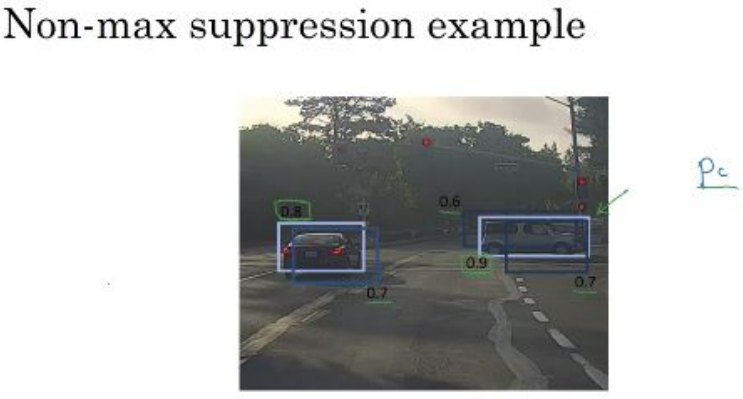
Summarizing, IoU is used to quantify the accuracy of our object-detection algorithm but, in general, it is a measure of how similar these 2 boxes are to each other.

Non-max Suppression

- One of the problems of object detection is that it might find the same object more than once. Non-max suppression is a way to ensure the algorithm only detects one object once. Consider the image below. If we superimpose a 19x19 grid we might have different cells that claim the midpoint for the 2 cars in the pic. For the car on the right it might be those 3 yellow cells; for the car on the left it might be the 3 green cells.



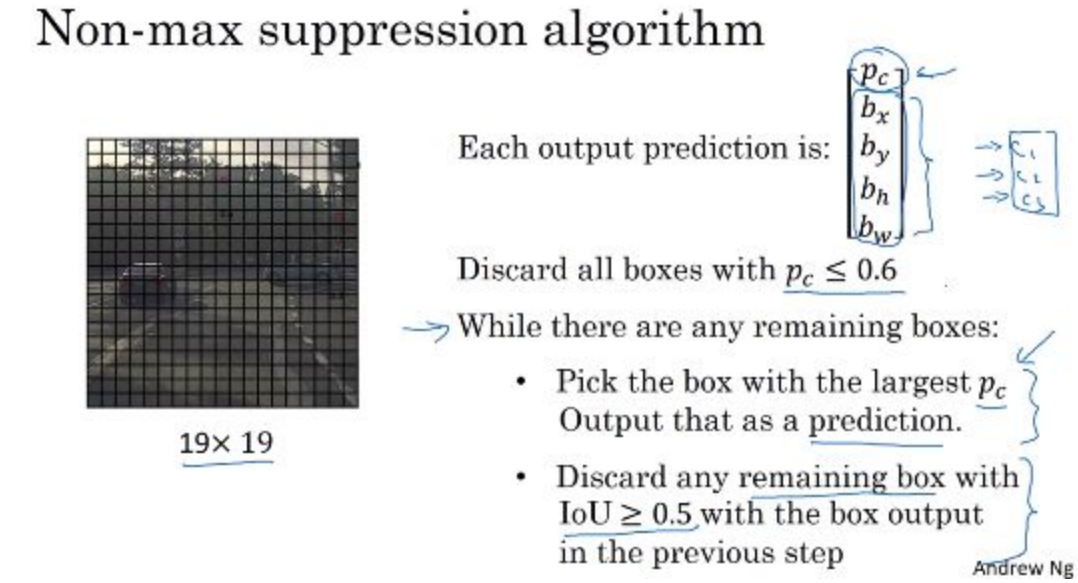
- Let's see how non-max suppression works. In our 19x19 grid we will have multiple predictions because many of our cells will think they have an object in it. For example, we might end up with 3 predictions for the car on the right and 2 predictions for the car on the left. So what non-max suppression does is that it looks at the probabilities of each prediction and picks the prediction with the highest probability. Other predictions with close probabilities are eliminated.



In the image above that means that for the car on the right we pick the prediction with probability = 0.9 and eliminate the ones with 0.6 and 0.7. We do a similar process for the car on the left. So our algorithm is called non-max suppression because its result is the maximum probabilities and it suppresses the close ones that are non-maximal.

