Al-Azhar University

Faculty of Engineering

Electronics & Communications Department

Self-Driving Car

Supervised by: Dr. Ahmed Salah.

Team Members

Mosaab Muhammad (mosaabmohammed@outlook.com)

Umar Sayed (umar.s.hamouda@gmail.com)

Mostafa Saleh (mostafatoema96@gmail.com)

Ahmed Yasser (ahmed10_17@yahoo.com)

Mahmoud Sabry (Mahmoudsabry996@gmail.com)

Acknowledgment

- This work won't be done without the guidance of Allah and the support of our parents and the guidance of the senior people.

- We would like to express our immense gratitude to Dr. Ahmed Salah for his constant support and motivation along the way that has encouraged us to come up with this project.

Abstract

- For decades self-driving cars have been a sci-fi dream and now they're becoming a reality. By 2020, it's estimated that 10 million cars with autonomous features will be on the road, Google; the biggest network has started working on the self-driving cars since 2010 and still developing new features and changes to give a whole new-brand level to the automated vehicles.

- Because this project is trendy, a lot of companies wants to take a leap in this field to control the market, such as in the image:

- The aim and the goal for self-driving cars is that it could eliminate all road related accidents. While this would be a tremendous thing for humanity.

- So for that reason, we decided to make our project about self-driving car.

- The function of self-driving car boils down into 4 functions:

1) Finding Lane Line: by making the car be able to find the lane line of the road using image processing and computer vision through the camera.

2) Traffic-Sign detection: by making the car be able to recognize the traffic sign and make action based on that decision.

3) Behavioral Cloning: by making the level of driving looks like there is someone driving the car.

4) Advanced Lane Line Detection: by making the car recognize the curvature of the lane lines and make better decisions.

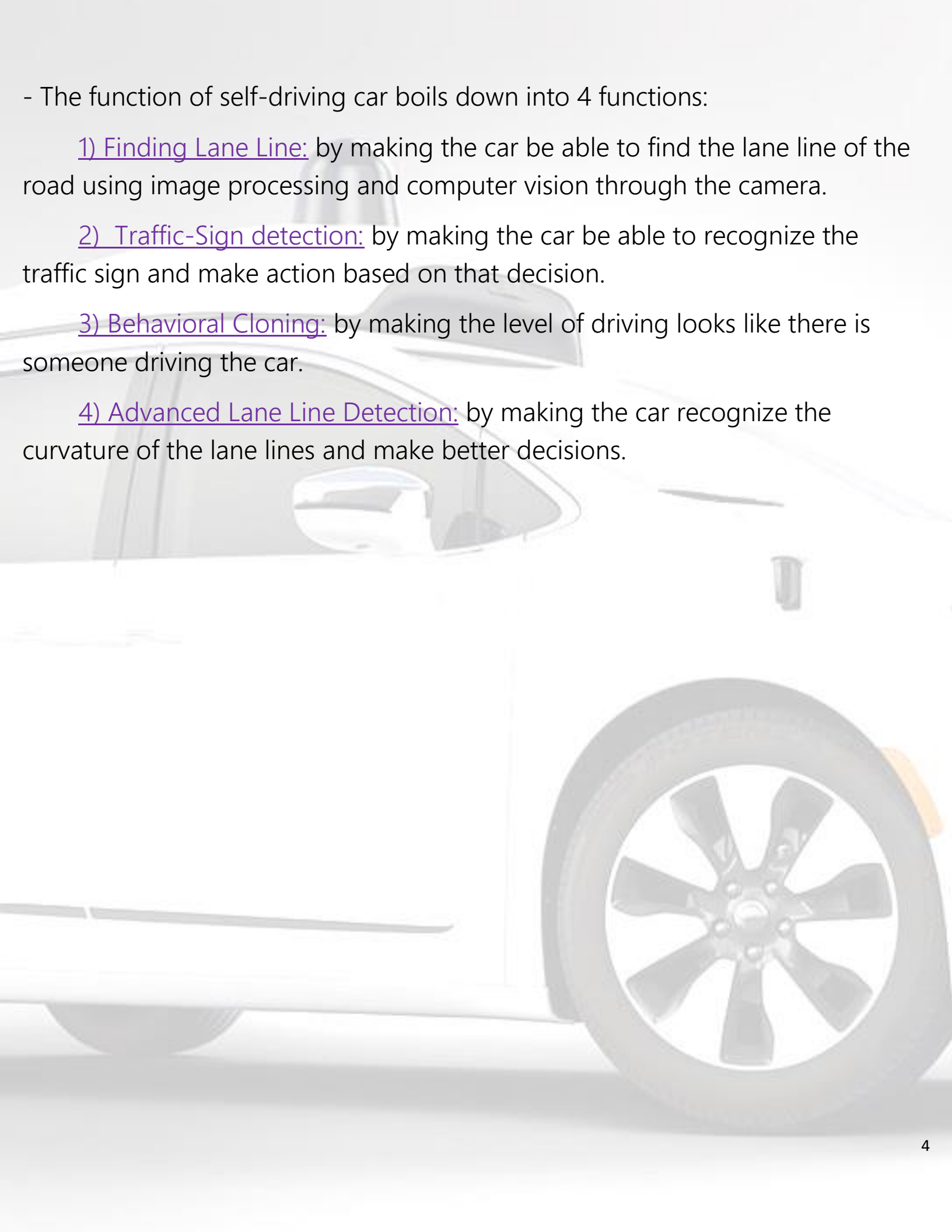
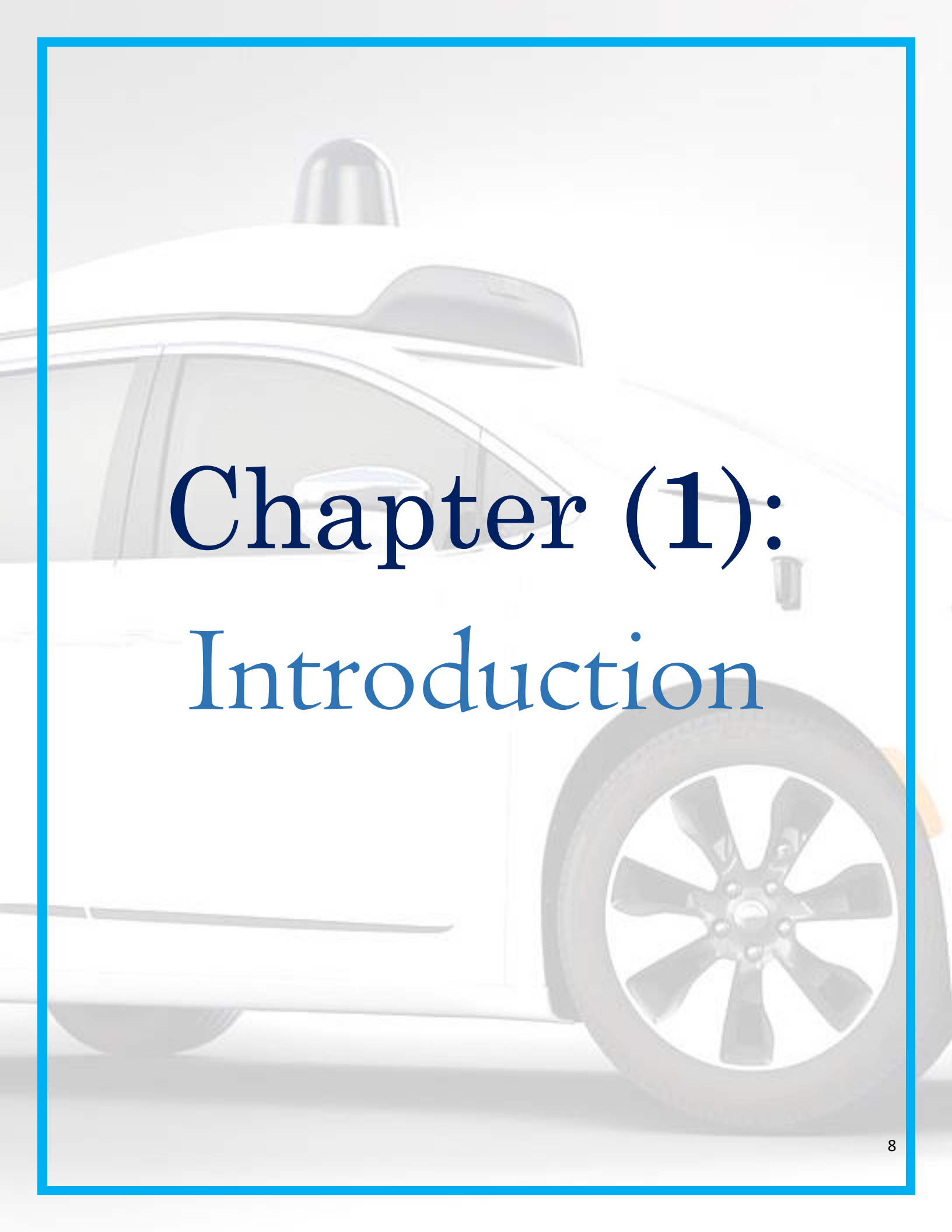


Table of Content



Chapter (1):
Introduction

The modern vehicle features increase not only the driver's comfort by providing automated features, but these functions are also responsible for the reduction of accidents. In common with these functions is that they depend heavily on perceived autonomous miniature vehicles need to demonstrate the reliable capability to follow their lane, to overtake vehicles blocking on their lane, to handle intersections safely according to the right-of-way traffic rule, and to park the vehicle at a sideways parking strips as fast as possible.

Though a truly driverless car is most likely still years away from being available to consumers, they are closer than many people think. Current estimates predict that by 2025 the world will see over 600,000 self-driving cars on the road, and by 2035 that number will jump to almost 21 million. Trials of self-driving car services have actually-begun in some cities in the United States. And even though fully self-driving cars are not on the market yet, current technology allows vehicles to be more autonomous than ever before.

1.1 What is self-driving car?

Self-driving Cars may seem like the logical step in the evolution of personal transportation. They are conceptually simple, easy to understand and in many ways, very desirable. But they are not cars in the usual sense.

- These cars are complex systems. They are the most complex robots that humans have built.

- A self-driving car (driverless, autonomous, robotic car) is a vehicle that is capable of sensing its environment and navigating without human input.

- Self-driving cars can detect environments using a variety of techniques such as radar, GPS, computer vision ,intricate systems of cameras and interconnected communication between vehicles, some models of cars now offer features that can advanced control systems interpret sensory information to identify appropriate navigational paths, as well as obstacles and relevant signage. Self-driving cars have control systems that are capable of analyzing sensory data to distinguish between different cars on the road. This is very useful in planning a path to the desired destination.

1.1.2 Autonomous Car Safety and Challenges:

- Autonomous cars must learn to identify countless objects in the vehicle's path, from branches and litter to animals and people. Other challenges on the road are tunnels that interfere with Global Positioning Systems (GPS), construction projects that cause lane changes, or complex decisions, like where to stop to allow emergency vehicles to pass.

- The systems need to make instantaneous decisions on when to slow down, swerve or continue acceleration normally. This is a continuing challenge for developers and there are reports of self-driving cars

hesitating and swerving unnecessarily when objects are detected in or near the roadways.

1.1.3 Levels of autonomy in self-driving cars:

The U.S. National Highway Traffic Safety Administration (NHTSA) lays out six levels of automation, beginning with zero, where humans do the driving, through driver assistance technologies up to fully autonomous cars. Here are the five levels that follow zero automation:

1. **Level 1**: Advanced driver assistance system (ADAS) aid the human driver with either steering, braking or accelerating, though not simultaneously. ADAS includes rearview cameras and features like a vibrating seat warning to alert drivers when they drift out of the traveling lane.
2. **Level 2**: An ADAS that can steer and either brake or accelerate simultaneously while the driver remains fully aware behind the wheel and continues to act as the driver.

3. **Level 3**: An automated driving system (ADS) can perform all driving tasks under certain circumstances, such as parking the car. In these circumstances, the human driver must be ready to re-take control and is still required to be the main driver of the vehicle.

4. **Level 4**: An ADS is able to perform all driving tasks and monitor the driving environment in certain circumstances. In

those circumstances, the ADS are reliable enough that the human driver needn't pay attention.

5. **Level 5**: The vehicle's ADS act as a virtual chauffeur and do all the driving in all circumstances. The human occupants are passengers and are never expected to drive the vehicle.

1.2 Algorithm of Self-driving Car:

The self-driving is the process of capturing the images through web camera that provides machine vision capabilities to the system as well as a plethora of sensors and microcontrollers which are placed as an antenna at the top of the vehicle.

1.2.1 Making Decision:

- Fully autonomous cars can make thousands of decisions for every mile traveled. They need to do so correctly and consistently. Currently, AV designers use a few primary methods to keep their cars on the right path.

1. **Neural networks**. To identify specific scenarios and make suitable decisions, today's decision-making systems mainly employ neural networks. The complex nature of these networks can, however, make it difficult to understand the root causes or logic of certain decisions.
2. **Rule-based decision making**. Engineers come up with all possible combinations of if-then rules and then program vehicles accordingly in rule-based approaches. The significant time and effort required, as well as the probable inability to include every potential

case, make this approach unfeasible.

3. **Hybrid approach**. Many experts view a hybrid approach that employs both neural networks and rule-based programming as the best solution. Developers can resolve the inherent complexity of neural networks by introducing redundancy—specific neural networks for individual processes connected by a centralized neural network. If-then rules then supplement this approach.

- The hybrid approach, especially combined with statistical-inference models, is the most popular one today.

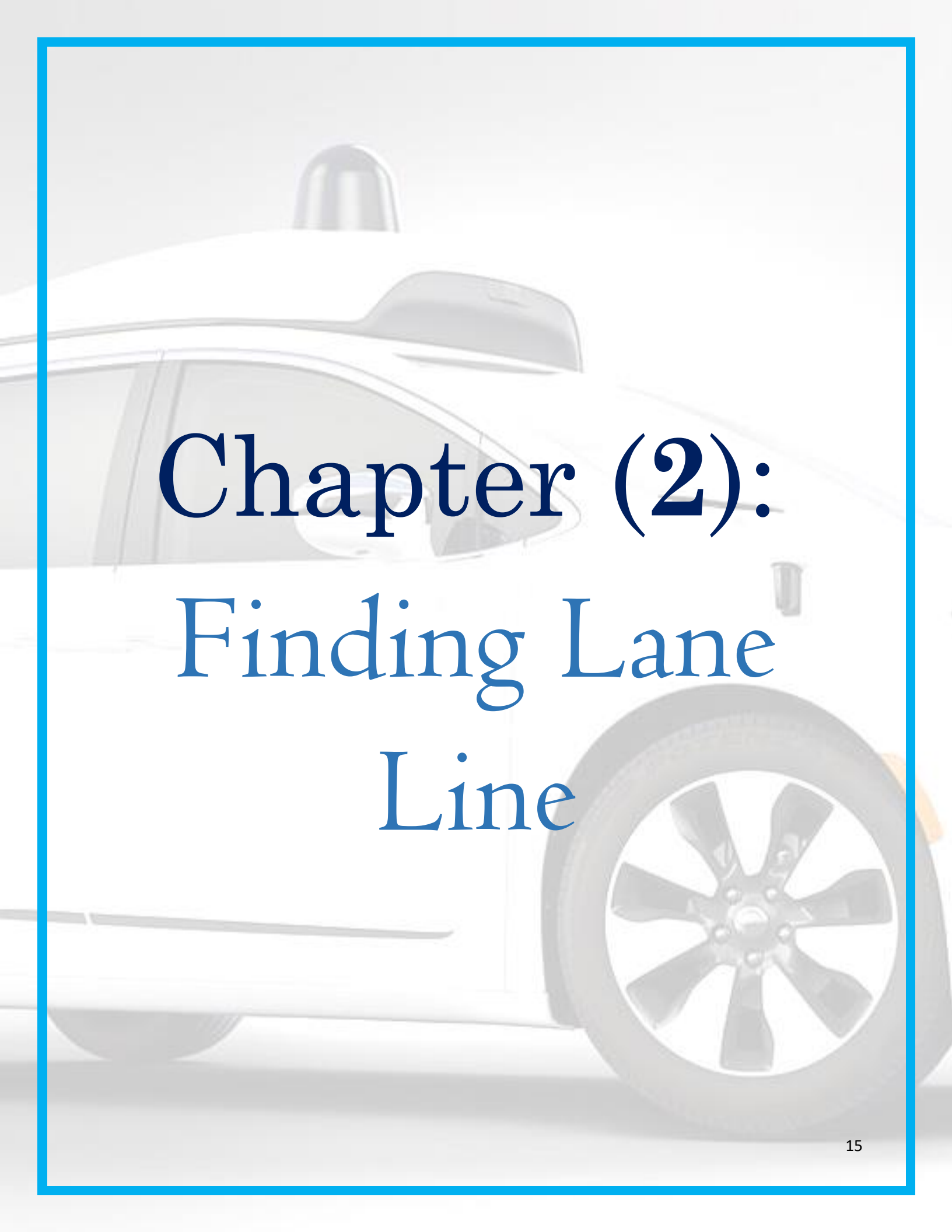
1.3 Potential Advantages:

- being able to get things done while in traffic or on the road.
- Increase road capacity.
- Fewer traffic collisions. Experts estimate 300,000 lives can be saved Per decade.
- Higher speed limits.
- Reduction in traffic police.
- Removal of limitations on drivers — age and sobriety won't be an Issue.

1.4 Potential Obstacles:

- Liability for damage.
- Resistance by individuals to forfeit control of their cars.

● Software reliability.
● Implementation of legal framework and establishment of government regulations for self-driving cars.
● Drivers being inexperienced if situations arose requiring manual driving.
● Loss of driving-related jobs.
● Loss of privacy.



Chapter (2):

Finding Lane Line

2.1 Setting up the problem:

- We as humans, use our eyes to know how to drive and to see the world around us.

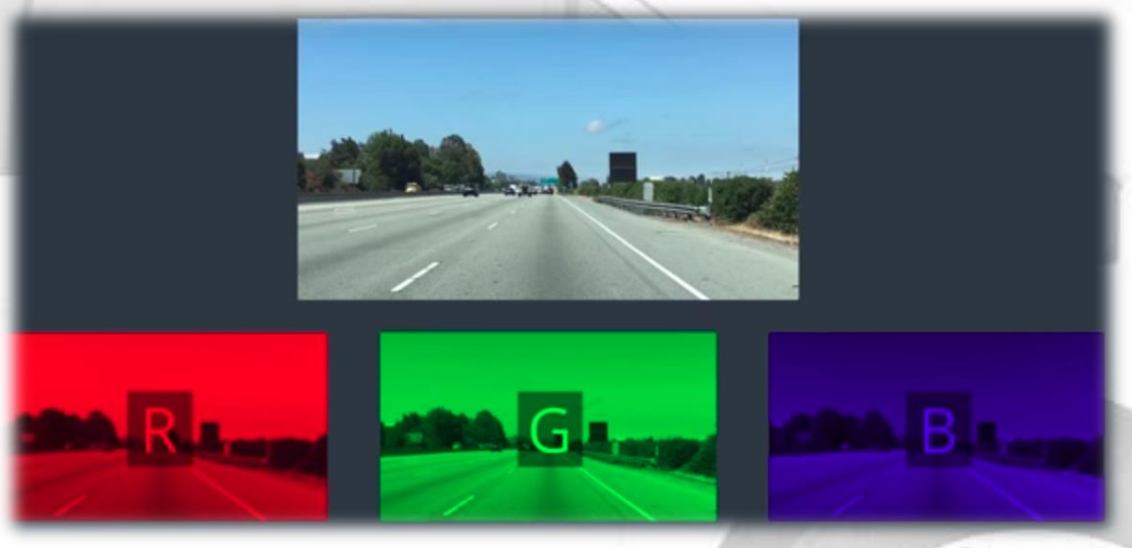
- But to make this happens in a car, we use cameras and other sensors to achieve a similar function.

- Then our goal here, is to teach a car how to see the lane line to be able to drive.

2.2 Color Selection:

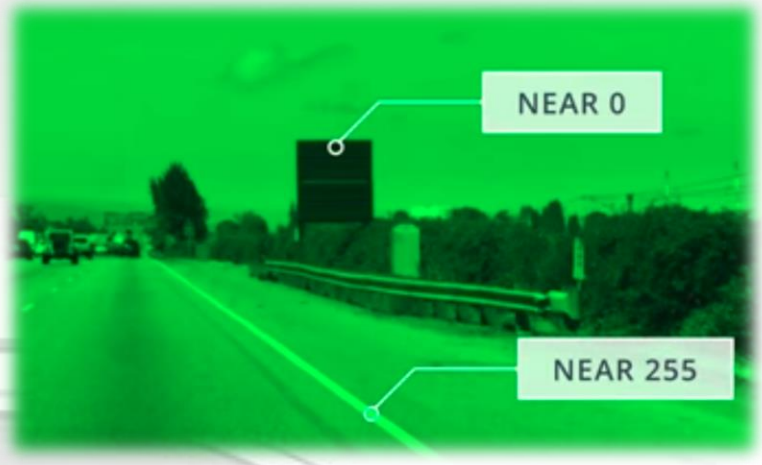
- First, we are going to start with a question? How to select the color of the lane line (white) from an image?.

- To answer this question, first we need to know the layers of a colored image.



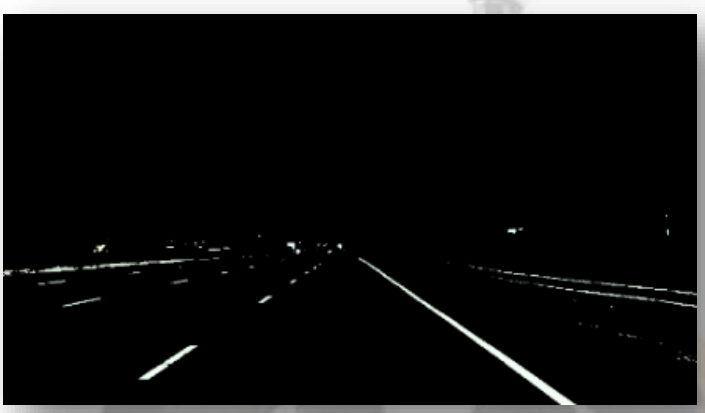
- Each colored image consists of 3 layers (Red, Green, Blue), each of these layer called “Color Channel”.

- Each of these color channels contains pixels whose values range from zero to 255. Where **0 -> Darkest** and 255 -> Brightest.



- But here's the question, what representation of pure white in the 3 layers White -> [255, 255, 255].

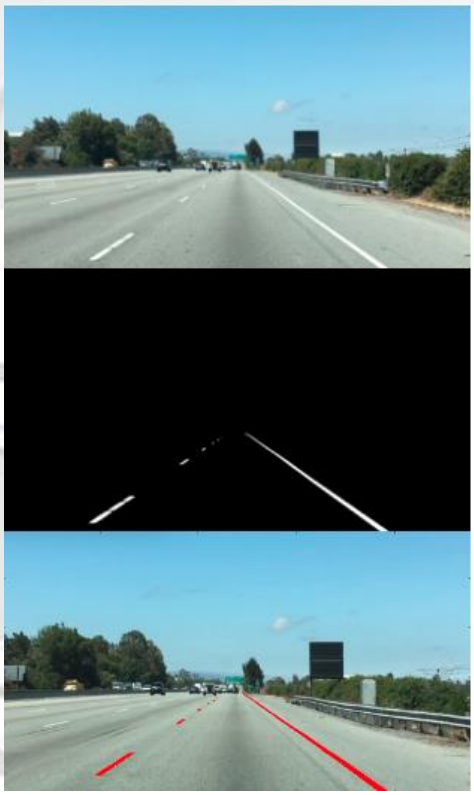
2.3 Color Selection Code Example:



- You can check the code from (Finding Lane Line Folder > Color Selection Code Example.py)
- And here's the resultant image:

2.4 Color & Region Selection:

- You can see the code that achieve the following image in (Color & Region Combined).



2.5 What is Computer Vision? (Canny Edge Detection):

- The name Canny refers to John F. Canny who developed this algorithm in 1986.

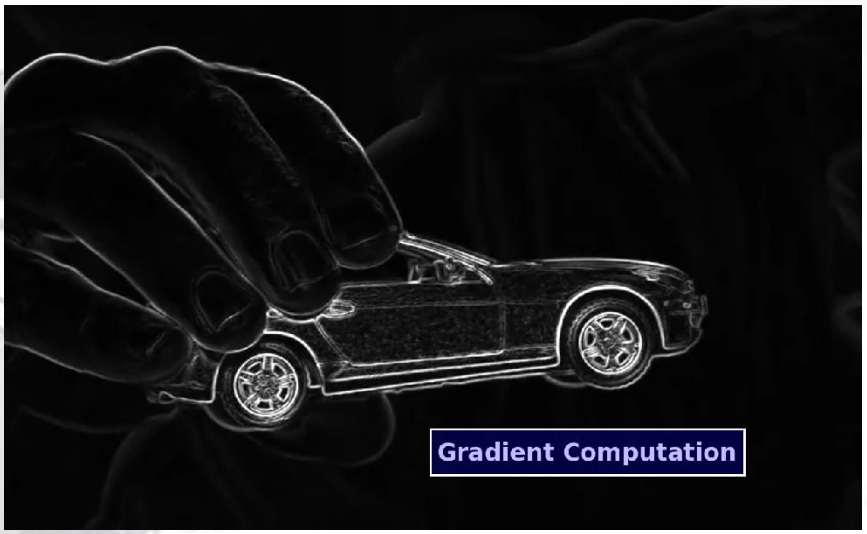
- With edge detection, the goal is identify the boundaries of an object in an image. So to do that we do the following:

  - Convert the image to Gray Scale. (Left image)

  - Compute the gradient of the resultant image, where the brightness of each pixel corresponds to the strength of the gradient at that point.

  - Then, we're going to find edges by tracing out the pixels that follows the strongest gradients.

- By identifying edges, we can more easily detect objects by their shape.



- Now, it's the time to see Canny function in OpenCV:

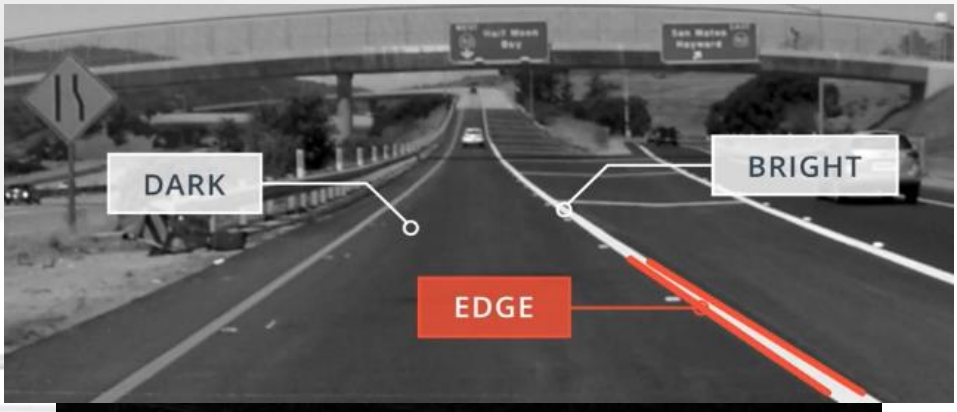
- gray: is the image in gray scale.

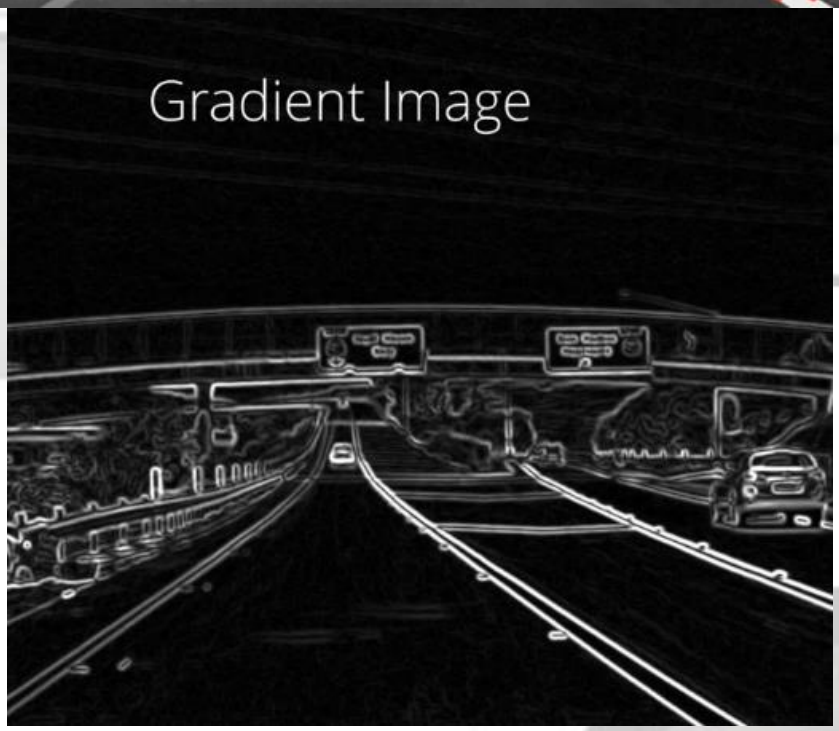
- low_threshold & high_threshold: how different the values are in adjacent pixels in the image; really just the strength of the gradient.

```
edges = cv2.Canny(gray,
low_threshold, high_threshold)
```

- Now, how we compute gradient on an image?

- Rapid

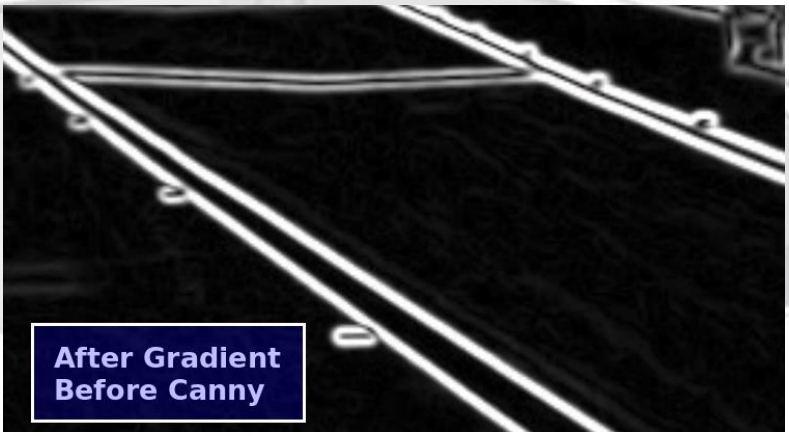


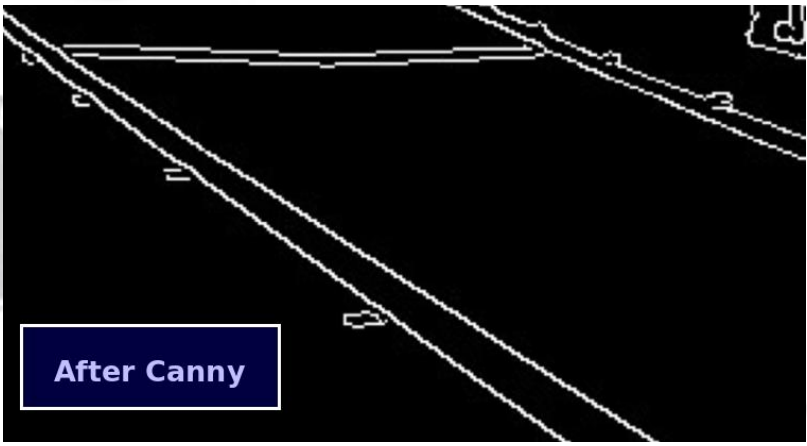


changes in brightness are where we find the edges.

- Our image is just a function of X and Y which means we can perform function on it, for example, we can take its derivative which is just a measure of change of this function.
- A small derivative, a small change. A big derivative, a big change.
- To perform derivative on an image, we'll take it with respect to X and Y simultaneously.
- Gradient on an image means we're measuring how fast pixel values are changing at each point.

- And as you can see, the direction they're changing most rapidly is in the edges!
- Computing the gradient gives us thick edges, with Canny algorithm, we will thin out these edges to find just the individual pixels that follow the strongest gradients.
- Then, we'll extend those strong edges to include pixels all the way down to a lower threshold that we defined when calling the Canny function.





NOTES:

- Canny algorithm will only include pixels between low_threshold and high_threshold and the other range will be blocked.

- Gray image leaves us with 2^8 = 265 possible values of each pixel; this range implies that derivatives will be on the scale of tens and hundreds. So, a reasonable range for your threshold parameters would also be in the tens and hundreds.

- John Canny himself recommended a low to high ratio of 1:2 or 1:3.

- Before applying Canny we need to apply Gaussian smoothing to suppress noise and spurious gradients by averaging.

- Canny algorithm apply Gaussian smoothing internally, but we do it manually to not get a different result by applying further smoothing.

- Kernel_size parameter in BlurGaussian could be an odd number. A larger Kernel_size implies averaging, or smoothing, over a larger area.

- See the resultant code in Canny function.ipynd

2.6 Hough Space:

- Now, after detecting the edges, we need to connect the dots of the image to form a line which supposed to represent the lane of the road.

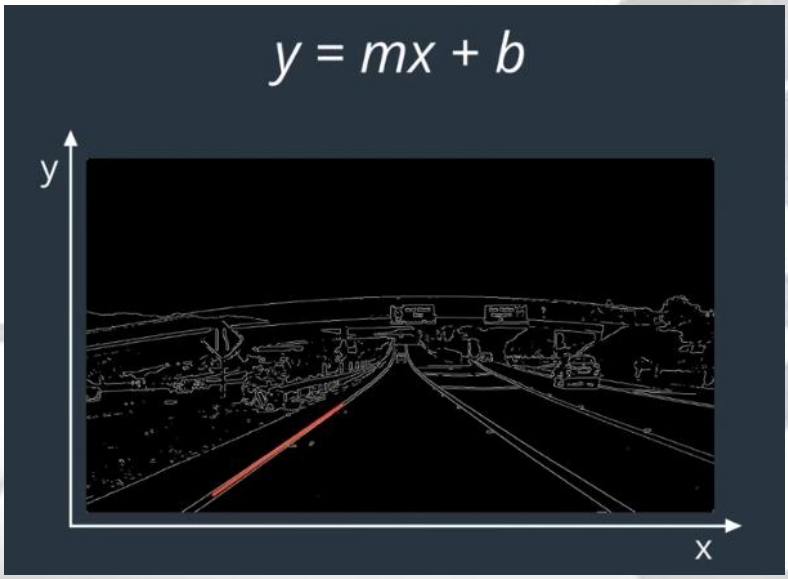
- We'll face a problem with a segment of unconnected lines but if we connect it as line, it will be a complete line.

