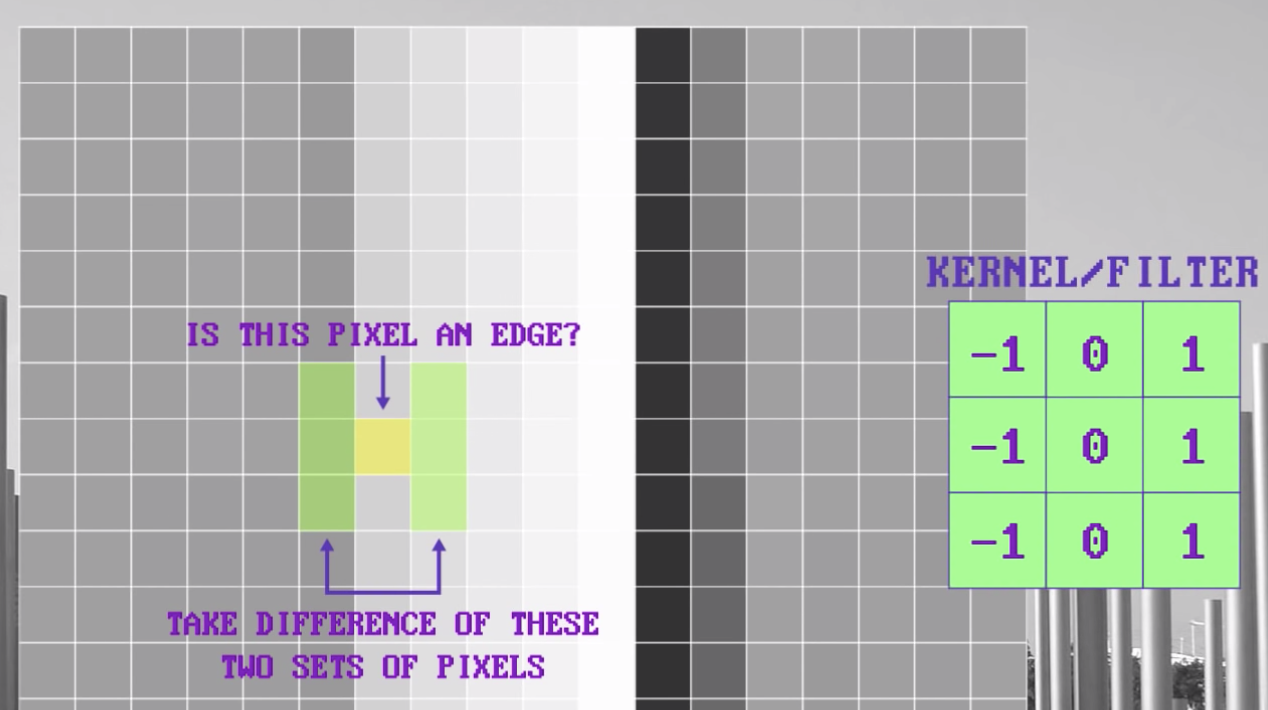
**Some notes of Computer Vision**

Basic concepts

* Color tracking (single pixel)
* Patches (small regions of pixels)
  + greyscale, detect edges



* <https://blog.algorithmia.com/introduction-to-computer-vision/>

**What is Computer Vision?**

* Computer Vision is the broad parent name for any computations involving visual content – that means images, videos, icons, and anything else with pixels involved. But within this parent idea, there are a few specific tasks that are core building blocks:
  + In **object classification**, you train a model on a dataset of specific objects, and the model classifies new objects as belonging to one or more of your training categories.
  + For **object identification**, your model will recognize a specific instance of an object – for example, parsing two faces in an image and tagging one as Tom Cruise and one as Katie Holmes.
* Use cases
  + Video **motion analysis** uses computer vision to estimate the velocity of objects in a video, or the camera itself.
  + In **image segmentation**, algorithms partition images into multiple sets of views.
  + **Scene reconstruction**creates a 3D model of a scene inputted through images or video (check out [Selva](https://www.selva3d.com/)).
  + In **image restoration**, noise such as blurring is removed from photos using Machine Learning based filters.
* Any other application that involves understanding pixels through software can safely be labeled as computer vision.
* Machines interpret images very simply: as a series of pixels, each with their own set of color values. Consider the simplified image below, and how grayscale values are converted into a simple array of numbers:
* <https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/>
* 8 different trackers in OpenCV 3.4.1
* What is Object Tracking? Locating an object in successive frames of a video is called **tracking**.
* **Object Tracking** ideas
  1. dense optical flow
     + estimate the motion vector of every pixel in a video frame
  2. sparse optical flow
     + like the KLT (Kanade-Lucas-Tomashi) feature tracker
     + track the location of a few feature points in an image
  3. Kalman Filtering
     + signal processing algorithm used to predict the location of a moving object based on prior motion information
     + used on Apollo 11 lunar module to the moon
  4. Meanshift and Camshift
     + locate the maxima of a density function
     + also used for tracking
  5. Single object tracker
     + the first frame is marked using a rectangle to indicate the location fo the object we want to track
     + the object is then tracked in subsequent frames using the tracking algorithm
     + used in conjunction with an object detector
  6. Multiple object track finding algorithms
     + when we have a fast object detector, it makes sense to detect multiple objects in each frame
     + run a track finding algorithm that identifies which rectangle in one frame corresponds to a rectangle in the next frame
* **Tracking vs Detection**
  1. Tracking is faster than Detection
  2. A good tracking algorithm will use all info it has about the object up to that point **while** a detection algo always starts from scratch
  3. object detection is run on every nth frame **while** the tracking algo is employed in the n-1 frames in between
  4. you can also lose track of an object when they go behind an obstacle for an extended period of time or if they move so fast that the tracking algorithm cannot catch up
  5. It is also common for tracking algorithms to accumulate errors and the bounding box tracking the object slowly drifts away from the object it is tracking
  6. To fix these problems with tracking algorithms, a detection algorithm is run every so often.
  7. Detection algorithms are trained on a large number of examples of the object. They, therefore, have more knowledge about the general class of the object. **On the other hand,** tracking algorithms know more about the specific instance of the class they are tracking.
* **Tracking can help when detection fails**
  1. A good tracking algo will handle some level of **occlusion**
     + If you are running a face detector on a video and the person’s face get’s occluded by an object, the face detector will most likely fail.
* **Tracking preserves identity**
  1. the output of object detection is an array of rectangles that contain the object, **while** there is no identity attached to the object
  2. tracking provides a way to literally connect the dots
* **OpenCV 3 Tracking API**
  + 8 different trackers — BOOSTING, MIL, KCF, TLD, MEDIANFLOW, GOTURN, MOSSE and CSRT
  + we define a bounding box containing the object for the first frame and initialize the tracker with the first frame and the bounding box
  + then read frames from the video and just update the tracker in a loop to obtain a new bounding box for the current frame
* **Goal of tracking**
  + to find an object in the current frame given we have tracked the object successfully in all (or nearly all) previous frames
  + Since we have tracked the object up until the current frame, we know how it has been moving
  + So we know the parameters of the **motion model**
    - The motion model is just a fancy way of saying that you know the location and the velocity (**speed + direction** of motion) of the object
    - if you knew nothing else about the object, you could predict the new location based on the current motion model and you would be pretty close to where the new location of the object is
  + we have more info than just the motion of the object
    - we know how the object looks in each of the previous frames, on which we can build an **appearance model** that encodes what the object looks like
    - the **appearance model** can be used to search in a small neighborhood of the location predicted by the **motion mode**l to more accurately predict the location of the object
    - The **motion model** predicts the approximate location of the object.
    - The **appearance model** fine tunes this estimate to provide a more accurate estimate based on appearance.
    - If the object was very simple and did not change its appearance much, we can use a simple template as an appearance model and look for that template. (However, real life is not that simple.)
      * To tackle this problem, in many modern trackers, the appearance model is a **classifier** that is trained in an **online** manner.
* **Classifier**
  + To classify a rectangular region of an image as either an object or background
  + ~ takes in an image patch as input and returns a score between 0 and 1 to indicate the probability that the image patch contains the object
  + The score is 0 when it is absolutely sure the image patch is the background and 1 when it is absolutely sure the image patch is the object
* In ml, we use "**online**" to refer to algo that are trained on the fly at run time.
  + An **offline classifier** may need thousands of examples to train a classifier
  + while an **online classifier** is typically trained using a very few examples at run time.
* A classifier is trained by feeding it positive (**object**) and negative (**background**) examples.
  + if you want to build a classifier for detecting cats, you train it with thousands of images containing cats and thousands of images that do not contain cats.
  + this way the classifier learns to differentiate what is a cat and what is not.\
* **however, while building an online classifier, we do not have the luxury of having thousands of examples of the positive and negative classes**

