CHAPTER 1

ABSTRACT

An ICAR is a vehicle sensing it’s environment, where there’s no human input at all. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and traffic sign and signal. The car will understand the road surface and respond to all the curves and turns in the road. These things are done with image processing. Recent advances in Deep Neural Networks (DNNs) have led to the development of DNN-driven autonomous cars that, using sensors like camera, etc., can drive without any human intervention.

Most major manufacturers including Tesla, GM, Ford, BMW, and Waymo/Google are working on building and testing different types of autonomous vehicles. Considering the future of autonomous vehicles is an ambitious era of safe and comfortable transportation, hence Instead of adding much add-ons like LiDAR, sensor, etc.

We are making it more streamlined and productive using Deep neural networks and image processing.

CHAPTER 2

INTRODUCTION

In this program, you'll sharpen your Python skills, apply C++, apply matrices and calculus in code, and touch on computer vision and machine learning. These concepts will be applied to solving self-driving car problems. At the end, you'll be ready for our Self-Driving Car Engineer Nanodegree program.

EVOLUTION

Fully autonomous or "self-driving" vehicles are an emerging technology that may hold tremendous mobility potential for individuals who are visually impaired who have been previously disadvantaged by an inability to operate conventional motor vehicles.

Prior studies however, have suggested that these consumers have significant concerns regarding the accessibility of this technology and their ability to effectively interact with it. We present the results of a quasi-naturalistic study, conducted on public roads with 20 visually impaired users, designed to test a self-driving vehicle human–machine interface. This prototype system, ATLAS, was designed in participatory workshops in collaboration with visually impaired persons with the intent of satisfying the experiential needs of blind and low vision users.

Our results show that following interaction with the prototype, participants expressed an increased trust in self-driving vehicle technology, an increased belief in its likely usability, an increased desire to purchase it and a reduced fear of operational failures. These findings suggest that interaction with even a simulated self-driving vehicle may be sufficient to ameliorate feelings of distrust regarding the technology and that existing technologies, properly combined, are promising solutions in addressing the experiential needs of visually impaired persons in similar contexts.

CHAPTER 3

COMPUTER VISION – LANE FINDING

The project consisted of the following stages:

1. Camera calibration to remove lens distortion effects.
2. Image pre-processing to detect lane lines.
3. Perspective transform on road to aid in detection.
4. Lane Line Detection on transformed road.
5. Vehicle position and lane radius of curvature calculation.
6. Generate video of results.

Camera Calibration

The output from the camera is a video, which in essence is a time-series of images. Due to the nature of photographic lenses, images captured using pinhole camera models are prone to radial distortions which result in inconsistent magnification depending on the object's distance from the optical axis.

The following is an example image from OpenCV showcasing the two main types of radial distortions:

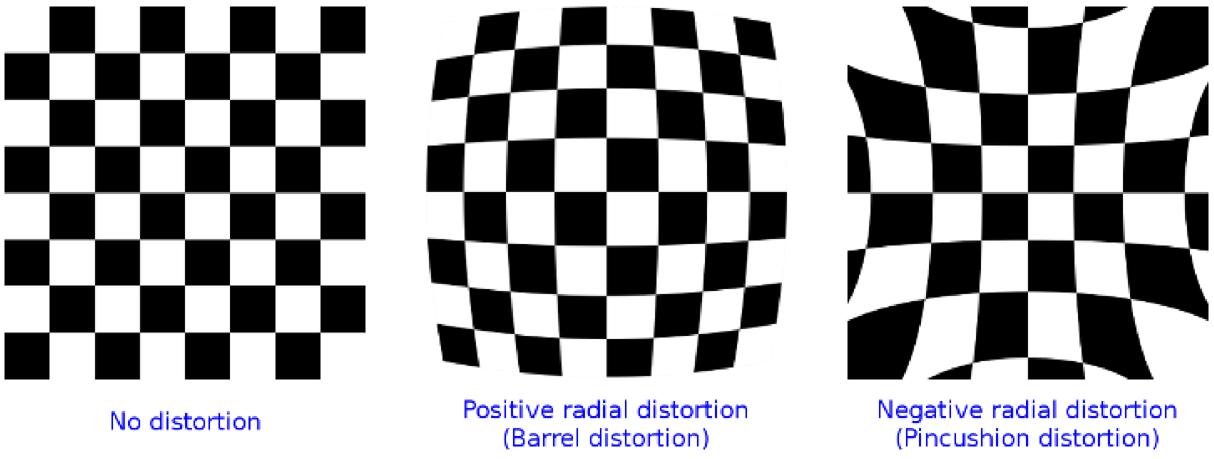


Figure 1. Examples of radial distortion.

In order to correctly detect the lane lines in the image, we first need to correct for radial distortion.Computer vision researchers have come up with a way to correct this radial distortion. The camera to be calibrated is used to capture images of checkerboard patterns, where all the white and black boxes in the pattern are of the same size. If the camera suffers from distortions, the captured images will incorrectly show the measurements of the checkerboard.

To correct the effects of distortion, the corners of the checkerboard are identified and deviations from the expected checkerboard measurements are used to calculate the distortion coefficients. These coefficients are then used to remove radial distortion from any image captured using that camera.

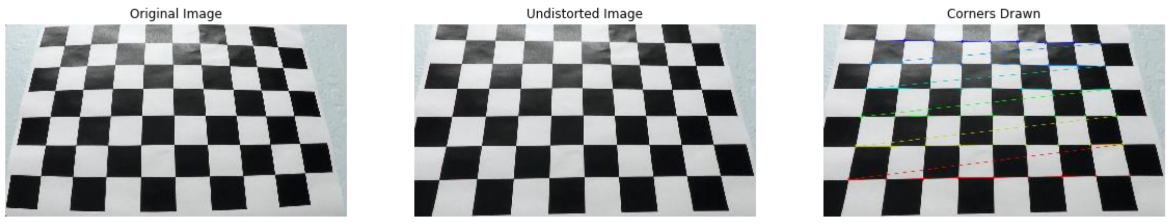


Figure 2. Checkerboard before and after fixing for distortion.

In the diagram above, the leftmost image shows the original distorted image, the rightmost image shows the corners drawn on top of the distorted image, and the middle image shows the resultant undistorted image after camera calibration.

The OpenCV functions findChessboardCorners and calibrateCamera were used to achieve the above camera calibration process.

Now that we've calibrated our camera, I tested the results on actual footage from the car video. The below image shows the result of camera calibration:

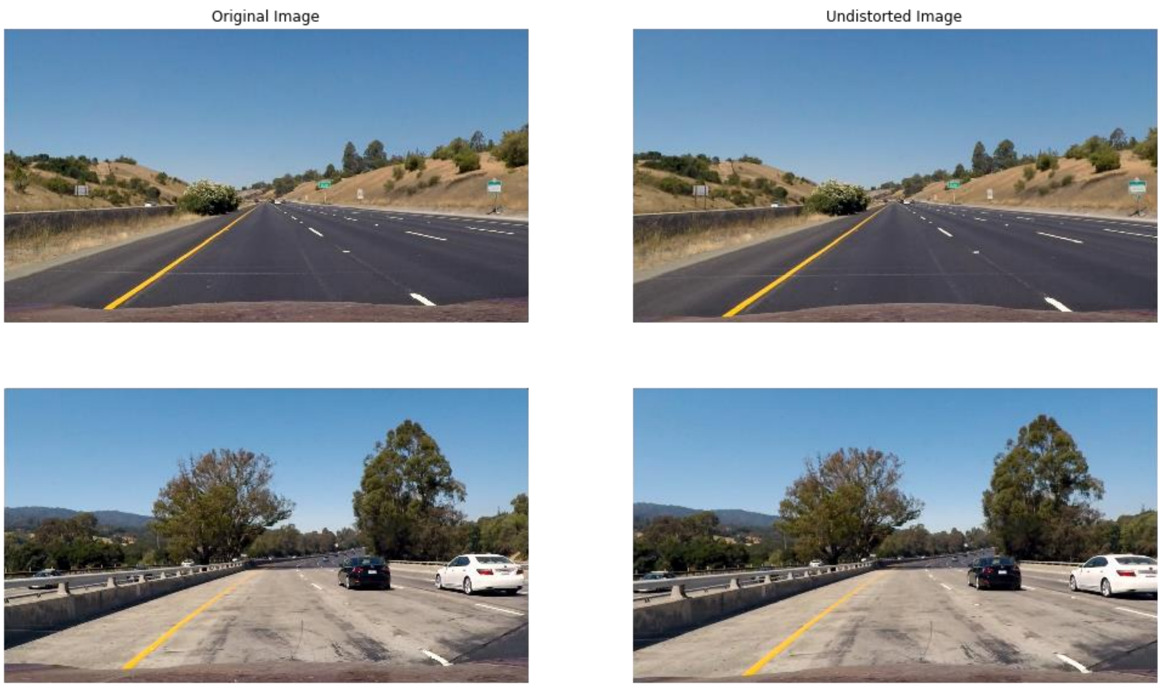


Figure 3. Distortion correct applied for snapshots from the car driving video.