- To do so, by transforming from image space to Hough Space.

- Where in Image Space the equation states as y = m0 * x + b0

- Where m0 is the slope and b0 is the value of y when x = 0

- In Hough Space, the axes are m and b as the image illustrates.



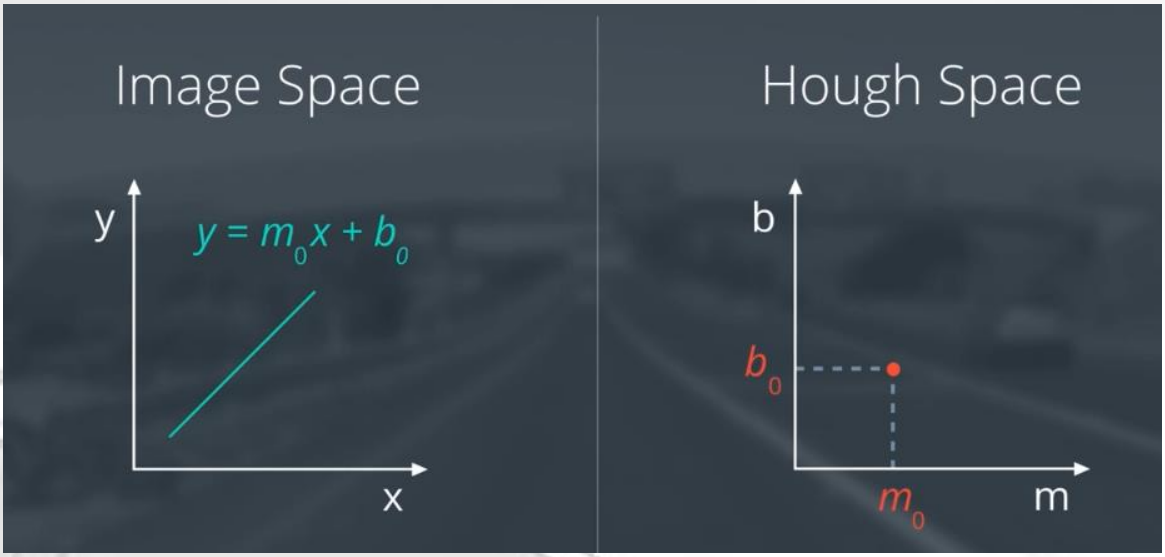


Image Space
y
$y = m_0 x + b_0$
x
Hough Space
b
b_0
m_0
m


Image Space
y
x
Hough Space
b
b_0
m
m

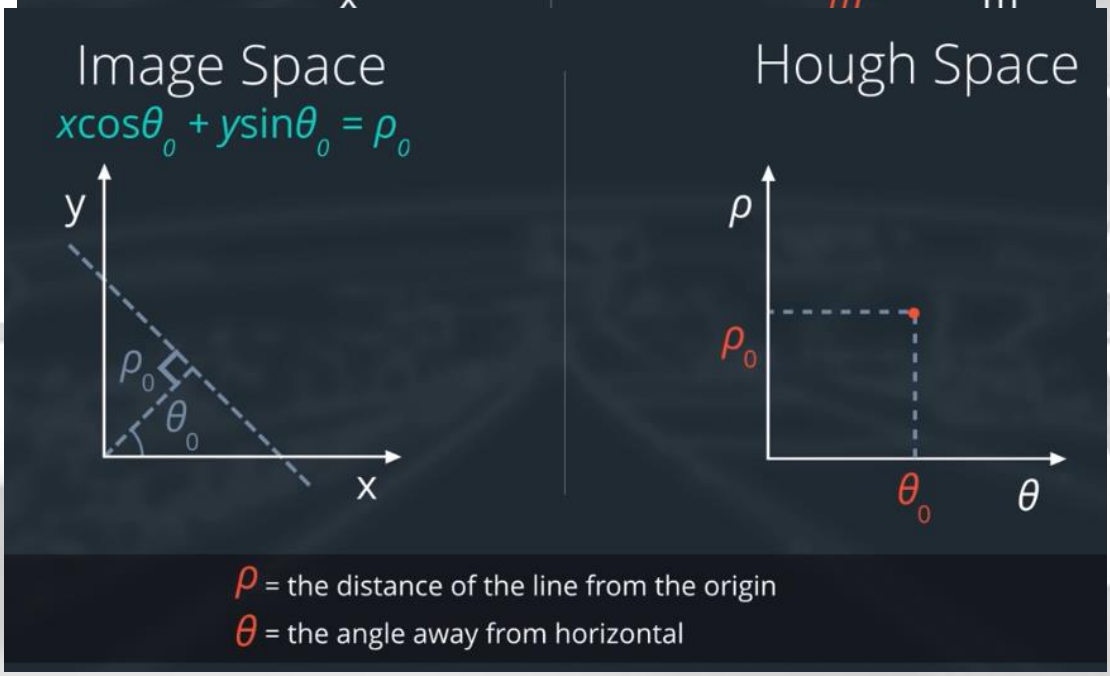
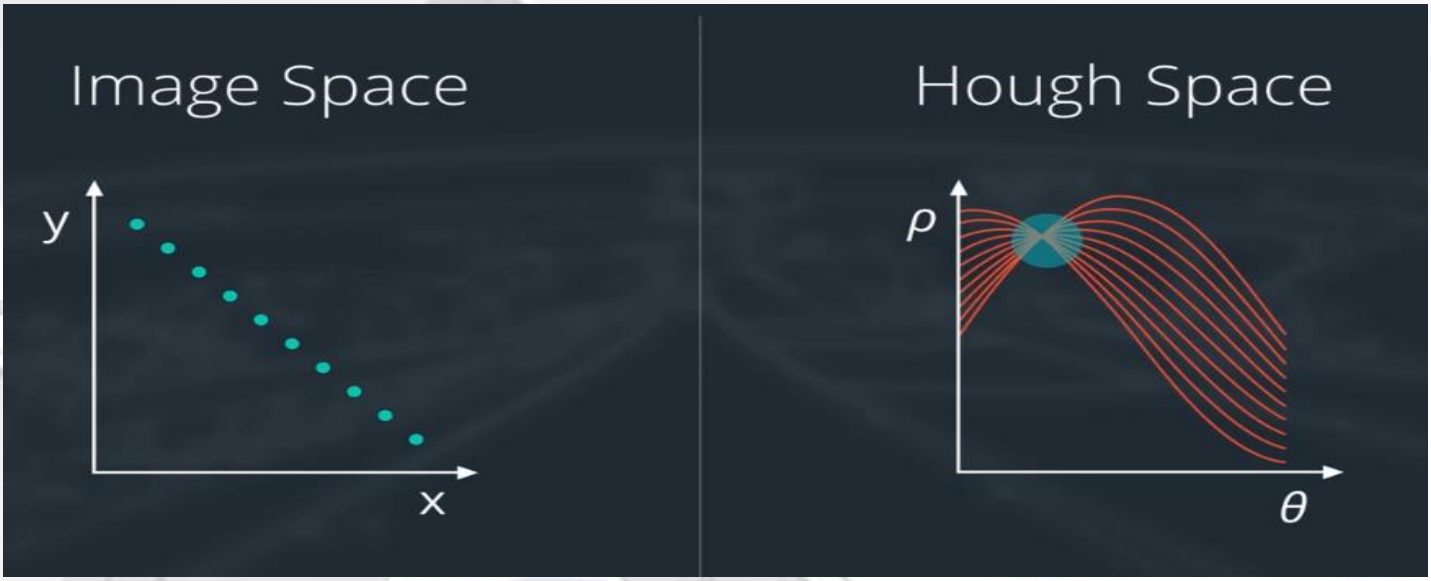


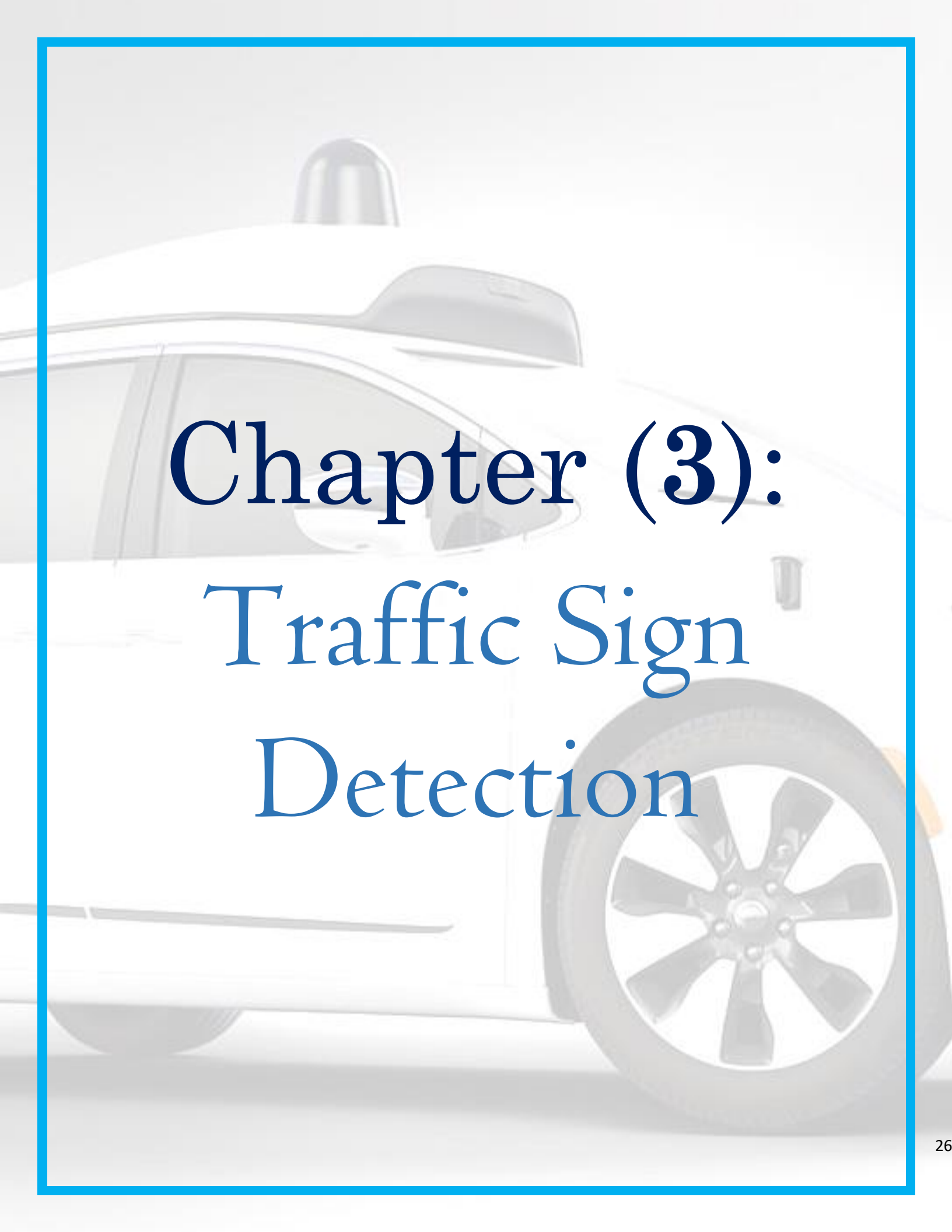
Image Space
$x\cos\theta_0 + y\sin\theta_0 = \rho_0$
y
ρ_0
θ_0
x
Hough Space
ρ
ρ_0
θ_0
θ
ρ = the distance of the line from the origin
θ = the angle away from horizontal



- So to connect the lines, we draw Hough Space in Grid, then see the center block that contain the intersection between all the lines.

- But what if we have a vertical line in Image Space, m (slope) will be infinity.

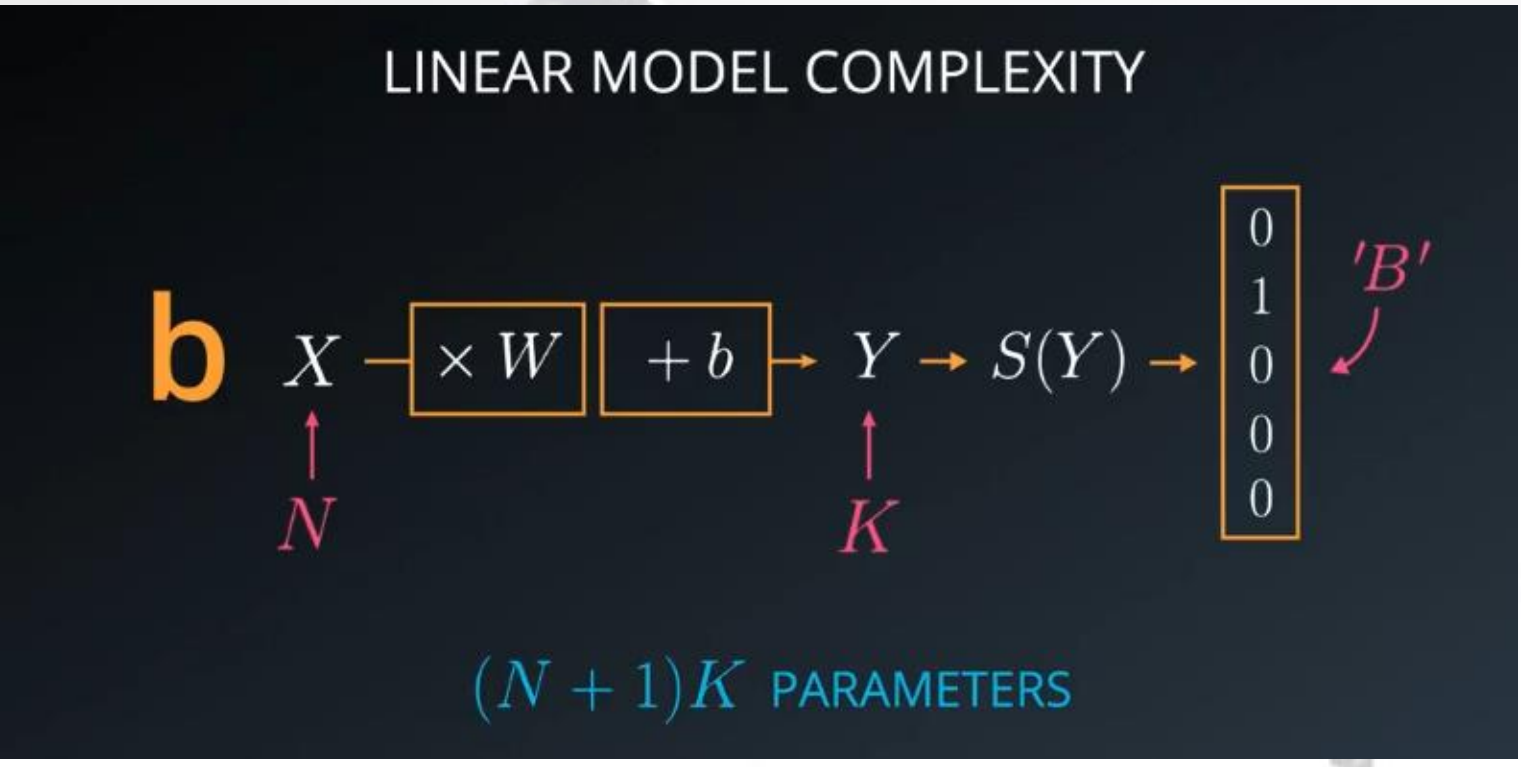
- The solution for this is to use polar coordinates and find the intersection between the sins to form a line.



Chapter (3):
Traffic Sign Detection

3.1 Deep Neural Network:

3.1.1 Linear Models are limited:

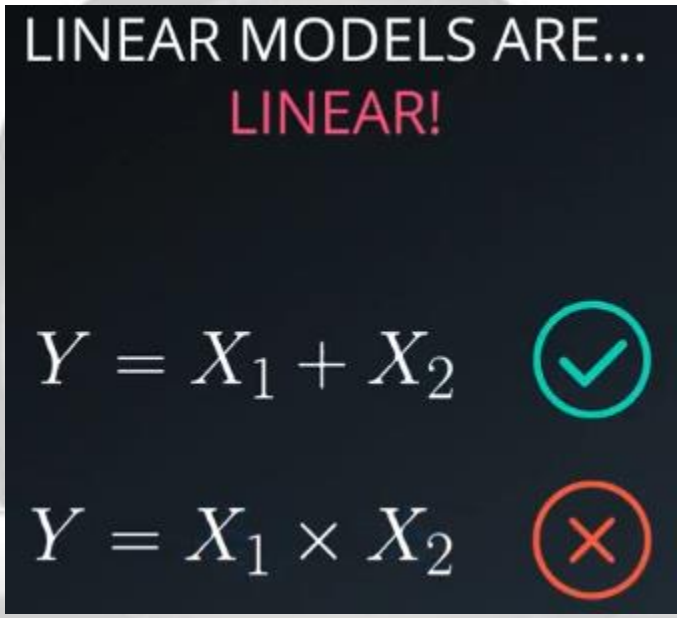


- To know the number of parameters, see the equation above.

- Also the relation is linear which means that the kind of interactions that you're capable of representing with that model is somewhat limited.

- For Example, if 2 inputs interact in an additive way, your model can represent them well as a matrix multiply.

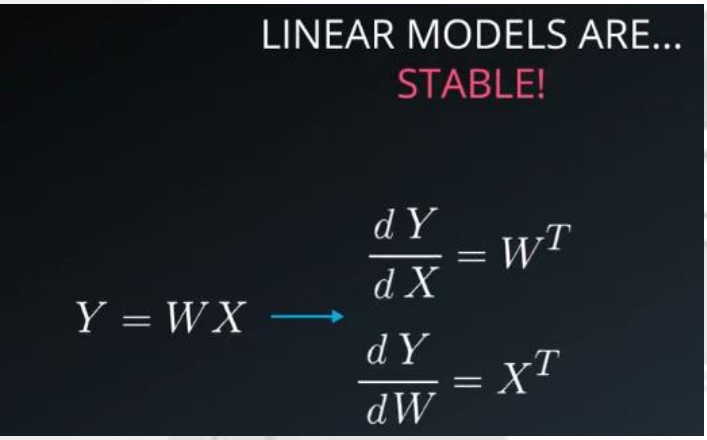
- But if 2 inputs interact in the way that the outcome depends on the product of the two, you won't be able to model that efficiently with a linear model.

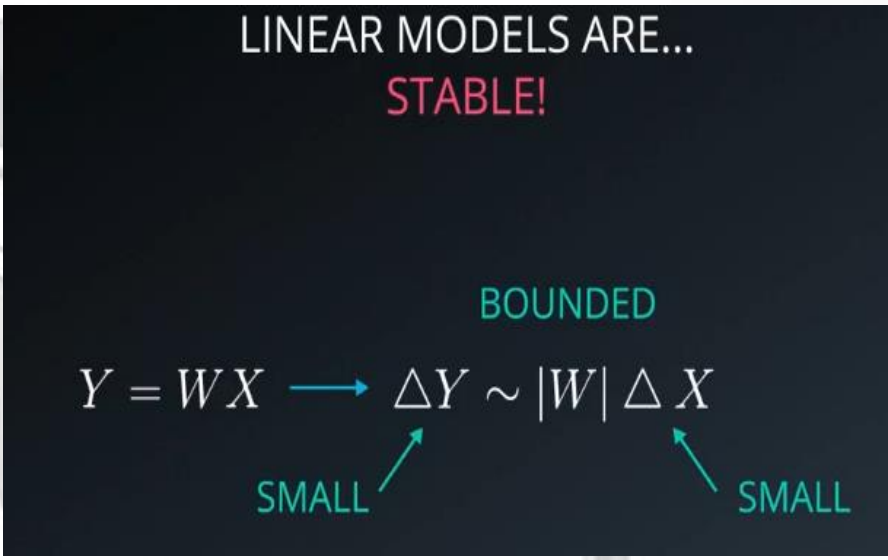


NOTE: Big matrix multiplies are exactly what GPUs were designed for.

- Numerically linear operations are very stable.

- Mathematically, you can show that small changes in the input can never yield big changes in the output.



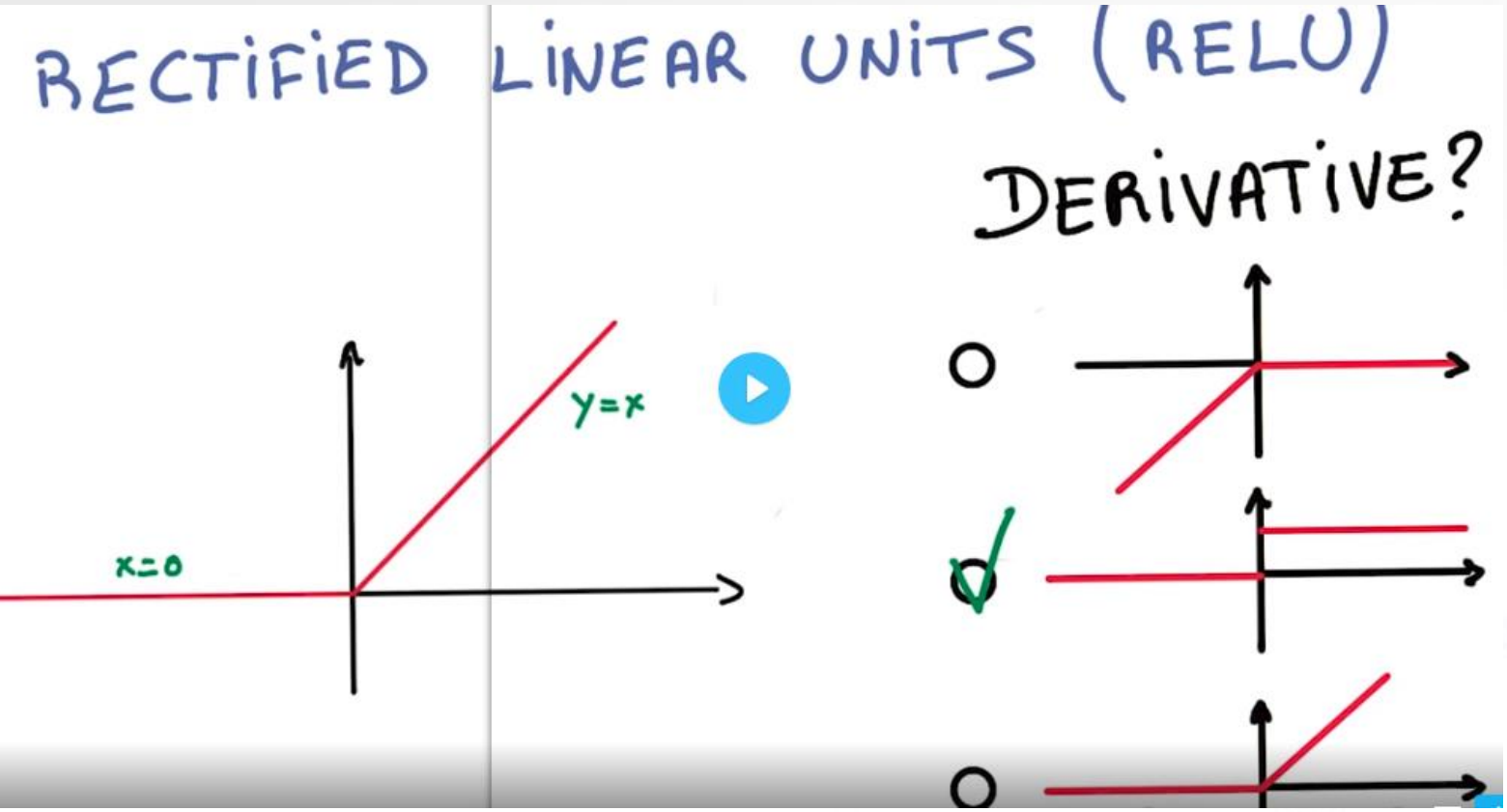


- The derivatives of linear function is constant, you can't get more stable numerically than a constant.

- So, we would like to keep our parameters inside big linear functions, but we would also want the entire model to be nonlinear.

$$Y = W_1 W_2 W_3 X = W X$$

3.1.2 Rectified Linear Units:



3.1.3 Network of ReLU:

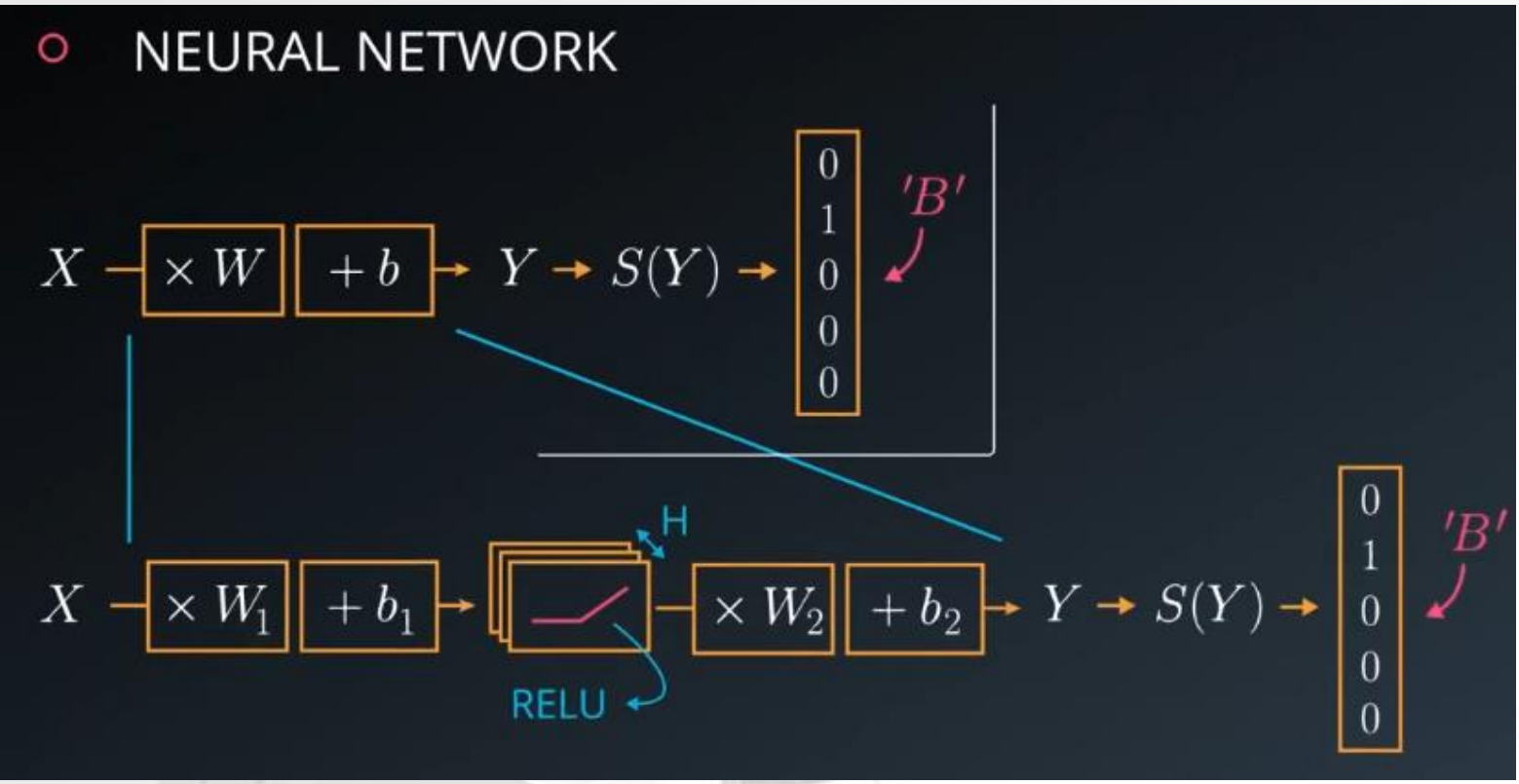
- Because we're lazy engineers, we're going to use our logistic classifier and do the minimal amount of change to make it nonlinear.

- Instead of having a single matrix multiply as our classifier, we're going to insert ReLU right in the middle.

- We now have 2 matrices, one going from the inputs to the ReLUs and another one connecting the ReLUs to the classifier.

- Because of that, we solved two of our problems:

    - Our function now is nonlinear because of the ReLU in the middle.



- We now have a new knob that we can tune this number (H) which corresponds to the number of ReLU units that we have in the classifier.

3.1.4: 2-Layer Neural Network:

- We're going to be adding a hidden layer to a network allows it to model more complex functions.

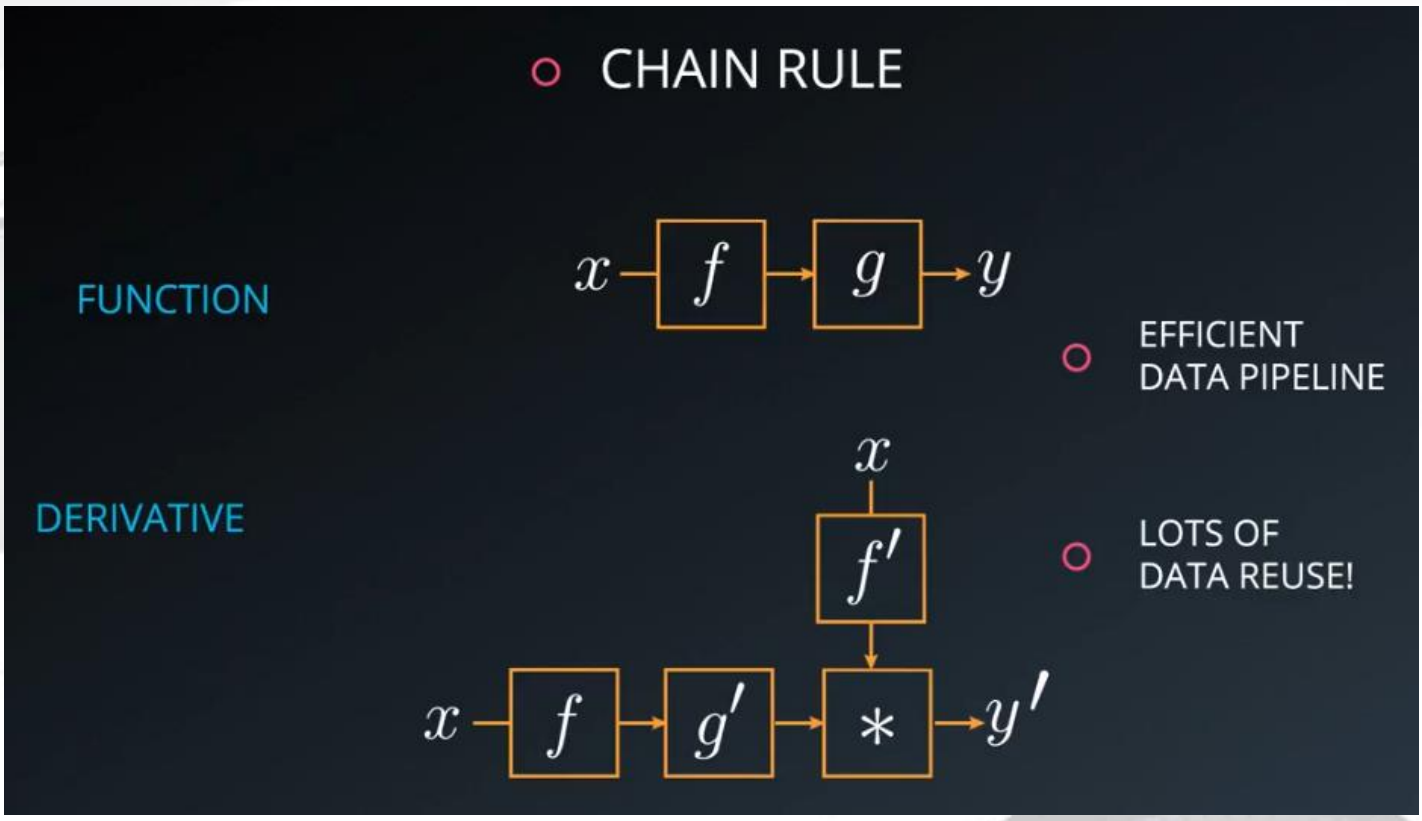
- Also using a non-linear activation function on the hidden layer lets it model non-linear functions.

- Now our network consists as the following:

  - The first layer effectively consists of the set of weights and biases applied to X and passed through `ReLUs`. The output of this layer is fed to the next one, but is not observable outside the network, hence it is known as a `hidden_layer`.

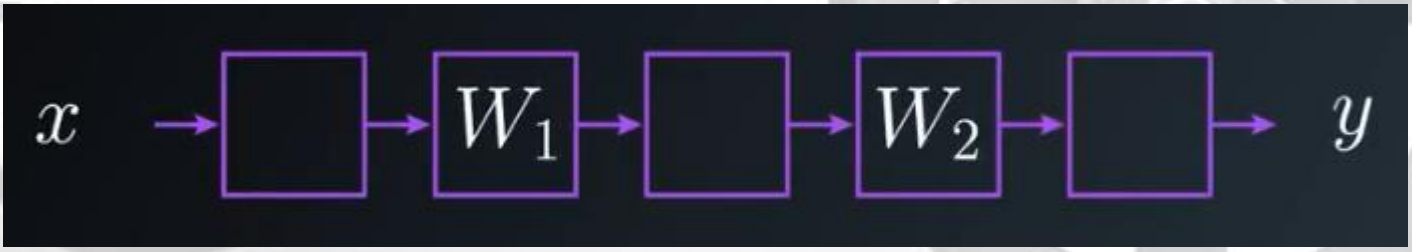
- The second layer consists of the weights and biases applied to these intermediate outputs, followed by the `softmax` function to generate probabilities.