- Now let's see how the algorithm works. Taking our same image from before we will only look for cars so our output prediction vector is as follows: $[p_c,\ b_x,\ b_y,\ b_h,\ b_w]$. So we will have multiple prediction which means we will have multiple boxes with different probabilities. Our first step is to discard all boxes where $p_c \leq 0.6.$ While there are any remaining boxes:
    - Pick the box with the largest p_c and output that as a prediction.
    - We then discard any remaining box with $IoU \geq 0.5$ that has high overlap with the box in the previous sets.

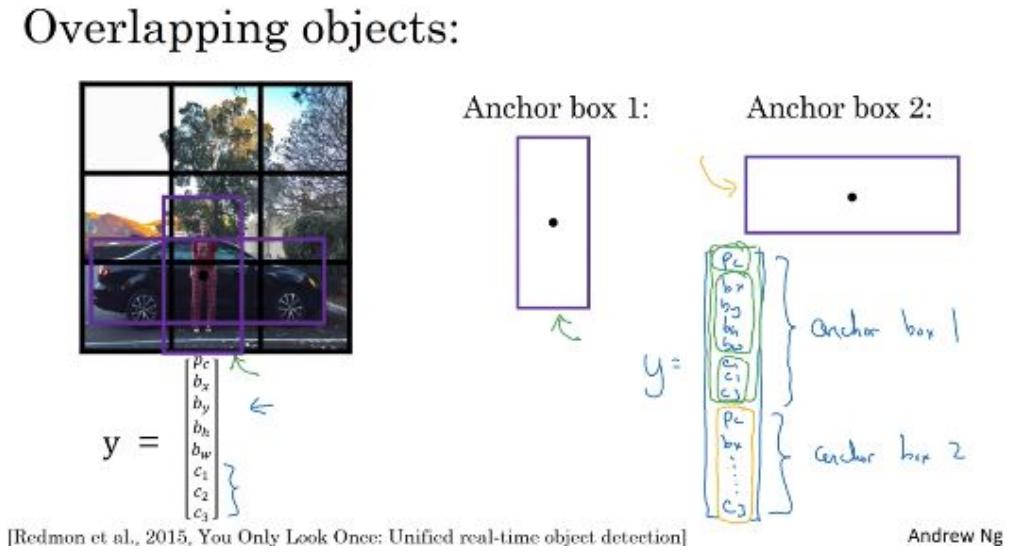
  And we repeat this while loop while there are any remaining boxes.



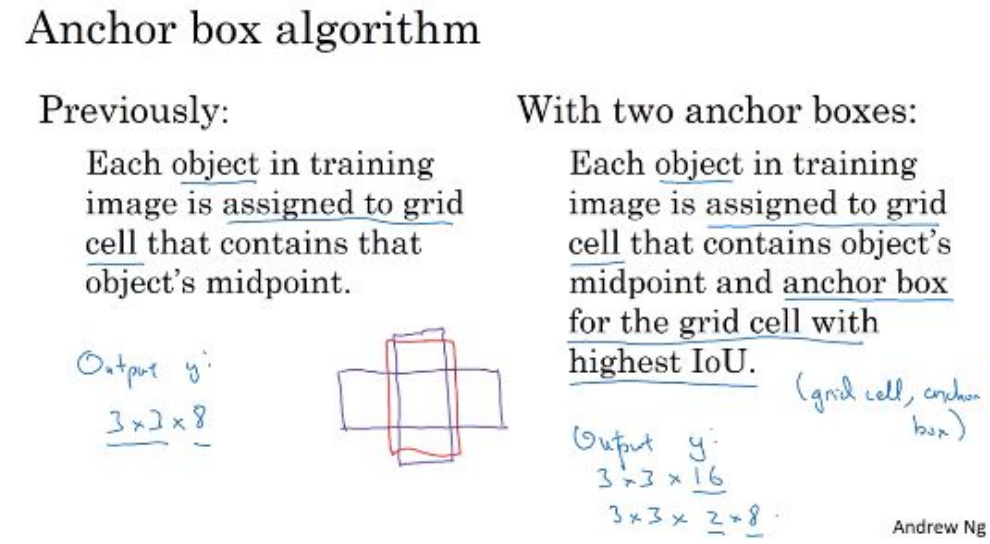
We described the algorithm using only one object. If we have more objects, say 3 we then apply non-max suppression 3 times independently, one for each object.