Image Preprocessing

With the undistorted images at hand, we now return to the main objective of detecting the lane lines on the road. One way to separate and detect objects in an image is to use colour transforms and gradients to generate a filtered-down thresholded binary image.

For color transforms, I experimented with three color spaces in order to find out which one is best at filtering the pixels representing the lane line on the road. Three colour spaces were tested:

- HSL: represents colour as three channels — hue, saturation, and lightness.
- LAB: represents colour as three channels — lightness, component a for green-red, and component b for blue-yellow.
- LUV: a transformation of the XYZ colorspace that attempts perceptual uniformity.

After some experimentation, I concluded that the **b channel** of the LAB colorspace and the **L channel** of the LUV colour space are the best combination for detecting the lane lines on the road.

The Sobel gradient filter was also considered. An image gradient measures the directional intensity of the colour change. Sobel is a type of gradient filter that uses Gaussian smoothing and differentiation operations to reduce the effects of noise.

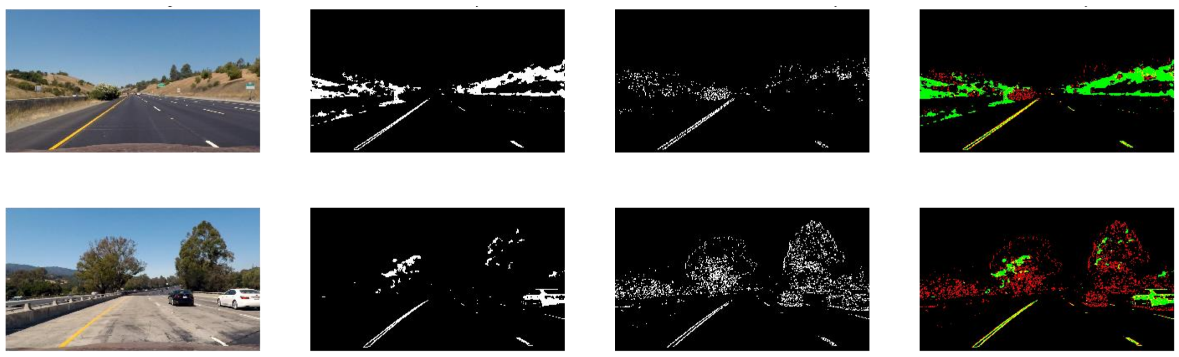


Figure 4. Original undistorted images in the first column, the b/L channel thresholding in the second column, the Sobel gradient filter in the third column, and the two filters combined in the last column.

Perspective Transform

We can now distinguish lane lines in the image, but the task of figuring out the exact angle/direction of the lane is difficult using the default camera view. In the default camera perspective, objects further away from the camera appear smaller and the lane lines appear to converge the further they are from the car, which isn't a true representation of the real world.

One way to fix this perspective distortion is to transform the perspective of the image such that we are looking at the image from above, also known as birds-eye view.

OpenCV provides functions get Perspective Transform and warp Perspective, which can be used to apply a perspective transformation to a segment in the image. Firstly, we pick the area in the image we would like to apply the transformation to. In the image below, I've picked the lane line segment in front of the car:

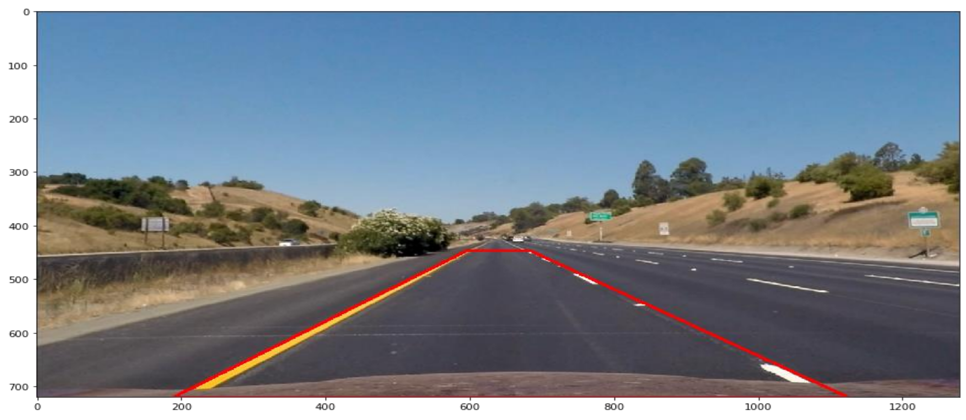


Figure 5. Source points we want to apply a perspective transform to.

Then, we choose the points representing the destination space we would like to transform the segment to, in our case any rectangle would suffice. The function will then return a 3x3 transformation matrix which can be used to warp any segment into our chosen perspective using the warp Perspective function.

The following image shows lanes lines from two different segments of the road with the perspective transformation successfully applied:

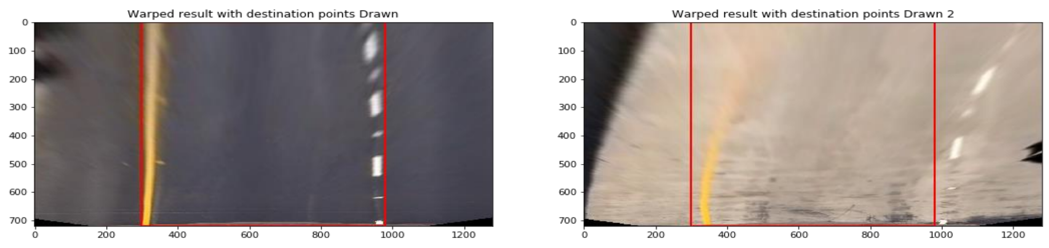

Warped result with destination points Drawn
Warped result with destination points Drawn 2

Figure 6. Perspective transform applied to two different sections of the road.

Notice how it is now much easier to determine the curvature of the lane line!

Lane Line Detection

We are now finally ready to fully detect the lane lines! As a start, we apply the binary thresholding discussed in the Image Preprocessing section to the perspective transformed lane line segment. We now have an image where the white pixels represent parts of the lane line we are trying to detect.

Next, we need to find a good starting point to look for pixels belonging to the left lane line and pixels belonging to the right lane line. One approach is to generate a histogram of the lane line pixels in the image. The histogram should have two peaks each representing one of the lane lines, where the left peak is for the left lane line and the right peak is for the right lane line. The image below shows two example histograms generated from two binary images:

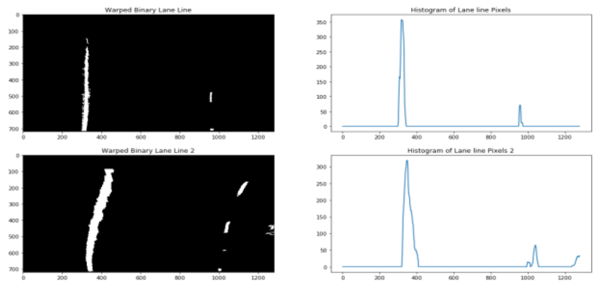


Figure 7. Binary thresholder images and histogram of pixels in thresholder images.

The locations of the two peaks are then used as a starting point to search for pixels belonging to each lane line. We employ a sliding window search technique that starts from the bottom and iteratively scans all the way to the top of the image, adding detected pixels to a list. If a sufficient number of pixels is detected in a window, the next window will be centred around their mean position, that way we are following the path of the pixels throughout the image.

After we've detected the pixels belonging to each lane line, we then fit a polynomial through the points, generating a smooth line which acts as our best approximation of where the lane line is.