3.1.5: Chain Rule:



- As long as you know how to write the derivatives of your individual functions, there is a simple graphical way to combine them together and compute the derivative for the whole function.

3.1.6: Backprop:

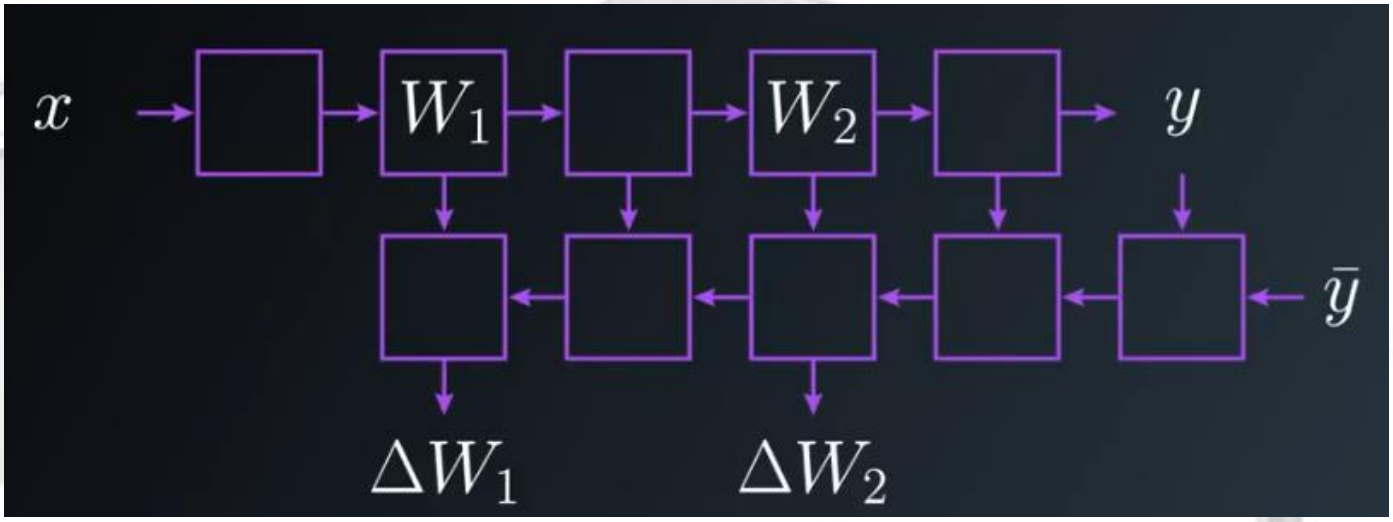


- Let's start with an example, Imagine your network is a stack of simple operations; like `linear transforms`, `ReLUs`, etc.

- Some have parameters, like the matrix transforms, some don't, like the ReLU.

- When you apply your data to some input `x`, you have data flowing through the stack up to your predictions `y` like the image above.

- To compute the derivatives, you create another graph that looks like this below.



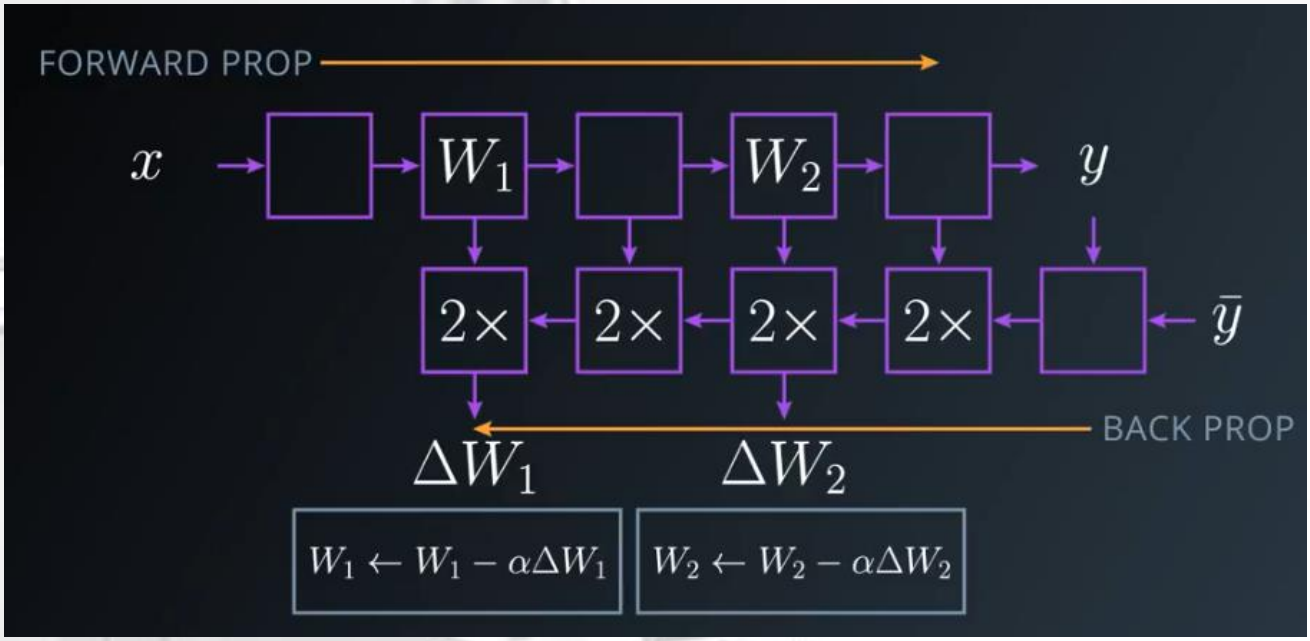
-

The data in that new graph flows backwards through the network, gets combined using the `chain rule` that we saw before, and produces gradients.

- That graph can be derived completely automatically from the individual operations in your network.

- So most deep learning frameworks will just do it for us. This is called `back propagation` and it's a very powerful concept.

- It makes computing derivatives of complex function very efficient, as long as the function is made up of simple blocks with simple derivatives.



- Running the model up to the predictions is often called the `forward prop`. And the model that goes backwards is called the `back prop`.

- So to recap, to run `stochastic gradient descent`. For every single little batch of your data in your training set, you're going to run the forward prop and then the back prop and that will give you gradient for each of your weights in your model.

- Then you're going to apply those gradients with learning rates to your original weights and update them.

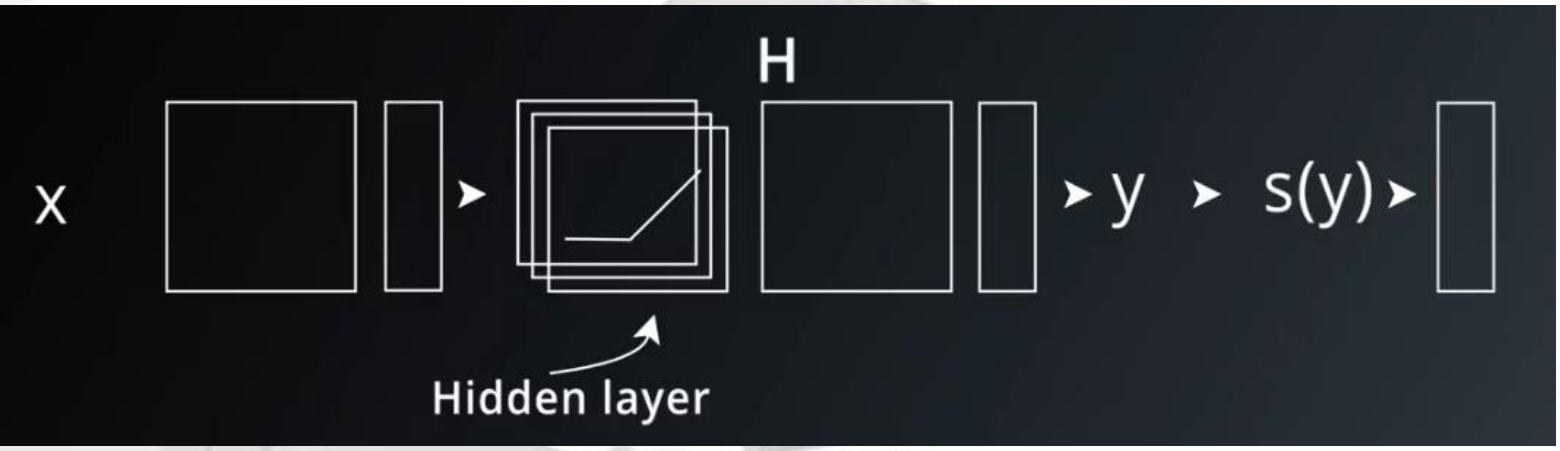
- And you're going to repeat that all over again many, many times. This is how your entire model gets optimized.

NOTE: each block of the back prop often takes about twice the memory that's needed for the forward prop and twice to compute.

3.1.7: Training Deep Neural Network:

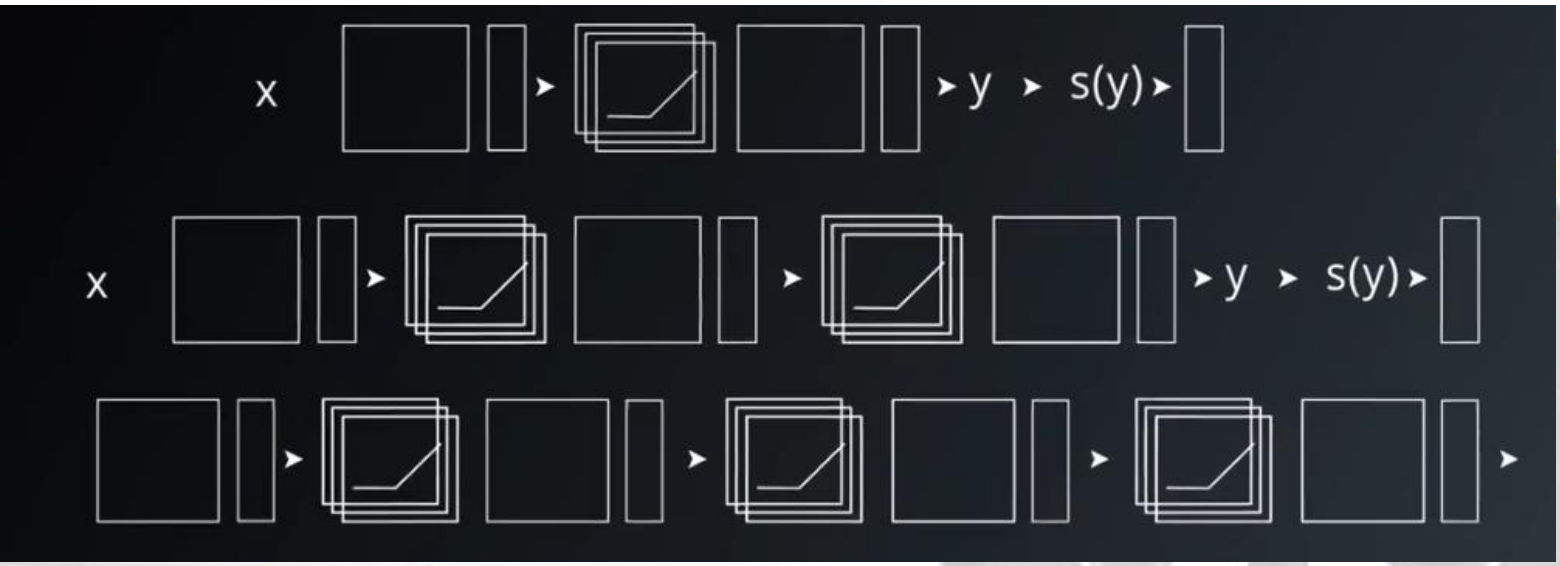
- So now we have a simple neural network consists of 2 layers. We can make it bigger, more complex by increasing the size of that hidden layer in the middle.

- BUT it turns out that increasing this H is not particularly efficient in general. You need to make it very big and it gets really hard to train.

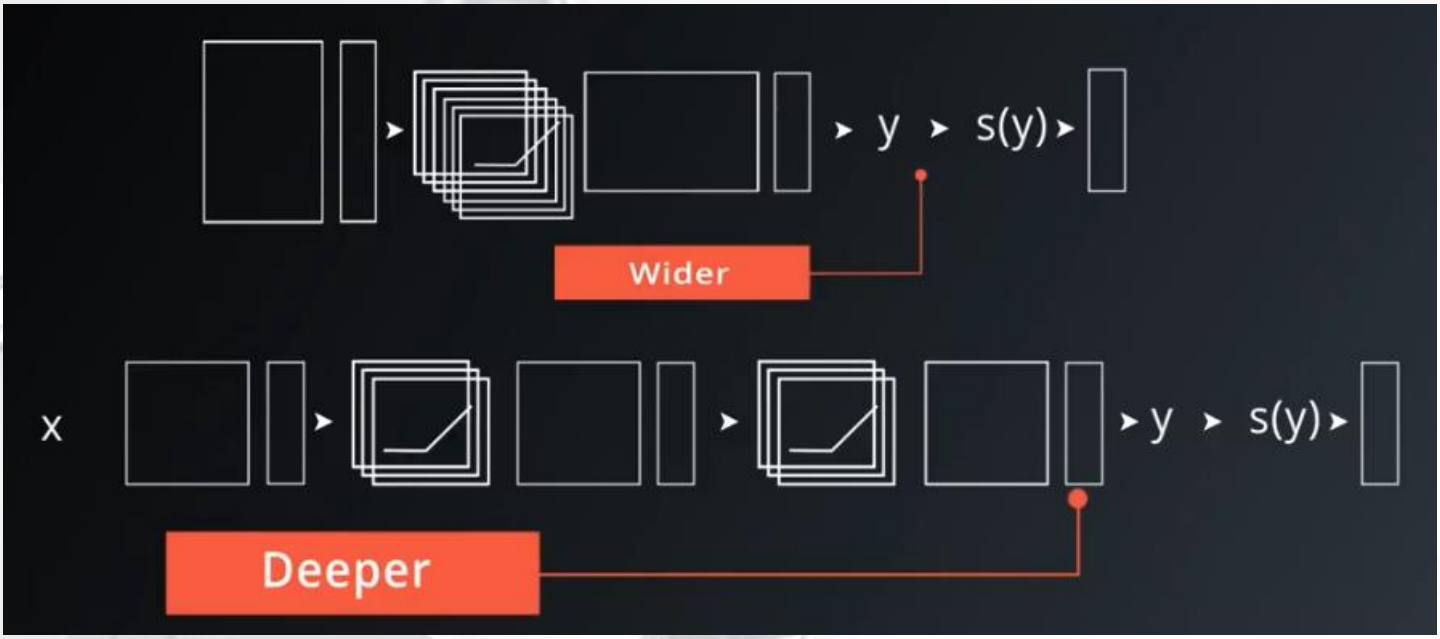


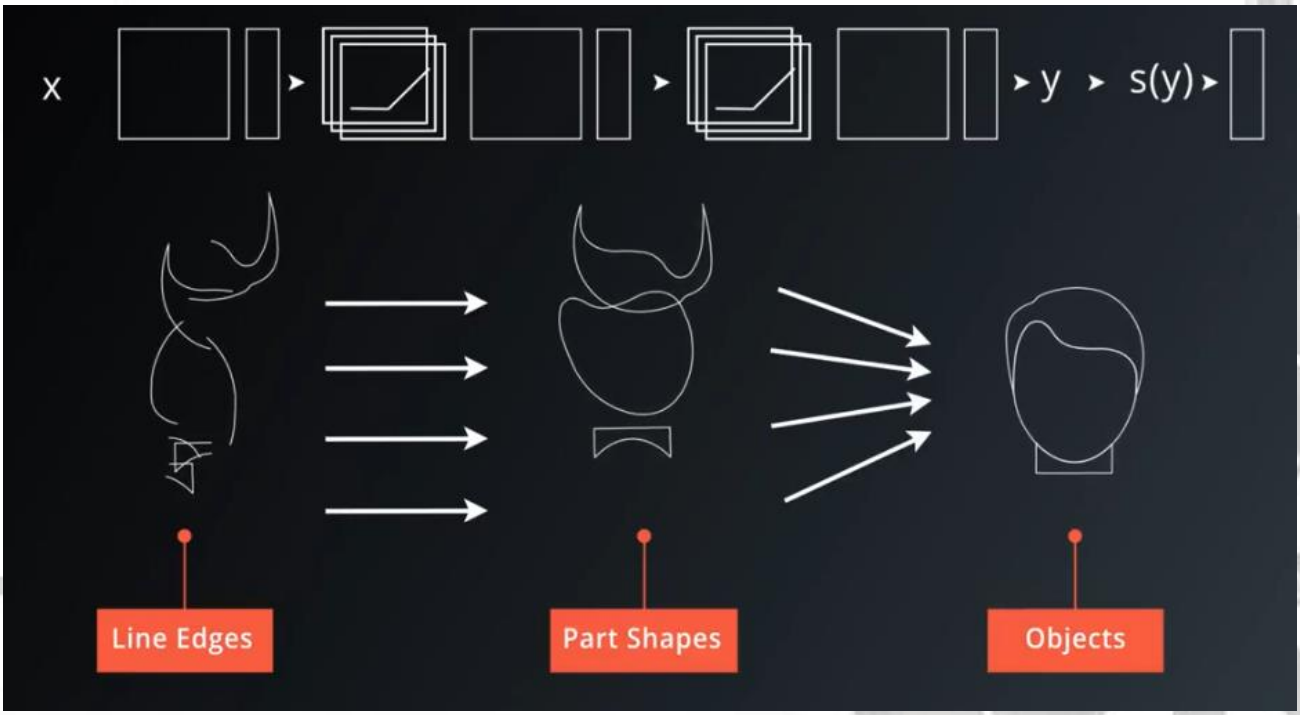
- Here's when deep learning comes in to play.

- Instead we can add more layers and make our model deeper. There are lots of good reasons to do that.



- One is parameter efficiency. You can typically get much more performance with pure parameters by going deeper, rather than wider.





- Another one is that a lot of natural phenomena tend to have a hierarchical structure which deep models naturally capture.

- If you poke at a model for images, for example, and visualize what the model learns. You'll often find very simple things at the lowest layers, like lines or edges.

- Once you move up, you tend to see more complicated things like geometric shapes.

- Go further up, and you start seeing things like objects, faces.

- This is very powerful, because the model structure matches the kind of abstractions that you might expect to see in your data, and as a result the model has an easier time learning them.

3.1.8: Regularization Intro:

- Why did we not figure out earlier that deep models were effective?

- Many reasons, but mostly because deep models only really shine if you have enough data to train them.

- It's only in recent years that large enough data sets have made their way to the academic world.

- Other reason is now we know better how to train very very big models using better regularization techniques.

- There is a general issue when you're doing numerical optimization which called "Skinny Jeans Problem".

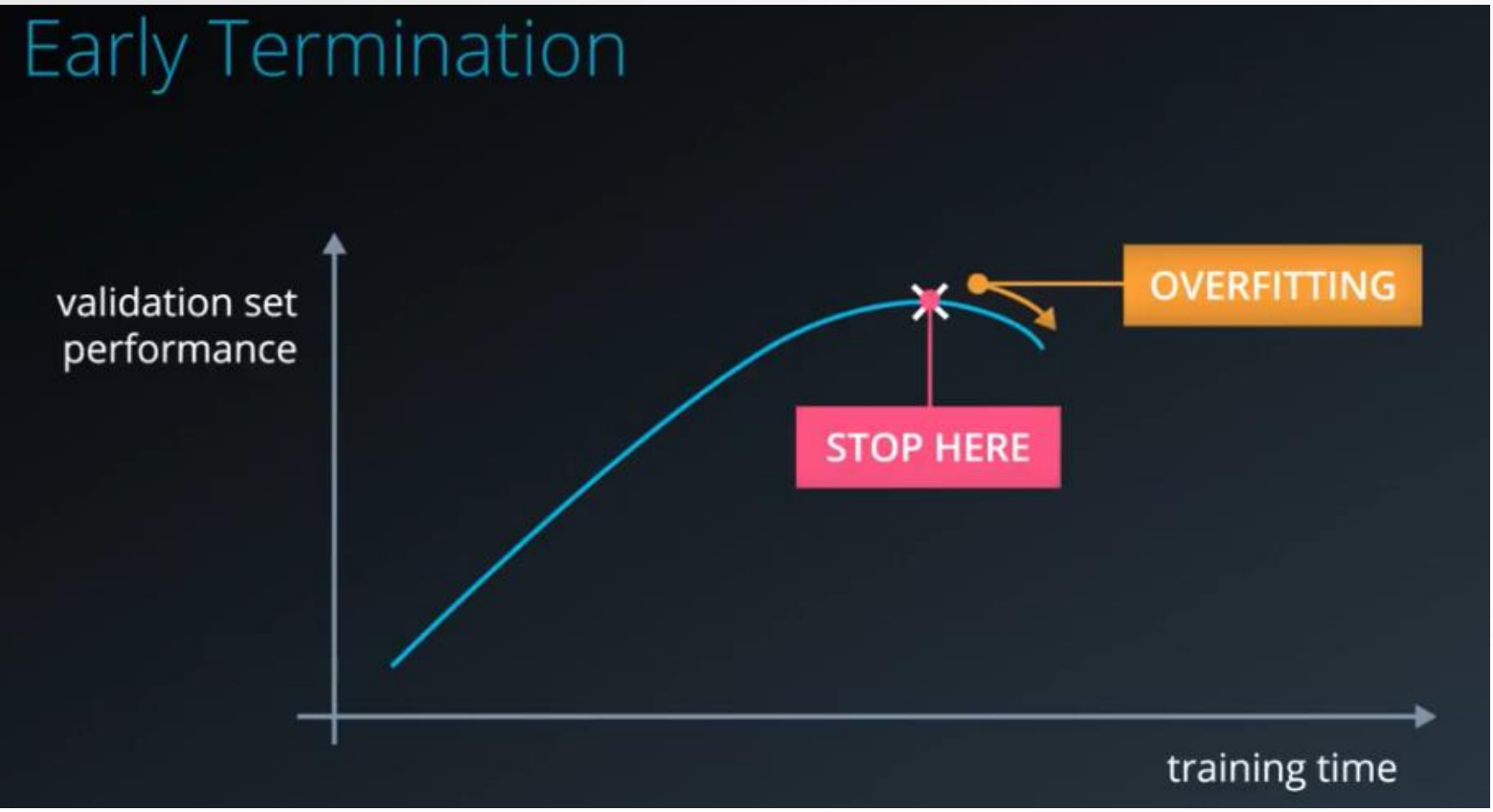
- Skinny Jeans look great, they fit perfectly, but they're really hard to get into.

- So most people end up wearing jeans that are just a bit too big. It's exactly the same with deep networks!

- The network that's just the right size for your data is very hard to optimize.

- So in practice, we always try networks that are way too big for our data and then we try our best to prevent them from over fitting.

3.1.9: Regularization:



- The first way we prevent over fitting, is by looking at the performance on our validation set. And stopping to train, as soon as we stop improving.

- It's called "Early Termination", and it's still the best way to prevent your network from over optimizing on the training set.

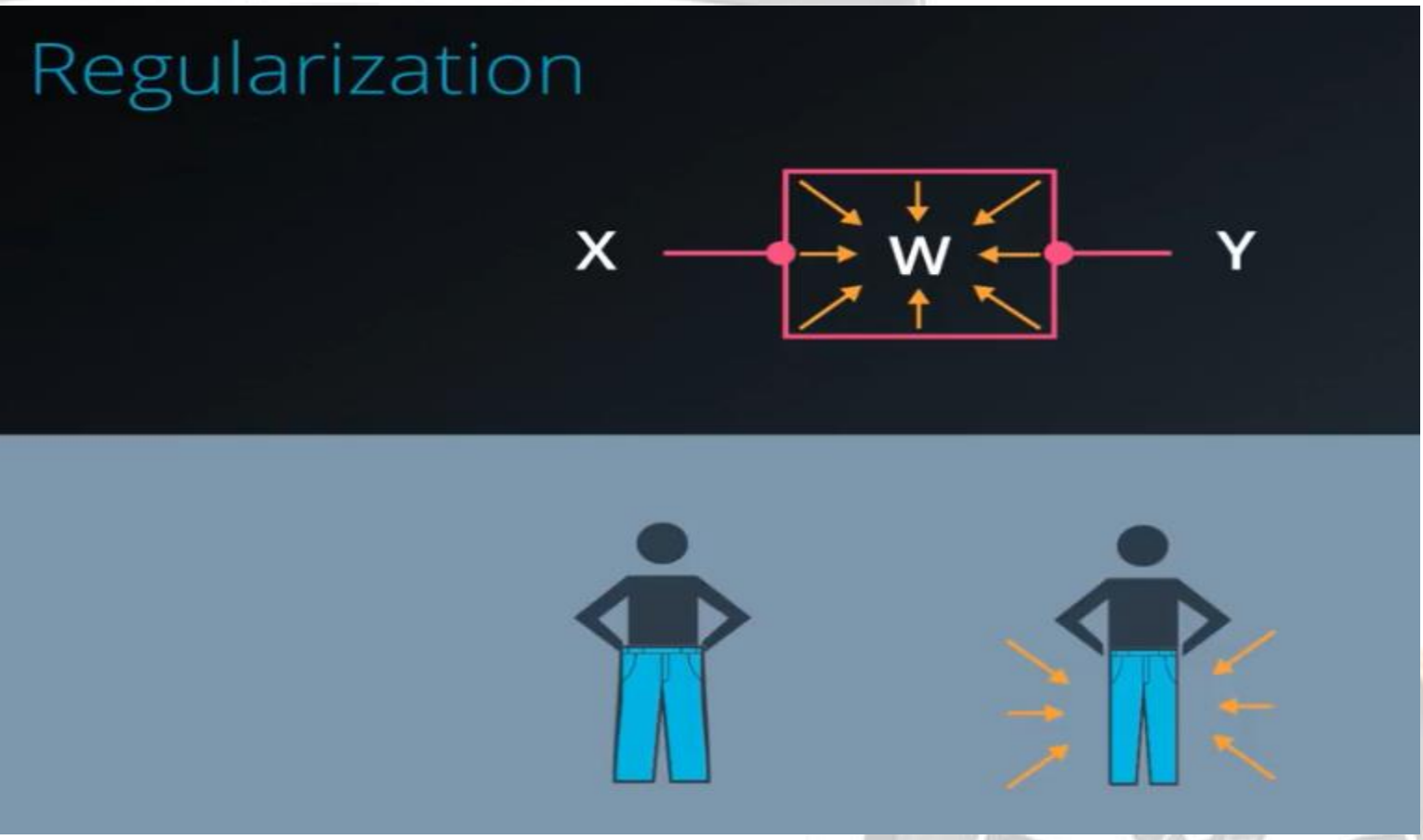
- Another way is to apply regularization. Regularizing means applying artificial constraints on your network that implicitly reduce the number of free parameters.

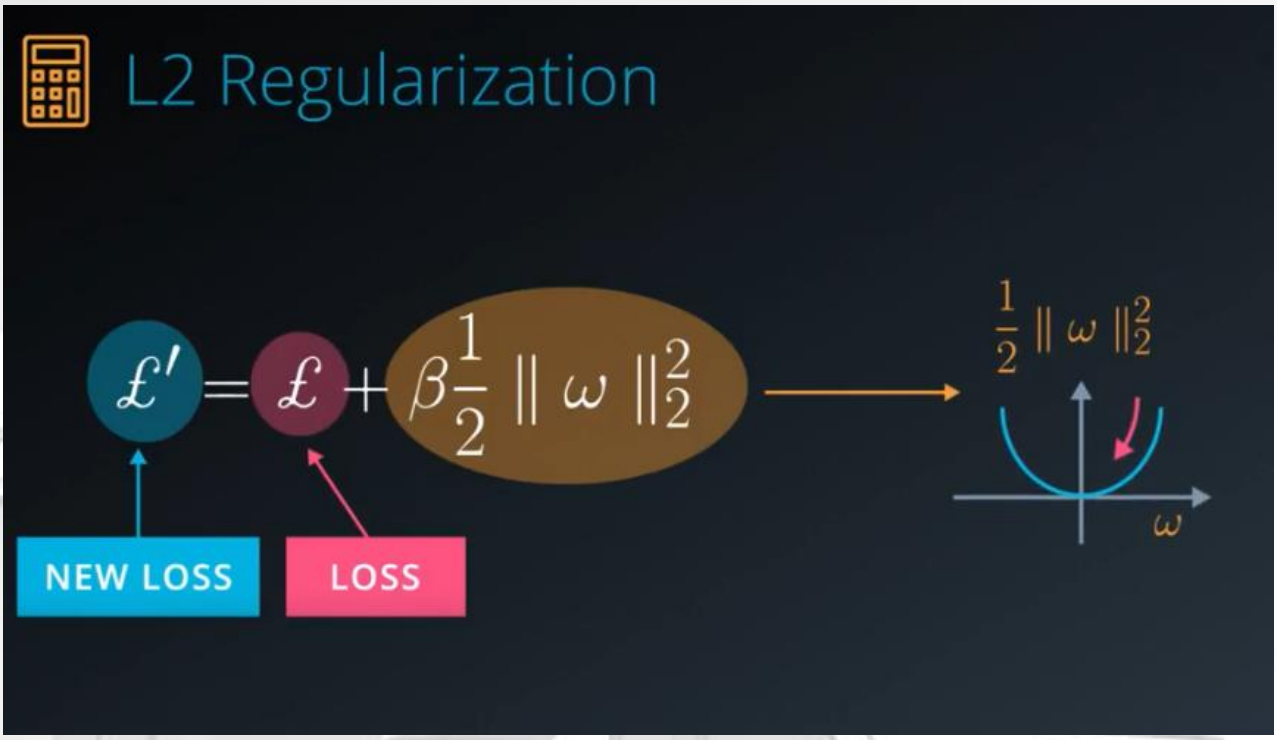
- While not making it more difficult to optimize.

- In the skinny jeans analogy, think stretch pants. They fit just as well, but because they're flexible, they don't make things harder to fit in.

- The stretch pants of deep learning are called “L2 Regularization”.

- The idea is to add another term to the loss, which penalizes large weights. It’s typically achieved by adding the L2 norm of your weights to the loss, multiplied by a small constant.





- And yes, yet another hyper parameter to tune in, sorry about that.

3.1.10: Regularization:

- The nice thing about L2 Regularization is that it's very simple.

- Because you just add it to your loss, the structure of your network doesn't have to change.

- You can even compute its derivative by hand.

- Remember that L2 norm stands for the sum of the squares of the individual elements in a vector.

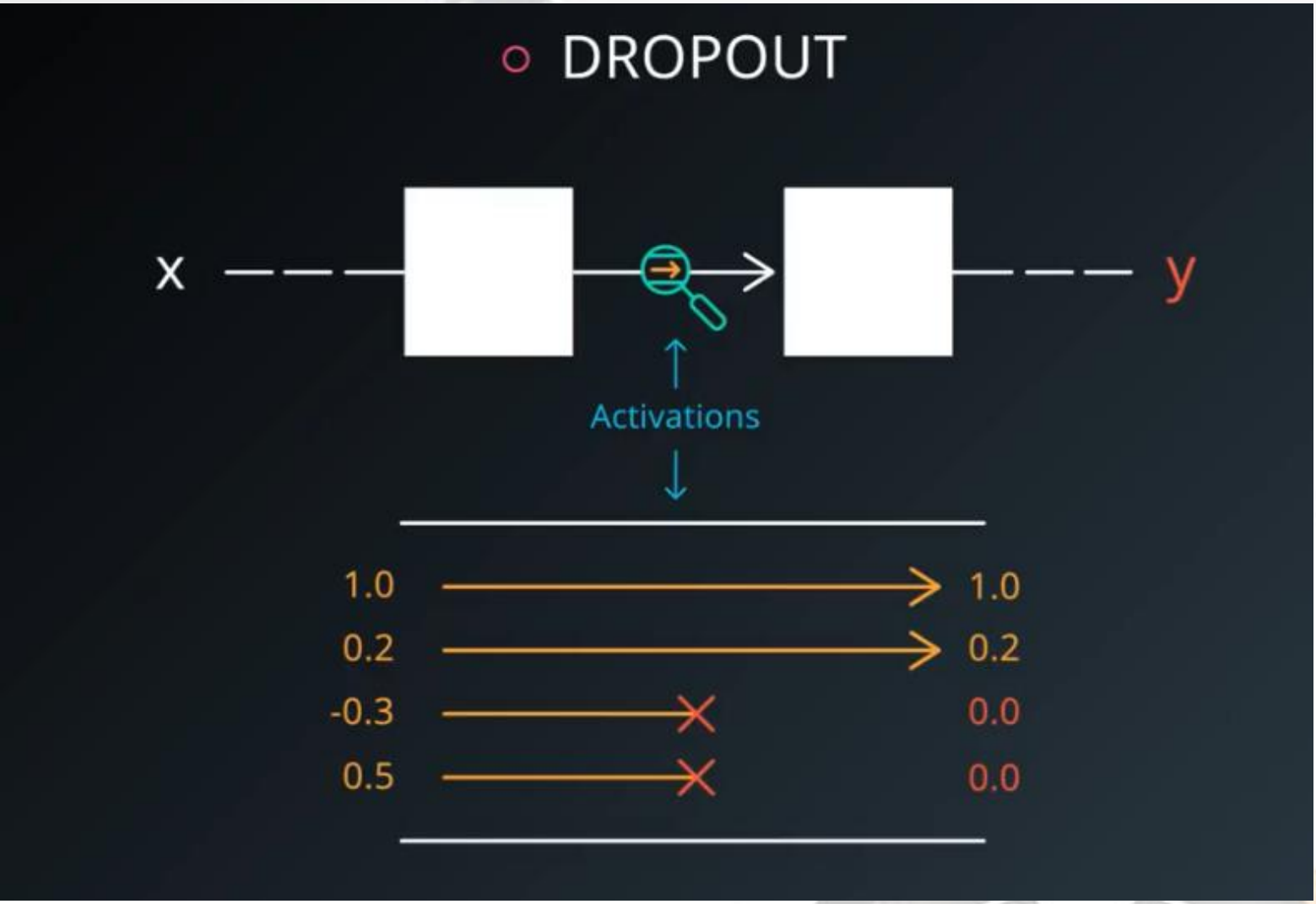
3.1.11: Dropout:

- There's another important technique for regularization, than only emerged relatively recently and works amazingly well.