Anchor Boxes

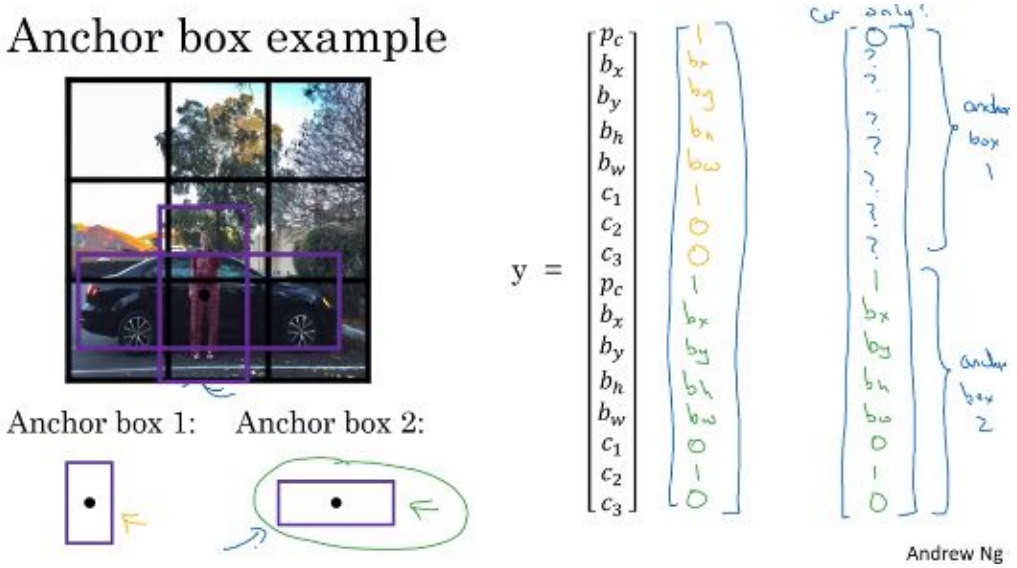
- One of the problems with our current understanding of Object Detection is that each cell can only detect one object, what happens if there are multiple objects we need to detect? We use anchor boxes. Let's how they work: in the image below the midpoint for both the pedestrian and the car is probably in the same cell. What we do to solve the problem is that we have 2 anchor boxes. One vertical one, anchor box 1 and one horizontal, anchor box 2. Our y then changes to basically have twice the number of parameters so $y = [p_c,\ b_x,\ b_y,\ b_h,\ b_w,\ c_1,\ c_2,\ c_3, p_c,\ b_x,\ b_y,\ b_h,\ b_w,\ c_1,\ c_2,\ c_3]$ where the first 8 parameters are for anchor box 1 and the second set is for anchor box 2.



- So what we had before is that each object was assigned to a grid cell that contains the midpoint. Now that we are using anchor boxes what we have is that each object is assigned to the grid cell that contains the object's midpoint and the anchor box for the grid cell with highest IoU. The output y before was 3x3x8 (it is 8-dimensional because we have 3 output classes) and now it's going to be 3x3x16 (or 3x3x2x8). With the anchor boxes then we assign the object to a $(grid\ cell,\ anchor\ box)$ pair.



- Let's go through the example below. In the image below we have the pedestrian and the car. What would be the output here, it would be $y = [1,\ b_x,\ b_y,\ b_h,\ b_w,\ 1, 0, 0, 1,\ b_x,\ b_y,\ b_h,\ b_w,\ 0,\ 1,\ 0]$. What if the grid cell only had the car and not a pedestrian? Y then would be $[0, ?, ?,\ ?,\ ?,\ ?, ?, ?, 1,\ b_x,\ b_y,\ b_h,\ b_w,\ 0,\ 1,\ 0]$