The image below shows the sliding window technique in action, with the polynomial fit through the detected lane pixels (red for left lane pixels and blue for right lane pixels):

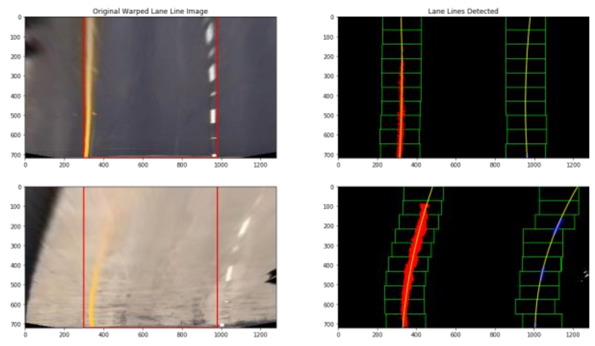


Figure 8. Warped images and polynomial fit produced using sliding window technique.

Below is another view of the sliding window search technique, with the search area highlighted and filled:

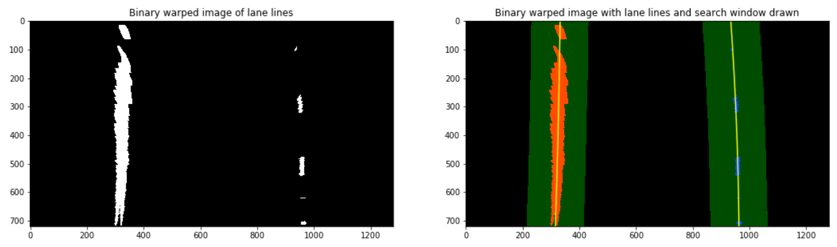


Figure 9. Sliding window technique compared to the binary image used as input.

Vehicle/Lane Position

Finally, using the location of the two detected lane lines and the assumption that the camera is located at the centre of the image, we then calculate the position of the car relative to the lane. Scale measurements to convert from pixels to meters have been calculated using the resolution of the image.

Furthermore, using the scale measurements, we can also calculate the curvature of the lane by fitting a new polynomial to the world space, then calculating the radii of curvature. The radius of curvature of the lane is then just the average of the two radii. Below image shows curve radius and centre offset for the two-lane lines (detection not visible in image):

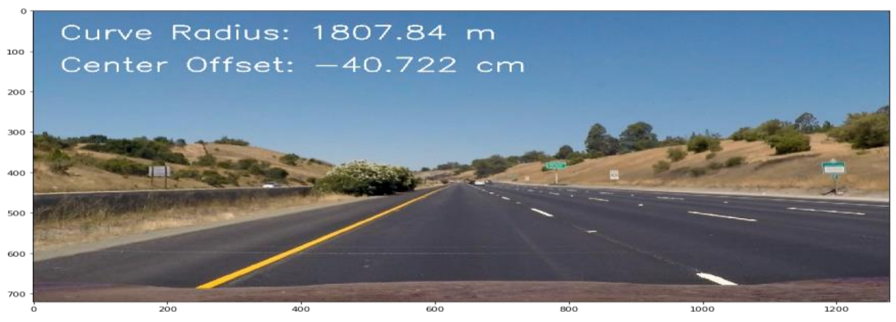


Figure 10. snapshot from the video with the lane curvature and car's centre offset superimposed.

CHAPTER 4

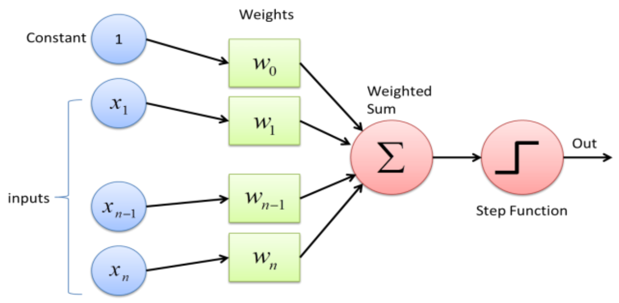
PERCEPTRON

Perceptron is a linear classifier (binary). Also, it is used in supervised learning.

The perceptron consists of 4 parts .

1. Input values or One input layer
2. Weights and Bias
3. Net sum
4. Activation Function

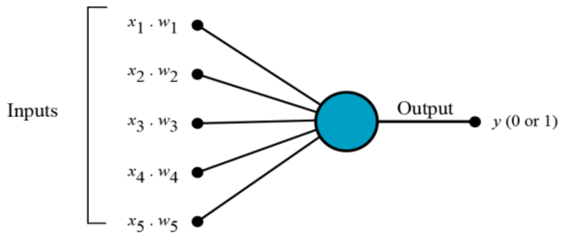
FYI: The Neural Networks work the same way as the perceptron. So, if you want to know how neural network works, learn how perceptron works.



Perceptron

The perceptron works on these simple steps

a. All the inputs **x** are multiplied with their weights **w**. Let’s call it **k.**



Multiplying inputs with weights for 5 inputs

b. **Add** all the multiplied values and call them **Weighted Sum.**

c. **Apply** that weighted sum to the correct **Activation Function**.

For Example : Unit Step Activation Function.

$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$

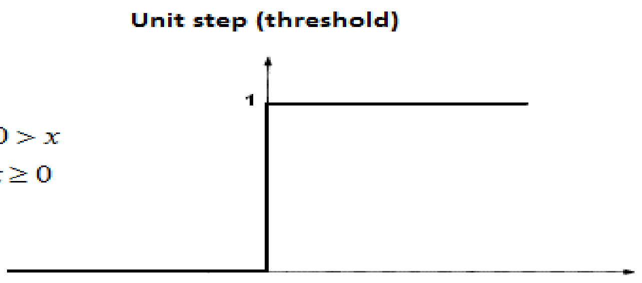
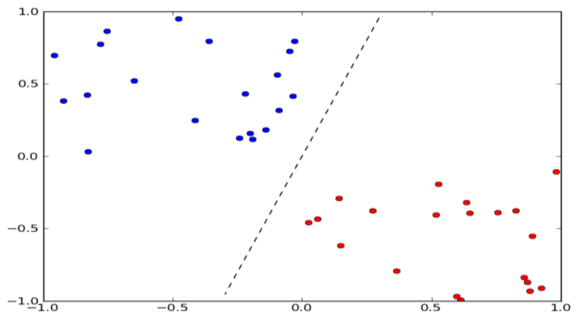


Fig: Unit Step Activation Function

Where we use Perceptron?

Perceptron is usually used to classify the data into two parts. Therefore, it is also known as a Linear Binary Classifier.



CHAPTER 5

KERAS

Keras is a high-level neural networks API, capable of running on top of Tensorflow, Theano, and CNTK. It enables fast experimentation through a high level, user-friendly, modular and extensible API. Keras can also be run on both CPU and GPU. Keras was developed and is maintained by Francois Chollet and is part of the Tensorflow core, which makes it Tensorflows preferred high-level API.

This article is the first of a little series explaining how to use Keras for deep learning.In this article, we will go over the basics of Keras including the two most used Keras models (Sequential and Functional), the core layers as well as some preprocessing functionalities.

Loading in a dataset

Keras provides seven different datasets, which can be loaded in using Keras directly. These include image datasets as well as a house price and a movie review datasets.

In this article, we will use the MNIST dataset, which contains 70000 28x28 grayscale images with 10 different classes. Keras splits it in a training set with 60000 instances and a testing set with 10000 instances.

To feed the images to a convolutional neural network we transform the dataframe to four dimensions. This can be done using numpys reshape method. We will also transform the data into floats and normalize it.

The easiest way of creating a model in Keras is by using the sequential API, which lets you stack one layer after the other. The problem with the sequential API is that it doesn't allow models to have multiple inputs or outputs, which are needed for some problems.

Nevertheless, the sequential API is a perfect choice for most problems.

To create a convolutional neural network we only need to create a Sequential object and use the add function to add layers.

The code above first of creates a Sequential object and adds a few convolutional, maxpooling and dropout layers. It then flattens the output and passes it two a last dense and dropout layer before passing it to our output layer.

The sequential API also supports another syntax where the layers are passed to the constructor directly.

Augmenting Image data

Augmentation is a process of creating more data from existing once. For images you can to little transformations like rotating the image, zooming into the image, adding noise and many more.

This helps to make the model more robust and solves the problem of having not enough data. Keras has a method called Image Data Generator which can be used for augmenting images.

Fit a model

Now that we defined and compiled our model it’s ready for training. To train a model we would normally use the fit method but because we are using a data generator we will use fit generator and pass it our generator, X data, y data as well as the number of epochs and the batch size. We will also pass it a validation set so we can monitor the loss and accuracy on both sets as well as steps per epoch which is required when using a generator and is just set to the length of the training set divided by the batch size.