- It's called "Dropout". And it works like this.

- Imagine that you have one layer that connects to another layer. The values that go from one layer to the next are often called "activations".

- Now take those activations and randomly, for every example you train your network on, set half of them to 0.



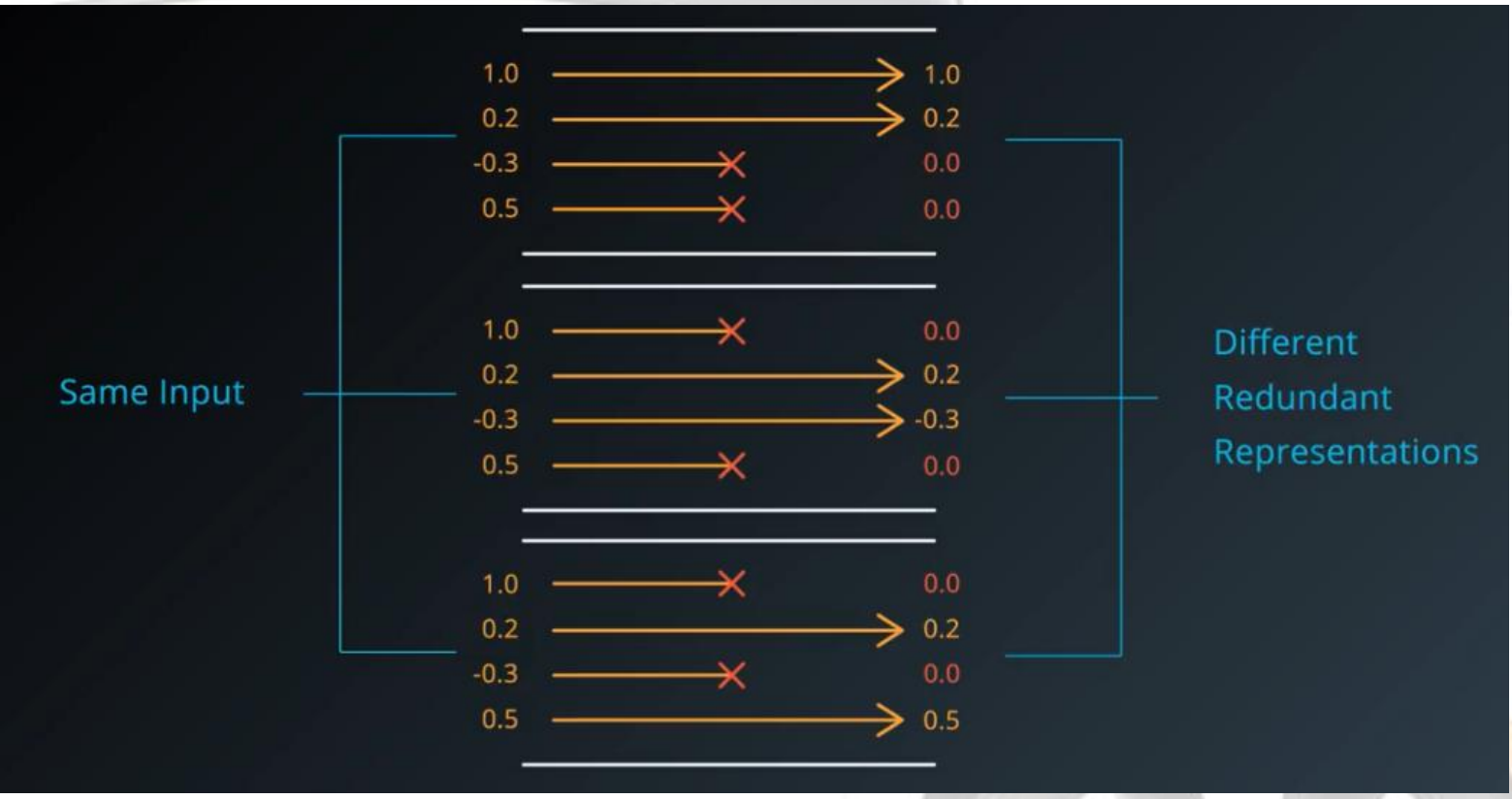
- Completely and randomly, you basically take half of the data that's flowing through your network, and just destroy it and then randomly again.

- So what happens with dropout? Your network can never rely on any given activation to be present, because they might be squashed at any given moment.

- So it is forced to learn a redundant representation for everything to make sure that at least some of the information remains. It's like a game of whack-a-mole.

- One activations get smashed, but there is always one or more that do the same job. And that don't get killed.

- So everything remains fine at the end.



- Forcing your network to learn redundant representations might sound very inefficient.

- But in practice, it makes things more robust, and prevents over fitting. It also makes your network act as if taking the consensus over an ensemble of networks, which is always a good way to improve performance.

3.1.12: Dropout part2:

- When you evaluate the network that's been trained with drop out, you obviously no longer want this randomness.

- You want something deterministic.

- Instead, you're going to want to take the consensus over these redundant models.

- You get the consensus opinion by averaging the activations.

- You want `Ye` here to be the average of all the `Yt`s that you got during training.

- Here's a trick to make sure this expectation holds.

- During training, not only do you use zero out so the activations that you drop out, but you also scale the remaining activations by a factor of 2.

- This way, when it comes time to average them during evaluation, you just remove these dropouts and scaling operations from your neural net.

- And the result is an average of these activations that is properly scaled.

3.2) CNN:

3.2.1) Intro to CNN:

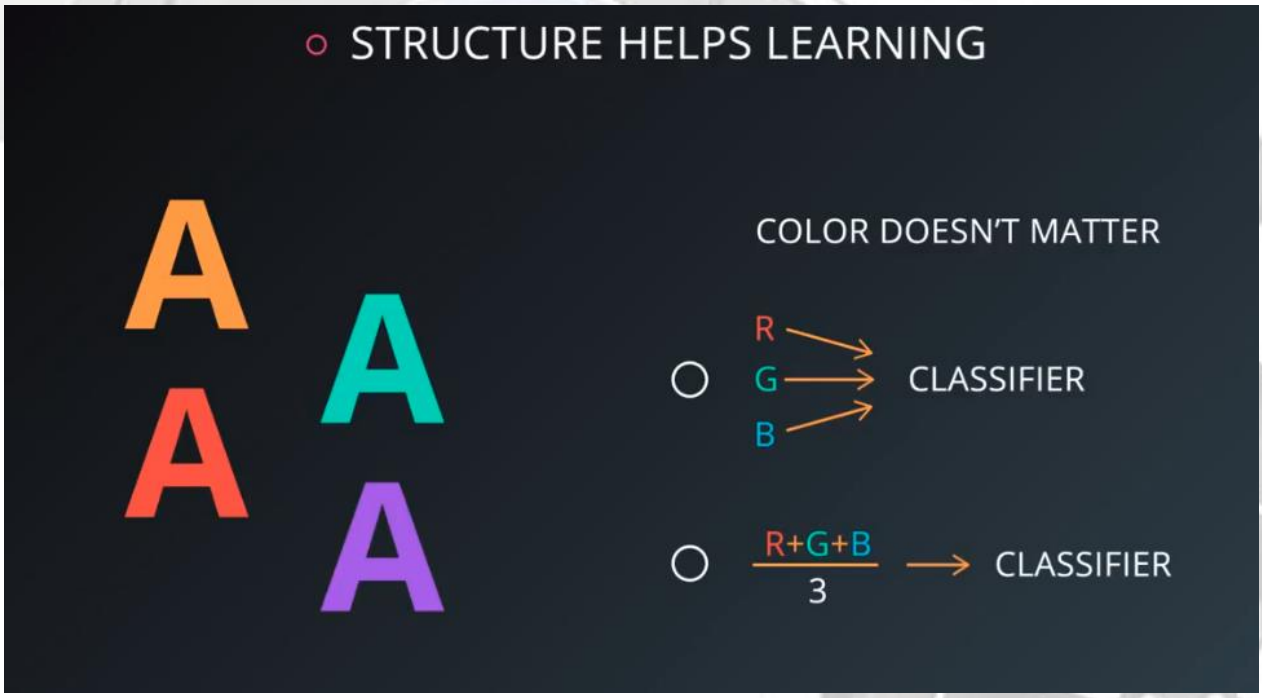
- If you know something about your data, for example, if it's an image or a sequence of things. We can do a lot better!

3.2.2: Color:

- The idea is very simple. If your data has some structure, and your network doesn't have to learn that structure from scratch, it's going to perform better.

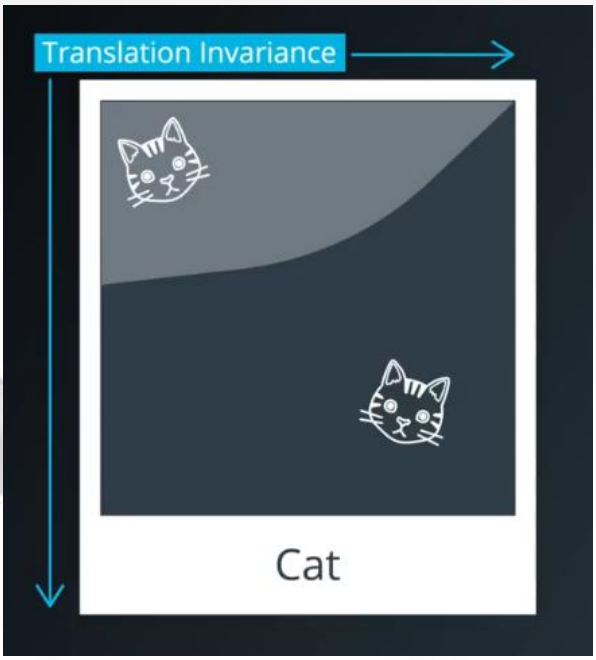
- Imagine, for example, that you're trying to classify those letters, and you know that color is really not a factor in what makes an A an A.

- What do you think would be easier for your classifier to learn? Answer:



Second choice

3.2.3: Statistical Invariance:



- Here's another example, you want your network to say it's an image with a cat in it.

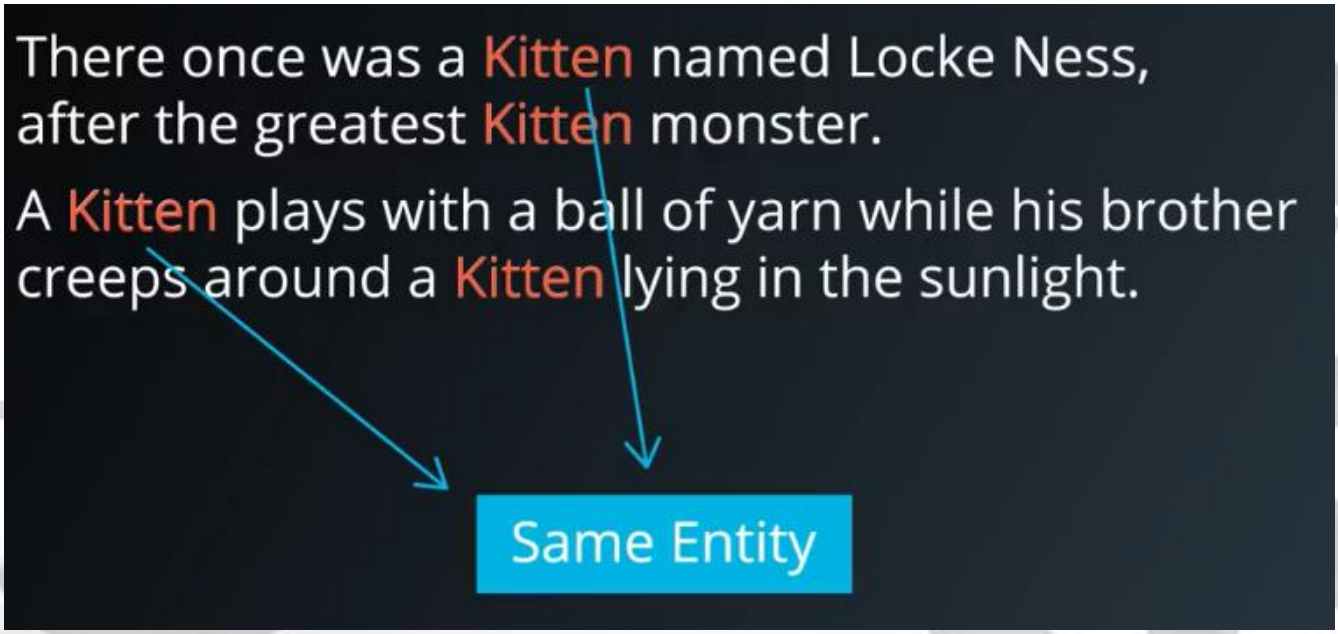
- It doesn't really matter where the cat is, it's still an image with a cat.

- If your network has to learn about kittens in the left corner, and about kittens in the right corner independently, that's a lot of work that it has to be done.

- How about you telling it, instead explicitly, that objects and images are largely the same whether they're on the left or on the right of the picture.

- That's what's called "Translation Invariance". Different positions, same kitten.

- Yet another example, Imagine you have a long text that talks about kittens.

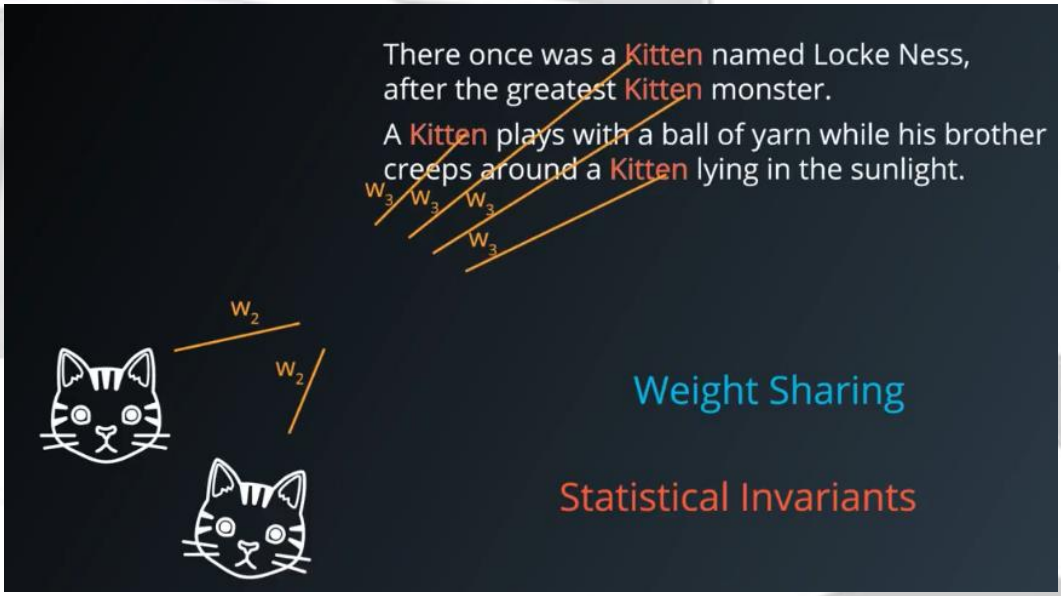


Does the meaning of kitten change depending on whether it's in the first sentence or in the second one? Mostly not, so it you're trying to network on

text, maybe you want the part of the network that learns what a kitten is to be reused every time you see the word kitten, and not have to re-learn it every time.

- The way you achieve this in your own networks, is using what's called "weight sharing".

- When you know that two inputs can contain the same kind of information, then you share their weights. And train the weights jointly for those inputs.

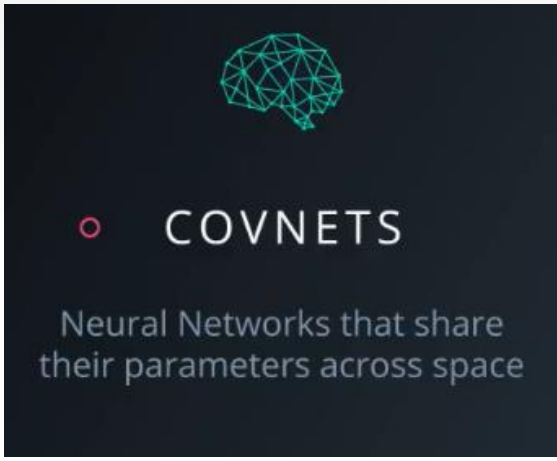


- Statistical invariants, things that don't change on average across time or space, are everywhere.

- For images, the idea of weight sharing will get us to study convolutional networks.

- For text and sequences in general, it will lead us to embedding and recurrent neural networks.

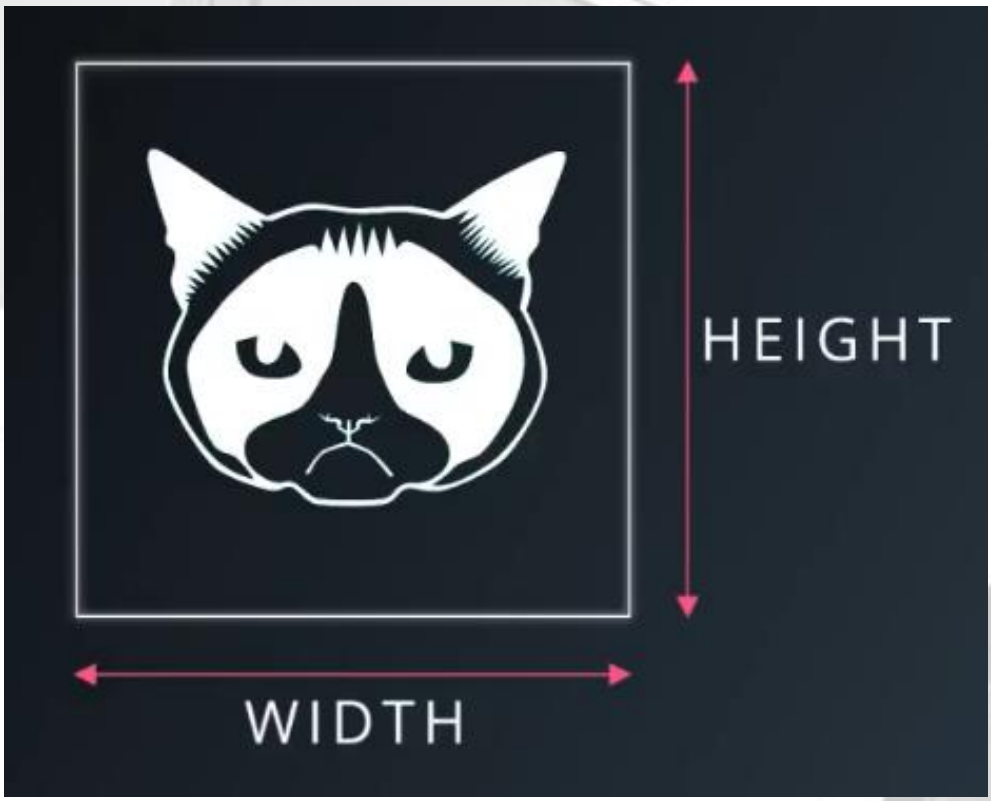
3.2.4: Convolutional Networks:



- Let's talk about Convolutional Networks, or ConNets.

- CovNets are neural networks that share their parameters across space.

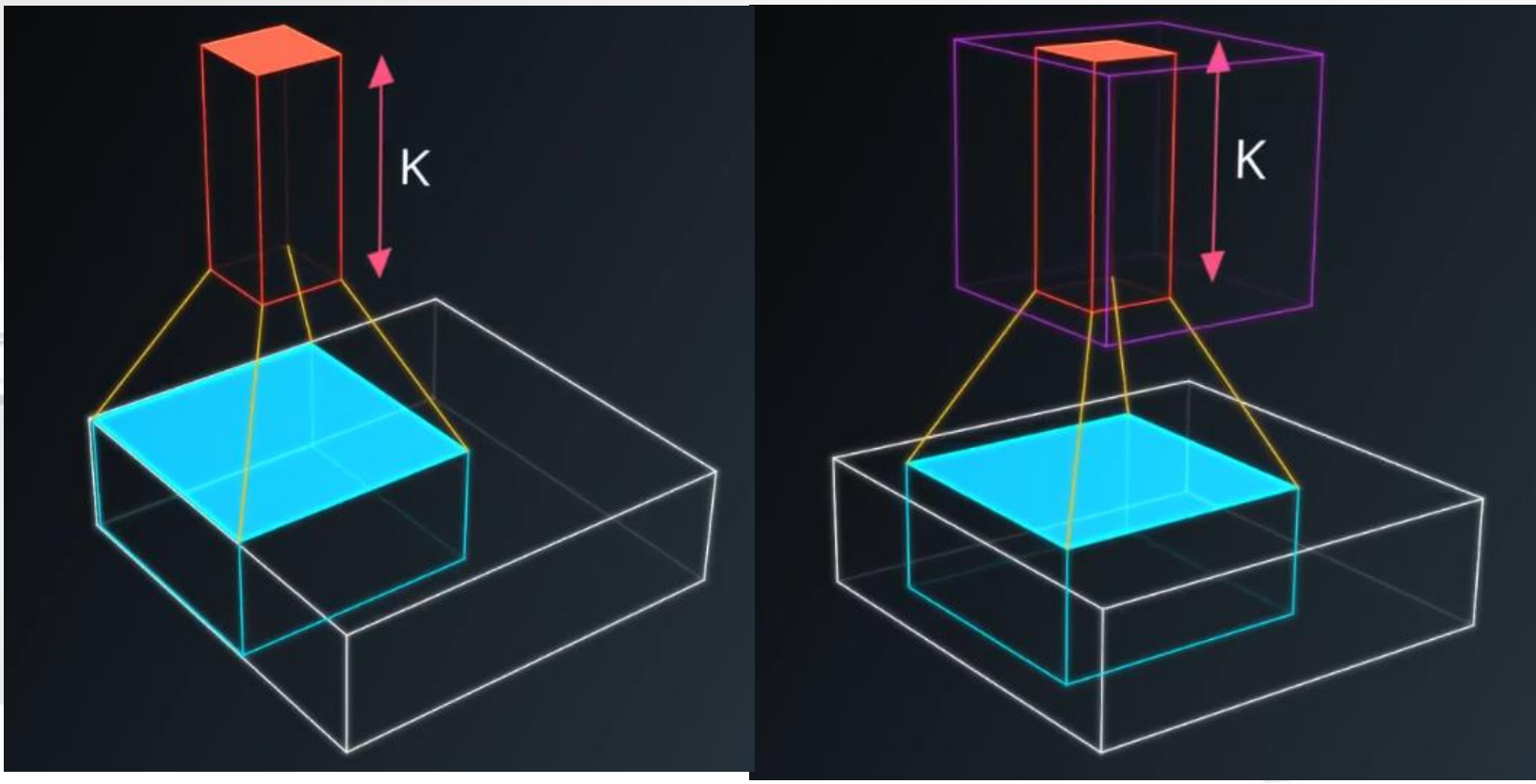
- Imagine you have an image. It can be represented as a flat pancake.



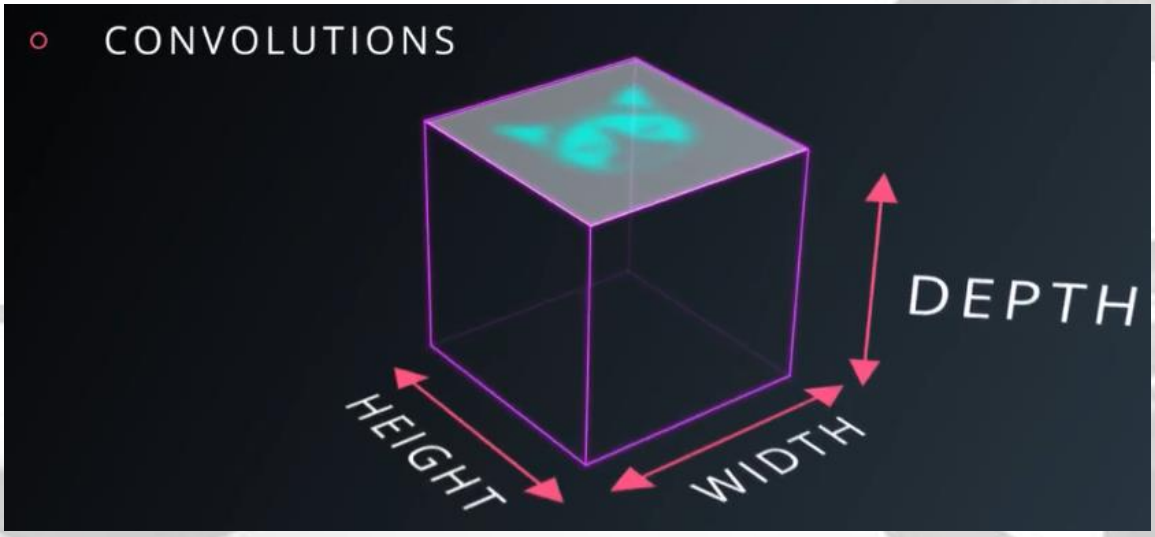
- It has a width, a height, and because you typically have red, green, and blue channels, it also has a depth. In this instance, depth is 3 (RGB). That's your input.

- Now, imagine taking a small patch of this image, and running a tiny neural network on it, with say, K outputs.

- Let’s represent those outputs vertically in a tiny column like this.



- Now, let’s slide that little neural network across the image without changing the weights.



- Just slide across invertically like we’re painting it with a brush.

- On the output, we've drawn another image. It's got a different width, a different height. And more importantly, it's got a different depth. Instead of just R, G, and B.

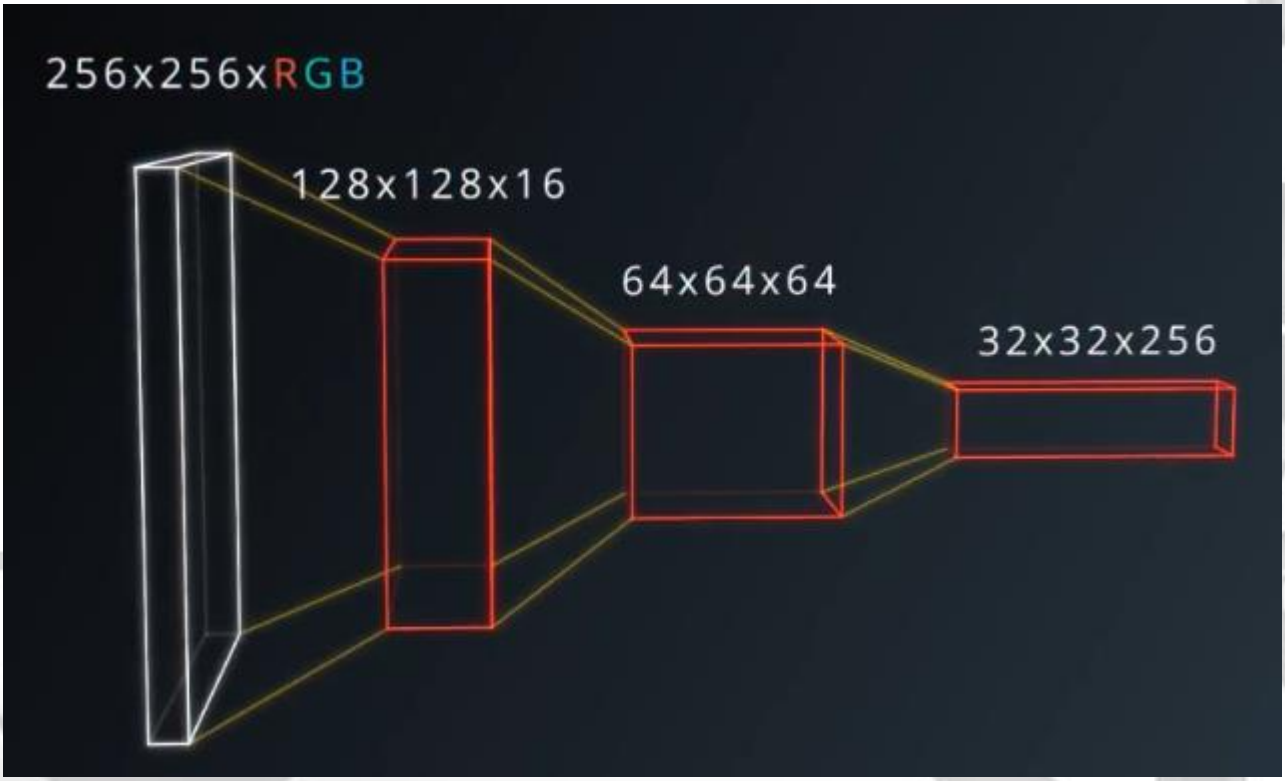
- Now, you have an output that's got many colored channels, K of them.

- This operation is called the convolution.

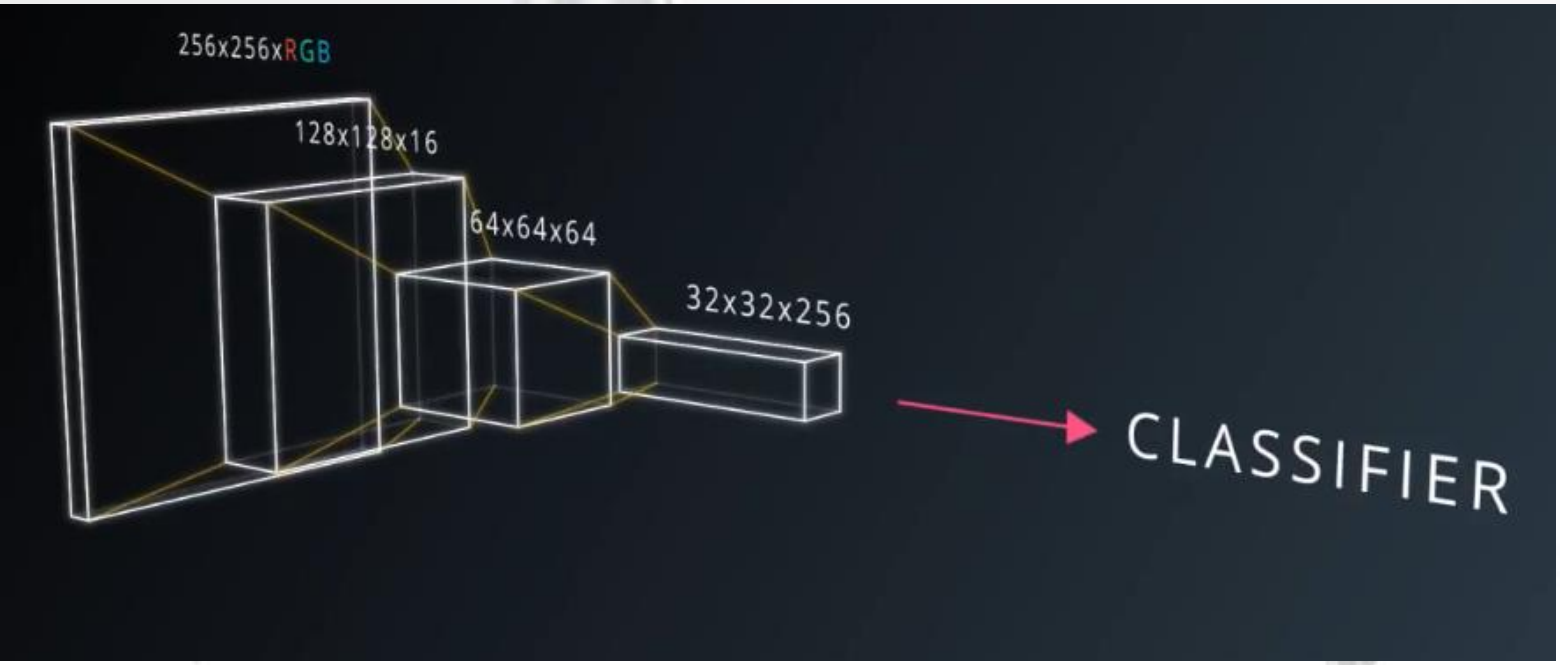
- If your patch size were the size of the whole image, it would be no different than the regular layer of a neural network.

- But because we have this small patch instead, we have many fewer weights and they are shared across space.

- A covenant is going to basically be a deep network where instead of having stacks of matrix multiply layers, we're going to have stacks of convolutions.



- The general idea is that they will form a pyramid. At the bottom, you have this big image, but very shallow just R, G, and B.



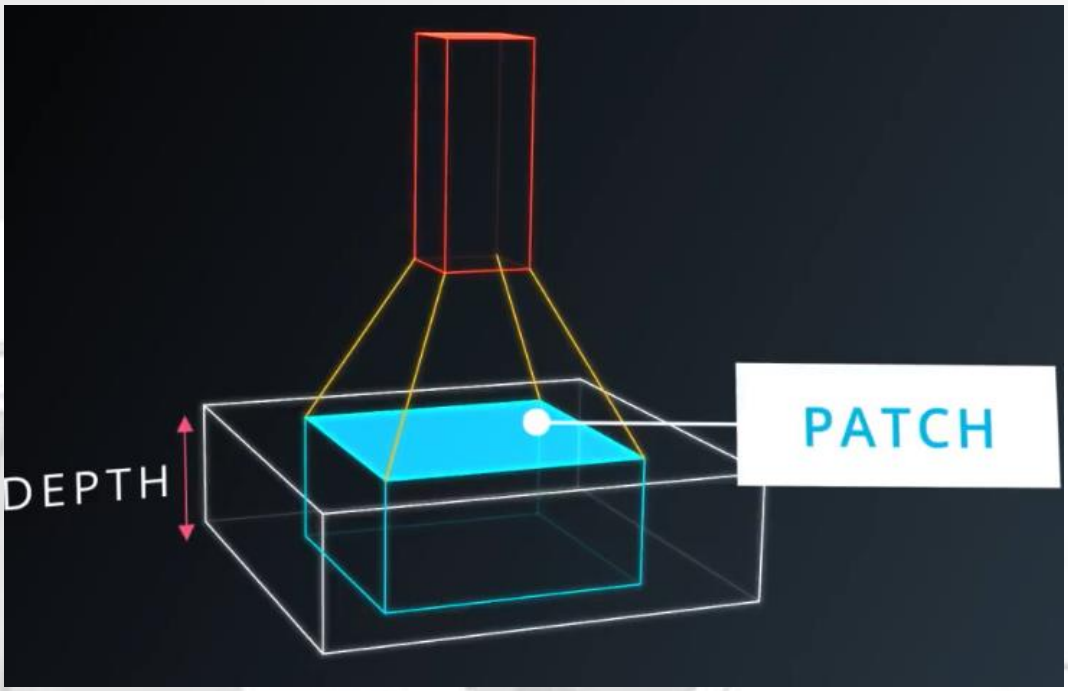
- You're going to apply convolutions that are going to progressively squeeze the special dimensions while increasing the depth which corresponds roughly to the semantic complexity of your representation.