**So we need different tracking algo approach online training**

1. **Boosting tracking**
   * based on an online version of AdaBoost -- the algo that HAAR cascade based face detector uses internally
   * need to be trained at runtime with positive and negative examples of the object
   * the initial **bounding box** supplied by the user / or by another object detection algo, is taken as the positive example for the **object**
   * and many image patches **outside the bounding box** are treated as the **background**
   * Given a new frame, the classifier is run on every pixel in the neighborhood of the previous location and the score of the classifier is recorded
   * the location with the maximum score is the new location os the object
   * as more frames come in, the classifier is updated with this additional data
   * **Pros**
     + **None** lol The algo is a decade old and works ok, but worse than MIL, KCF, which have similar principles
   * **Cons**
     + Tracking performance is mediocre. It does not reliably know when tracking has failed
2. **MIL tracker**
   * similar to Boosting tracker
   * while difference:
     + instead of considering only the current location of the object as a positive example, it looks in a small neighborhood around the current location to generate several potential positive examples
     + (with this idea, in most of these "positive" examples the object is not centered)
     + but this idea is good, because
       - in MIL, you do not specify positive and negative examples, but **positive and negative "bags"**
       - The collection of images in the positive bag are not all positive bags
         * **only one** image in the positive bag needs to be a positive example.
         * for example

a positive bag contains the patch centered on the current location of the object and also patches in a small neighborhood around it

even if the current location of the tracked object is not accurate, when samples from the neighborhood of the current location are put in the positive bag, there is a good chance that this bag contains at least one image in which the object is nicely centered

* + **Pros**
    - performance pretty good.
    - does not drift as much as the boosting tracker and it does a reasonable job under partial **occlusion**
    - if you are using OpenCV 3.0, this might be the **best tracker** available to you
    - but if you are using a higher version, consider **KCF**
  + **Cons**
    - Tracking failure is not reported reliably
    - does not recover from full occlusion

1. **KCF tracker**
   * **Kernelized Correlation Filters**
   * built on the idea of previous two trackers
   * the multiple positive samples used in the MIL tracker have large overlapping regions
   * this overlapping data leads to some nice mathematical properties that is exploited by this tracker to make tracking faster and more accurate at the same time
   * **Pros**
     + Accuracy and speed are both better than MIL and it reports tracking failure better than BOOSTING and MIL
     + if you are using OpenCV 3.1 and above, I recommend using this for most applications
   * **Cons**
     + Does not recover from full occlusion.
     + Not implemented in OpenCV 3.0
   * **Bug alert**
     + There is a bug in OpenCV 3.1 (Python only)
       - incorrect bounding boxes are returned
2. **TLD tracker**
   * **Tracking, learning and detection**
   * decomposes the long term tracking task

* <https://github.com/llSourcell/YOLO_Object_Detection/blob/master/YOLO%20Object%20Detection.ipynb>
* History of object detection
  + **Viola-Jones algorithm** 2001, using SVM (2001 - 2017)
    1. Invented by Paul Viola and Michael Jones
    2. The first time the facial detection really worked in real time on a webcam (Was the most stunning demonstration of computer vision and its potential at the time)
    3. **Hand coded feature**
    4. Implemented in OpenCV and face detection became synonymous with Viola and Jones
    5. **Four stages**
       1. Haar Feature selection
          - All human faces share similar properties, these regularities may be matched using **Haar features**

Eg. Eye region is darker than the upper-cheeks; the nose bridge region is brighter than the eyes

* + - * + Composition of properties forming matchable facial features

Location and size

Value: oriented gradients of pixel intensities

* + - * + Four features matched by the algo are then sought in the image of a face







* + - * + Rectangle features

Value = Σ (pixels in black area) - Σ (pixels in white area)