Visualizing the training process

We can visualize our training and testing accuracy and loss for each epoch so we can get intuition about the performance of our model. The accuracy and loss over epochs are saved in the history variable we got whilst training and we will use Matplotlib to visualize this data.

CHAPTER 6

TRAFFIC SIGNS CLASSIFICATION WITH A CNN

we still must rely upon them as long as autonomous vehicles with different levels of autonomy co-exist harmoniously with human drivers on the road. To accomplish such a task, it would mean that autonomous vehicles should have the ability to read and understand the traffic signs, and to determine the appropriate action. This project focuses on the former: To develop a neural network that reads traffic signs and classifies them correctly.

These are the steps I followed to accomplish the task:

1. Explore and Visualize the dataset

2. Pre-process and augment the dataset, if needed

3. Develop a CNN model

4. Train and validate the model

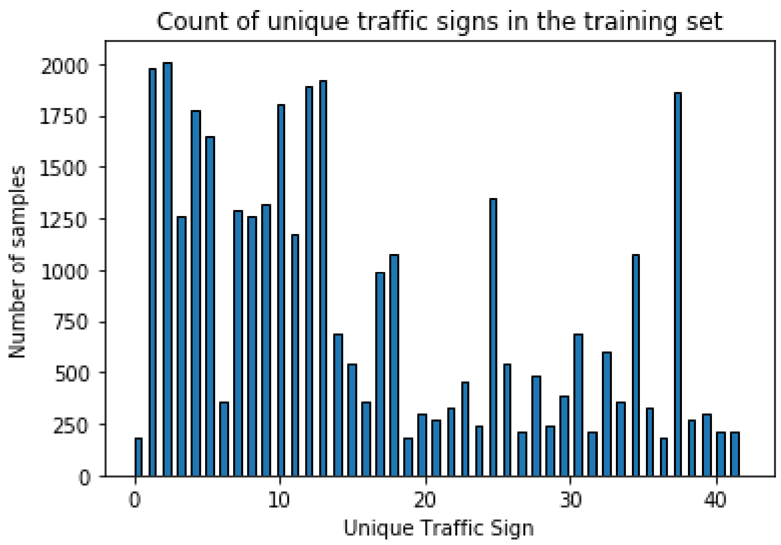
5. Optimize the model by experimenting with different hyper-parameters

6. Test the model with the test dataset

Let’s delve deeper into each of the steps above to understand the process better.

Explore & Visualize the dataset:

The dataset used for this project is German Traffic Signs. There are 43 unique traffic signs in the dataset and a quick look at the histogram of the number of samples for each traffic sign shows that they are unevenly distributed. Each image in the dataset is 32 pixels x 32 pixels x 3 Channels (one each for RGB).



DATA SUMMARY:

Number of training examples = 34799

Number of validation examples = 4410

Number of testing examples = 12630

Image shape = (32, 32, 3)

Here are some random images from the input dataset:

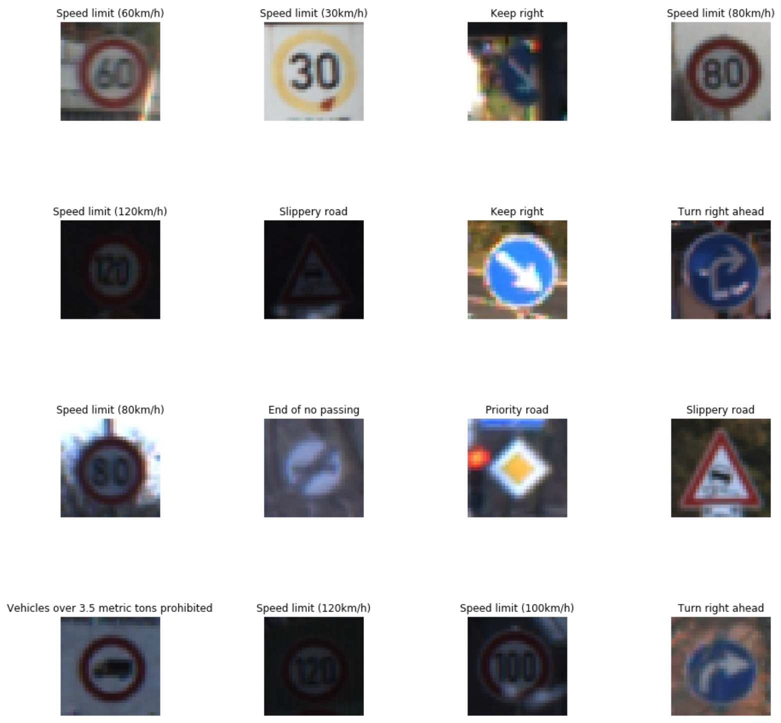


Figure 2: Random visualization of images in the training dataset

Pre-process and augment the dataset, if needed:

All the images in the dataset were normalized so that the data has a mean of zero and equal variance. This helps the model converge faster. I ran the model without any image augmentation and found out that the validation accuracy for the model was quite high when compared to the training accuracy, as seen in the 1st column of Fig. 5 at the end of this post. Since there was an uneven distribution of the number of images for each class as shown in the histogram of the training set images in the previous step, I augmented images for some classes such that the minimum number of images in each class is 250. A summary of the number of images that were added to those classes which had less than 250 images is shown below:

Adding 70 samples for class 0

Adding 70 samples for class 19

Adding 10 samples for class 24

Adding 40 samples for class 27

Adding 10 samples for class 29

Adding 40 samples for class 32

Adding 70 samples for class 37

Adding 40 samples for class 41

Adding 40 samples for class 42

For the augmented images, I applied a transform function which randomly rotates, translates, and shears the input image to avoid replicates of the input dataset. Credit goes to Vivek for this method. Adding more than 250 samples for each traffic sign did not produce any improvement in the results. The total number of training images before & after augmentation were 34799 & 35189 respectively. Here are some random images from the dataset after pre-processing and augmentation:

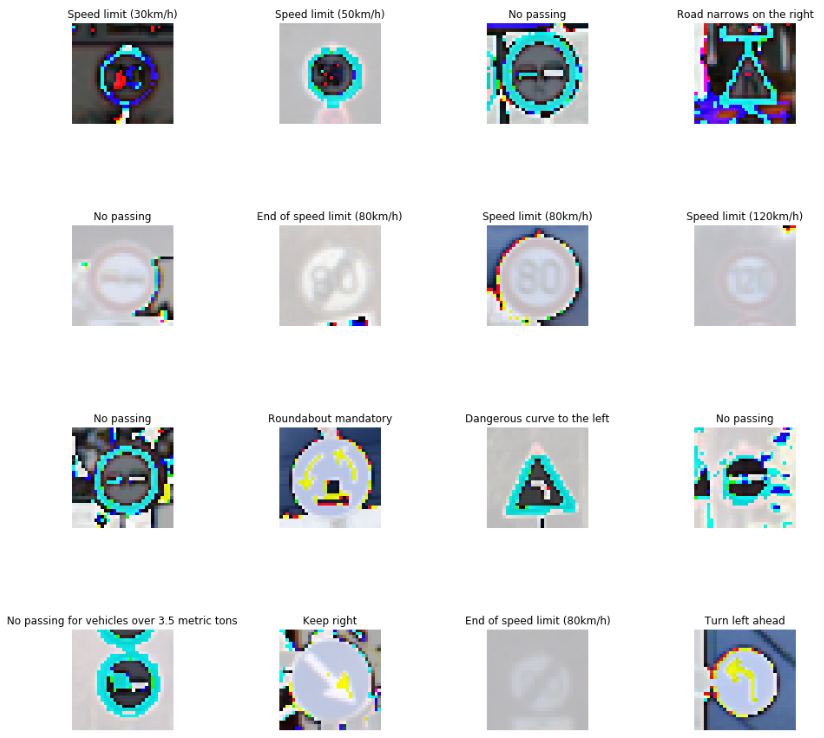


Figure 3: Random visualization of images in the training dataset after augmentation, pre-processing, & normalization

Develop a CNN model:

I chose the LeNet architecture (Fig. 4 below) and adapted it for this project to start with, since it is already a well established and proven model. The image below shows the number of layers, layer sizes, & connectivity of the model. The goal for this project is to ensure at least 93% in model accuracy with the validation dataset.