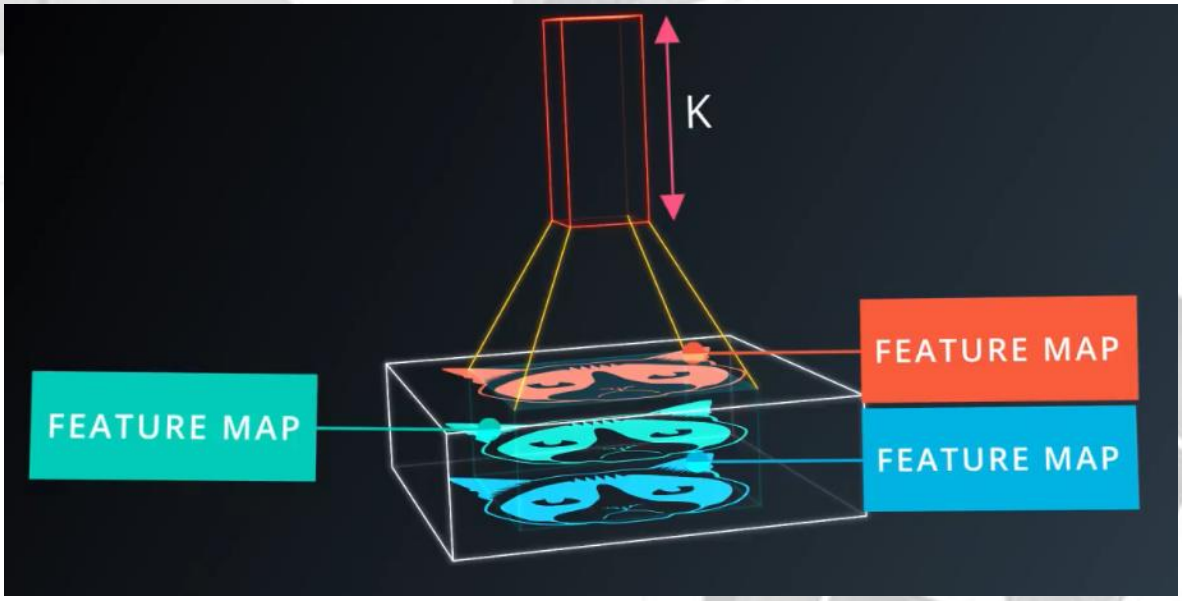
- At the top, you can put your classifier.

- You have a representation where all this special information has been squeezed out, and only parameters that map to content of the image remain.

- So that's the general idea. If you're going to implement this, there are lots of little details to get right and a fair bit of lingo to get used too.

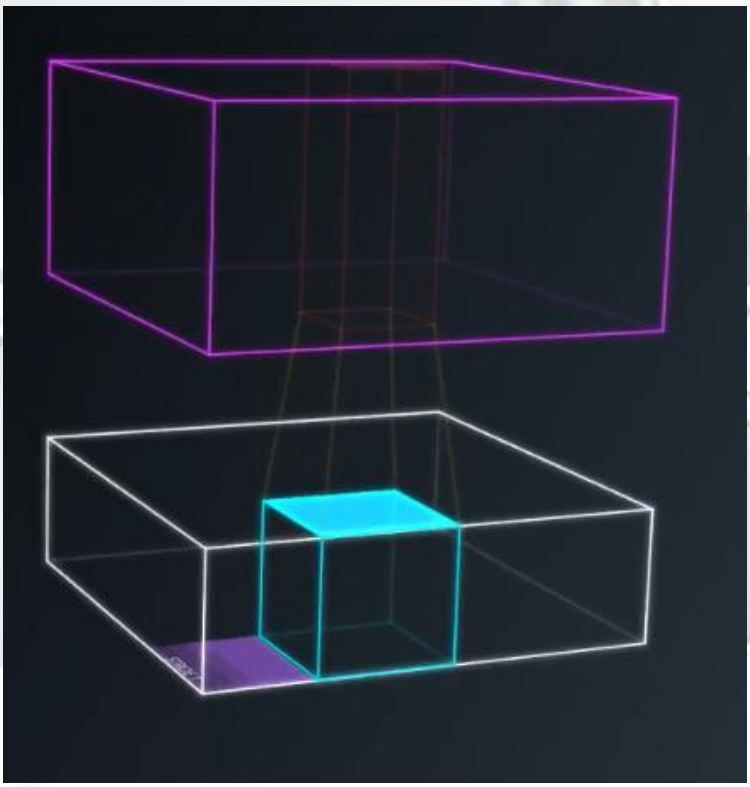
- You've met the concept of Patch and Depth.

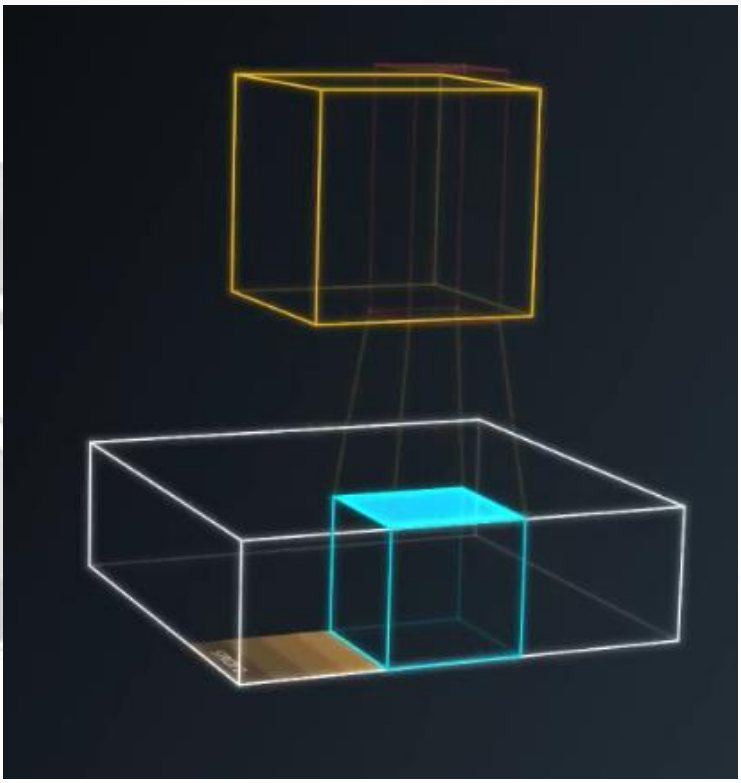




- Patches are sometimes called Kernels. Each pancake in your stack is called a feature map.

- Another term that you need to know is Stride. It's the number of pixels, so that you're shifting each time you move your filter.





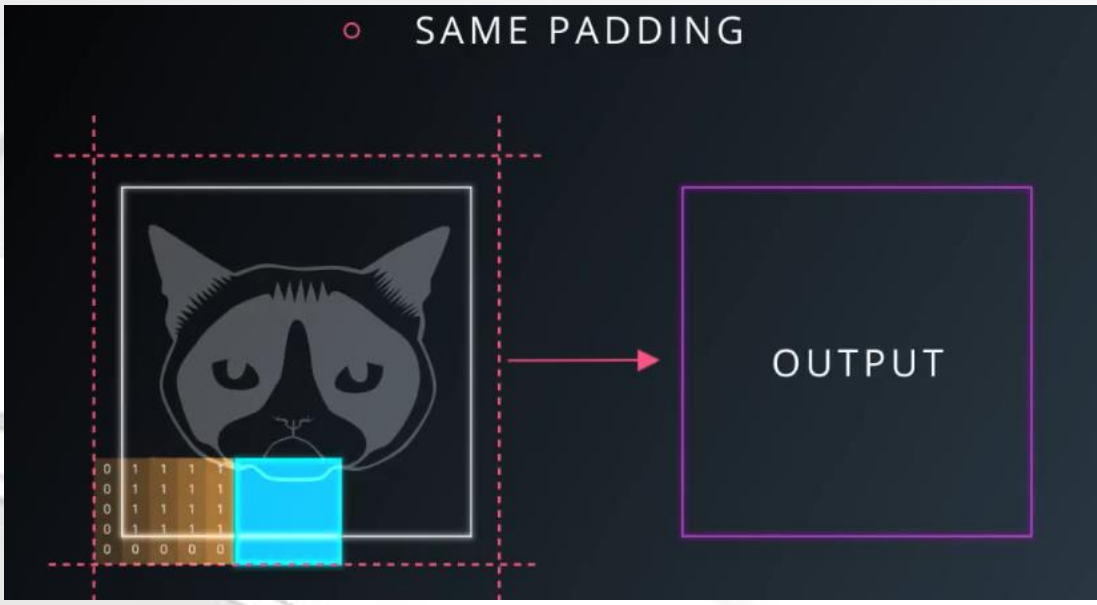
- The stride of one makes the output roughly the same size as the input.

- A stride of two means it's about half the size. I say roughly because it depends a bit about what you do at the edge of your image.

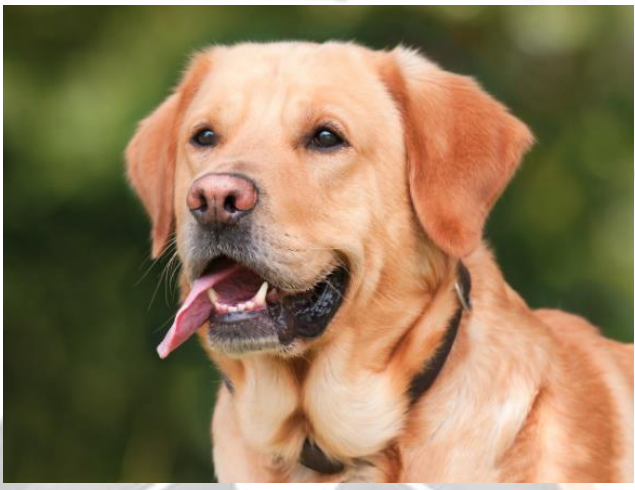
- Either you don't go pass the edge and it's often called "Valid Padding" as a shortcut.

- Or you go off the edge and pad with zeros in such a way that the output map size is exactly the same size as the input map.

- That is often called "Same Padding" as a shortcut.



3.2.5: Intuition:



- Let's say we wanted to classify the following image of a dog as a Golden Retriever.

- As humans, how do we do this?

- One thing we do is that we identify certain parts of the dog, such as:

    - Nose

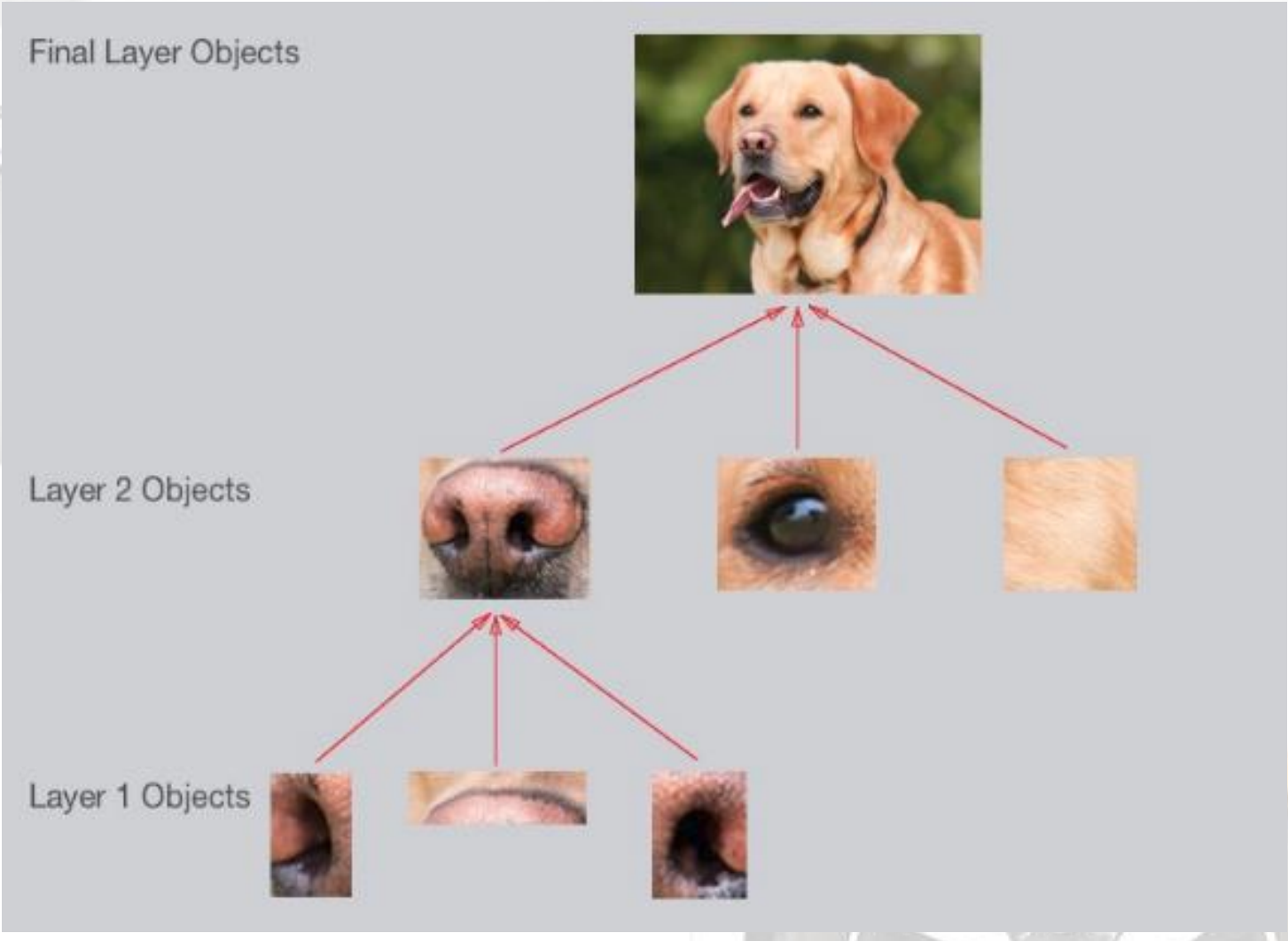
    - Eyes

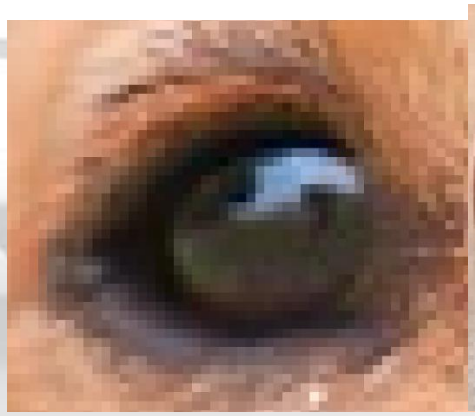
    - Fur

- We essentially bread up the image into smaller pieces, recognize the smaller pieces, and then combine those pieces to get an idea of the overall dog.

- Going One Step Further:

- But let's take this one step further. How do we determine what exactly a nose is? A Golden Retriever nose can be seen as an oval with 2 black holes inside it.





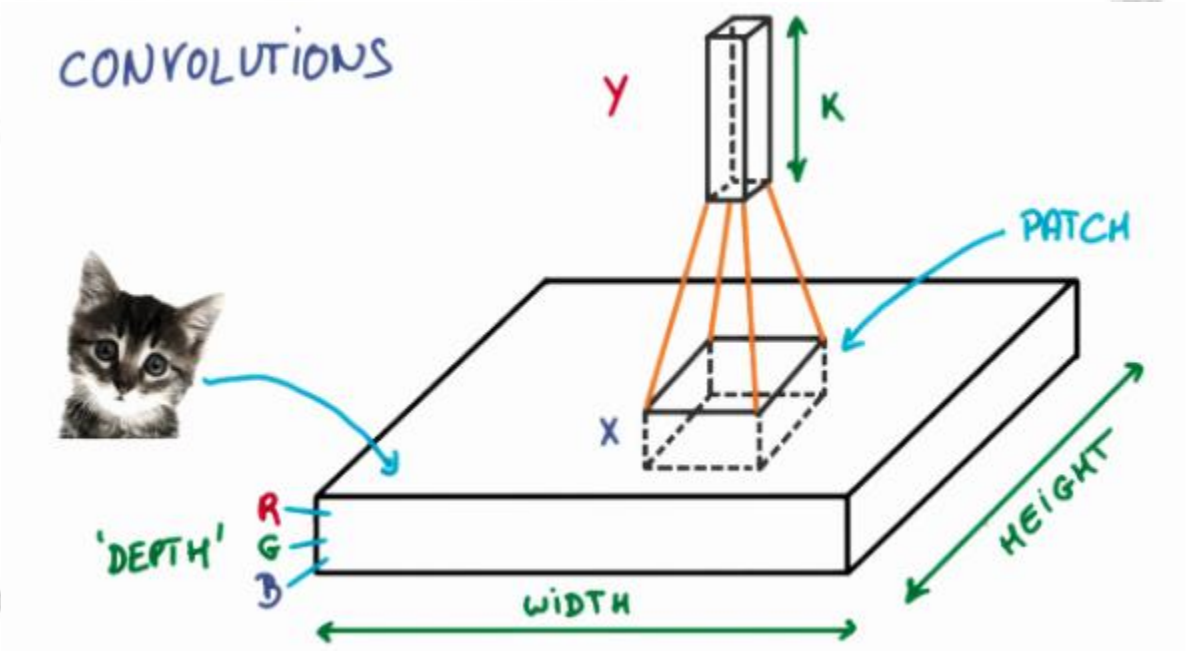
Thus, one way of classifying a Retriever's nose is to break it up into smaller pieces and look for black holes and curves that define an oval as shown below.

- Broadly speaking, this is what CNN learns to do. It learns to recognize basic lines and curves, then shapes and blobs, and then increasingly complex objects within the image. Finally, the CNN classifies the image by combining the larger, more complex objects.

NOTE: we never though program the CNN with information about specific features to look for, it learns on its own!

3.2.6: Filters:

- The first step for a CNN is to break up the image into smaller pieces. We do this by selecting a width and height that defines a filter.

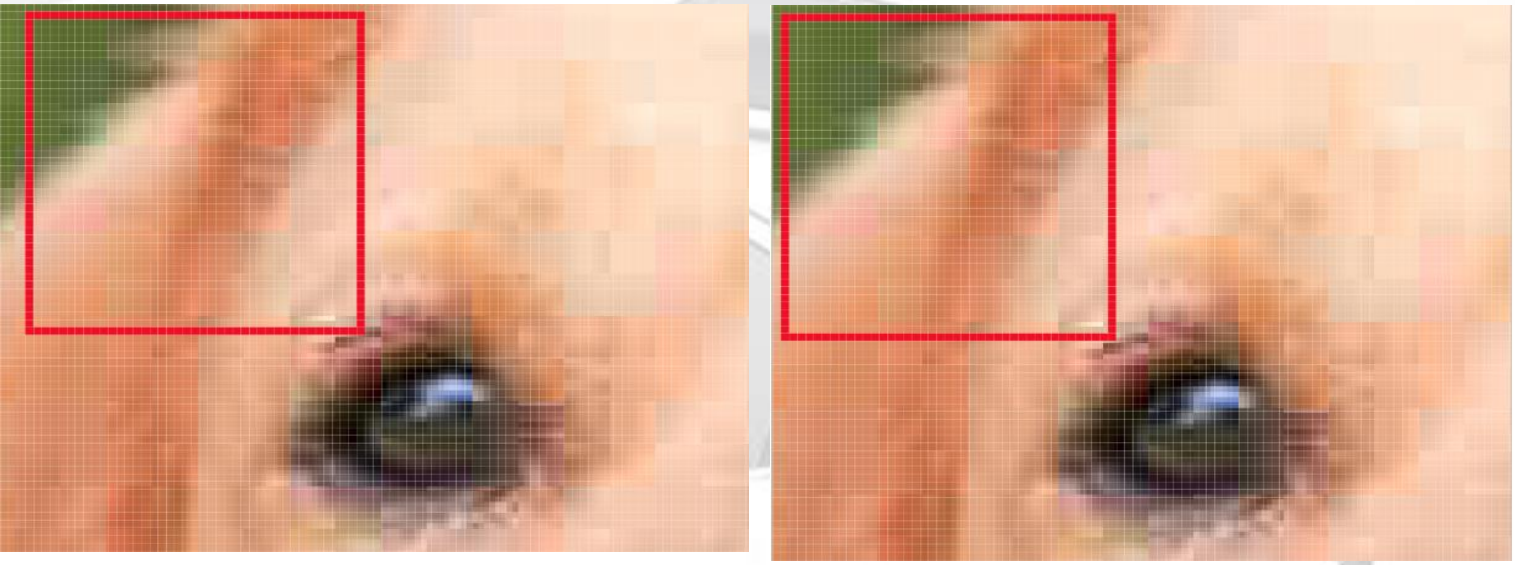


- The filter looks at small pieces, or patches, of the image. These patches are the same size as the filter.

- We then simply slide this filter horizontally or vertically to focus on a different piece of the image.

- The amount by which the filter slides is referred to as the 'Stride'. The stride is a hyper parameter (that we should tune).

- Increasing the stride reduces the size of your model by reducing the number of total patches each layer observes. However, this usually comes with a reduction in accuracy.



- Let's look at an example. In this zoomed in the image of the dog, we first start with the patch outlined in red. The width and height of our filter define the size of this square.

- We then move the square over to the right by a given stride (2 in this case) to get another patch.

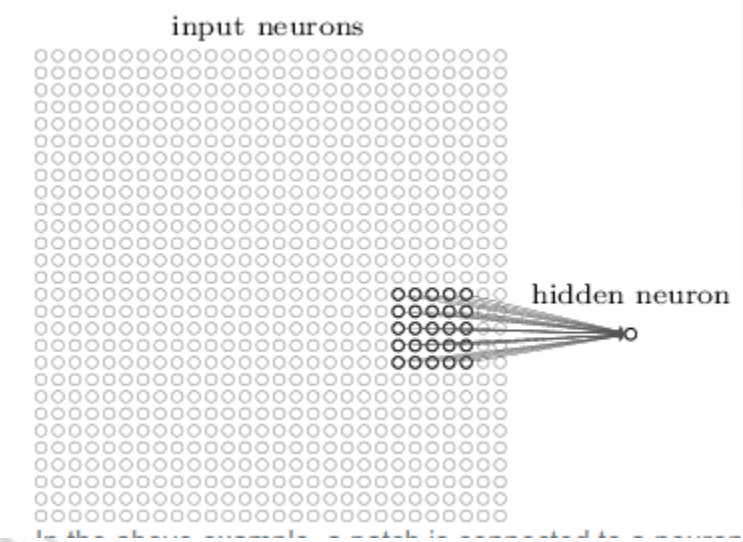
- What's important here is that we are grouping together adjacent pixels and treating them as a collective.

- In a normal, non-convolutional neural network, we would have ignored this adjacency. In a normal network, we would have connected every pixel in the input image to a neuron in the next layer. In doing so, we would not have taken advantage of the fact that pixels in an image are close together for a reason and have special meaning.

- By taking advantage of this local structure, our CNN learns to classify local patterns, like shapes and objects, in an image.

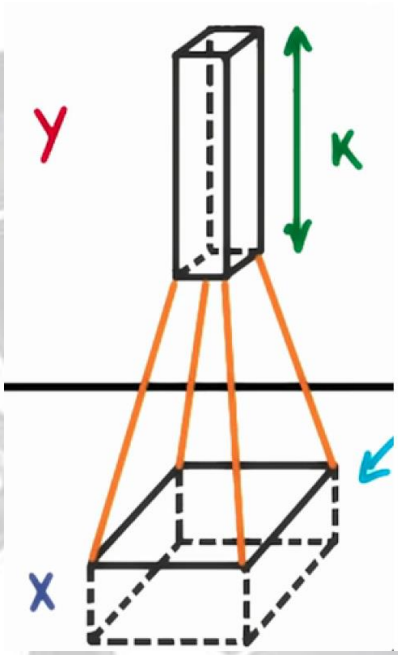
- Filter Depth:

- It's common to have more than one filter. Different filters pick up different qualities of a patch. For example, one filter might look for a particular color, while another might look for a kind of object of a specific shape. The amount of filters in a convolutional layer is called the filter depth.



- How many neurons does each patch connect to?

- That's dependent on our filter depth. If we have a depth of `K`, we connect each patch of pixels to `K` neurons in the next layer. This gives us the height of `K` in the next layer, as shown below. In practice, `K` is a hyper parameter we tune, and most CNNs tend to pick the same starting values.



- But why connect a single patch to multiple neurons in the next layer? Isn't one neuron good enough?

- Multiple neurons can be useful because a patch can have multiple interesting characteristics that we want to capture.

- For example, one patch might include some white teeth, some tongue whiskers, and part of a red tongue. In that case, we might want a filter depth of at least three – one for each teeth, whiskers, and tongue.

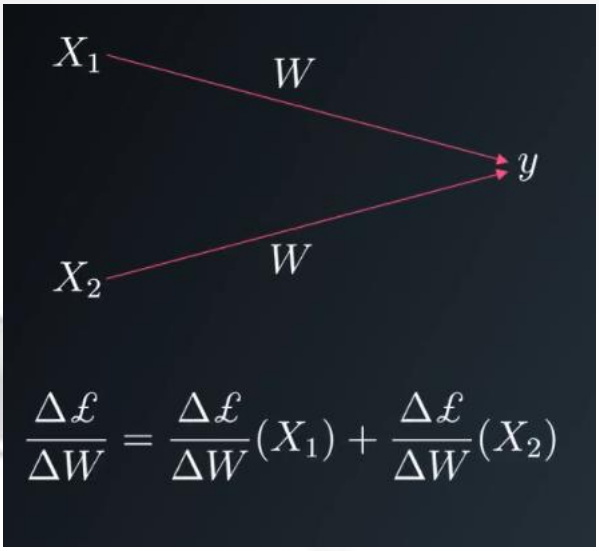
3.2.7: Feature Map Size:

3.2.8: Convolutions Cont.:

- You may be wondering what happens to the chain rule, when we use shared weights like this?

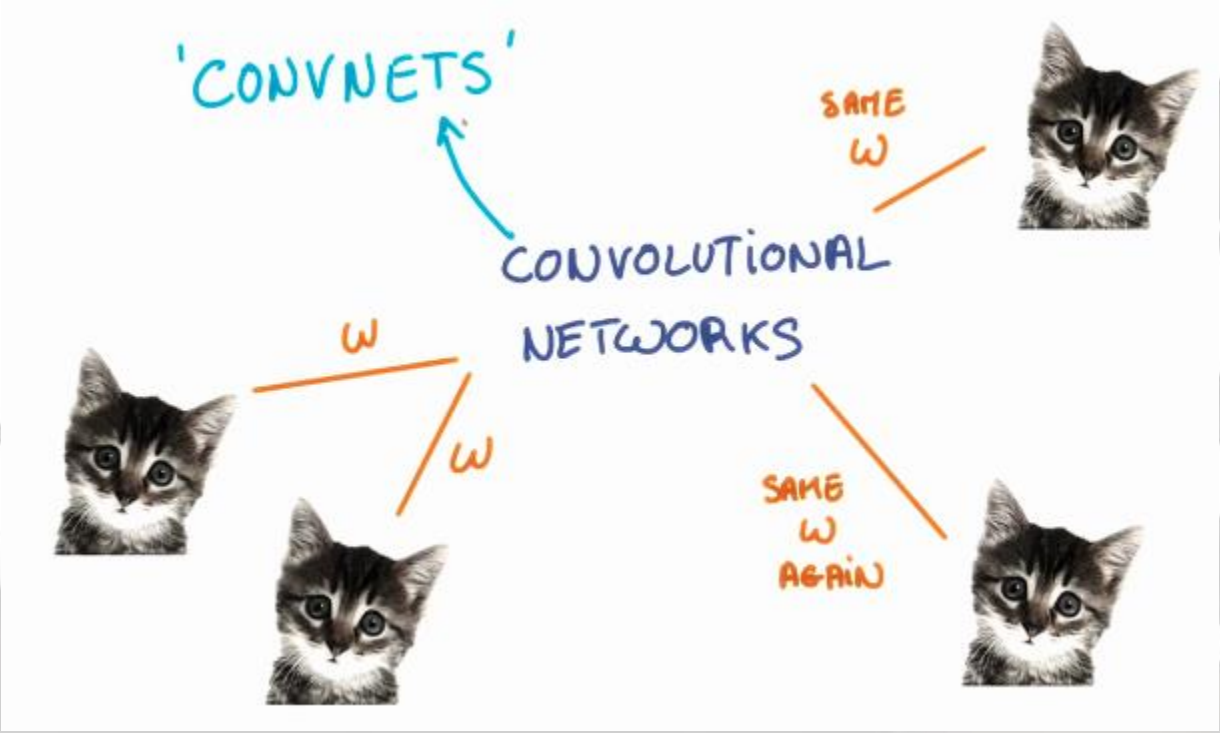
- Nothing really happens, the math just works.

- You just add up the derivatives for all the possible locations on the image.



Note: a "Fully Connected" layer is a standard, non-convolutional layer, where all inputs are connected to all output neurons. This is also referred to as a "dense" layer, and is what we used in the previous 2 lessons.

3.2.9: Parameter Sharing:



- When we are trying to classify a picture of a cat, we don't care where in the image a cat is. It's still a cat in our eyes.

- We would like our CNNs to also possess this ability known as Translation invariance. How can we achieve this?

- If we want a cat that's in the top left patch to be classified in the same way as a cat in the bottom right patch, we need the weights and biases corresponding to those patches to be the same, so that they are classified the same way!

- This is exactly what we do in CNNs. The weights and biases we learn for a given output layer are shared across all patches in a given input layer.

- Note that as we increase the depth of our filter, the number of weights and biases we have to learn still increases, as the weights aren't shared across the output channels.

- There's an additional benefit to sharing our parameters. If we did not reuse the same weights across all patches, we would have to learn new parameters for every single patch and hidden layer neuron pair.

- This does not scale well, especially for higher fidelity images. Thus, sharing parameters not only helps us with translation invariance, but also gives us a smaller, more scalable model.

- How can we calculate the number of neurons of each layer in our CNN?

- Given:

    - Our input layer has a width of W and a height of H.

    - Our convolutional layer has a filter size F.

    - We have a stride of S.

    - A padding of P.

- And the number of filters `K`,

- Then:

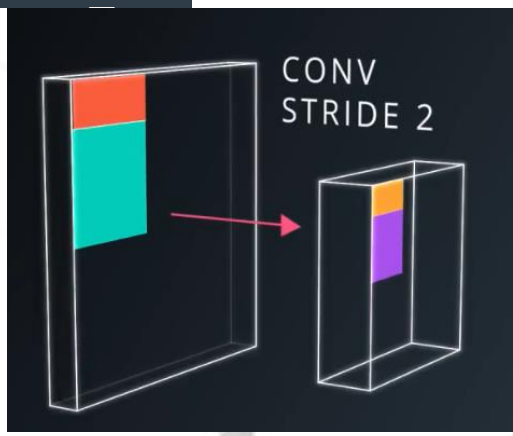
  - The width of the next layer: `W_out = [ (W-F+2P)/S ] + 1`

  - The output height would be: `H_out = [ (H-F+2P)/S ] + 1`

  - The output depth would be: `D_out = K`

  - The output volume would be: `W_out * H_out * D_out`

3.2.10: Explore the Design Space:



- The first improvement is a better way to reduce the spatial extent of your future maps in the convolutional pyramid.

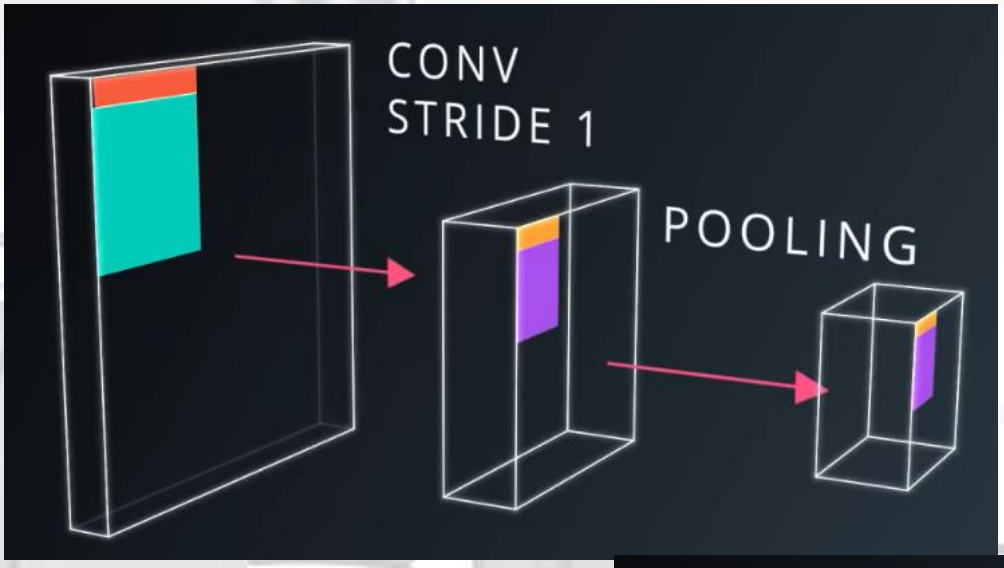
- Until now, we've used striding to shift the filters by a few pixels each time and reduce the feature map size.