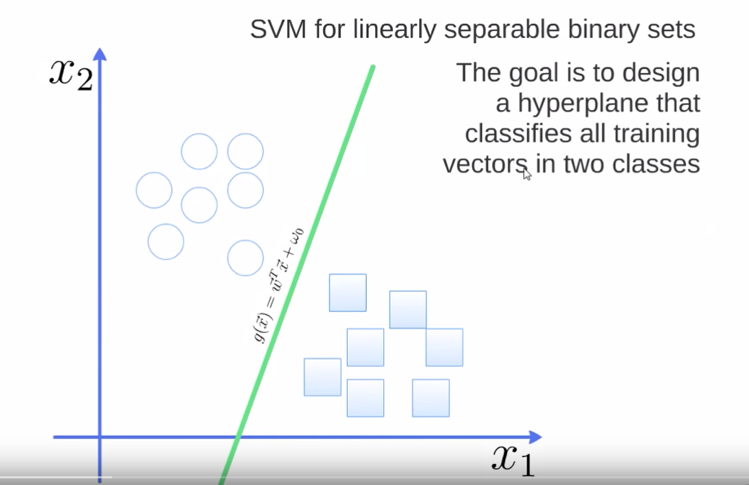
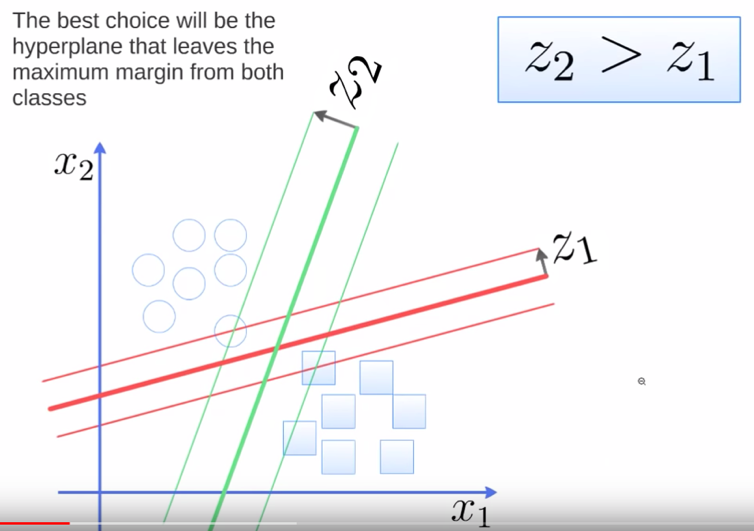
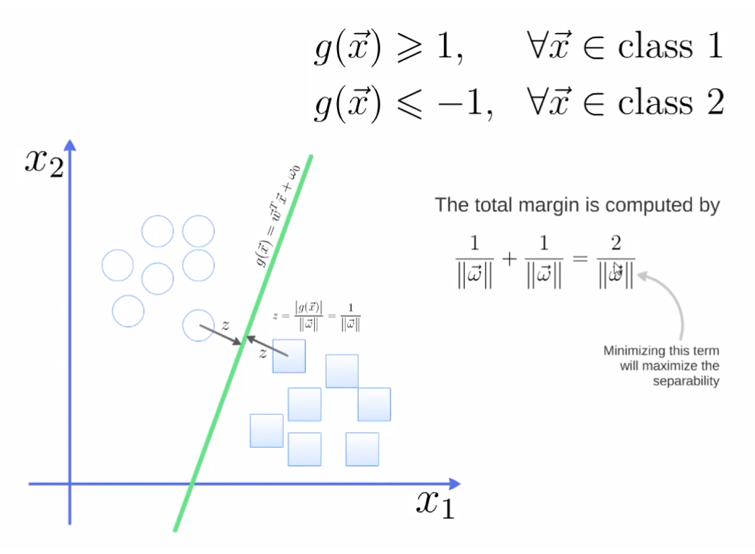
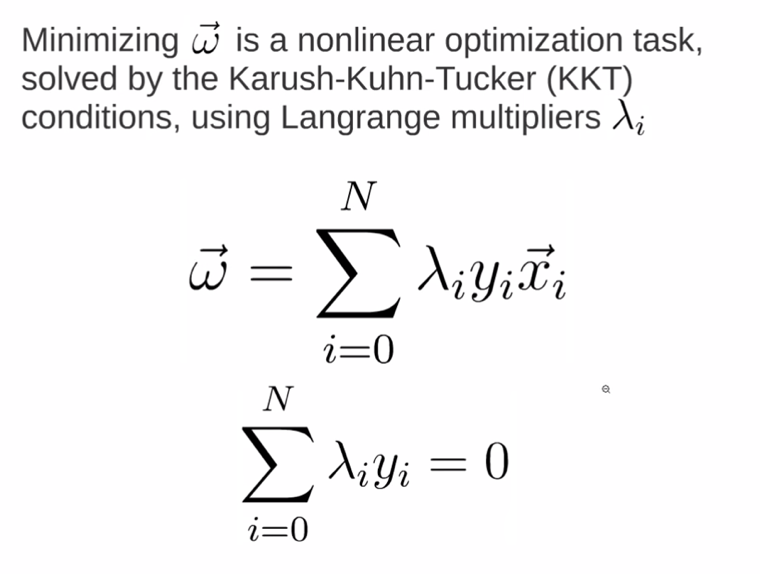
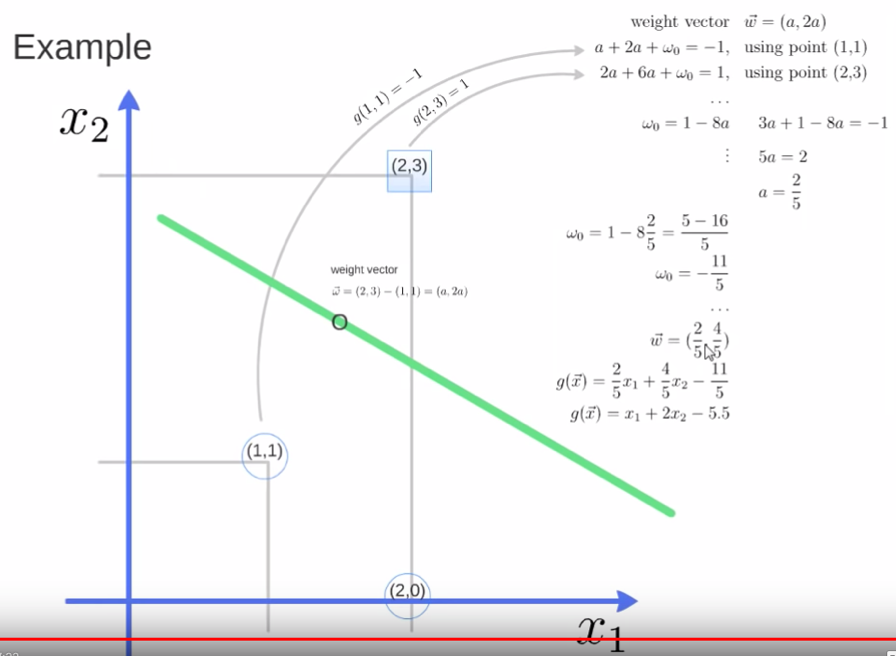