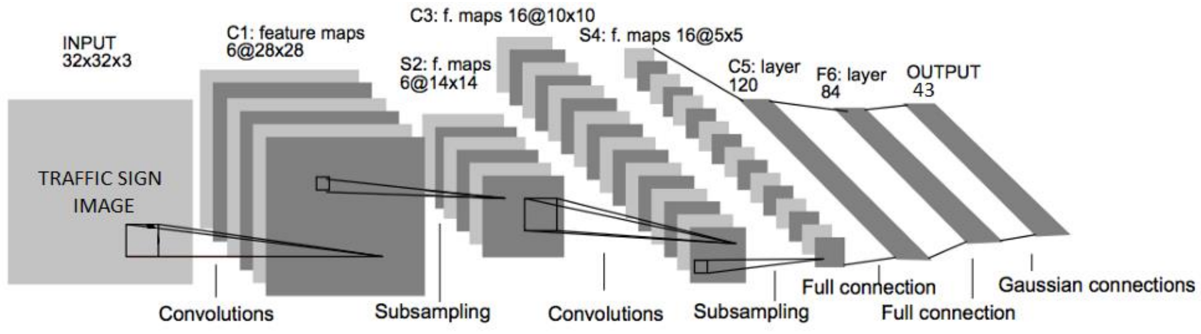


Figure 4: LeNet Model (Source: Yann LeCun)

Train and validate the model:

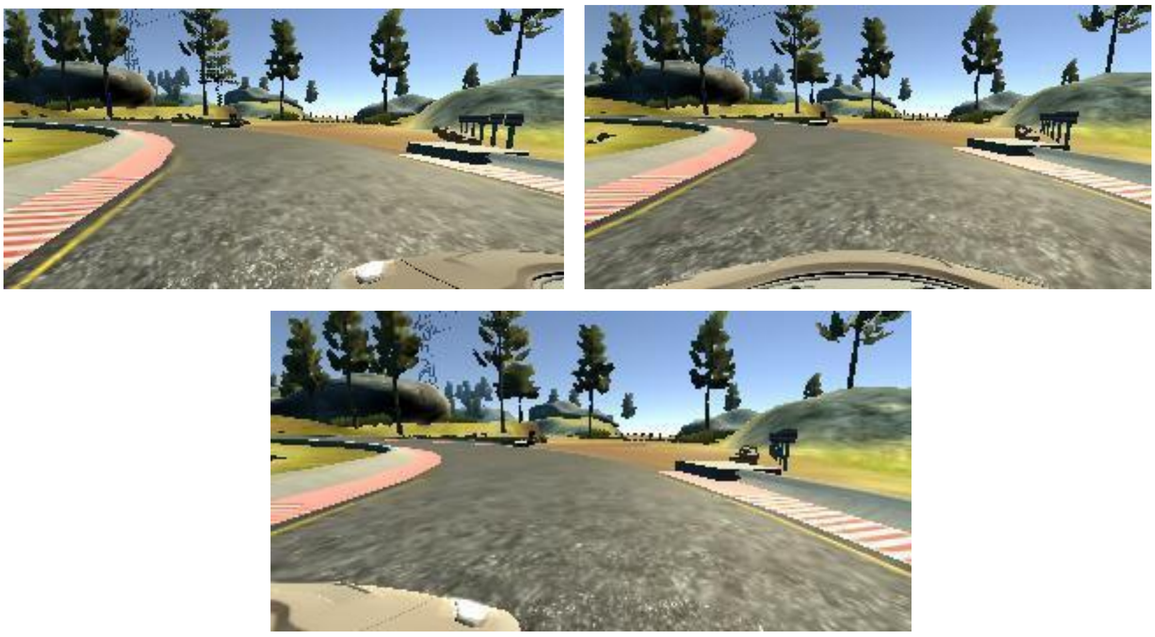
I started with the LeNet architecture and trained the model with the training dataset. To start with, I did not augment any additional images to the training set to establish a baseline. When I ran the model with the validation dataset, the results were not good: see the 1st column of Fig. 5 below. The training accuracy was quite high, but the validation accuracy was poor. A high accuracy on the training set but low accuracy on the validation set implies overfitting. To address this overfitting problem, there are a number of recommended approaches in the toolkit that I could experiment with:

CHAPTER 7

BEHAVIOURAL CLONING — MAKE A CAR DRIVE LIKE YOURSELF

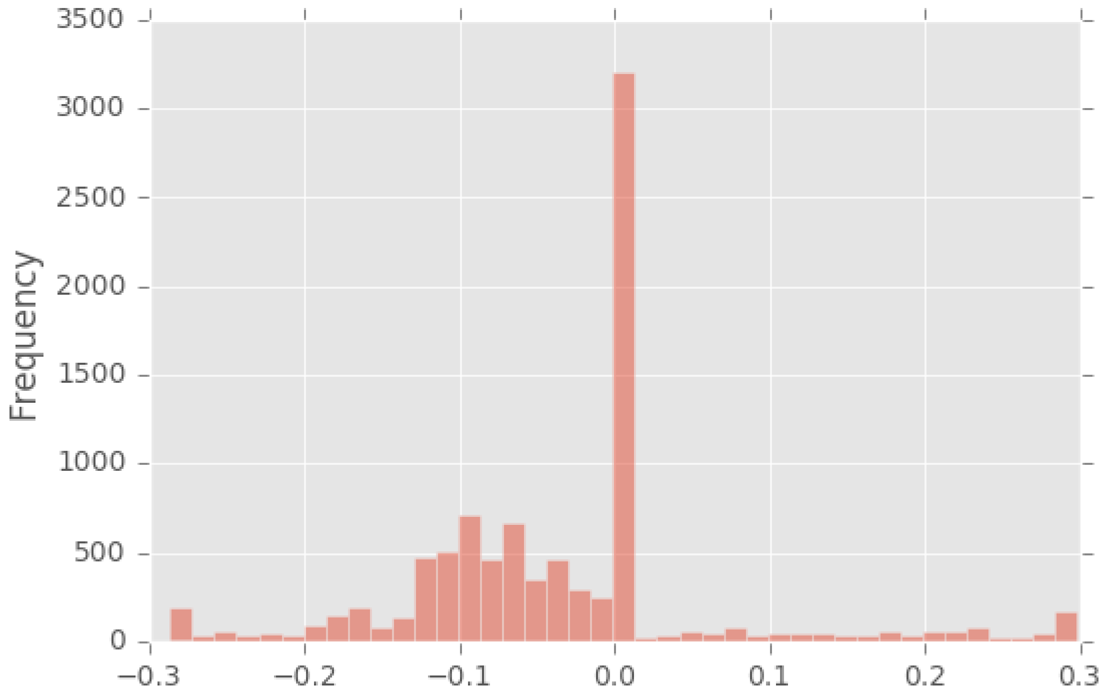
This is an article to provide my thoughts on an interesting project I did for the Udacity Self-Driving Car Nanodegree. The code and technical details can be found . The goal is to teach a Convolutional Neural Network (CNN) to drive a car in a simulator provided by Udacity. The car is equipped with three cameras that provide video streams and records the values of the steering angle, speed, throttle and brake. The steering angle is the only thing that needs to be predicted, but more advanced models might also want to predict throttle and brake. This turns out to be a regression task, which is very different from usual applications of CNNs for classification purposes.

I collected the training data by driving the car on the flat terrain training track. The performance of the CNN can then be checked by letting the car drive autonomously on the same track or ideally on a second track that is considerably more windy with steep hills and that should not be used for training. Below are some pictures from the different cameras on the car on the training track.



Unbalanced data

While driving the car under normal conditions the steering angle is very close to zero most of the time. This can clearly be seen in the raw training data. Below are the steering angles I recorded while driving the car around the track while staying as close as possible to the middle of the lane. This is all the data I collected for training the final model (~9000 images or driving 3–4 laps).



Histogram of steering angles: the range [-1,1] corresponds to the angle range ±25°

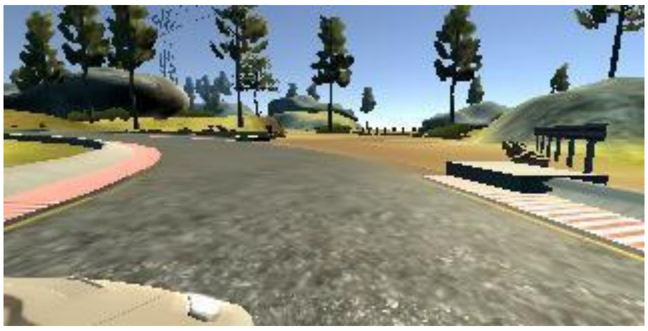
The left/right skew is due to driving the car around the track in one direction only and can be eliminated by flipping each recorded image and its corresponding steering angle. More troublesome is the bias to driving straight: the rare cases, when a large steering angle recorded are also the most important ones if the car is to stay on the road. One possible solution would be to let the car drift to the edge of the road and recover before a crash occurs. I tried this, but found it an unsatisfactory solution, because in that case the car still goes straight most of the time — direction “off the track” — with a few large steering angles sprinkled on top. As a result, a CNN trained on such data typically does not even complete the training track, unless the training data is ‘just right’. I got a CNN to drive the car around the training track this way, but the model failed on the test track.