- This is a very aggressive way to down sample an image. It removes a lot of information.

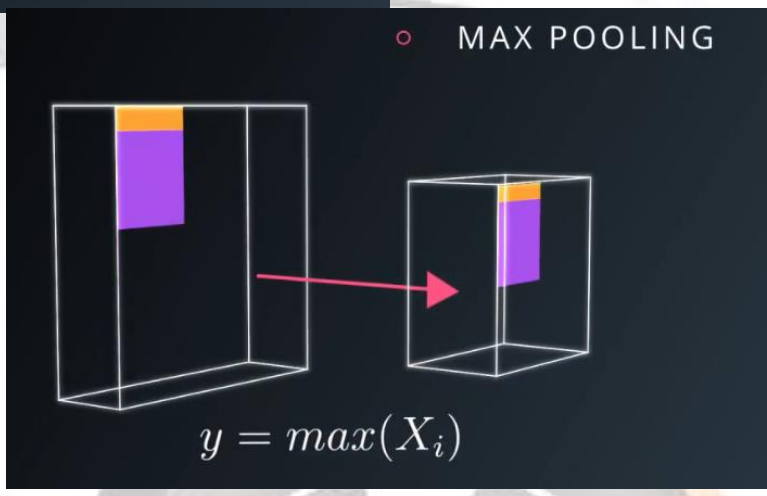
- What if instead of skipping one in every two convolutions, we still run with a very small stride, say for example, one but then took all the convolutions in a neighborhood and combine them somehow?

- That operation is called “pooling”, and there are a few ways to go about it.

- The most common is the max pooling.



- At every point of on the feature map, look at a small neighborhood around that point and compute the maximum of all the responses around it.

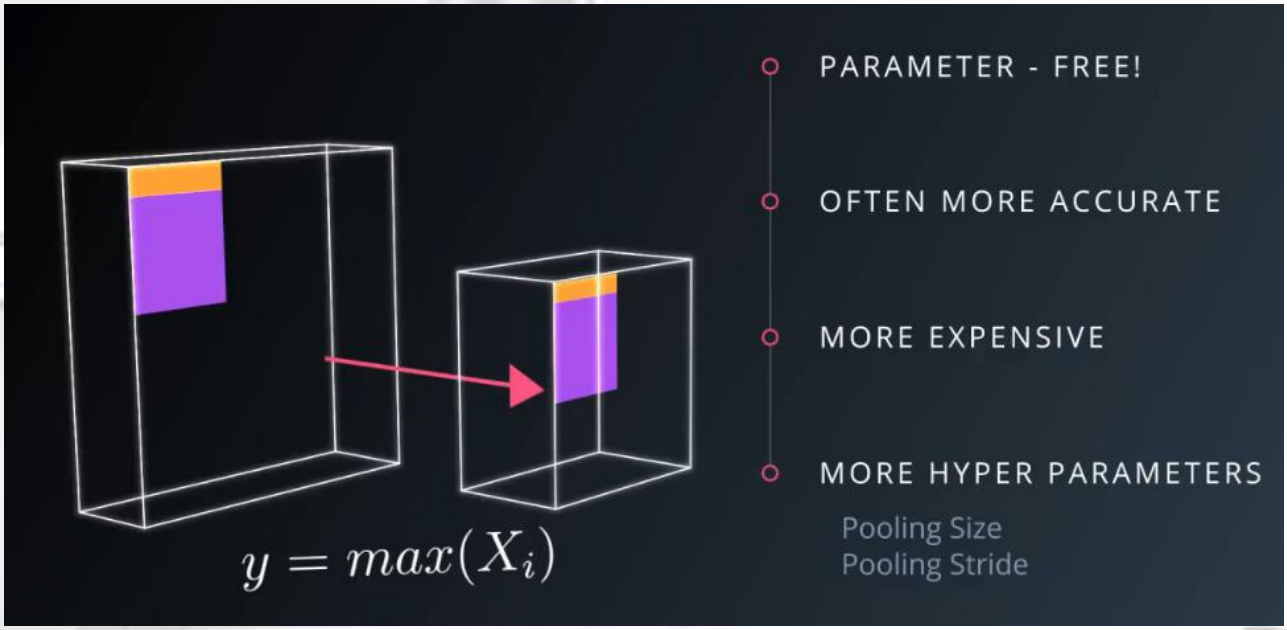


- There are some advantages to using “`max pooling`”.

- First, it doesn’t add to your number of parameters, so you don’t risk an increase in over fitting.

- Second, it simply often yields a more accurate model. However, since the convolutions that went below run at lower stride, the muddle then becomes a lot more expensive to compute.

- And now, you have even more hyper parameters to worry about, the `pooling region size` and the `pooling g stride`. And no, they don't have to be the same.



- A very typical architecture for a covnet is a few layers alternating convolutions and max pooling, followed by a few fully connected layers at the top.

- The first famous model to use this architecture was "LENET-5" designed by Yan Lecun to do character recognition back in 1998.

- Modern convolutional networks, such as AlexNet, which famously won the competitive ImagNet object recognition challenge in 2012, used the same architecture with a few wrinkles.

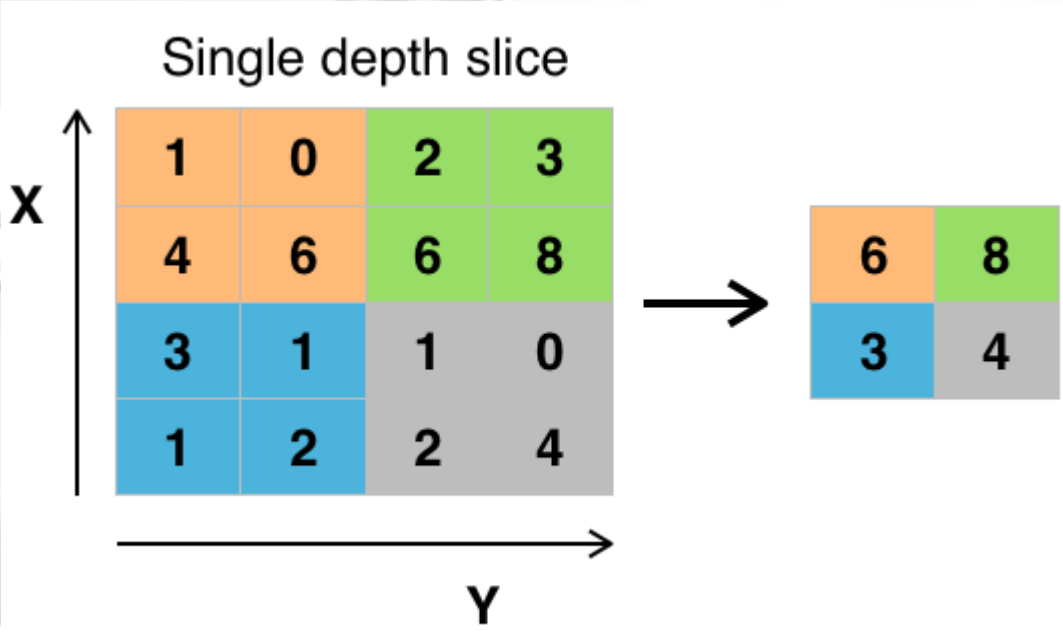
- Another notable form of pooling is average pooling.

- Instead of taking the max, just take an average over the window of pixels around a specific location.

- It's a little bit like providing a blurred low resolution view of the feature map below.

3.2.13: TensorFlow Max Pooling:

- The image above is an example of max pooling with a 2x2 filter and stride of 2. The four 2x2 colors represent each time the filter was applied to find the maximum value.



- Conceptually, the benefit of the max pooling operation is to reduce the size of the input, and allow the neural network to focus on only the most important elements. Max pooling does this by only retaining the maximum value for each filtered area, and removing the remaining values.

3.2.15: Pooling Mechanics:

Setup

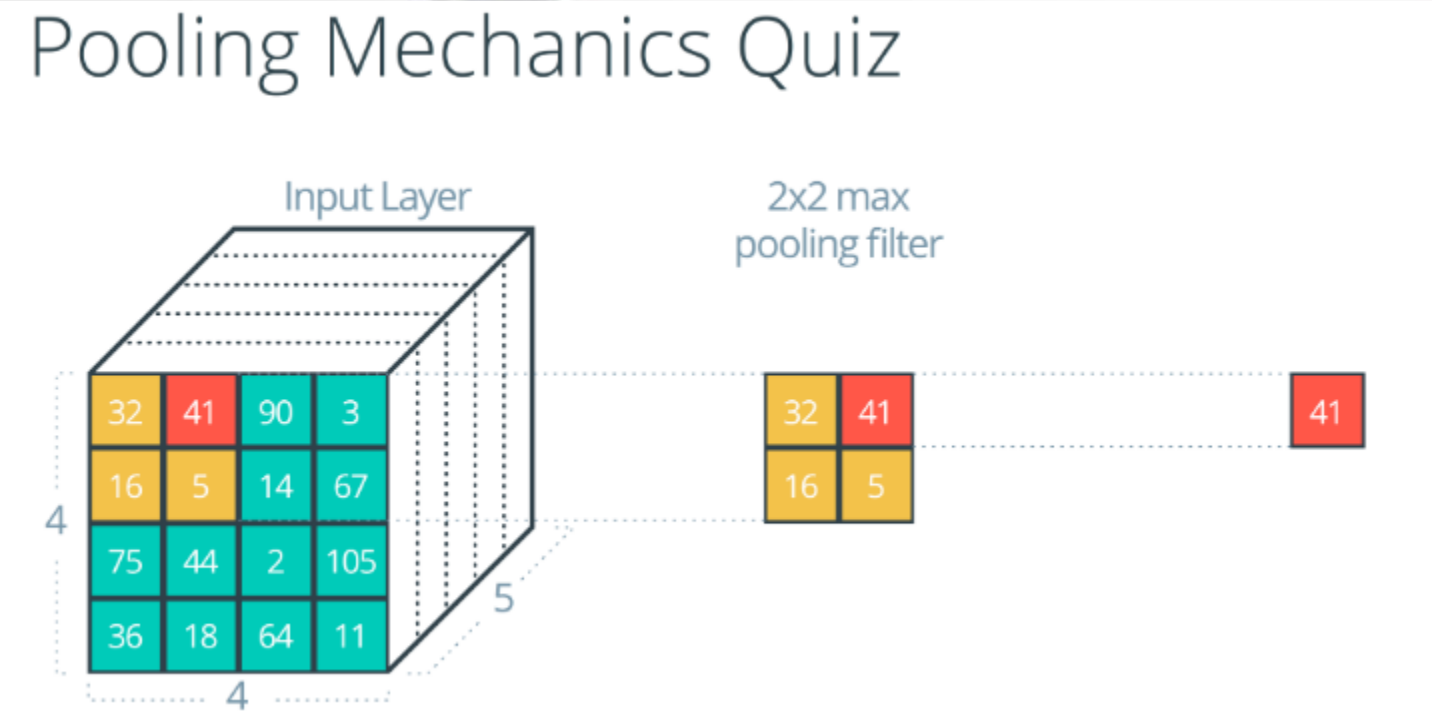
H = height, W = width, D = depth

- We have an input of shape 4x4x5 (HxWxD)
- Filter of shape 2x2 (HxW)
- A stride of 2 for both the height and width (S)

Recall the formula for calculating the new height or width:

```
new_height = (input_height - filter_height)/S + 1
new_width = (input_width - filter_width)/S + 1
```

NOTE: For a pooling layer the output depth is the same as the input depth. Additionally, the pooling operation is applied individually for each depth slice.



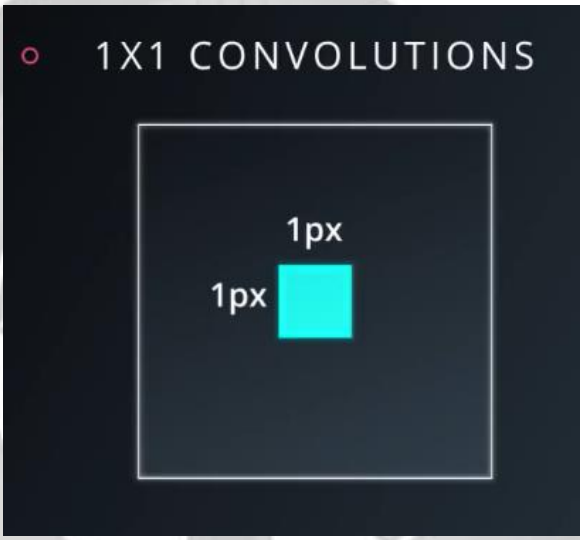
- The shape of the output 2x2x5.

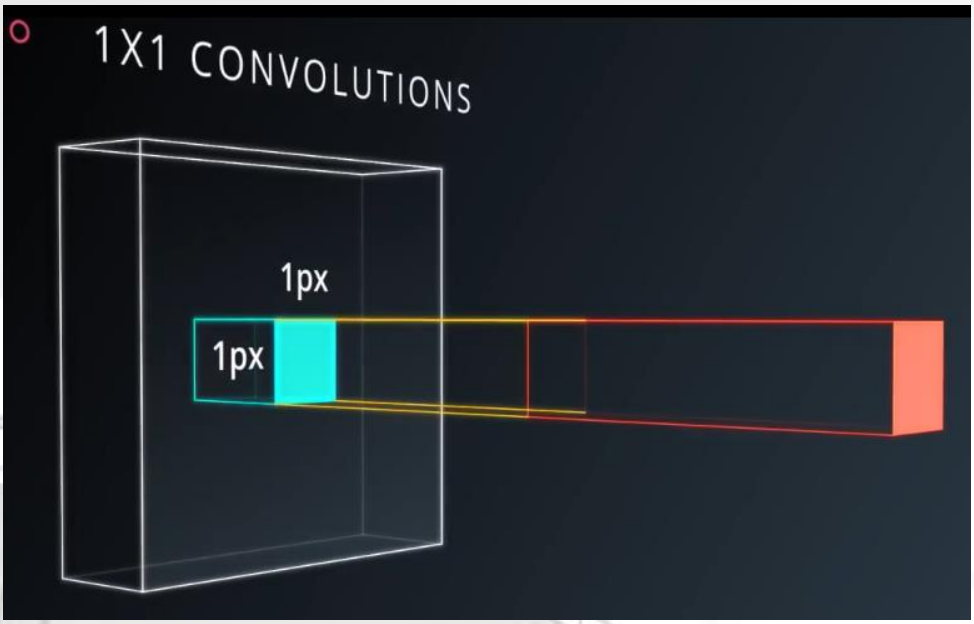
3.2.16: 1x1 Convolutions:

- But first, I want to introduce you to another idea.

- It's the idea of 1x1 convolutions. You might wonder, why one would ever want to use 1x1 convolutions?

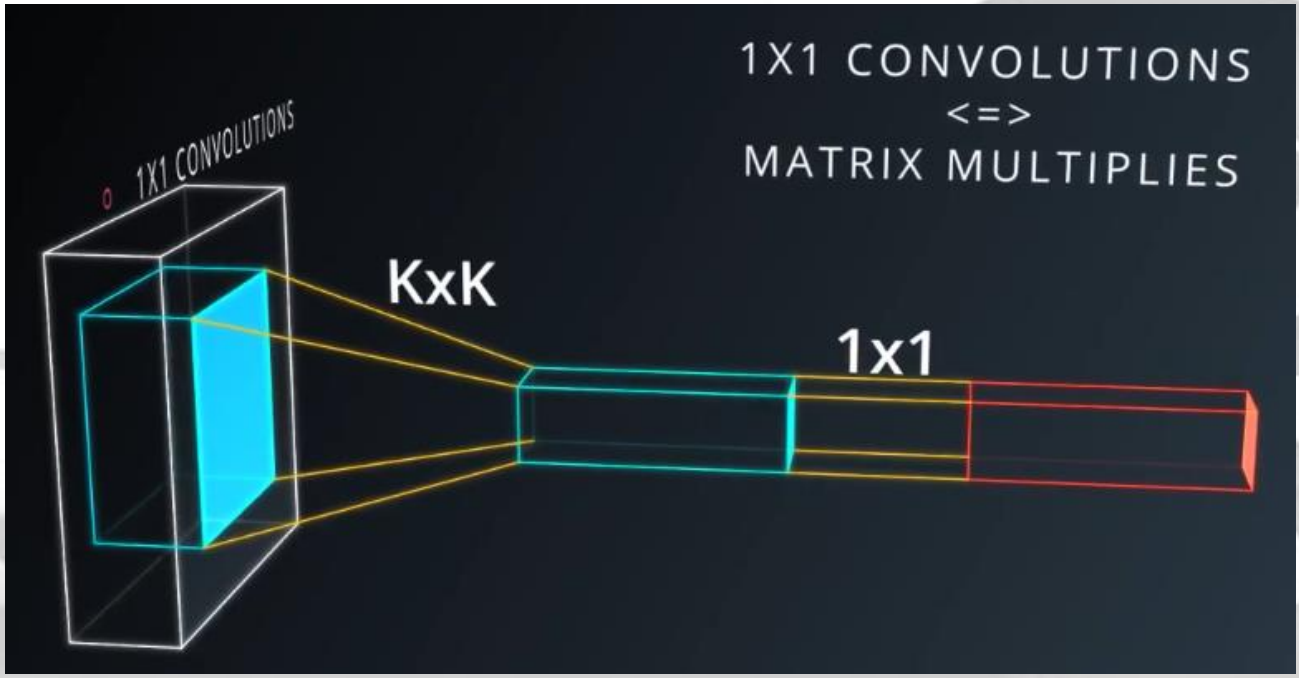
- They're not really looking at a patch of the image just that one pixel.





- Look at the classic convolution setting, it's basically a small classifier for a patch of the image but it's only a linear classifier.

- But if you add a 1x1 convolution in the middle, suddenly you have a mini neural network running over the patch instead of linear classifier.



- Interspersing your convolutions with 1x1 convolutions is a very inexpensive way to make your models deeper and have more parameters, without completely changing their structure.

- They're also very cheap because if you go through their math, they're not really convolutions at all, they're really just and they have relatively few parameters.

- I mentioned all of these, average pooling and 1x1 convolutions because I want to talk about the general strategy that has been very successful at creating cognates that are both smaller and better than cognates that simply use a pyramid of convolutions.

3.2.17: Inception Module:

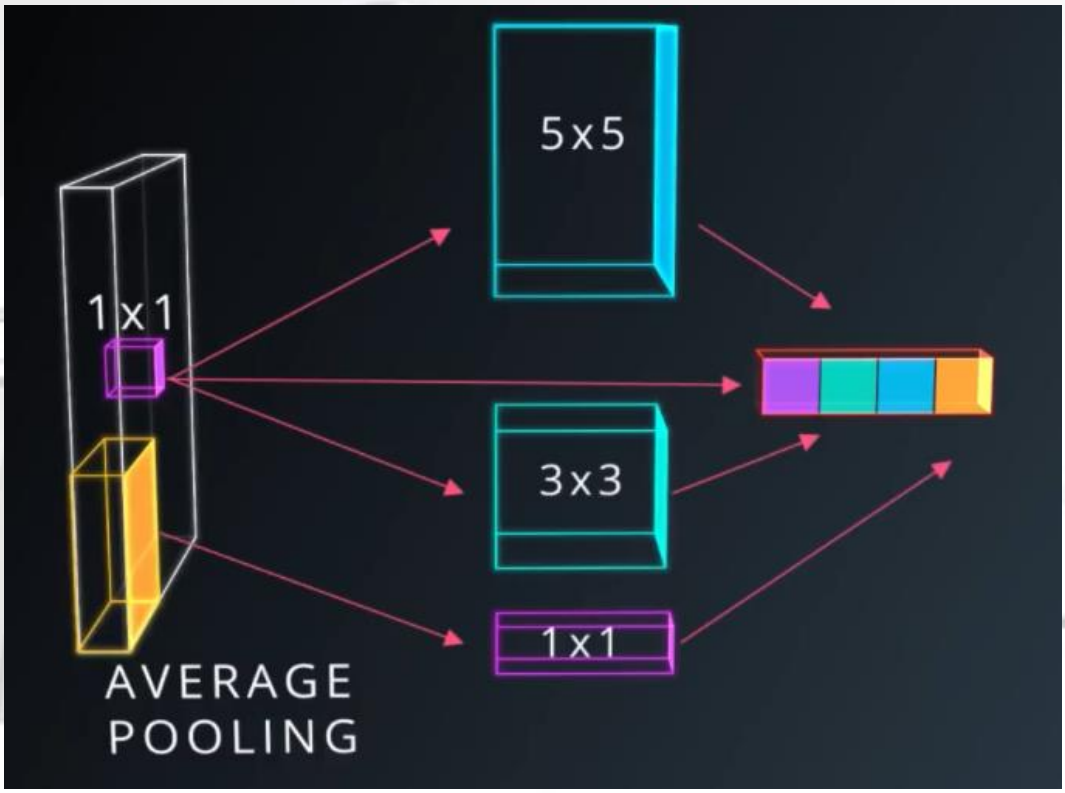
- It's called an inception module, it's going to look a little more complicated.

- The idea is that at each layer of your convent, you can make a choice having a pooling operation, have a convolution.

- And then you need to decide, it is a 1x1 or a 3x3 or a 5x5.

- All of these area actually beneficial to the modeling power of your network, so why choose?

- Let's use them all!



-Here's what an inception module looks like. Instead of having a single convolution, you have a composition of average pooling followed by 1x1, then a 1x1 convolution then a 1x1 followed by 3x3. Then a 1x2 followed by a 5x5.

- And at the top, you simply concatenate the output of each of them. It looks complicated.

- But what's interesting is that you can chose these parameters in such a way that the total number of parameters in your model is very small.

- Yet the model performs better than if you had a simple convolution.

3.3) Transfer Learning:

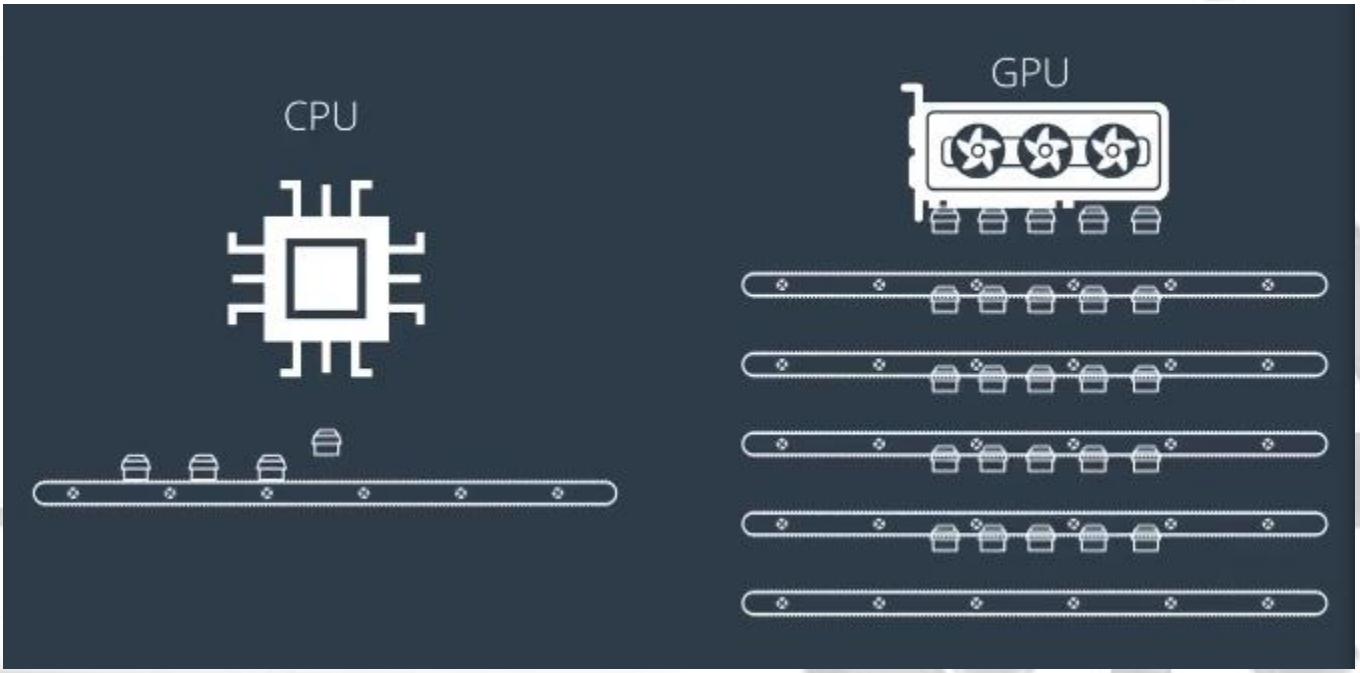
3.3.1: GPU & CPU:

- You might think of GPUs as devices built for rendering graphically intensive video games and that's true but GPUs have also become extraordinarily important for deep learning.

- GPUs are optimized for high throughput computation whereas CPUs are mostly optimized for latency running a single thread of instructions as quickly as possible.

- GPUs are optimized for throughput running as many simultaneous computations as possible

- Throughput computing is important for computer graphics because we want to update lots of pixels on the screen at the same time and it turns out that throughput computing is also important for deep learning because the



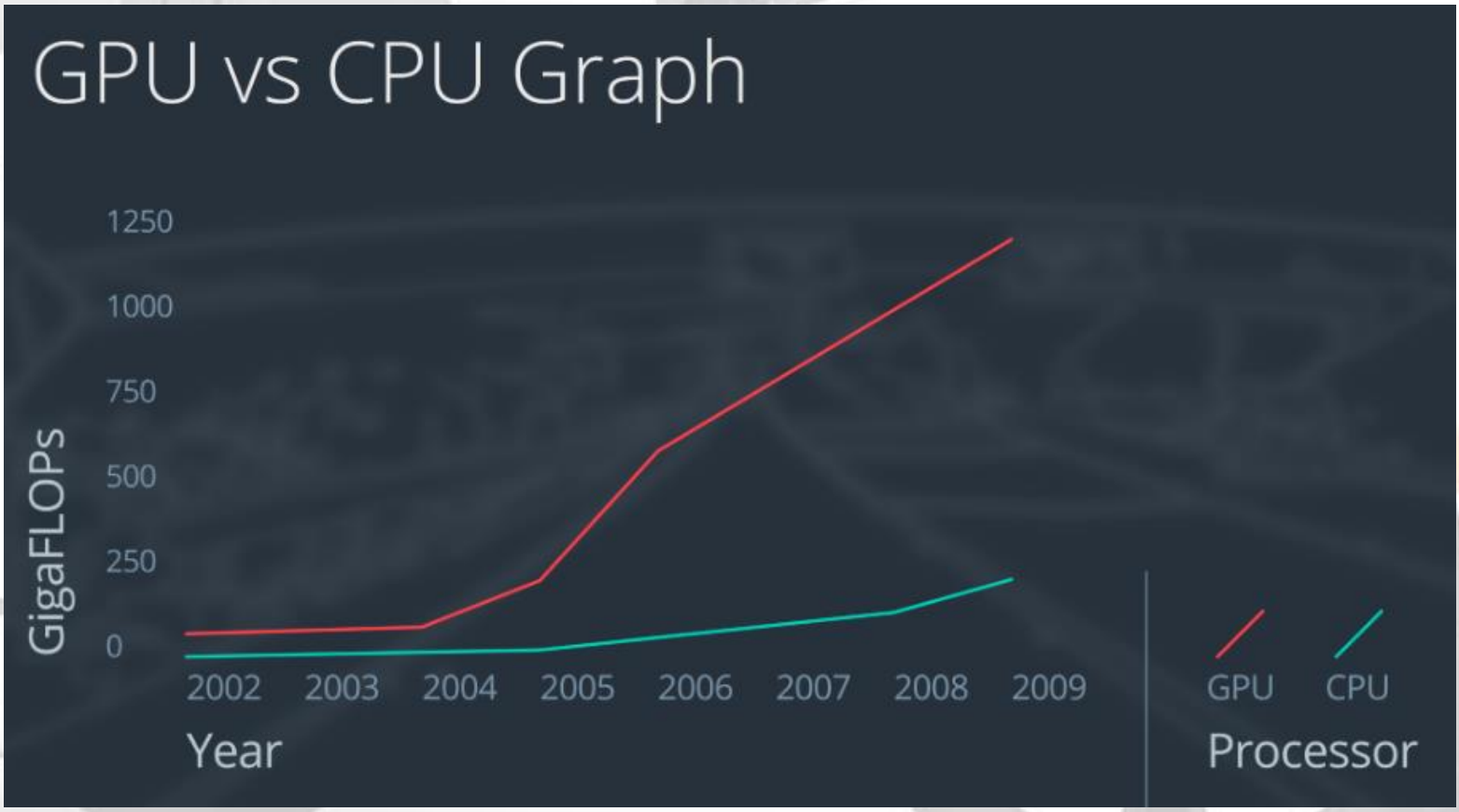
computations fundamental to deep leaning have a lot of parallelism.

- QUESTION: What level of acceleration do you typically see when you move from training a network on a CPU to a GPU?

- ANSWER: It depends on a lot of factors including how the software running has been designed and the precise CPU and GPU are comparing.

- For example, the low-power processor in your laptop is going to be much slower than a big server processor but a rule of thumb would be that Network strain about five times faster on a GPU that on a CPU.

- One point though, is to use transfer learning instead of starting from scratch, you start from trained model to begin you training with and you can change the last layer to match you needs.



3.3.2: Transfer Learning:

- When you're tackling a new problem with a neural network, it might help to start with an existing network that was built for a similar task and then try to fine-tune it for your own problem.

- There are a couple of good reasons to do this.

- First, existing neural networks can be really useful. If somebody has taken days or weeks to train a network already, then a lot of intelligence is stored in that network.

- Taking advantage of that work can accelerate your own progress.

- Second, sometimes the data set for the problem you'll work on might be small. In those cases, look for an existing network that's designed for a problem similar to your own. If that network has already been trained on a larger data set, then you can use it as a starting point to help your own network generalize better.

- Transfer learning involves taking a pre-trained neural network and adapting the neural network to a new, different data set.

- Depending on both:

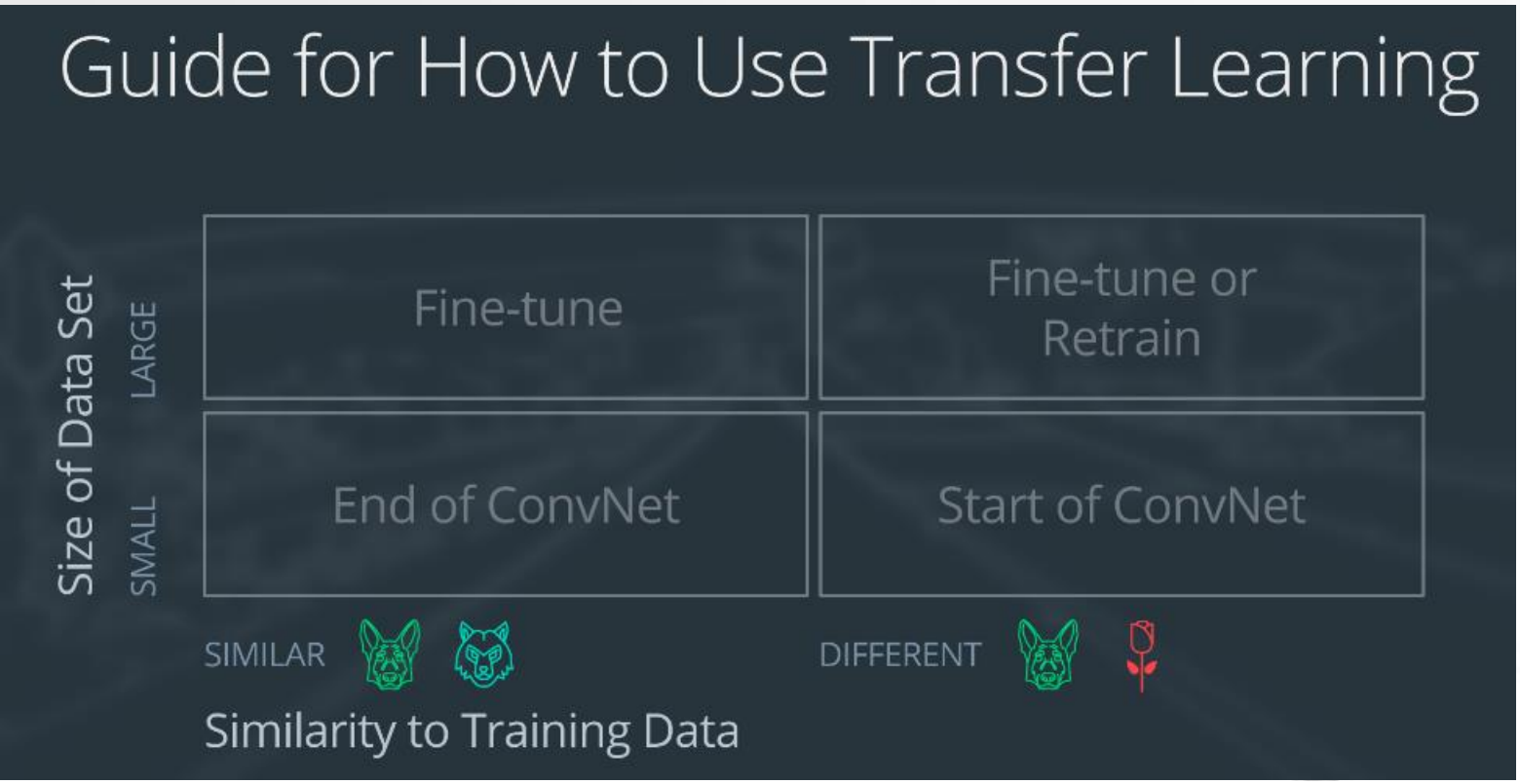
    - The size of the new dataset, and

    - The similarity of the new data set to the original data set.