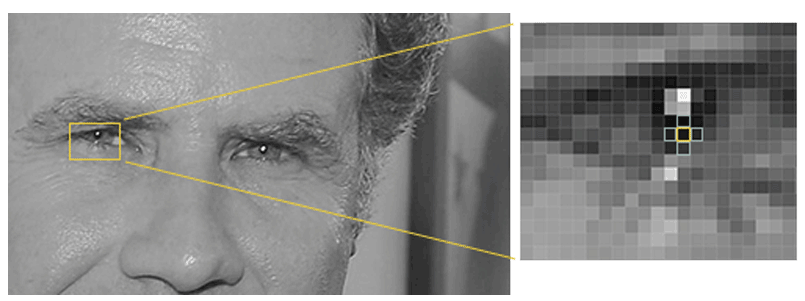
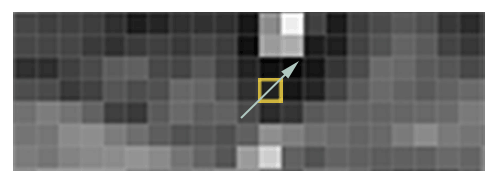
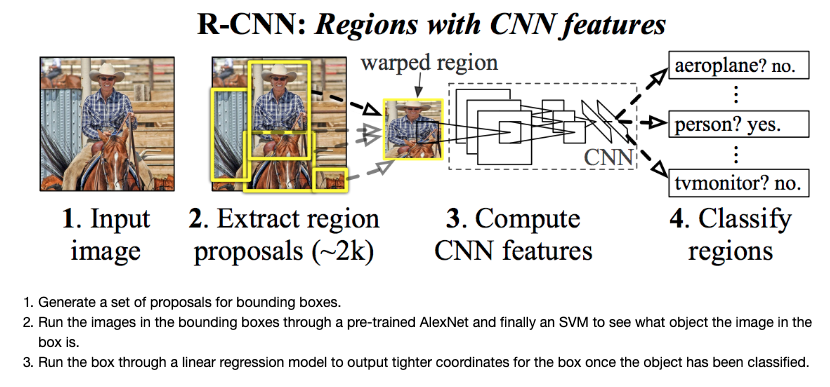
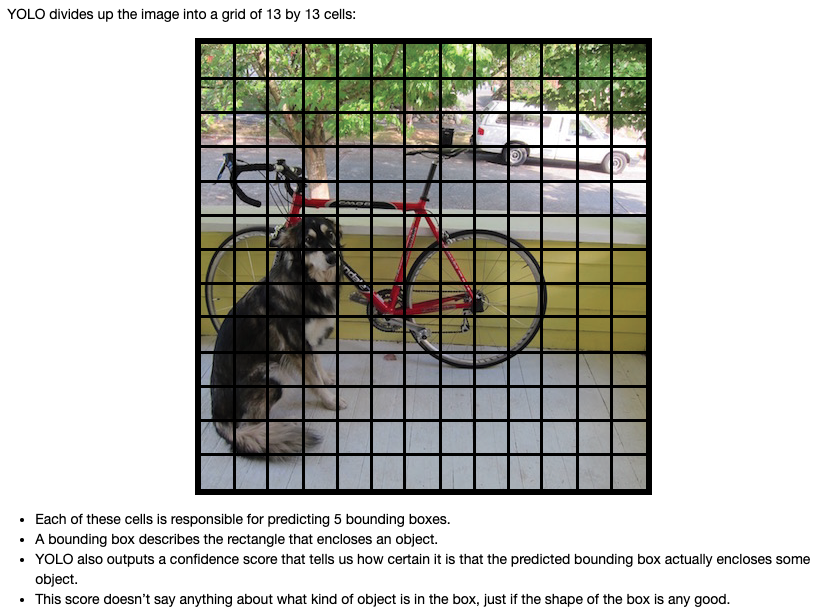
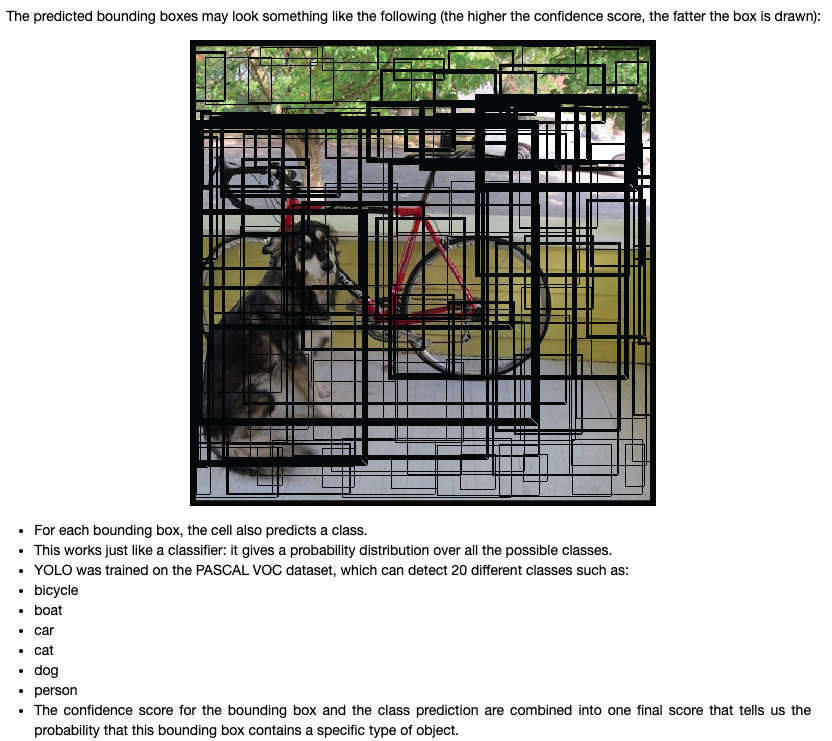