Another solution would be to sample the extreme angle events more often than the small angles ones. However, since they are rare, a large amount of training data may need to be collected for the model to avoid overfitting.

Inspired by this post, I therefore decided to simulate *all* recovery events synthetically. For training I drove the car as smoothly as possible right in the middle of the road. The rationale behind this was always to record the ideal steering angle. Recovery events were then simulated by distorting and cropping the recorded camera images and adjusting the steering angle. By chaining image distortions, random brightness corrections and crops together a practically infinite number of training images could thereby be generated from the little training data I had gathered. This worked surprisingly well.

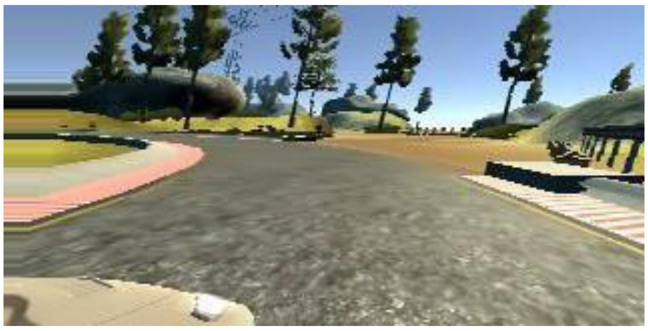
Data augmentation

I used images from all three cameras. Images taken from the side cameras are akin to parallel translations of the car. To account for being off-centre I adjusted the steering angle for images taken from the side cameras as follows: ignoring perspective distortions one could reason that if the side cameras are about 1.2 meters off-centre and the car is supposed to get back to the middle of the road within the next 20 meters the correction to the steering should be about 1.2/20 radians (using $\tan(\alpha)$~α). This turned out to be a quite powerful means to make the car avoid the sides of the road.



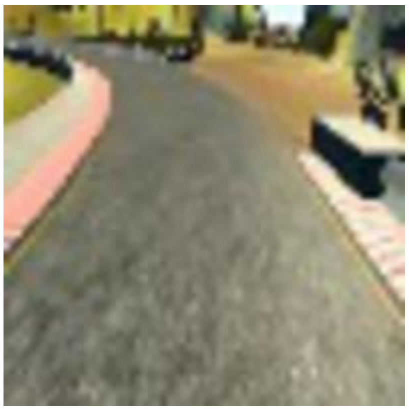
Random camera view. Steering angle: -2.8°, corrected:-3.4°

In the next stage each image got sheared horizontally. The pixels at the bottom of the image were held fixed while the top row was moved randomly to the left or right. The steering angle was changed proportionally to the shearing angle. This had the effect of making curvy road pieces appear just as often in the training set as straight parts.



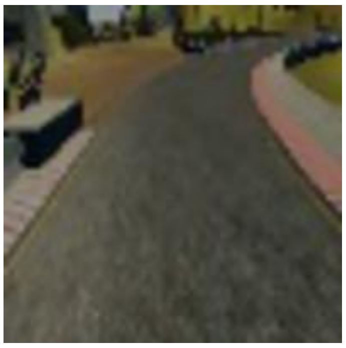
Random shearing: corrected steering angle: -1.0°

So far all transformations had been performed on the full 320x160 pixel images coming from the cameras. In the next stage I chose a window of 280x76 pixels that eliminated the bonnet from the bottom of the image and cropped off the part above the horizon in flat terrain at the top. For each image the location of this window was displaced randomly from the centre along the x-and y-axes by up to ±20 and ±10 pixels, respectively. The steering angle was changed proportionally to the lateral displacement of the crop. Thereby, also images of the car being in hilly terrain were simulated and a greater variety of images than available from the side cameras could be obtained. The result was then resized to 64x64 pixels.



Random crop: corrected steering angle: -2.1°

Finally, each image was randomly flipped (horizontally) with equal probability in order to make left and right turns appear as frequently. Also brightness was randomly adjusted.



Random brightness and flip: corrected steering angle: 2.1°

Chaining all these transformations can be done efficiently in batches that get generated on the fly during training from the recorded images.

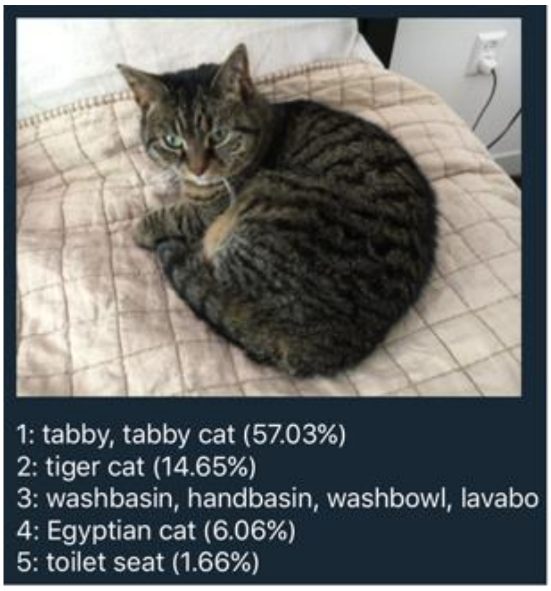
Model architecture and training

The CNN I chose is a pretty standard CNN consisting of 4 convolutional layers with ReLU activations, followed by two fully connected layers with dropout regularization. Finally a single neuron formed the output that predicted the steering angle. One noteworthy thing is the absence of pooling layers. The rationale behind avoiding pooling layers was that pooling layers make the output of a CNN to some degree invariant to shifts of the input, which is desirable for classification tasks, but counterproductive for keeping a car in the middle of the road. I trained the network on an Ubuntu 16.04 system using an NVIDIA GTX 1080 GPU. For any given set of hyperparameters the loss typically stopped decreasing after a few epochs (200000 images each).

CHAPTER 8

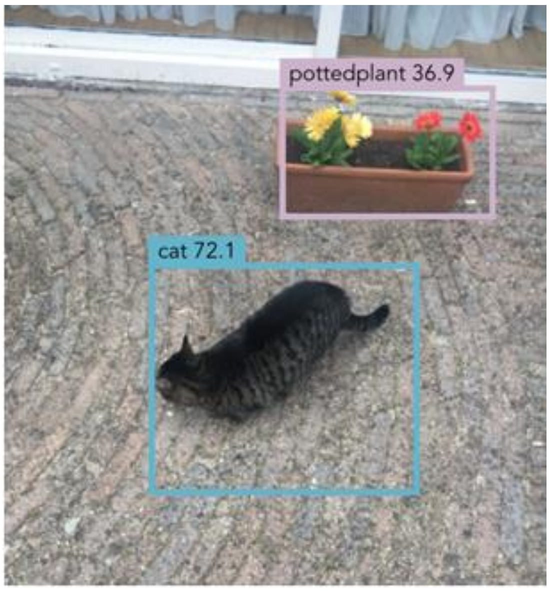
YOLO

Detection is a more complex problem than classification, which can also recognize objects but doesn't tell you exactly where the object is located in the image — and it won't work for images that contain more than one object.



```
1: tabby, tabby cat (57.03%)
2: tiger cat (14.65%)
3: washbasin, handbasin, washbowl, lavabo
4: Egyptian cat (6.06%)
5: toilet seat (1.66%)
```

Classification



Object detection

YOLO is a clever neural network for doing object detection in real-time.

In this blog post I'll describe what it took to get the "tiny" version of YOLOv2 running on iOS using Metal Performance Shaders.

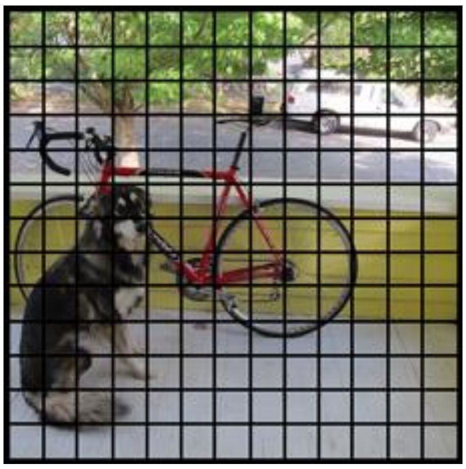
How YOLO works

You can take a classifier like VGG Net or Inception and turn it into an object detector by sliding a small window across the image. At each step you run the classifier to get a prediction of what sort of object is inside the current window. Using a sliding window gives several hundred or thousand predictions for that image, but you only keep the ones the classifier is the most certain about.