- The approach for using transfer learning will be different. There are four main cases:

1. new data set is small, new data is similar to original training data
2. new data set is small, new data is different from original training data
3. new data set is large, new data is similar to original training data

4. new data set is large, new data is different from original training data



- A large data set might have one million images. A small data could have two-thousand images. The dividing line between a large data set and small data set is somewhat subjective. Over-fitting is a concern when using transfer learning with a small data set.

- Images of dogs and images of wolves would be considered similar; the images would share common characteristics. A data set of flower images would be different from a data set of dog images.

- Each of the four transfer learning cases has its own approach. In the following sections, we will look at each case one by one.

Case 1: Small Data Set, Similar Data:

- If the new data set is small and similar to the original training data:

1) Slice off the end of the neural network.

2) Add a new fully connected layer that matches the number of classes in the new data set.

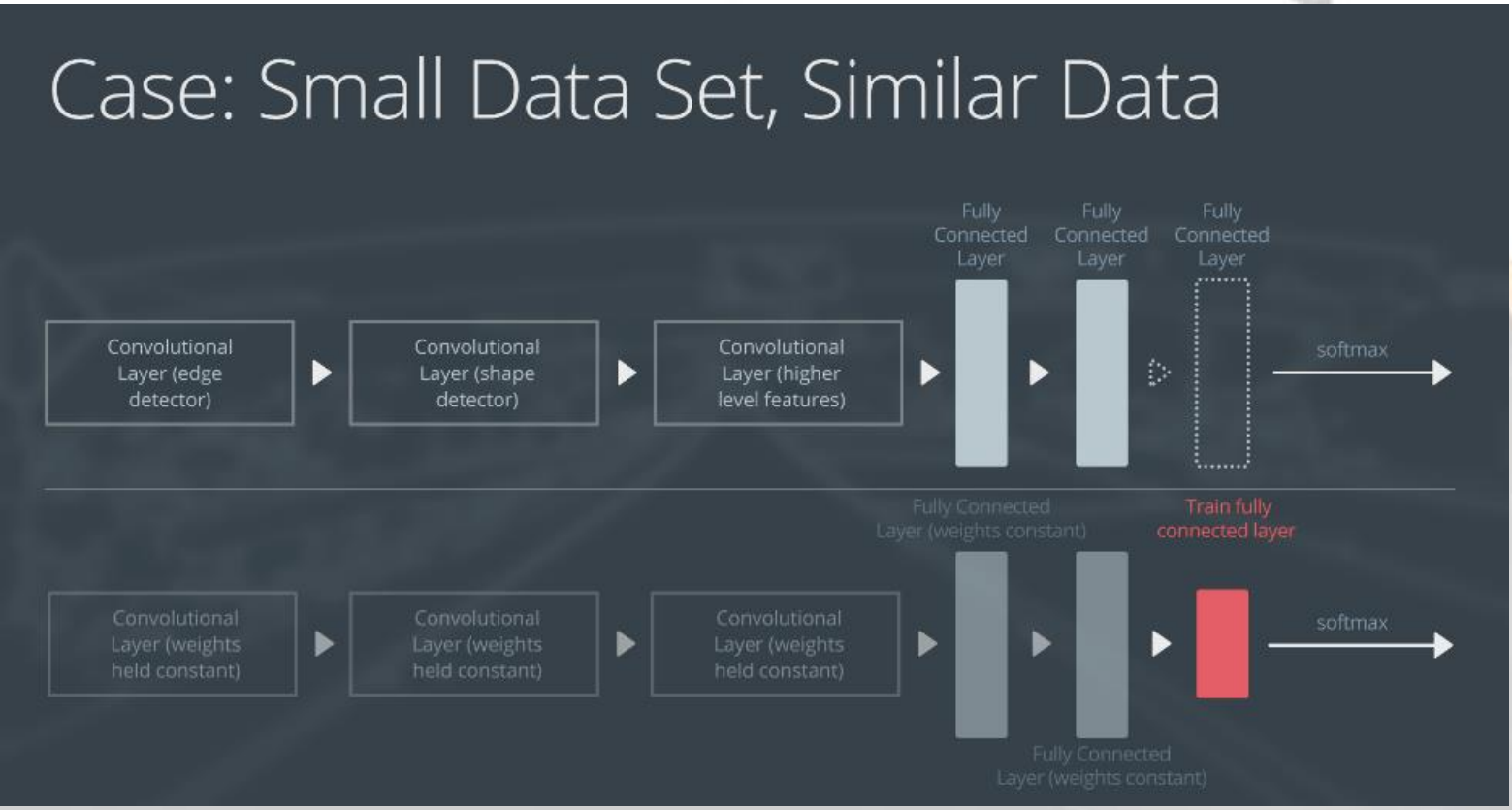
3) Randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network.

4) Train the network to update the weights of the new fully connected layer.

NOTE: To avoid over-fitting on the small data set, the weights of the original network will be held constant rather than re-training the weights.

- Since the data sets are similar, images from each data set will have similar higher level features.

Therefore most or all of the pre-trained neural network layers already contain relevant information about the new data set and should be kept.



Case 2: Small Data Set, Different Data:

- If the new data set is small and different from the original training data:

1) Slice off most of the pre-trained layers near the *beginning* of the network.

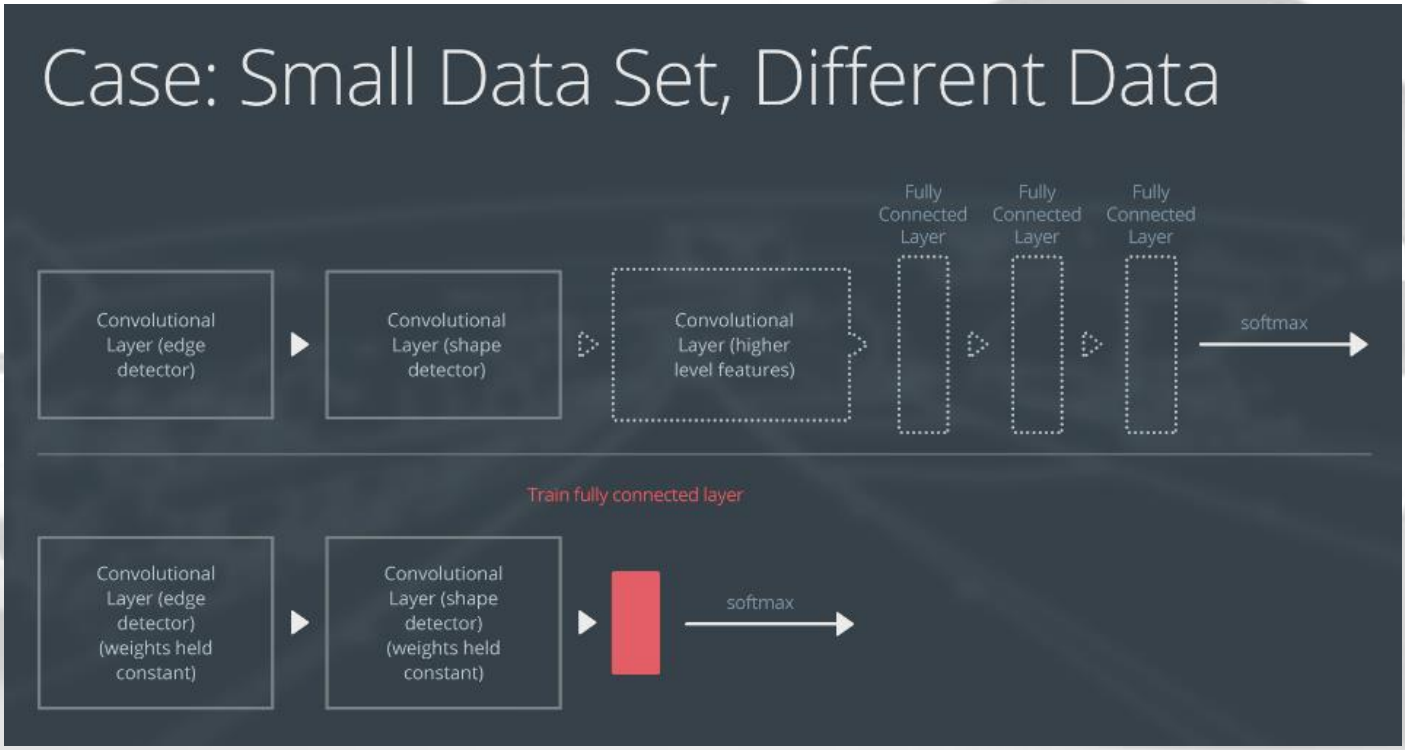
2) Add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set.

3) Randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network.

4) Train the network to update the weights of the new fully connected layer.

- Because the data set is small, over-fitting is still a concern. To combat over-fitting, the weights of the original neural network will be held constant, like in the first case.

- But the original training set and the new data set do not share higher level features. In this case, the new network will only use the layers containing *lower level features*.



Case 3: Large Data Set, Similar Data:

- If the new data set is large and similar to the original training data:

1) Remove the last fully connected layer and replace with a layer matching the number of classes in the new data set.

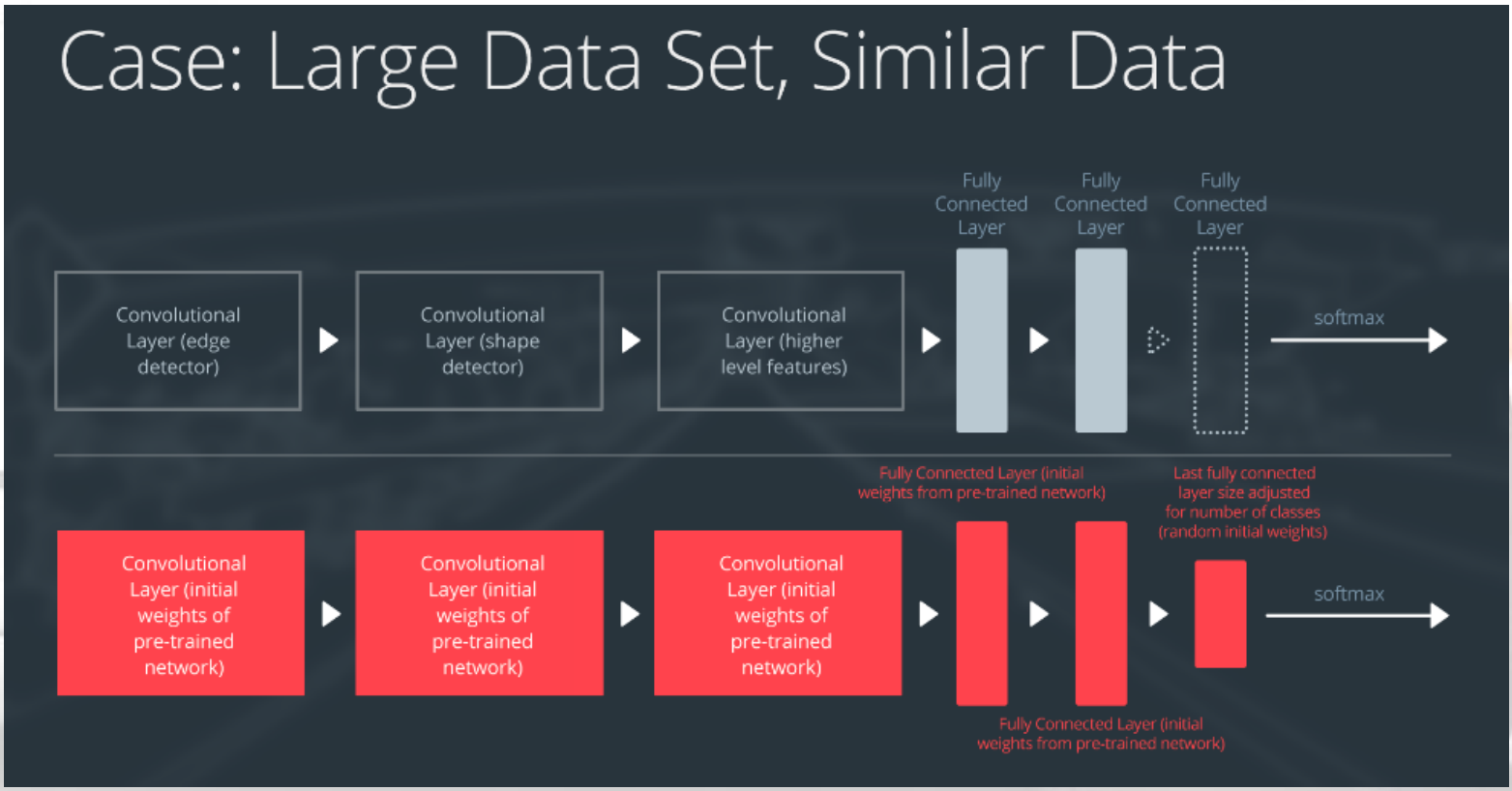
2) Randomly initialize the weights in the new fully connected layer.

3) Initialize the rest of the weights using the pre-trained weights.

4) Re-train the entire neural network.

- Because of Large Dataset, Over-fitting is not as much of a concern when training on a large data set; therefore, you can re-train all of the weights.

- Because of the original training set and the new data set share higher level features, the entire neural network is used as well.



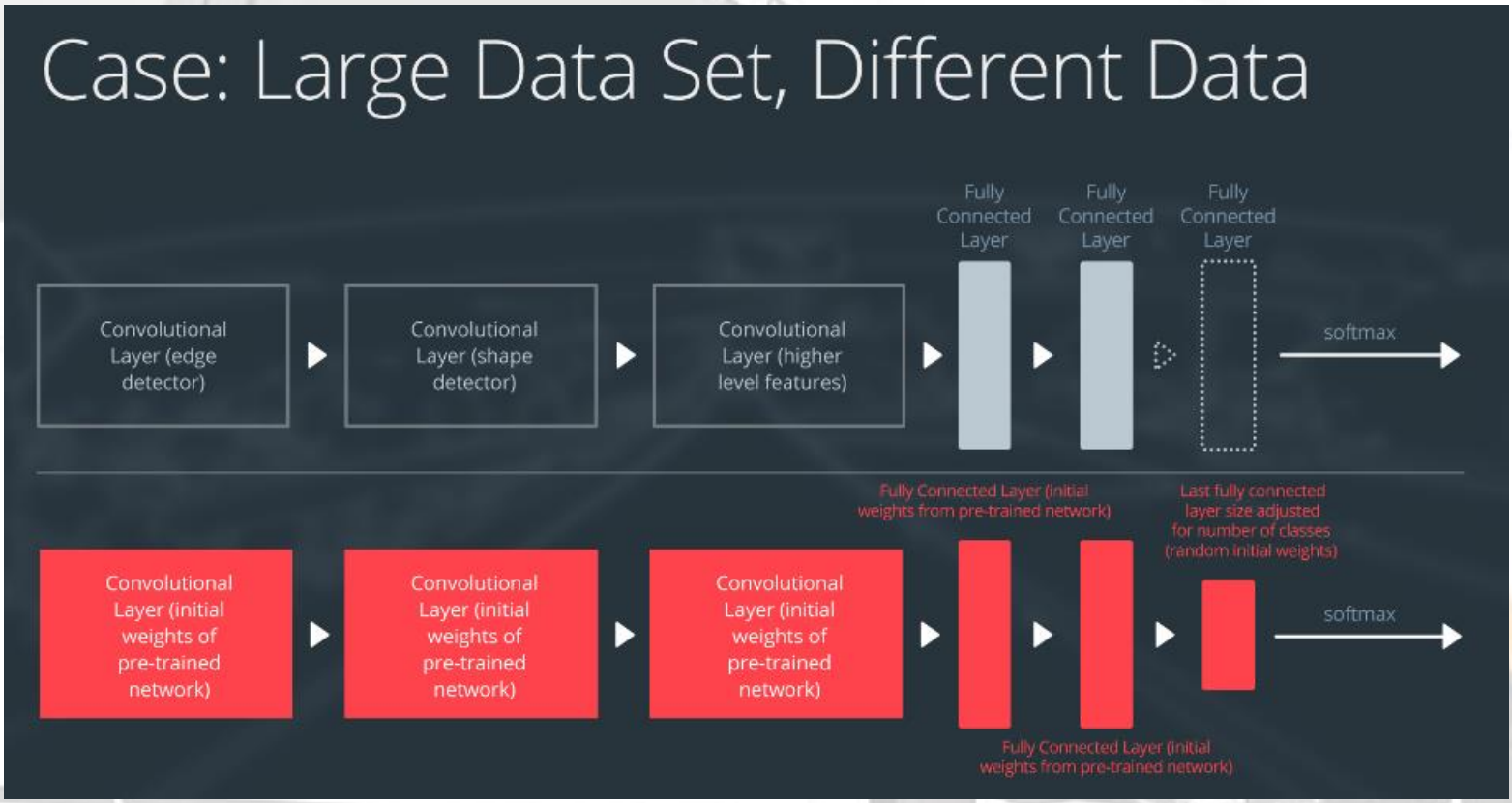
Case 4: Large Data Set, Different Data:

- If the new data set is large and different from the original training data:

1) Remove the last fully connected layer and replace with a layer matching the number of classes in the new data set.

2) Retrain the network from scratch with randomly initialized weights.

3) Alternatively, you could just use the same strategy as the "large and similar" data case.



- Even though the data set is different from the training data, initializing the weights from the pre-trained network *might* make training faster. So this case is exactly the same as the case with a large, similar data set.

- If using the pre-trained network as a starting point does not produce a successful model, another option is to randomly initialize the convolutional neural network weights and train the network from scratch.

3.3.3: Deep Learning History:

- One of the fascinating things about neural networks, is how long they've been taken to become an overnight success.

- Deep learning has only really taken off in the last five years. And that boils down to the increased availability of label data along with the greatly increased computational throughput of modern processors.

- For a long time, we didn't have the huge label data sets that we needed to make deep learning work.

- Those data sets only became widely available with the rise of the internet which made collecting and labeling huge data sets feasible.

- But even we had big data sets, we often didn't have enough computational power to make use of them.

- It's only been in the last five years that processors have gotten big enough and fast enough to train large scale neural networks.

3.3.4: ImageNet:

- "Aughts" or "Oughts" refers to the first decade of a century. "Late Aughts" refers to the latter part of the first decade of 2000's.

- By the late aughts, the Internet had made it easier to generate and collect images, storage cost had dropped so that It was cheap to save large collections of images, and services like Amazon's Mechanical Turk had even made it more cost-effective to label images.

- That confluence of factors gave rise to ImageNet, a huge dataset of hand-labeled images.

- And the ImageNet database gave rise to the ImageNet Large Scale Visual Recognition Competition.

- This competition is most famous as an annual competition , where teams from industry and academia try to build the best networks for object detection and localization.

- Kind of like "Guess Image".

- This spurred really intense competition between teams in industry and academia to produce the best image classification network.

- In 2012, the winning model really changed the field, it's called "AlexNet" and it looked a lot like Yann LeCun's neural network from way back in 1998.

3.3.5: AlexNet:

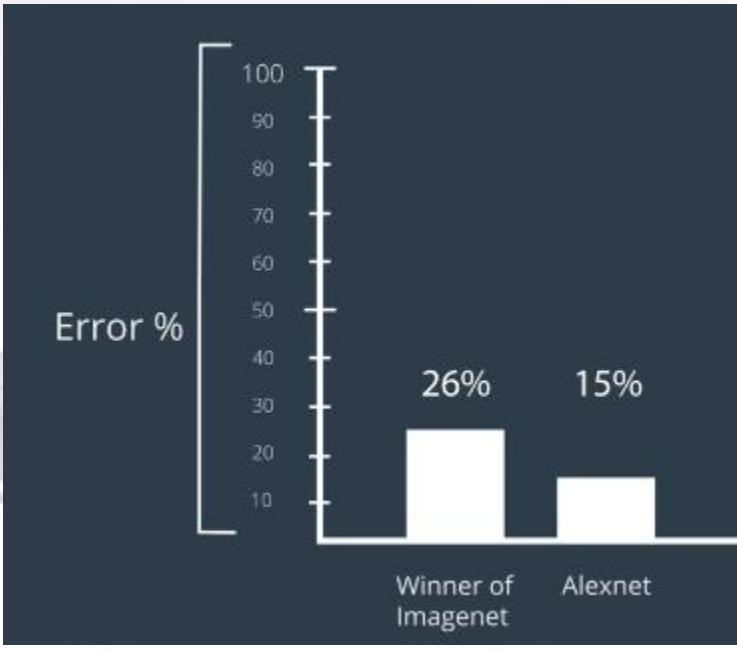
- AlexNet was developed at the University of Toronto by Alex Krizhevsky, Ilya Sutskever and their professor Geoffrey Hinton.

- Although the fundamental architecture of AlexNet resembled LeNet from 1998, AlexNet was a breakthrough in serveral respects.

- First and formost, AlexNet used the massive parallelism afforded by GPUs to accerlerate training.

- Using the best GPUs available in 2012, the AlexNet team was able to train the network in about a week.

- Additionally, AlexNet pioneered the use of Rectified Linear Units as an activation function and dropout is a technique for avoiding over-fitting.



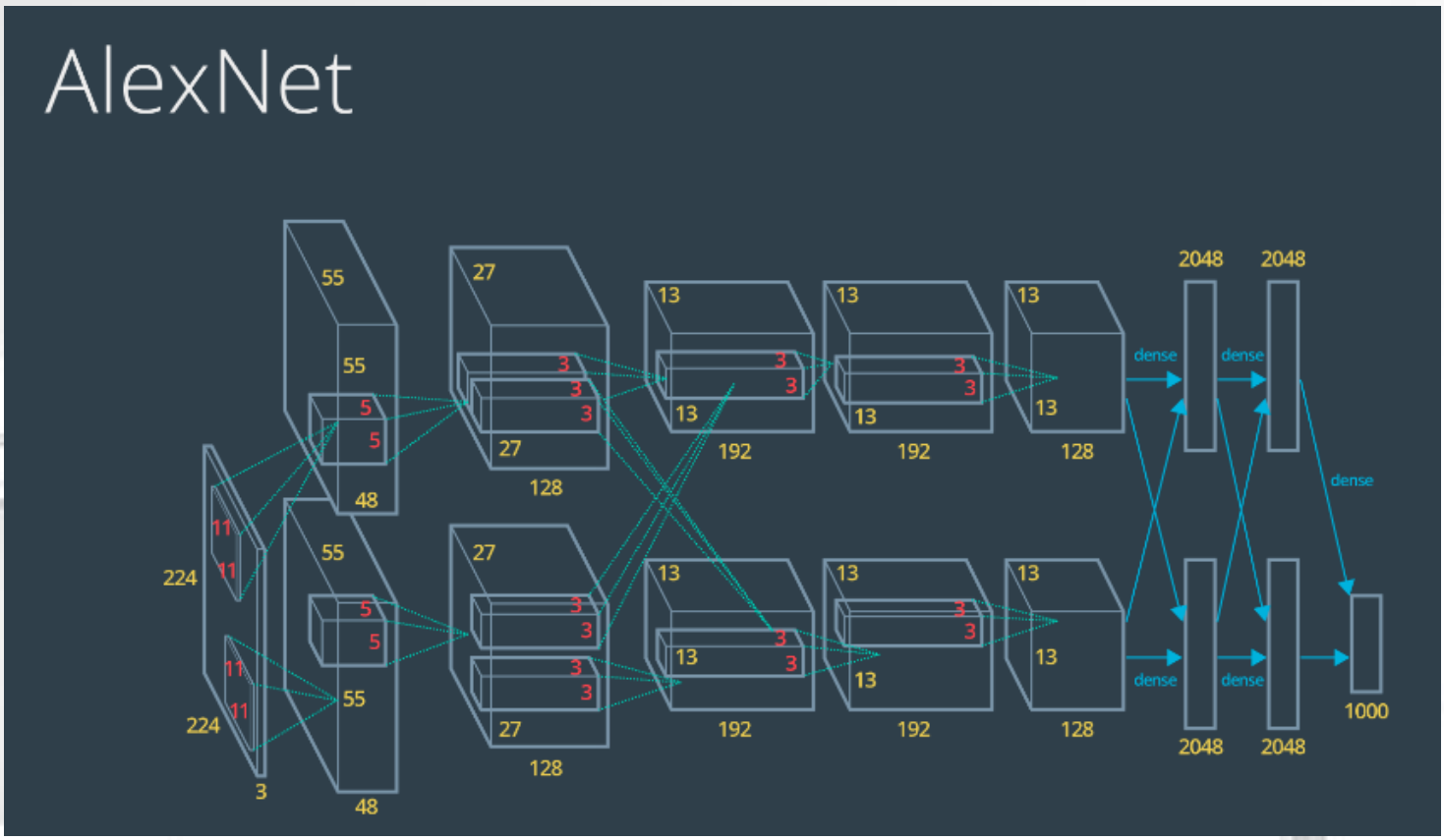
- In 2011, the year before AlexNet was developed, the winner of the ImageNet competition successfully classified 74% of images, or the terminology of the competition, its error was 26%.

- The next year AlexNet lowered its error to 15%. This was a huge leap forward.

- AlexNet puts the network on 2 GPUs, which allows for building a larger network.

- Although most of the calculations are done in parallel, the GPUs communicate with each other in certain layers.

- The original research paper on AlexNet said that parallelizing the network decreased the classification error rate by 1.7% when compared to a neural network that used half as many neurons on one GPU.

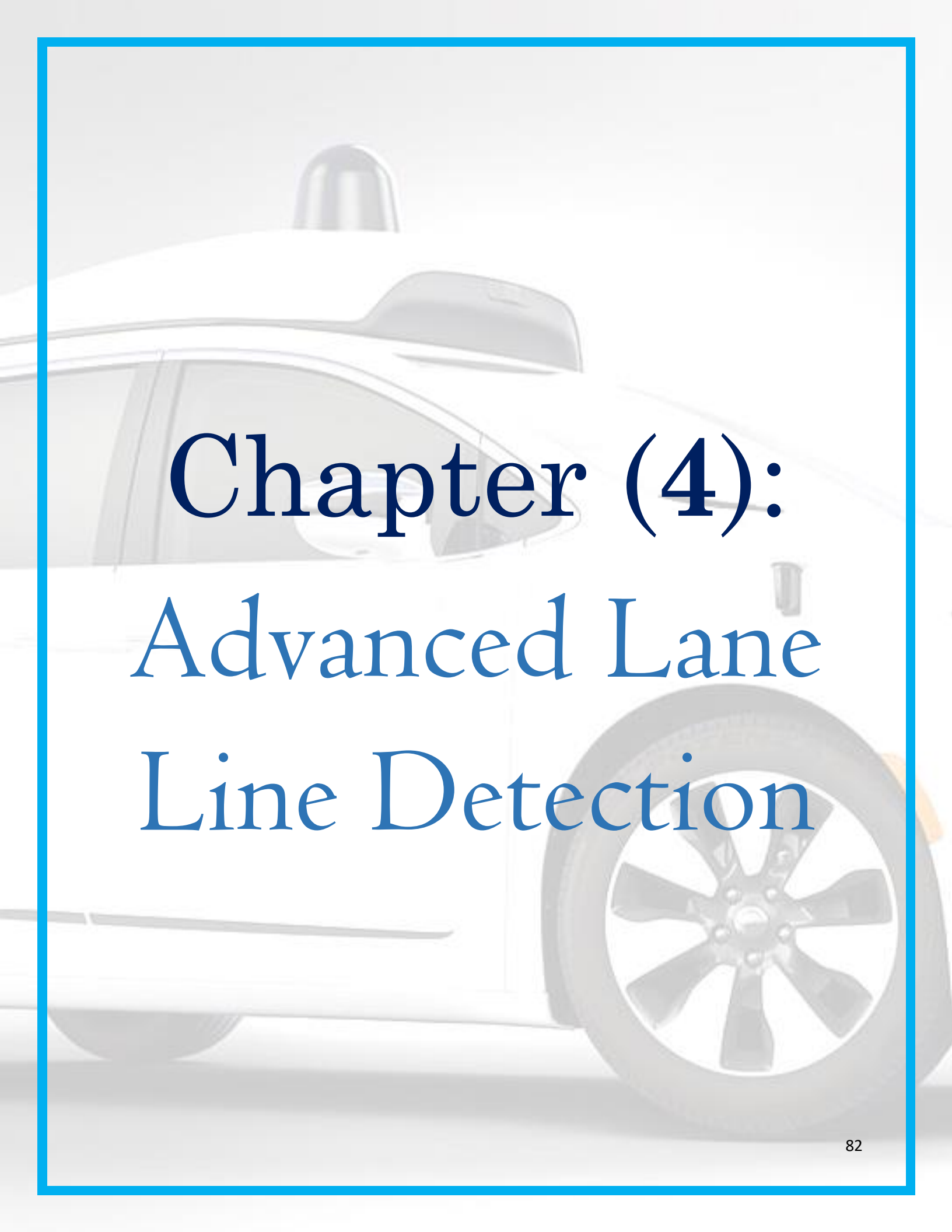


3.3.6: AlexNet:

QUESTION: How do we apply transfer learning?

- 2 popular methods are:

1) Feature Extraction: Take a pre-trained neural network and replace the final (classification) layer with a new classification layer, or perhaps even a small feed-forward network that ends with a new classification layer. During training the weights in all the pre-trained layers are frozen, so only the weights for the new layer(s) are trained. In other words, the gradient doesn't flow backwards past the first new layer.



Chapter (4):
Advanced Lane Line Detection

4.1) Camera Calibration:

4.1.1: Welcome to Computer Vision:

- Robotics can essentially be broken down into a three step cycle.

First step is to sense or perceive the world.

- The second step is to decide what to do based on that perception.

- The third step is to perform an action

to carry out that decision.

- Computer Vision is a major part of the perception step in that cycle.

- 80% of building a self-driving car is *perception*.

- Computer Vision is the art and science of perceiving and understanding the world around you through images.

- In the case of self-driving cars, computer vision helps us detect lane markings , vehicles, pedestrians, and other elements in the environment in order to navigate safely.

- So, you might ask, "*Why bother doing this at all with camera images when we have more sophisticated instruments at our disposal?*"

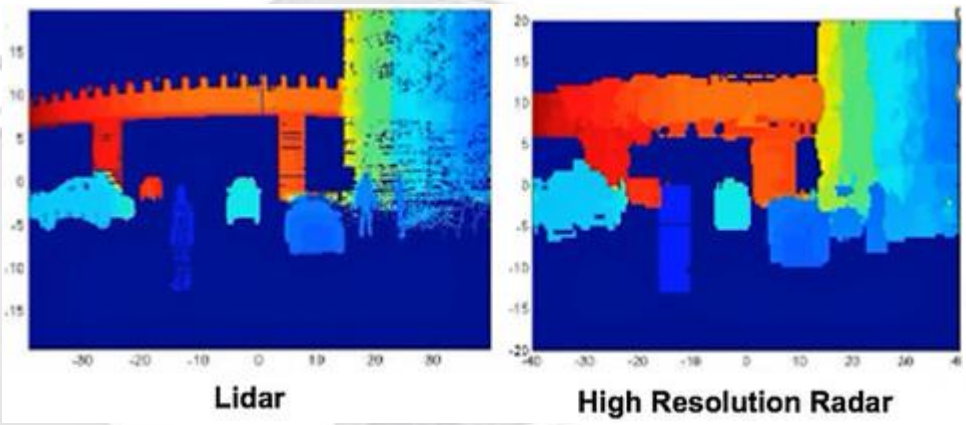
- The answer is self-driving cars employ a suit of sophisticated sensors, but humans do the job of driving with just two eyes and one good brain.

- Now let's take a closer look why we use cameras instead of other sensors might be an advantage in developing self-driving cars.

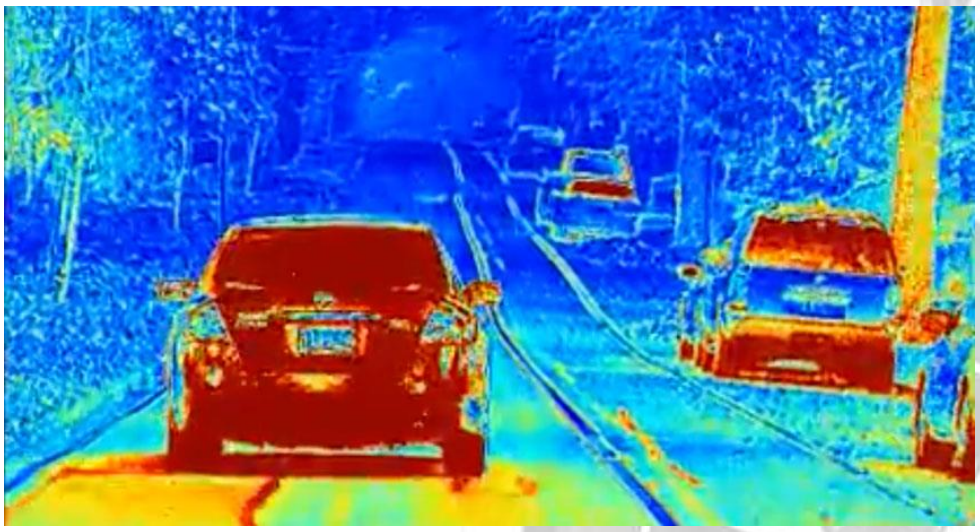
Sensors	Spatial Resolution	3D	Cost
Radar + Lidar	Low	Yes	$$$
Single Camera	High	No	$

NOTE: Spatial Resolution means how many pixels in the image?

1) Spatial Resolution in Lidar and Radar:



2) Spatial Resolution in Camera:



- Radar and Lidar see the world in 3D, which can be a big advantage for knowing where you are relative to your environment.

- A camera sees in 2D, but at much higher spatial resolution that Radar and Lidar such that it's actually possible to infer depth information from camera images.

- The big difference, however comes down to cost, where cameras are significantly cheaper.

- It's altogether possible that self-driving cars will eventually be outfitted with just a handful of cameras and a really smart algorithm to do the driving.

4.1.2: Overview:

- In the very first module of these nano-degree program, we used some basic computer vision techniques to find the lane lines on the road.