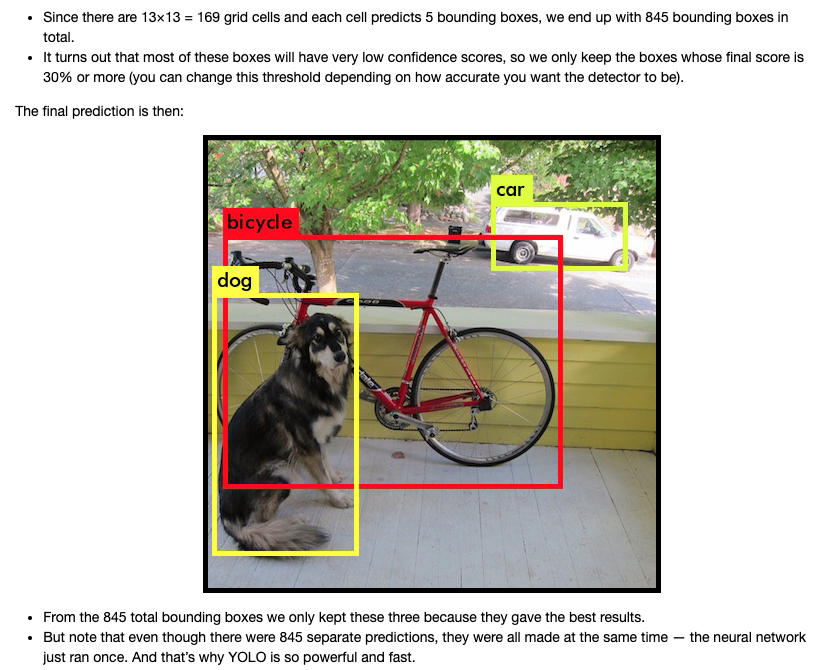
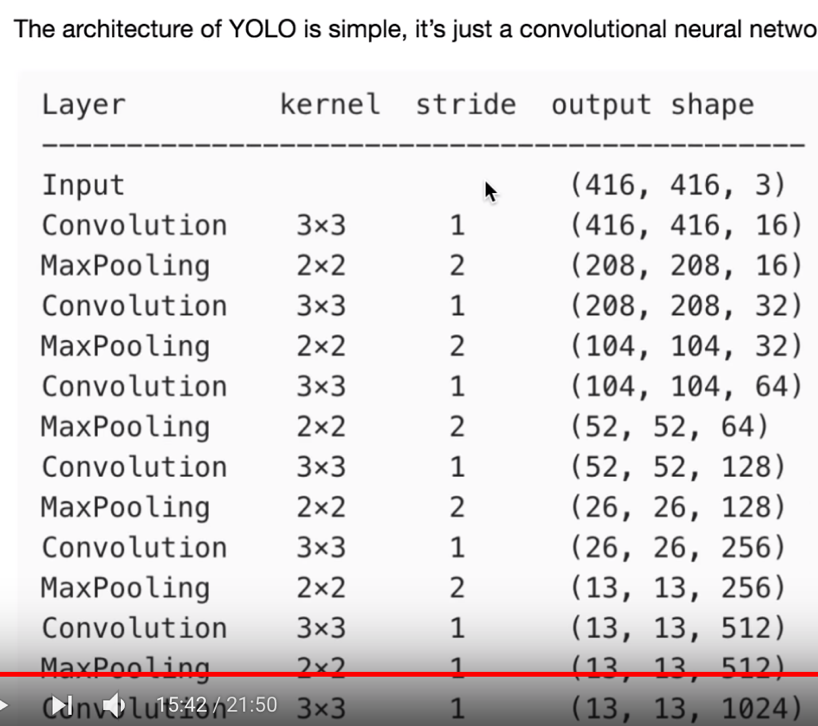
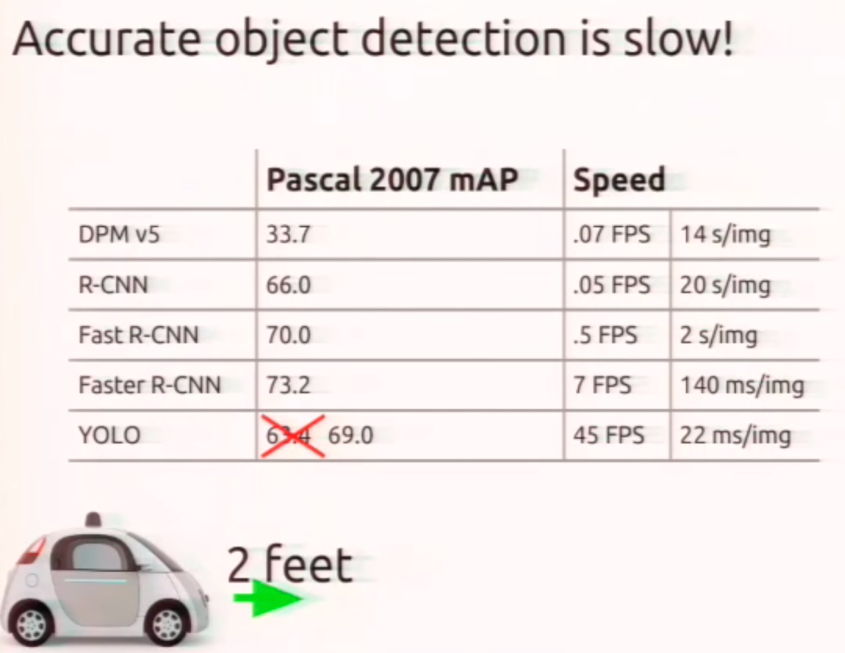
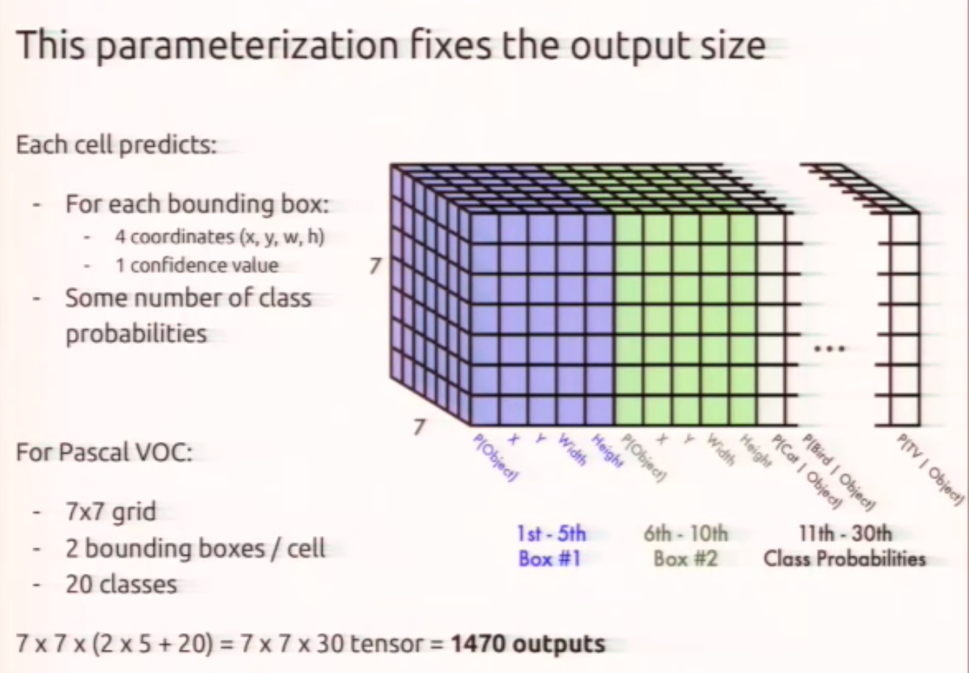
Each feature is related to a special location in the sub-window