This approach works but it's obviously going to be very slow, since you need to run the classifier many times. A slightly more efficient approach is to first predict which parts of the

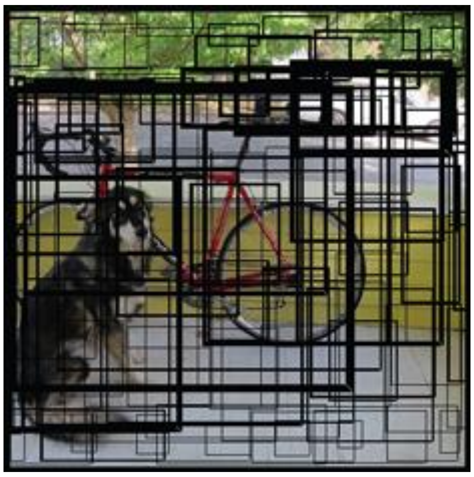
image contain interesting information — so-called region proposals — and then run the classifier only on these regions. The classifier has to do less work than with the sliding windows but still gets run many times over.
YOLO takes a completely different approach. It's not a traditional classifier that is repurposed to be an object detector. YOLO actually looks at the image just once (hence its name: You Only Look Once) but in a clever way.

YOLO divides up the image into a grid of 13 by 13 cells:



Each of these cells is responsible for predicting 5 bounding boxes. A bounding box describes the rectangle that encloses an object.

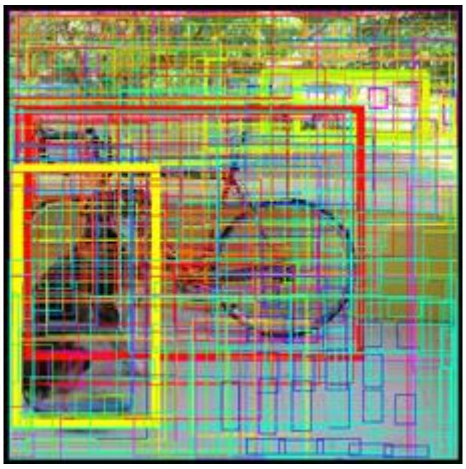
YOLO also outputs a confidence score that tells us how certain it is that the predicted bounding box actually encloses some object. This score doesn't say anything about what kind of object is in the box, just if the shape of the box is any good.
The predicted bounding boxes may look something like the following (the higher the confidence score, the fatter the box is drawn):



For each bounding box, the cell also predicts a class. This works just like a classifier: it gives a probability distribution over all the possible classes. The version of YOLO we're using is trained on the PASCAL VOC dataset, which can detect 20 different classes such as:

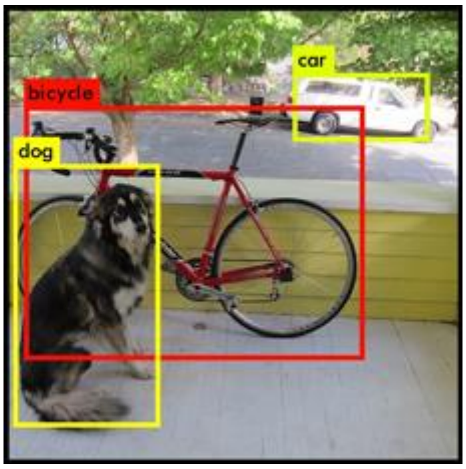
- bicycle
- boat
- car
- cat
- dog
- person
- and so on…

The confidence score for the bounding box and the class prediction are combined into one final score that tells us the probability that this bounding box contains a specific type of object. For example, the big fat yellow box on the left is 85% sure it contains the object "dog":



Since there are 13×13 = 169 grid cells and each cell predicts 5 bounding boxes, we end up with 845 bounding boxes in total. It turns out that most of these boxes will have very low confidence scores, so we only keep the boxes whose final score is 30% or more (you can change this threshold depending on how accurate you want the detector to be).

The final prediction is then:



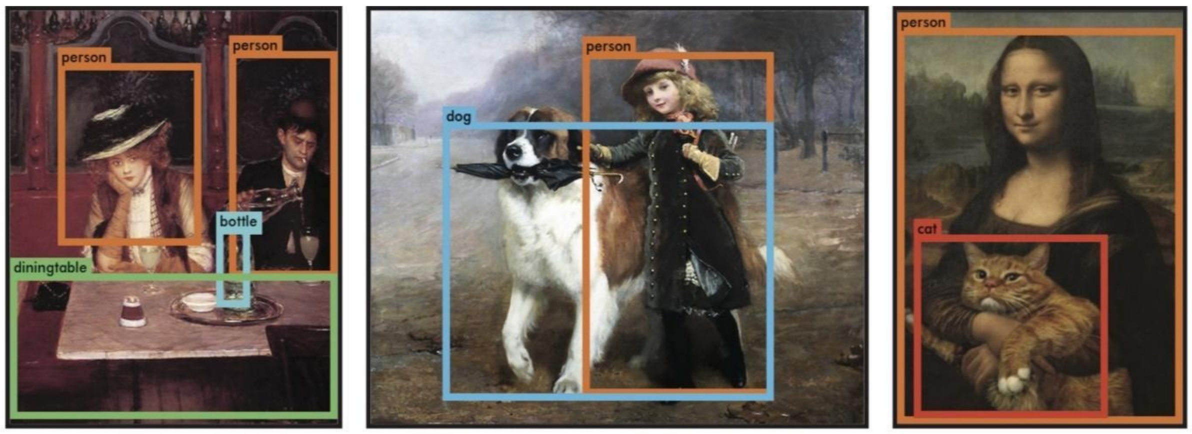
From the 845 total bounding boxes we only kept these three because they gave the best results. But note that even though there were 845 separate predictions, they were all made at the same time — the neural network just ran once. And that's why YOLO is so powerful and fast.

Benefits of YOLO

- Fast. Good for real-time processing.

- Predictions (object locations and classes) are made from one single network. Can be trained end-to-end to improve accuracy.

- YOLO is more generalized. It outperforms other methods when generalizing from natural images to other domains like artwork.

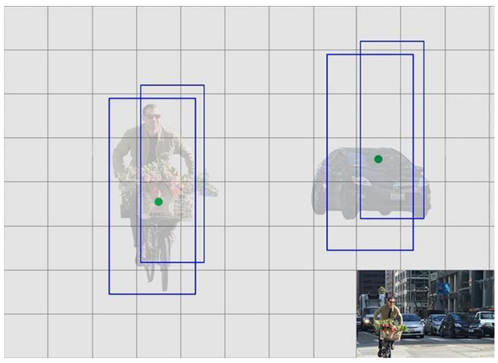


- Region proposal methods limit the classifier to the specific region. YOLO accesses to the whole image in predicting boundaries. With the additional context, YOLO demonstrates fewer false positives in background areas.

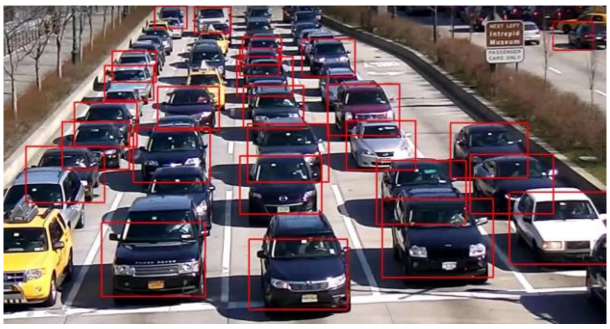
- YOLO detects one object per grid cell. It enforces spatial diversity in making predictions.

Convolutional with Anchor Boxes

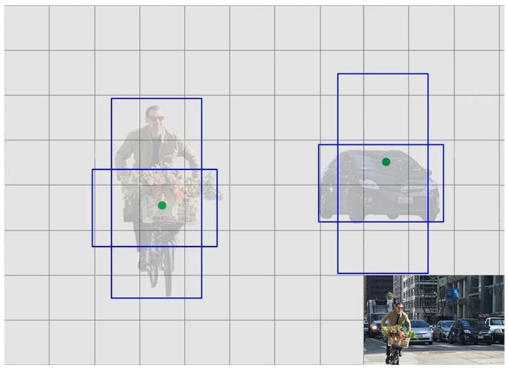
As indicated in the YOLO paper, the early training is susceptible to unstable gradients. Initially, YOLO makes arbitrary guesses on the boundary boxes. These guesses may work well for some objects but badly for others resulting in steep gradient changes. In early training, predictions are fighting with each other on what shapes to specialize on.