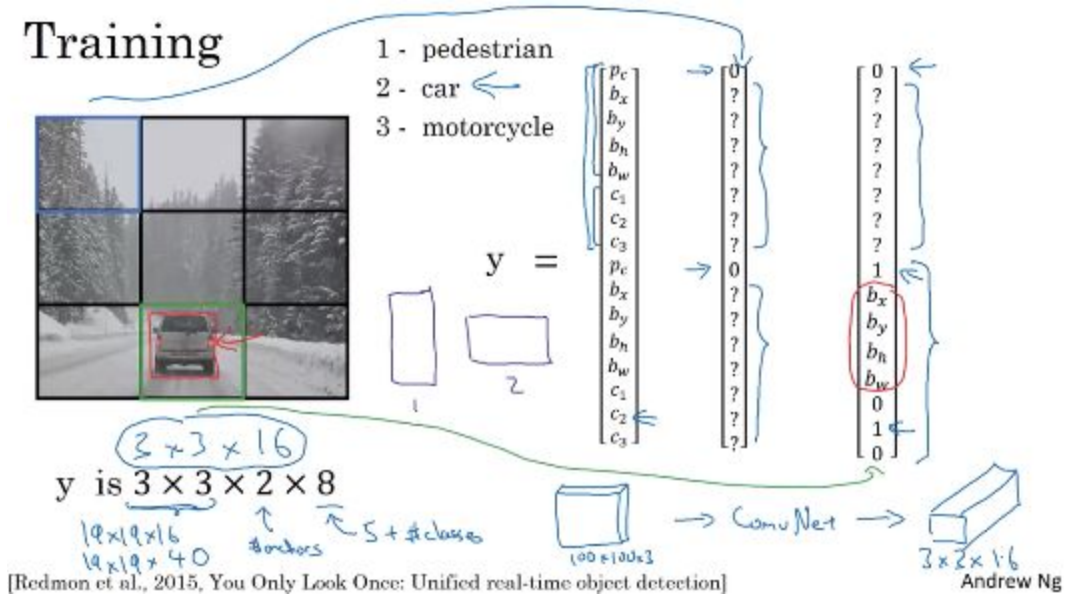


What if we had 3 objects? This algorithm doesn't handle that case so we would implement a default tiebreaker if we run into this situation. We talked about anchor boxes as a way to handle multiple object detection in the same cell. However, in practice, this does not happen frequently especially if we use a 19x19 grid. A better motivation for anchor boxes is that it allows our learning algorithm to specialize better. Some of the outputs can specialize on 'fat objects' like cars and some on 'thin objects' like pedestrians.