* + - 1. Creating an Integral Image
         * An image representation
         * Evaluate rectangular features in constant time

Each feature’s rectangular area is always adjacent to at least one other rectangle

* + - 1. Adaboost Training
         * Select the best features
         * Train classifiers that use the best features
         * Constructing a **strong** classifier as a linear combination of weighted simple **weak** classifiers

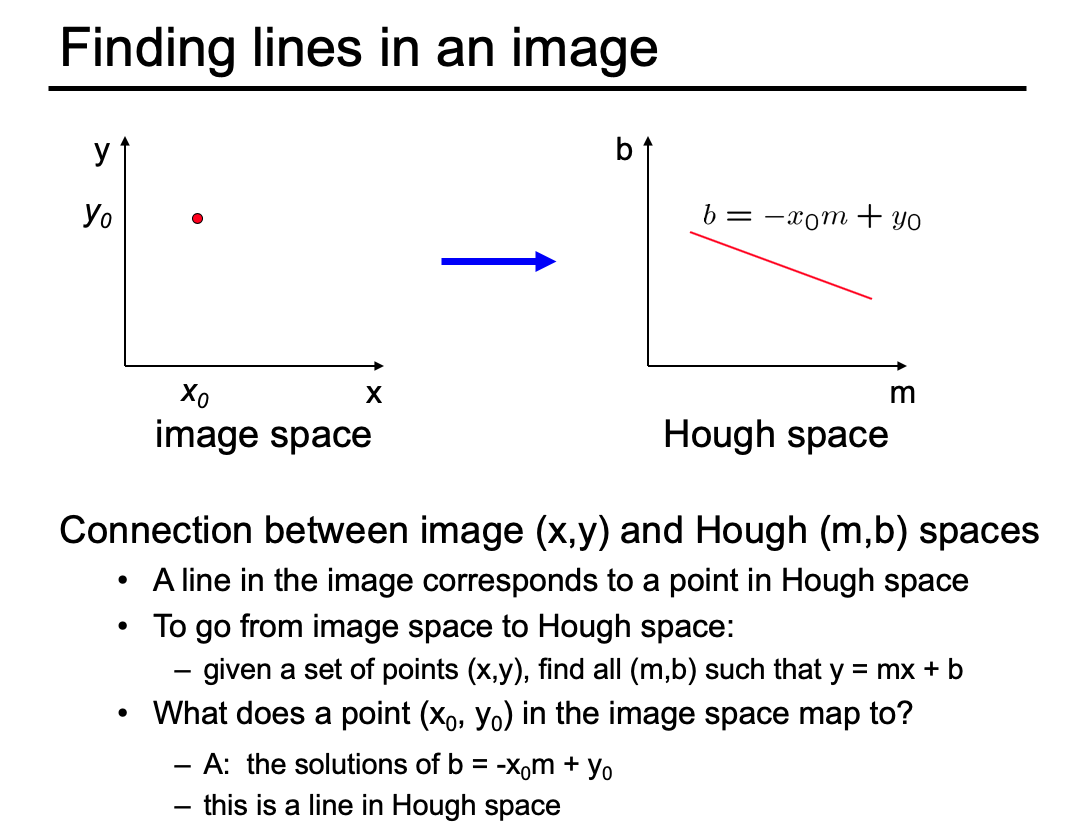
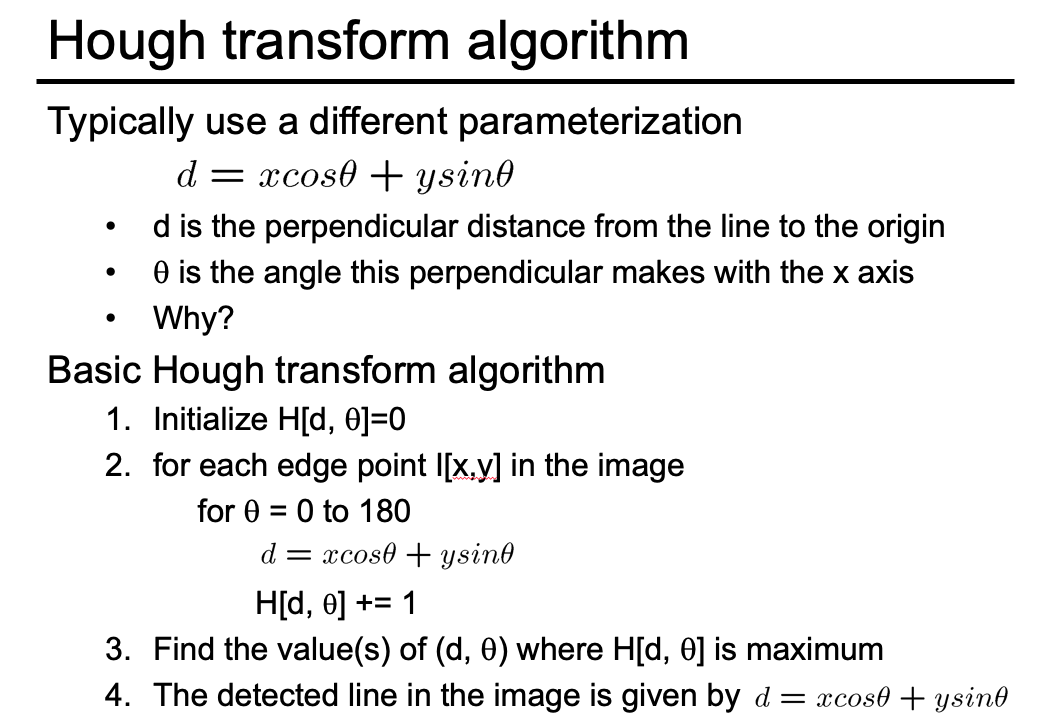
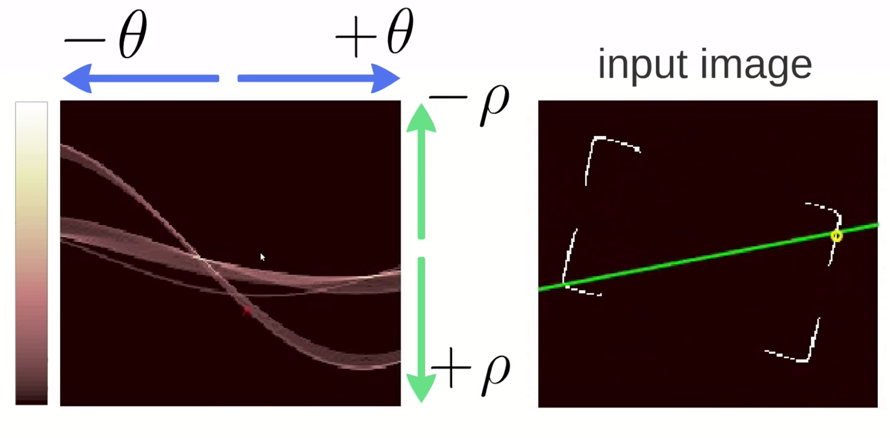
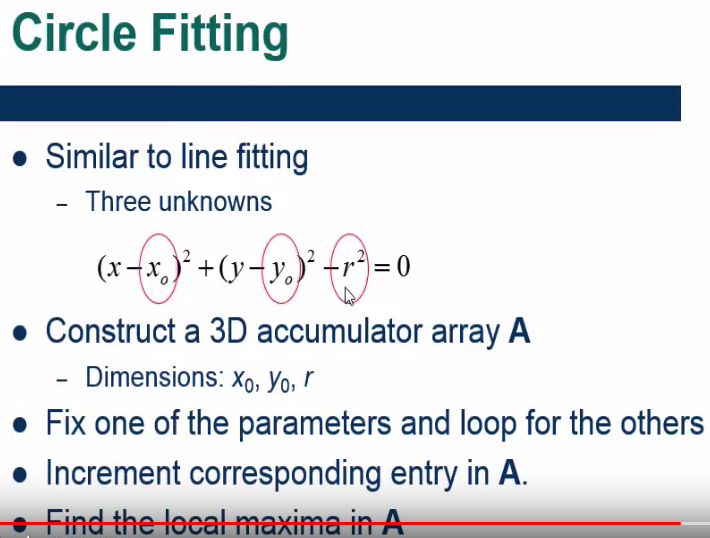
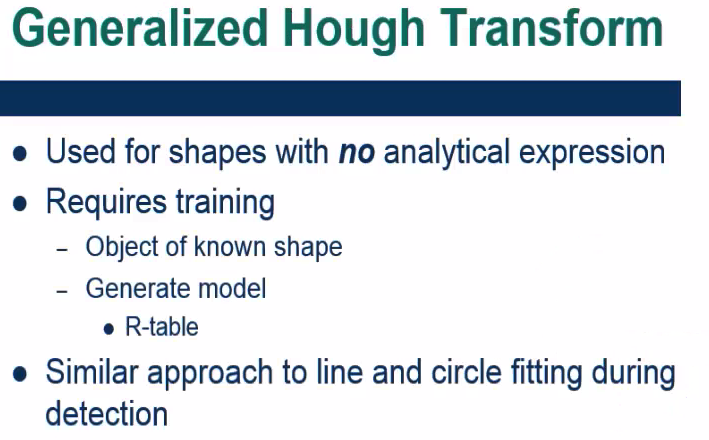
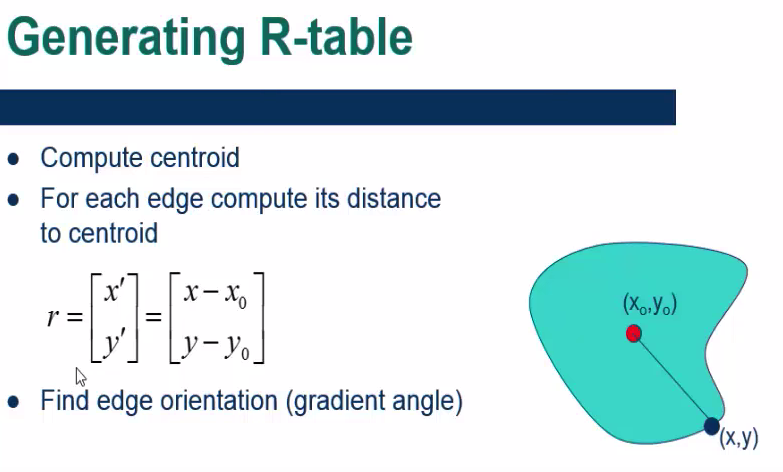
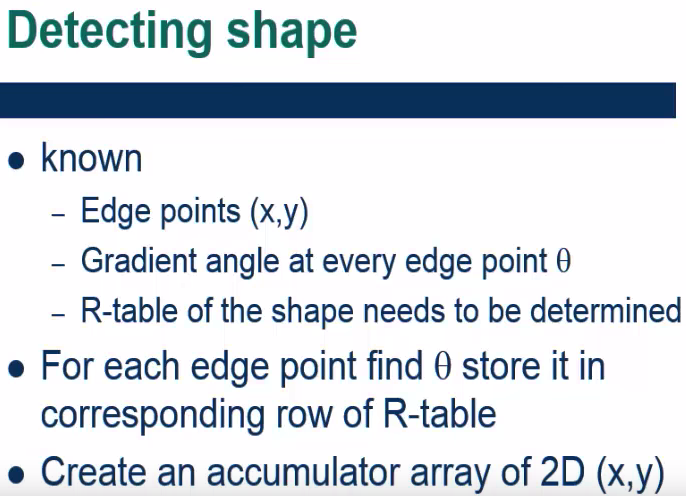
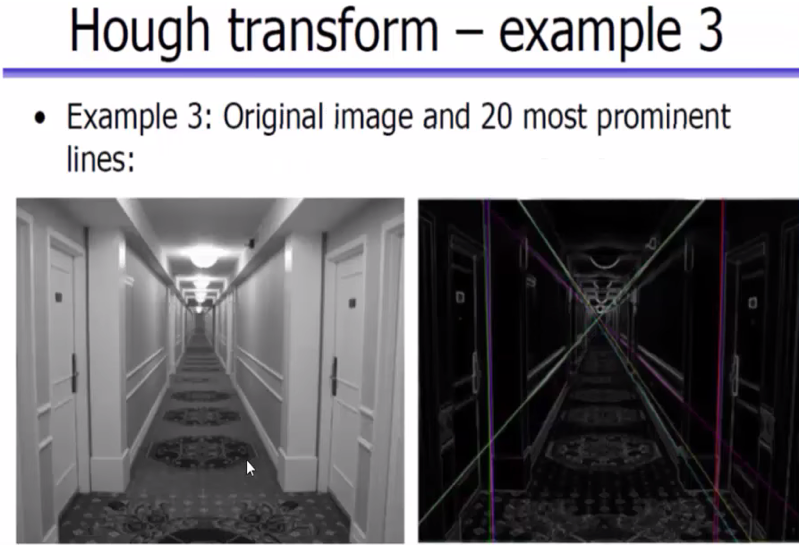
Each weak classifier is a threshold function