In the real-life domain, the boundary boxes are not arbitrary. Cars have very similar shapes and pedestrians have an approximate aspect ratio of 0.41.

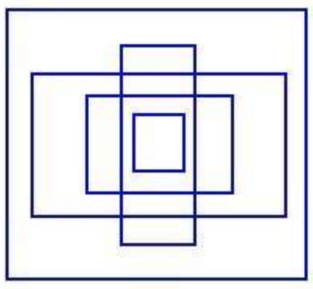


Since we only need one guess to be right, the initial training will be more stable if we start with diverse guesses that are common for real-life objects.



More diverse predictions

For example, we can create 5 **anchor** boxes with the following shapes.



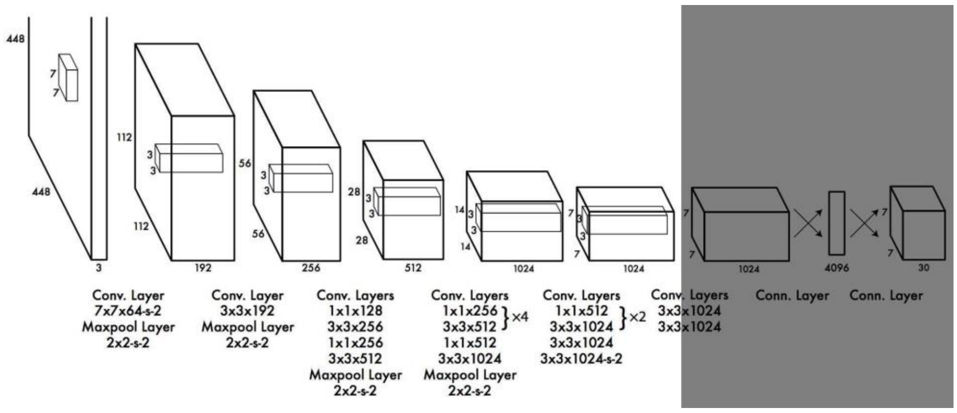
5 anchor boxes

Instead of predicting 5 arbitrary boundary boxes, we predict offsets to each of the anchor boxes above. If we **constrain** the offset values, we can maintain the diversity of the predictions and have each prediction focuses on a specific shape. So the initial training will be more stable.

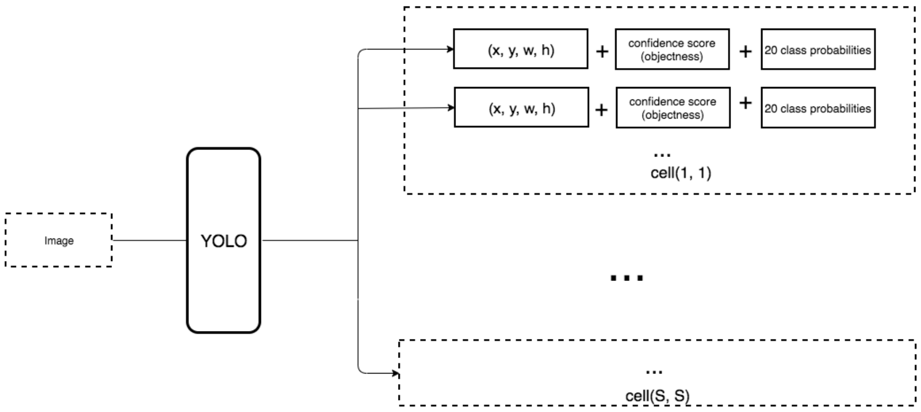
In the paper, anchors are also called **priors**.

Here are the changes we make to the network:

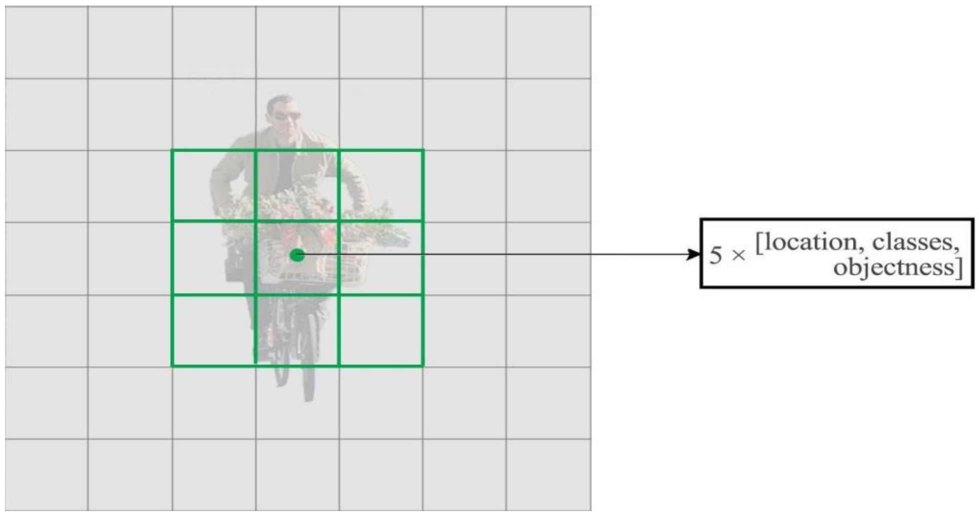
- Remove the fully connected layers responsible for predicting the boundary box.



- We move the class prediction from the cell level to the boundary box level. Now, each prediction includes 4 parameters for the boundary box, 1 box confidence score (objectness) and 20 class probabilities. i.e. 5 boundary boxes with 25 parameters: 125 parameters per grid cell. Same as YOLO, the objectness prediction still predicts the IOU of the ground truth and the proposed box.

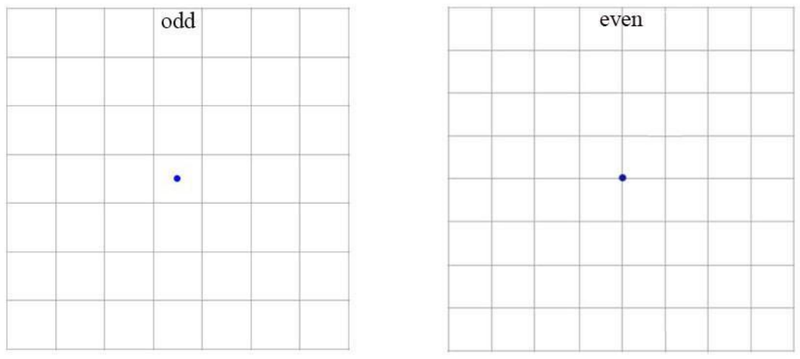


- To generate predictions with a shape of 7 × 7 × 125, we replace the last convolution layer with three 3 × 3 convolutional layers each outputting 1024 output channels. Then we apply a final 1 × 1 convolutional layer to convert the 7 × 7 × 1024 output into 7 × 7 × 125. (See the section on DarkNet for the details.)



Using convolution filters to make predictions.

- Change the input image size from 448 × 448 to 416 × 416. This creates an odd number spatial dimension (7×7 v.s. 8×8 grid cell). The center of a picture is often occupied by a large object. With an odd number grid cell, it is more certain on where the object belongs.



- Remove one pooling layer to make the spatial output of the network to **13×13** (instead of 7×7).

Anchor boxes decrease mAP slightly from 69.5 to 69.2 but the recall improves from 81% to 88%. i.e. even the accuracy is slightly decreased but it increases the chances of detecting all the ground truth objects.

CHAPTERS 9

CONCLUSION

Driverless cars appear to be an important next step in transportation technology. The challenge for driverless car designers is to produce control systems capable of analyzing sensory data in order to provide accurate detection of other vehicles and the road ahead. Developments in autonomous cars is continuing and the software in the car is continuing to be updated.

Though it all started from a driverless thought to radio frequency, cameras, sensors, more semi-autonomous features will come up, thus reducing congestion, increasing safety with faster reactions and fewer errors. If the people's thought hasn't changed about the self-driving cars being safe, these cars are already safe and are becoming safer. Only if they believe and give a try to technology, they get to enjoy the luxury of computerized driving.