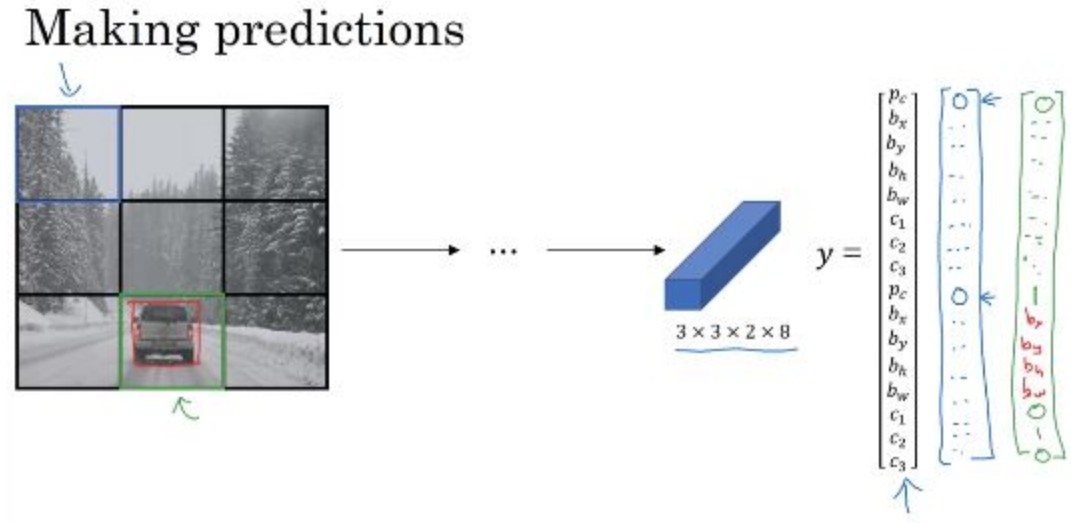
YOLO Algorithm

- Now let's put everything together. First we'll construct the training set. In the image below with the 3x3 grid we would go cell by cell and fill out y; for the first cell $y = [0, ?, ?, ?, ?, ?, ?, ?, ?, 0, ?, ?, ?, ?, ?, ?, ?, ?]$. For the cell that has a car $y = [0, ?, ?, ?, ?, ?, ?, ?, ?, 1, b_x, b_y, b_h, b_w, 0, 1, 0]$. Why the second set of 8 numbers? Because we have 2 anchor boxes (1 & 2 shown below) and the car has a slightly higher IoU with anchor box 2 so it is assigned to that box. So our output volume is 3x3x16. In practice,

using a 19x19 grid & 5 anchor boxes, our output volume would be 19x19x40.



- Now let's see how the algorithm will make predictions. Given an image, our convnet will output a 3x3x16 volume where each of the 9 grids will have a vector y like the one shown below. Our neural network cannot output don't cares so, for the cells where objects are not detected, we would have a y with 2 zero's and a bunch of numbers we would not pay attention to. For the 8th grid, $y = [\ 0,\ ?, ?, ?, ?, ?, ?, ?, 0,\ b_x, b_y, b_h, b_w, 0, 1, 0]$.



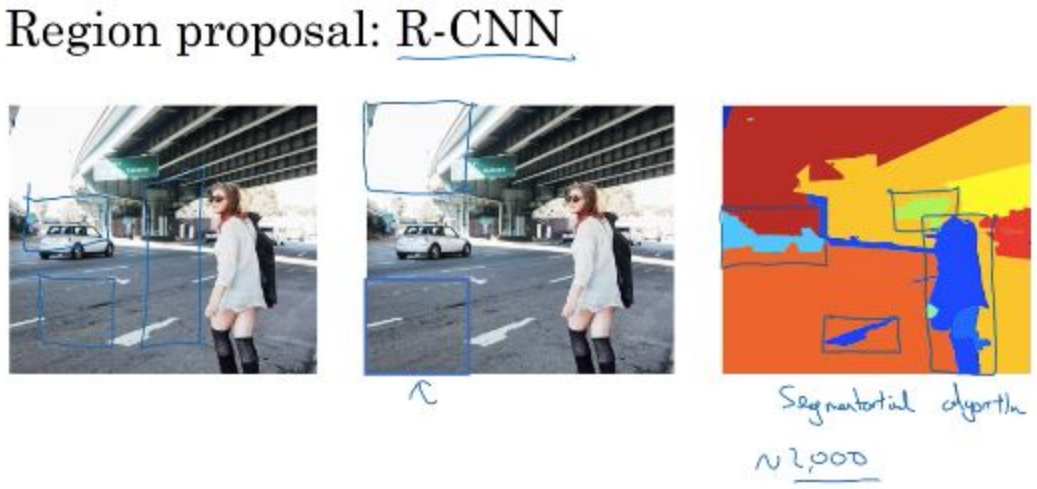
- Finally, we run our predictions through non-max suppression. For each grid cell we would get 2 predicted boxes. First we get rid of the low probability predictions and then for each class (assume we have 3 classes: pedestrian, car, motorcycle) we use non-max suppression to generate the final predictions. And that's it for the YOLO algorithm.

Outputting the non-max supressed outputs

- For each grid call, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

Region Proposals

- We are now going to look at region proposals, a body of work that has been very influential in computer vision. Region proposal or R-CNN posits that you don't need to run your object detection algorithm in every part of your input image. What it does is that it takes the input image, runs a segmentation algorithm on that input and ends up with an image like the one on the right. Then what happens is we run our classifier on the blobs that look promising, maybe around 2,000 blobs. However, R-CNN is very slow.



- There have been efforts to speed up R-CNN. The first one is called Fast R-CNN which uses a conv implementation of sliding windows to classify all the regions. The second effort is Faster R-CNN which uses a convolutional network to propose regions. In general, most of these implementations are still slower than YOLO. However, we look at them because we might run into them in the real world.

Faster algorithms

R-CNN:	Propose regions. Classify proposed regions one at a time. Output label + bounding box.
Fast R-CNN:	Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions.
Faster R-CNN:	Use convolutional network to propose regions.