* + - 1. SVM
         * 
         * 
         * 
         * 
         * 
* 
  + 1. **Cons**
       - Requires full view frontal upright faces
         * The entire face must point towards the camera and should not be tilted to either side
  + -> **HOG**
    - Histograms of Oriented Gradients, 2005
    - By Navneet Dalal and Bill Triggs
    - Initial for pedestrian detection
    - **Hand coded feature**
    - **Algorithm**
      * For every single pixel, we look at the pixels that directly surrounding it
        + 
      * Draw an arrow showing in which direction the pixel is getting darker to its surrounding pixels
        + 
      * Repeat the process for every single pixel in the image
      * Every pixel is replaced by an arrow. These arrows are called **gradients**.
      * We'll break up the image into small squares of 16x16 pixels each
      * In each square, we’ll count up how many gradients point in each major direction
      * Then we’ll replace that square in the image with the arrow directions that were the strongest.
      * End result? Original image converted into simple representation that captures basic structure of a face in a simple way:
      * Detecting faces means find the part of our image that looks the most similar to a known HOG pattern that was extracted from a bunch of other training faces:
        + 
    - **Pros**
      * Much more efficient detection technique
  + ... **The Deep Learning Era begins** (2012)
* Convolutional Neural Networks became the gold standard for image classification after Kriszhevsky's CNN's performance during ImageNet
  + -> **R-CNN, Regions with CNN features** (2015)
    - By Ross Girshick et al.
    - Before they feed into the CNN, R-CNN will use **selective search** to create 2000 bounding boxes / **region proposals** (in the paper)
      * <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>
      * Selective Search:  
        1. Generate initial sub-segmentation, we generate many candidate regions  
        2. Use greedy algorithm to recursively combine similar regions into larger ones   
        3. Use the generated regions to produce the final candidate region proposals
      * These 2000 candidate region proposals are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output. The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into an [SVM](https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47?source=post_page---------------------------) to classify the presence of the object within that candidate region proposal.
    - At a high level, **selective search** looks at the image through windows of different sizes
    - for each size tries to group together adjacent pixels by text, color, or intensity to identify object
    - **Cons**
      * It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
      * It cannot be implemented real time as it takes around 47 seconds for each test image.
      * The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.
* 
  + -> \*\***YOLO**\*\*  (<https://www.youtube.com/watch?v=NM6lrxy0bxs>)
  + <https://github.com/llSourcell/YOLO_Object_Detection/blob/master/YOLO%20Object%20Detection.ipynb>
  + 
  + 
  + 
  + 
  + How YOLO works is that we take an image and split it into an SxS grid, within each of the grid we take m bounding boxes. For each of the bounding box, the network outputs a class probability and offset values for the bounding box. The bounding boxes having the class probability above a threshold value is selected and used to locate the object within the image.
  + YOLO is orders of magnitude faster(45 frames per second) than other object detection algorithms. The limitation of YOLO algorithm is that it struggles with small objects within the image, for example it might have difficulties in detecting a flock of birds. This is due to the spatial constraints of the algorithm.
    1. divide up the image into a grid of 13 by 13 cells
    2. each cell are to be predicted for 5 bounding boxes
    3. a bounding box described the rectangle that encloses an object
    4. also confidence rate
    5. trained on the PASCAL VOC dataset, which can detect 20 different classes
    6. 
    7. end up with 125 channels for every grid cell. These 125 numbers (5 bounding boxes \* 25 data elements) contain the data for the bounding boxes and the class predictions.
    8. 
    9. If there is one object in the grid cell, then that object is a car (conditional logic)
    10. after we get all the bounding boxes weighted by their actual probabilities for containing that object
        - And we have a bunch of detections for this object
        - we have a lot of boxes with very low confidence value for any class
        - **so** we threshold the predictions, perform non-mac suppression to get rid of some duplicate detections
    11. 30 tensors
    12. predicting all these detections simultaneously
    13. yolo model implicitly incorporates global context in the detection process
        - **so** it can learn things about which objects tend to co-occur together this relative size and location of objects
    14. **Process of predicting full detection from a single image**
        - get an image
        - -> get some ground truth labels for that image
          * (The ground truth is what you measured for your target variable for the training and testing examples.
          * Nearly all the time you can safely treat this the same as the label.)
        - -> match each ground truth label with the appropriate grid cell that we want to predict that detection at test time
          * take the center of a bounding box
          * the cell that the center falls is responsible for predicting that detection
          * adjust that cell's class prediction (eg. predict dog)
          * adjust that cell's bounding box proposals so we look at the cell's predicted boxes and going to figure out which one overlaps **most** with our ground truth label
          * also, adjust that so we want to increase the confidence
          * we also want to adjust its coordinates
          * we also want to look at other bounding boxes predicted by that cell and decrease their confidence since they do not overlap the object
          * we will have a lot of cells in this image that do not have any ground truth detection is overlapping with them
          * we just want to look at all of the bounding boxes for those cells and decrease their confidence as well since they do not contain any objects
          * **notes**

we do not want to adjust the class probabilities or coordinates for those bounding boxes since they are not any actual ground truth labels that we want to predict there

* + 1. **Pros**
       - sped up detection pipeline
       - it can be the same speed as a classification pipeline
    2. **training methodology of neural networks**
       - **tech used**
         * pre trained on images
         * gradient descent
         * data augmentation
         * more details in the paper
* **(back in 2015, openCV 2) Basic motion detection and tracking with Python and OpenCV**
  + **two primary methods**
    1. Gaussian Mixture Model-based foreground
       - [*An improved adaptive background mixture model for real-time tracking with shadow detection*](http://www.ee.surrey.ac.uk/CVSSP/Publications/papers/KaewTraKulPong-AVBS01.pdf) by KaewTraKulPong et al., available through the**cv2.BackgroundSubtractorMOG**  function.
       - [*Improved adaptive Gaussian mixture model for background subtraction*](http://www.zoranz.net/Publications/zivkovic2004ICPR.pdf) by Zivkovic, and [*Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Subtraction*](http://www.zoranz.net/Publications/zivkovicPRL2006.pdf), also by Zivkovic, available through the **cv2.BackgroundSubtractorMOG2**  function.
    2. background segmentation
    3. in newer versions of OpenCV, we have Bayesian (probability) based foreground and background segmentation
       - implemented from Godbehere et al.’s 2012 paper, [*Visual Tracking of Human Visitors under Variable-Lighting Conditions for a Responsive Audio Art Installation*](http://goldberg.berkeley.edu/pubs/acc-2012-visual-tracking-final.pdf).
       - cv.**createBackgroundSubtractorGMG**
  + **All of these methods** are concerned with segmenting the background from the foreground
    1. even discern between actual motion and shadowing and small lighting changes
  + **Why do we care what pixels belong to the foreground and what pixels are part of the background?**
    1. in motion detection, we tend to make the following assumption:
    2. **The *background* of our video stream is largely *static and unchanging* over consecutive frames of a video. Therefore, if we can model the background, we monitor it for substantial changes. If there is a substantial change, we can detect it — this change normally corresponds to *motion* on our video.**
    3. **However, everything is changing in real life**, so the most successful background subtraction **/ foreground detection** systems utilize fixed mounted cameras and in controlled lighting conditions
  + if you guessed that it stores the first frame of the video file/webcam stream, you’re right.
  + ***Assumption:****The first frame of our video file will contain*no motion*and*just background*— therefore, we can model the background of our video stream using only the first frame of the video*.
  + Obviously we are making a pretty big assumption here. But again, our goal is to run this system on a Raspberry Pi, so we can’t get too complicated. And as you’ll see in the results section of this post, we are able to easily detect motion while tracking a person as they walk around the room.
  + **# resize the frame, convert it to grayscale, and blur it**
  + **# compute the absolute difference between the current frame and**
  + **# first frame**
  + **# dilate the thresholded image to fill in holes, then find contours**
  + **# on thresholded image**
  + **# loop over the contours**
  + **# if the contour is too small, ignore it**
  + **# compute the bounding box for the contour, draw it on the frame,**
  + **# and update the text**

Hough transform

* A voting scheme
* 
* 
* Just looking for edges
* 
* 
* 
* 
* 
* 

**YOLO**

Differences from previous approaches on object detection:

Prev: repurposes classifiers to perform detection

Yolo: frame object detection as a regression problem to spatially separated bounding boxes and associated class probabilities.

* A single neural network predicts bounding boxes **and** class probabilities directly from full images in one evaluation

Performance:

Processes images in real time at 45 frames per second

Pros:

1. Fast
2. Less likely to predict false positives on backgrounds

Cons:

1. Makes more localization errors

We reframe object detection as a single regression problem,

straight from image pixels to bounding box coordinates

and class probabilities. Using our system, you only

look once (YOLO) at an image to predict what objects are

present and where they are.

Processing images

with YOLO is simple and straightforward. Our system

(1) resizes

the input image to 448 \_ 448,

(2) runs a single convolutional network

on the image, and

(3) thresholds the resulting detections by

the model’s confidence.

A single

convolutional network simultaneously predicts multiple

bounding boxes and class probabilities for those boxes.

YOLO trains on full images and directly optimizes detection

performance.

Benefits of the unified model:

1. YOLO is extremely fast. Since we frame detection

as a regression problem we don’t need a complex pipeline.

We simply run our neural network on a new image at test

time to predict detections.

2. Furthermore, YOLO

achieves more than twice the mean average precision of

other real-time systems. For a demo of our system running

in real-time on a webcam please see our project webpage:

<http://pjreddie.com/yolo/>.

3. YOLO reasons globally about the image when

making predictions. Unlike sliding window and region

proposal-based techniques, YOLO sees the entire image

during training and test time so it implicitly encodes contextual

information about classes as well as their appearance.

Systems like deformable parts

models (DPM) use a sliding window approach where the

classifier is run at evenly spaced locations over the entire

image

R-CNN use region proposal methods to first generate potential bounding boxes in an image

and then run a classifier on these proposed boxes. After

classification, post-processing is used to refine the bounding

boxes, eliminate duplicate detections, and rescore the

boxes based on other objects in the scene

Unlike classifier-based approaches,

YOLO is trained on a loss function that directly corresponds

to detection performance and the entire model is trained

jointly.

**Unified Detection**

1. Our network uses features

from the entire image to predict each bounding box.

2. It also

predicts all bounding boxes across all classes for an image

simultaneously.

3. our network reasons globally

about the full image and all the objects in the image.

Our system divides the input image into an S \* S grid.

If the center of an object falls into a grid cell, that grid cell

is responsible for detecting that object.

**Each grid cell predicts** B bounding boxes and confidence

scores for those boxes. These **confidence scores** reflect how

confident the model is that the box contains an object and

also how accurate it thinks the box is that it predicts.

Formally

we define confidence as Pr(Object) \* IOUtruth pred .

If no object exists in that cell, the confidence scores should be

zero. Otherwise we want the confidence score to equal the

**intersection over union (IOU)** between the predicted box

and the ground truth.

Each bounding box consists of 5 predictions: x, y, w, h,

and confidence. The (x; y) coordinates represent the center

of the box relative to the bounds of the grid cell. The width

and height are predicted relative to the whole image. Finally

the confidence prediction represents the IOU between the

predicted box and any ground truth box.

**Each grid cell also predicts** C conditional class probabilities,

Pr(Classi | Object). These probabilities are conditioned

on the grid cell containing an object. We only predict

one set of class probabilities per grid cell, regardless of the

number of boxes B.

At test time we multiply the conditional class probabilities

and the individual box confidence predictions,



which gives us class-specific confidence scores for each

box.

These scores encode both the probability of that class

appearing in the box and how well the predicted box fits the

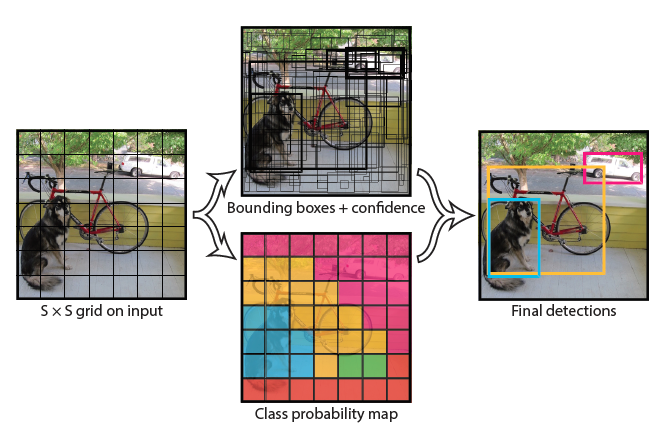
object.

The Model. Our system models detection as a regression

problem. It divides the image into an S \* S grid and for each

grid cell predicts B bounding boxes, confidence for those boxes,

and C class probabilities. These predictions are encoded as an

S \* S \* (B \* 5 + C) tensor.

**From paper** <https://arxiv.org/abs/1506.02640>

**Training**

We train

this network for approximately a week and achieve a single

crop top-5 accuracy of 88% on the ImageNet 2012 validation

set, comparable to the GoogLeNet models in Caffe’s

Model Zoo [24]. We use the Darknet framework for all

training and inference [26].

Our final layer predicts both class probabilities and

bounding box coordinates. We normalize the bounding box

width and height by the image width and height so that they

fall between 0 and 1. We parametrize the bounding box x

and y coordinates to be offsets of a particular grid cell location

so they are also bounded between 0 and 1.

YOLO predicts multiple bounding boxes per grid cell.

At training time we only want one bounding box predictor

to be responsible for each object. We assign one predictor

to be “responsible” for predicting an object based on **which**

**prediction has the highest current IOU with the ground**

**truth**. This leads to specialization between the bounding box

predictors. Each predictor gets better at predicting certain

sizes, aspect ratios, or classes of object, improving overall

recall.

So, to put it simple, you take an image as input, pass it through a neural network that looks similar to a normal CNN, and you get a vector of bounding boxes and class predictions in the output.

More formally, in order to apply Intersection over Union to evaluate an (arbitrary) object detector we need:

1. The *ground-truth bounding boxes* (i.e., the hand labeled bounding boxes from the testing set that specify *where* in the image our object is).
2. The *predicted bounding boxes* from our model.