- The first thing we'll do in this module, is to write a better lane finding algorithm that deals with complex scenarios like curving lines, shadows and changes in the color of the pavement.

- We'll measure things like how much the lane is curving and where our vehicle is with respect to center.

- These are the kind of measurements we ultimately need to make in order to control the car.

- After that, we'll implement vehicle detection and tracking.

- While deciding when to change lanes, when to accelerate, and when to hit the brakes, knowing where the other vehicles are on the road is crucial to the decision-making process.

4.1.3 Getting Started:

- So the question remains the same, "*How should we get started finding Lane Lines?*".

- *ANSWER*: our ultimate goal in this first section is to measure some of the quantities that need to be known in order to control the car.

- For example, to steer a car, we'll need to measure how much your lane is curving.

- To do that, we'll need to map out the lens in our camera images, after transforming them to a different perspective.

- One way, you're looking down on the road from above. But, in order to get his perspective transformation right, we first have to correct for the effect of image distortion.

- By image distortion, we mean something like this?

- But hopefully, the distortion we're dealing with isn't quite that bad, but yes, that's the idea.

- Cameras don't create perfect images.

- Some of the objects in the images, especially ones near the edges, can get stretched or skewed in various ways and we need to correct for that.

4.1.4 Distortion Correction:

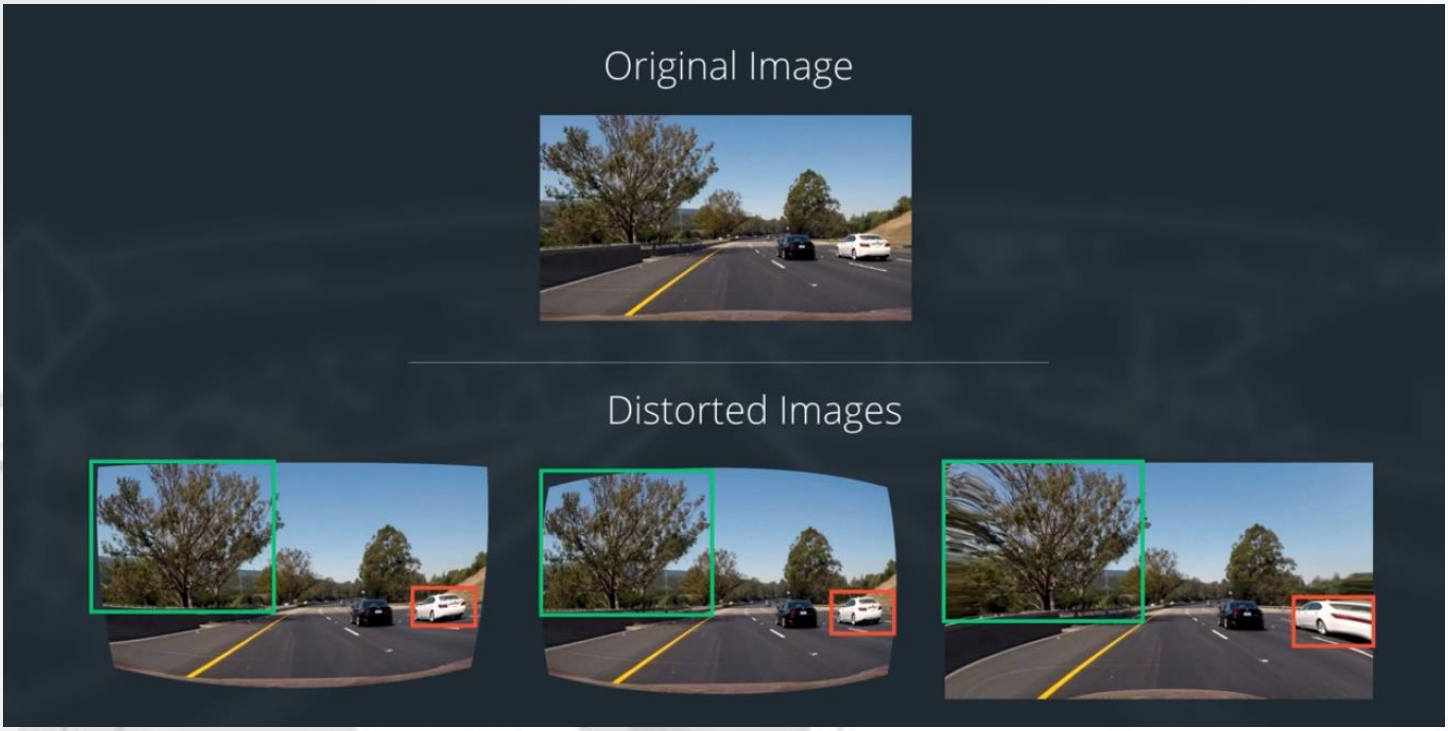
- When we talk about image distortion, we're talking about what happens when a camera looks at 3D objects in the real world and transform them into a 2D image.
- This transformation isn't perfect.

- For example, here's an image of a road and some images taken through different camera lenses that are slightly distorted.
- In these distorted images, you can see that the edges of the lanes are bent and sort of rounded or stretched outward.
- And distortion is actually changing what the shape and size of these objects appears to be.
- Because we're trying to accurately place the self-driving car in this world.

- Eventually, we'll want to look at the curve of a lane and steer the correct direction.
- But if the lane is distorted, we'll get the wrong measurement for curvature in the first place and our steering angle will be wrong

---

- So, the first step in analyzing camera images is to undo this distortion so that we can get correct and useful information out of them.



4.1.5: Effects of Distortion:

- **Question**: Why is it important to correct for image distortion?
- **ANSWER**:

1) Distortion can change the apparent size of an object in an image.

2) Distortion can change the apparent shape of an object in an image.

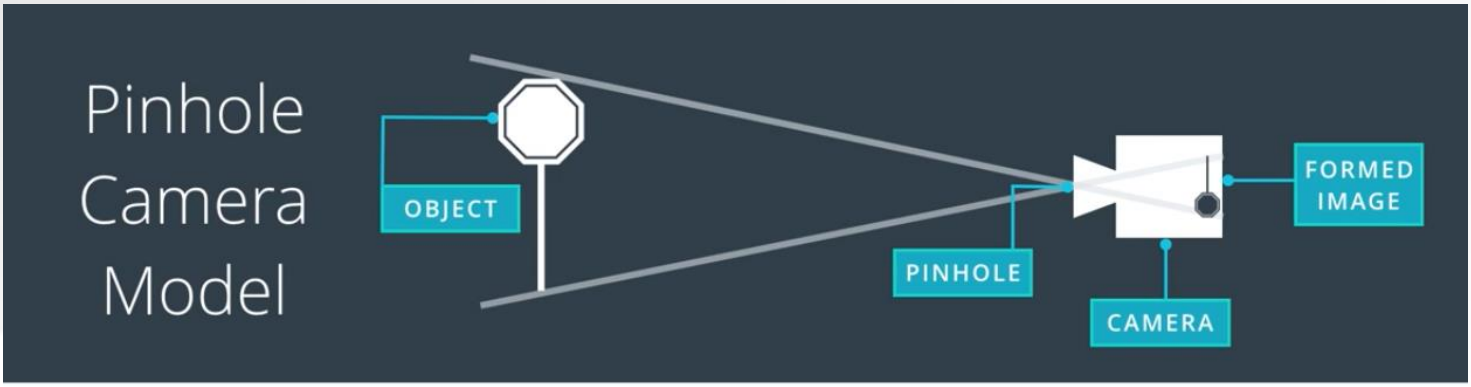
3) Distortion can cause an object's appearance to change depending on where it is in the field of view.

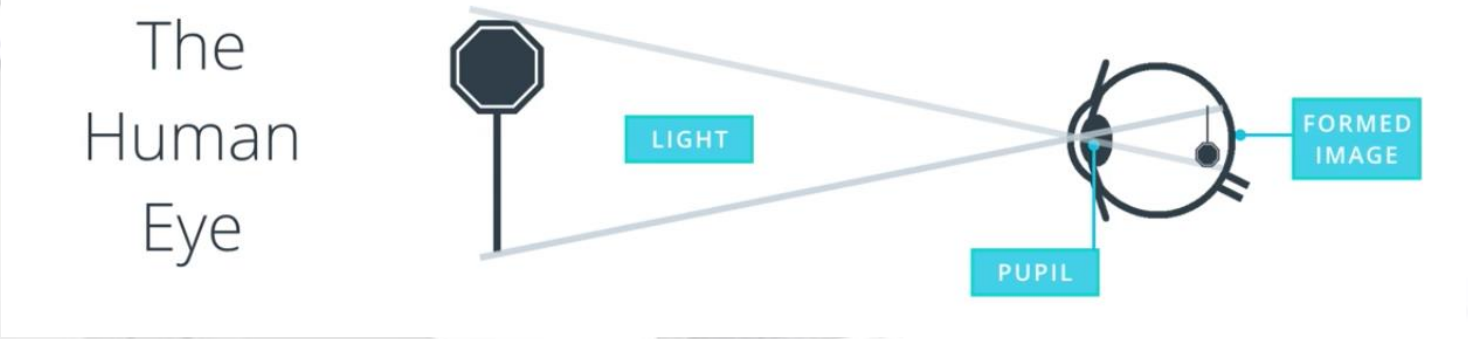
4) Distortion can make objects appear closer or farther away than they actually are.

4.1.6: Pinhole Camera Model and Types of Distortion:

- Before we get into the code and start correcting for distortion, let's get some intuition as to how this distortion occurs.

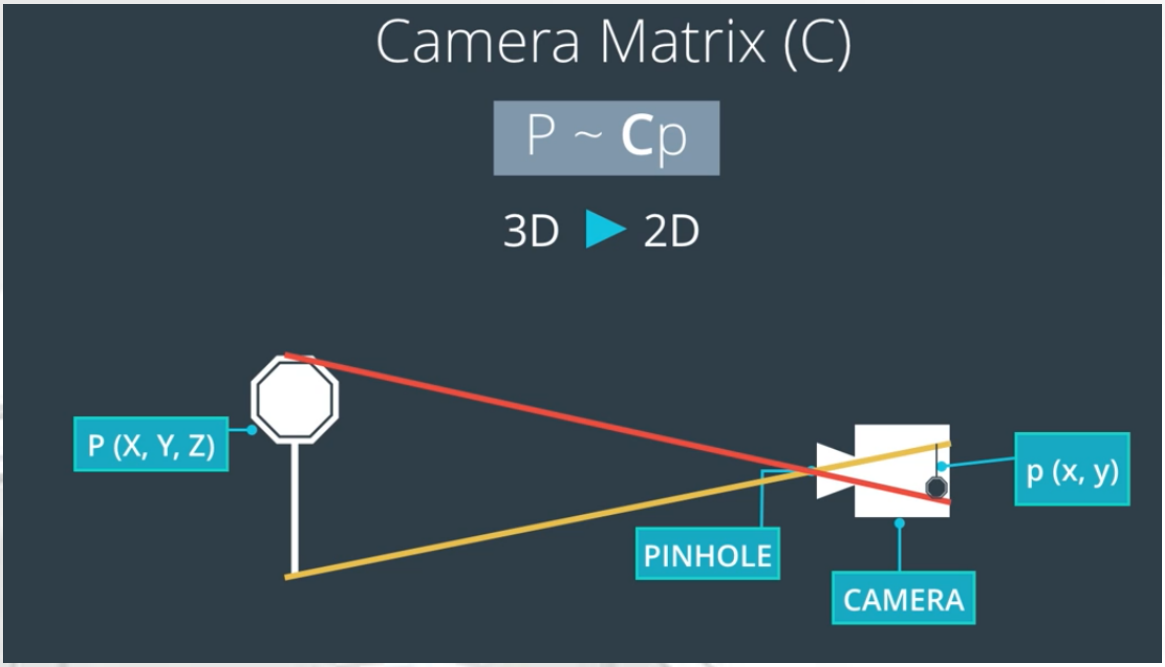
- Here's a simple model of a camera called the pinhole camera model.
- When a camera forms an image, it's looking at the world similar to how our eyes do. By focusing the light that's reflected off of objects in the world.



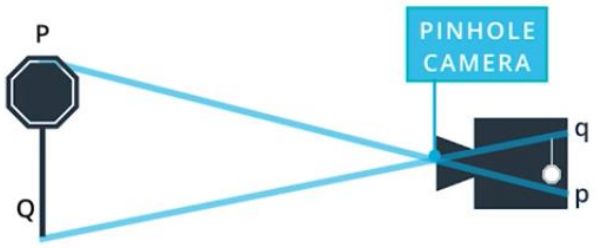


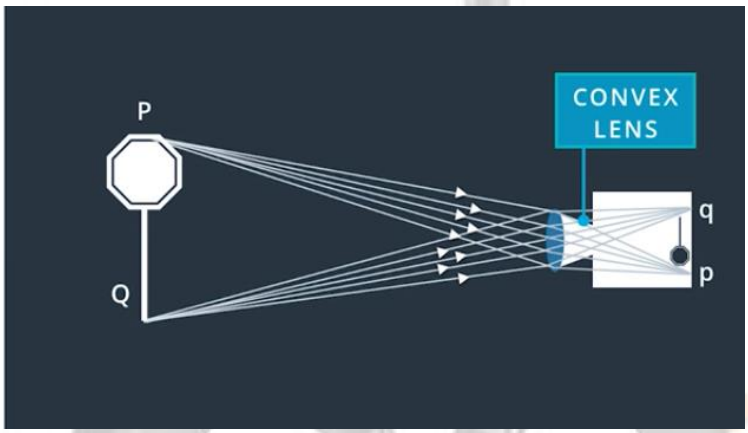
- In this case through a small pinhole, the camera focuses the light that's reflected off of a 3D traffic sign, and forms a 2D image at the back of the camera or a sensor or some film would be placed.
- In fact the image it forms here will be upside down, and reversed because rays of light that enter from the top of an object will continue on that angled path through the pinhole and end up at the bottom of the formed image.

- In math, this transformation from 3D object points, P of X, Y, and Z. To 2D image points, P of just X, and Y is done by a transformative matrix called "Camera Matrix". While we'll call C for camera and we'll need this to calibrate the camera later on.



- However, real cameras don't use tiny pinholes like this, they use LENSES to focus multiple light rays at a time. Which allows them to quickly form images. But lenses can introduce distortion too.





- Light rays often bend a little too much or too little at the edges of a curved lens of a camera, and this creates the effect we looked at earlier that distorts the edges of images.
- So that lines or objects appear, more or less, curved that they actually are. This is called "Radial Distortion", and it's the most common type of distortion.

- Another type, is tangential distortion, if the camera's lens is not aligned perfectly parallel to the imaging plane where the camera film or sensor is, this makes an image tilted. So that some objects appear further away or closer than they actually are. And this is "Tangential Distortion".

- There are even example of lenses that purposefully distort images like fisheye or wide angle lenses which keep radian distortion for stylistic effect.

- But for our purposes we are using this images to position ourself driving car and eventually steer it the right direction.
- So we need undistorted images that accurately reflect our real world surroundings.

$$\text{Distortion}_{\text{coefficients}} = (k_1\ k_2\ p_1\ p_2\ k_3)$$

– Luckily, this distortion can generally be captured by five numbers called "Distortion Coefficients", whose values reflect the amount of radial and tangential distortion in an image.
– In severely distorted cases, sometimes even more than five coefficients are required to capture the amount of distortion.

– If we know these coefficients, we can use them to calibrate our camera and un-distort our images. And the mathematical details of correcting for distortion are below.

4.1.7: Distortion Coefficients and Correction:

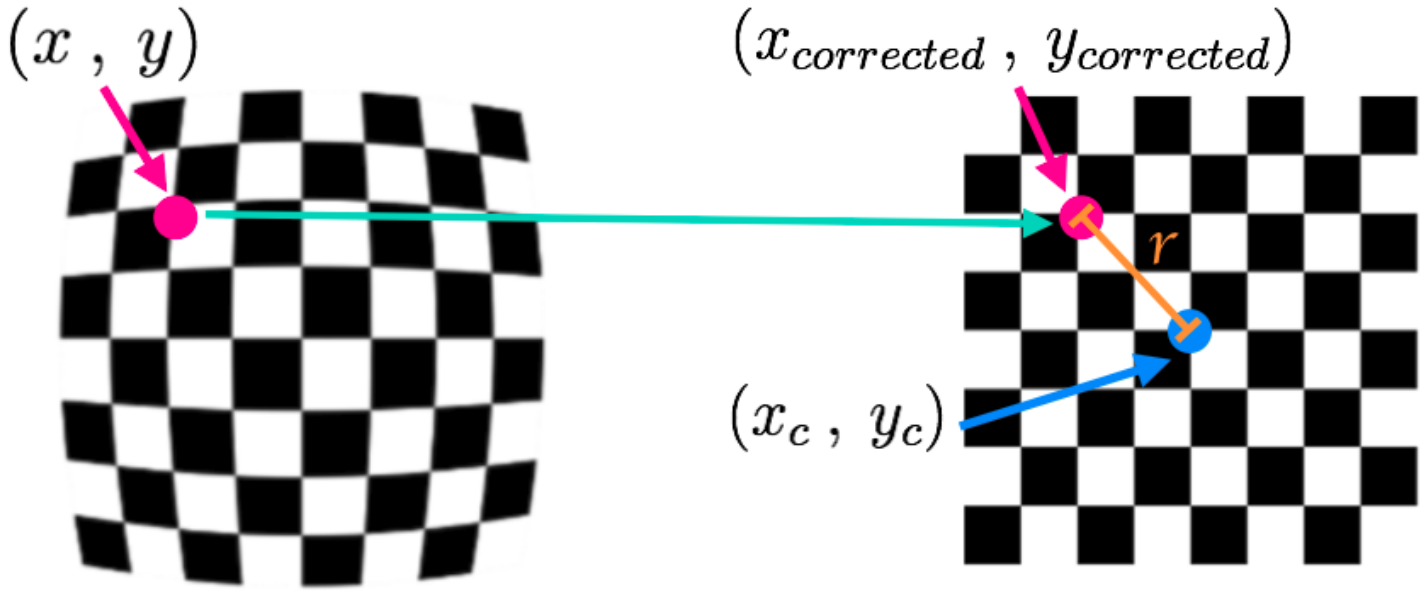
– There are 3 coefficients needed to correct for radial distortion: k1, k2, and k3.
– To correct the appearance of radially distorted points in an image, one can use a correction formula.

– In the following equations, (x, y) is a point in a distorted image. To undistort these points, Open-CV calculates r, which is the known distance between a point in an undistorted (corrected) image ($X_{corrected}$, $Y_{corrected}$) and the center of the image distortion, which is often the center of that image (X_c, Y_c). This center point (X_c, Y_c) is sometimes referred to as the distortion center. These points are pictured below.

– NOTE: The distortion coefficient k3 is required to accurately reflect major radial distortion (like in wide angle lenses). However, for minor radial distortion, which most regular camera lenses have, k3 has a value close to or equla to zero and is negligible. So, in Open-CV, you can

choose to ignore this coefficient this is why it appears at the end of the distortion values array: [k1, k2, p1, p2, k3].

– In this course, we will use it in all calibration calculations so that our calculations apply to a wider variety of lenses and can correct for both minor and major radial distortion.



– There are two more coefficient that account for tangential distortion: p1 and p2, and this distortion can be corrected using a different correction formula.

$$x_{corrected} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$$

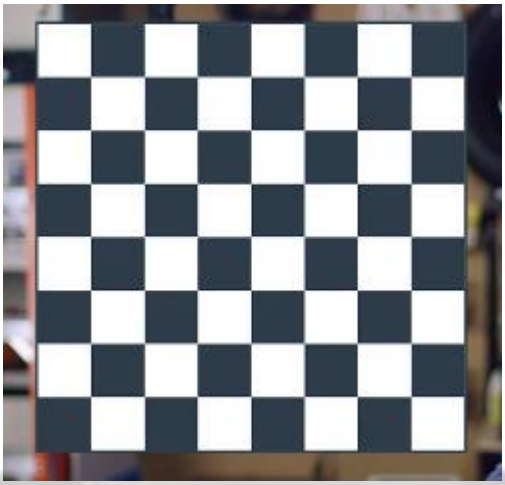
$$x_{distorted} = x_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$
$$y_{distorted} = y_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

4.1.8: Image Formation:

– Question: What is the fundamental difference between images formed with a pinhole camera and those formed using lenses?
– Answer: Pinhole camera images are free from distortion, but lenses tend to introduce image distortion.

4.1.9: Measuring Distortion:

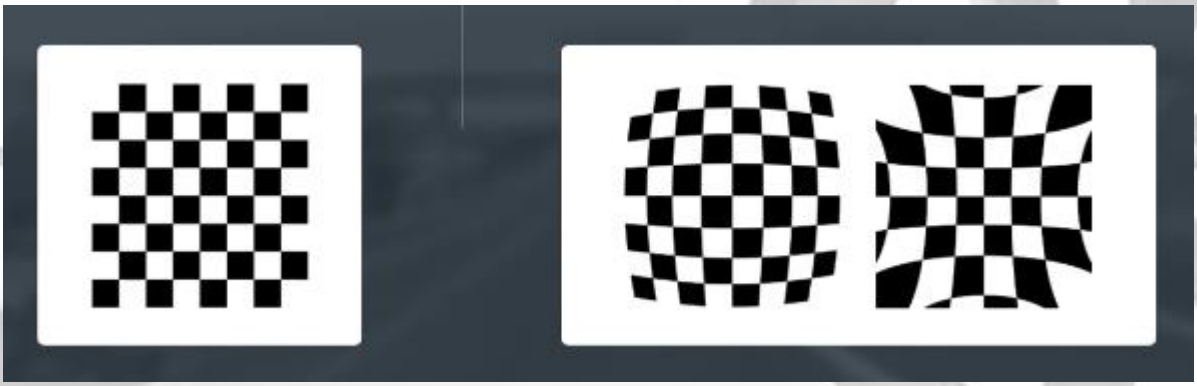
– So, we know that distortion changes the size and shapes of objects in an image, but how do we calibrate for that?
– Well, we can take pictures of known shapes, then we'll be able to detect and correct any distortion errors.



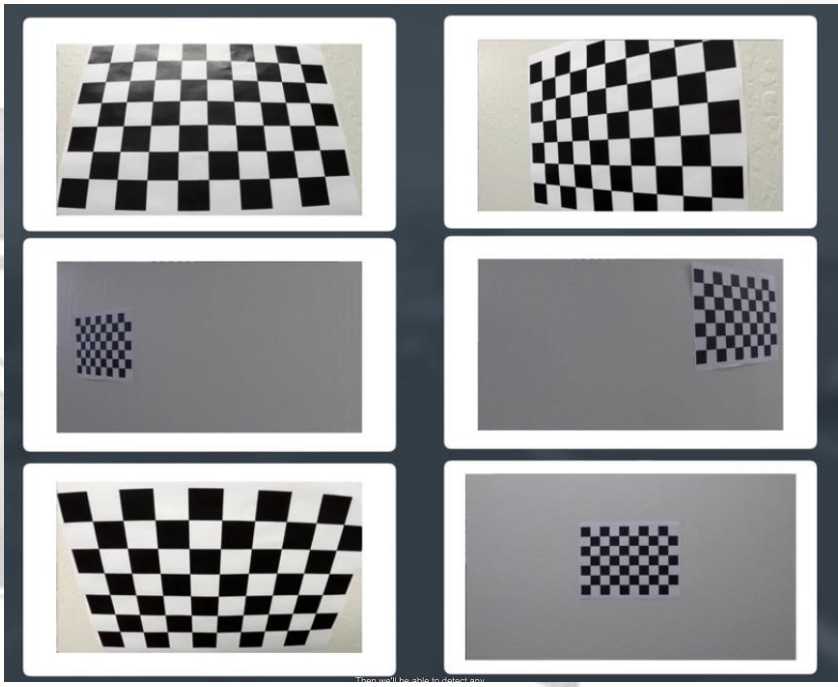
– We could choose any shapes to calibrate our camera, and we'll use a chessboard.

– A chessboard is great for calibration because its regular high contrast pattern makes it easy to detect automatically.

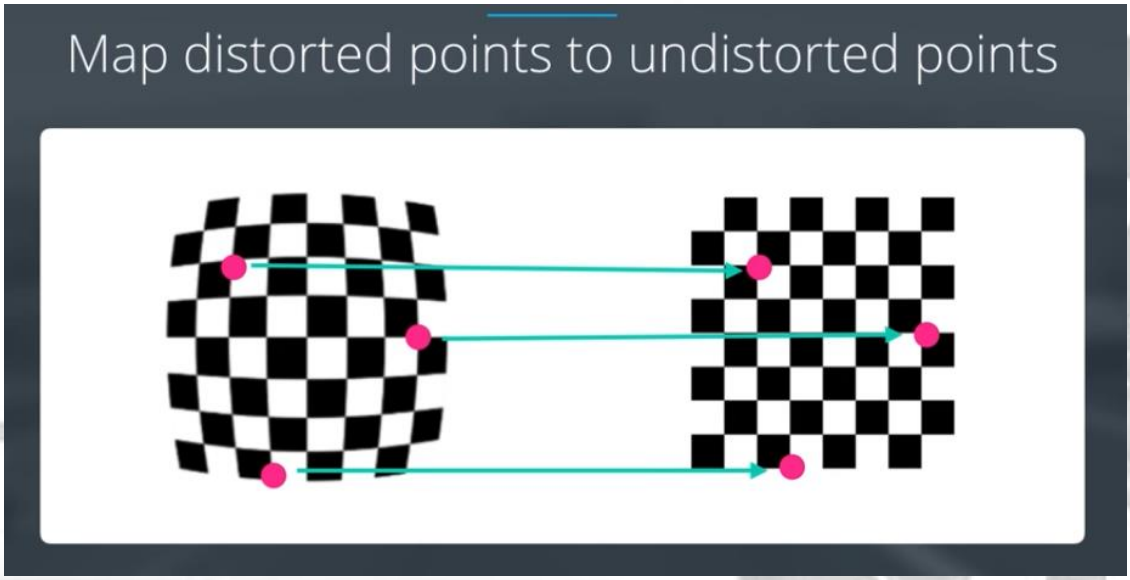
– And we know what an undistorted flat chessboard looks like.



– So, if we use our camera to take multiple pictures of a chessboard against a flat surface.
– Then we'll be able to detect any distortion by looking at the difference between the apparent size and the shape of the squares in these images, and the size and shape that they actually are.



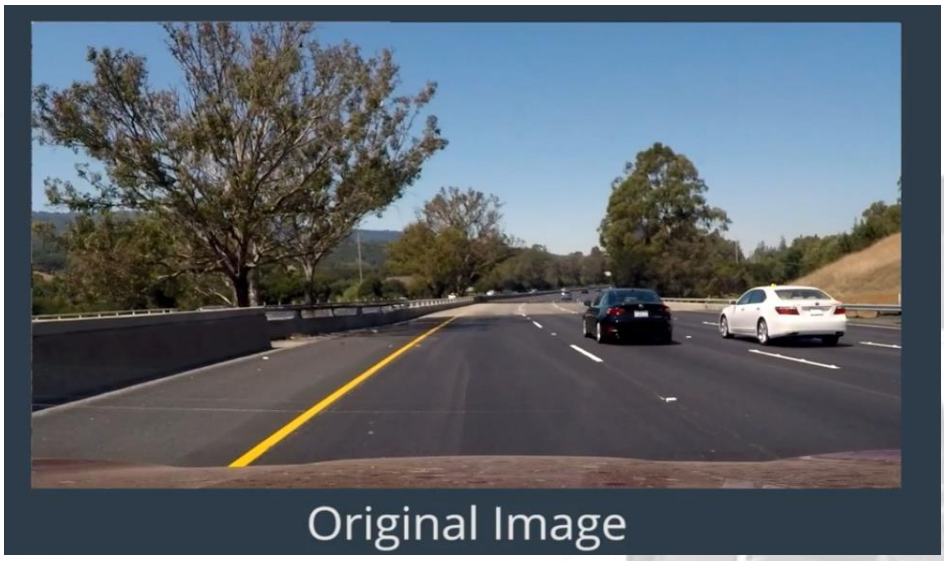
– Then we'll use that information to calibrate our camera.
– Create a transform that maps these distorted points to undistorted points. And finally, undistort any images.



4.1.10: Lane Curvature:

- Now after camera calibration, we can start to extract really useful information from images of the road.
- One really important piece of information is lane curvature.
- Think about how you drive on a highway, you take a look at the road in front of you and cars around you. You press the gas or break to go with the flow. And based on how much the lane is curving, left or right, you turn the steering wheel to stay in that lane.
- Now, how does this work for a self-driving car?
- Self-driving cars need to be told the correct steering angle to turn left or right and we can calculate this angle if we know a few things about the speed and dynamics of the car and how much the lane is curving.

- To determine the curvature, we'll go through the following steps.

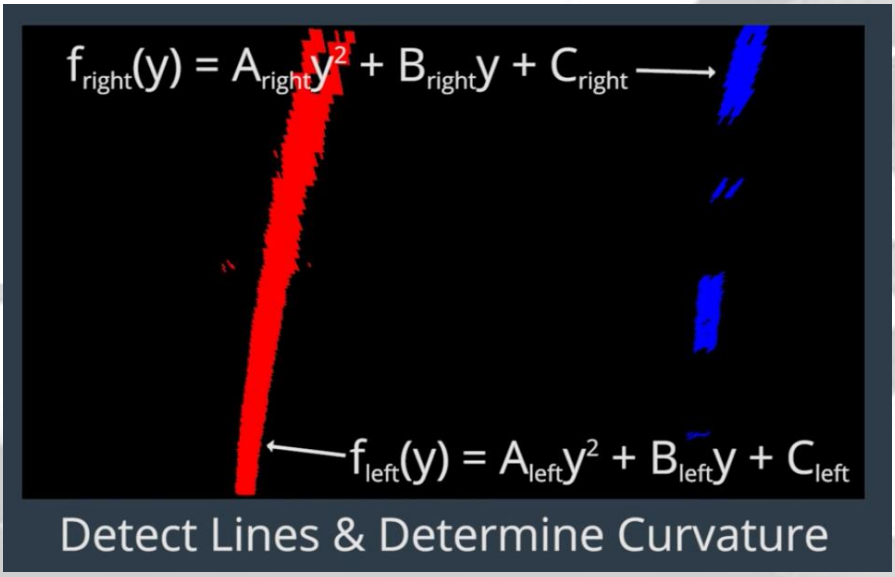


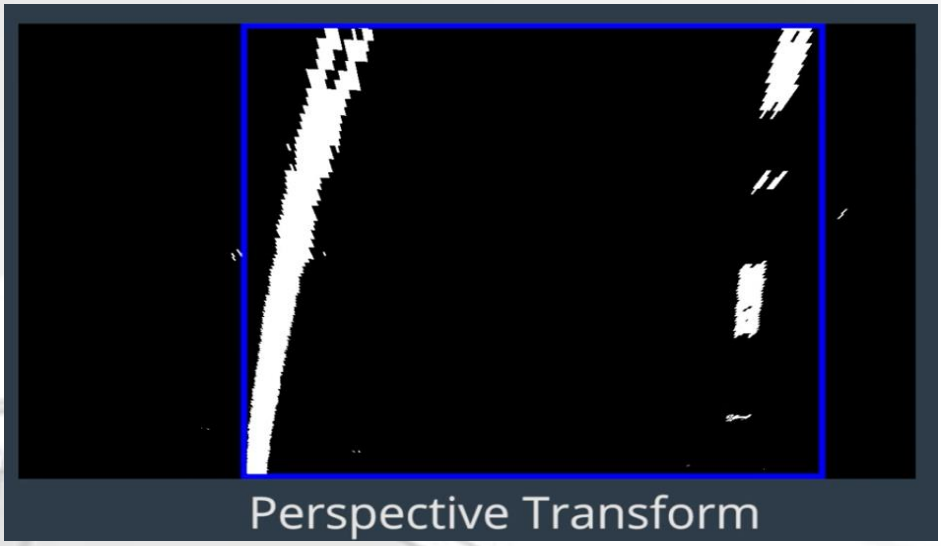
1) First, we'll detect the lane line using some masking and thresholding techniques.

2) Then, perform a perspective transform to get a birds eye view of the lane line. This let's us fit a polynomial to the lane lines.

3) Then, we can extract the curvature the lines from this polynomial with just a little math.

- So let's start by learning more about the perspective transform.



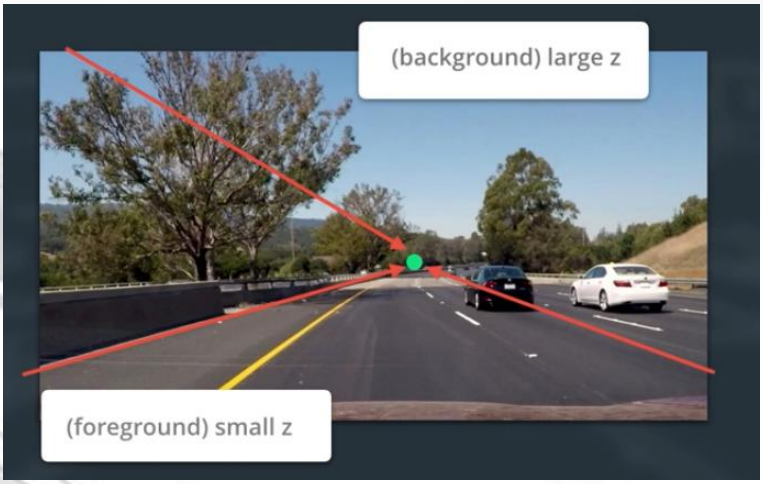


4.1.11: Perspective:

- In an image, perspective is the phenomenon where an object appears smaller the farther away it is from a viewpoint like a camera, and parallel lines appear to converge to a point.

- It's seen in everything from camera images to art. Many artist use perspective to give the right impression of an object's size, depth, and position when viewed from a particular point.
- And let's look at perspective in this image of the road.

- As you can see, the lane looks smaller and smaller the farther away it gets from the camera, and the background scenery also appears smaller than the trees closer to the camera in the foreground.

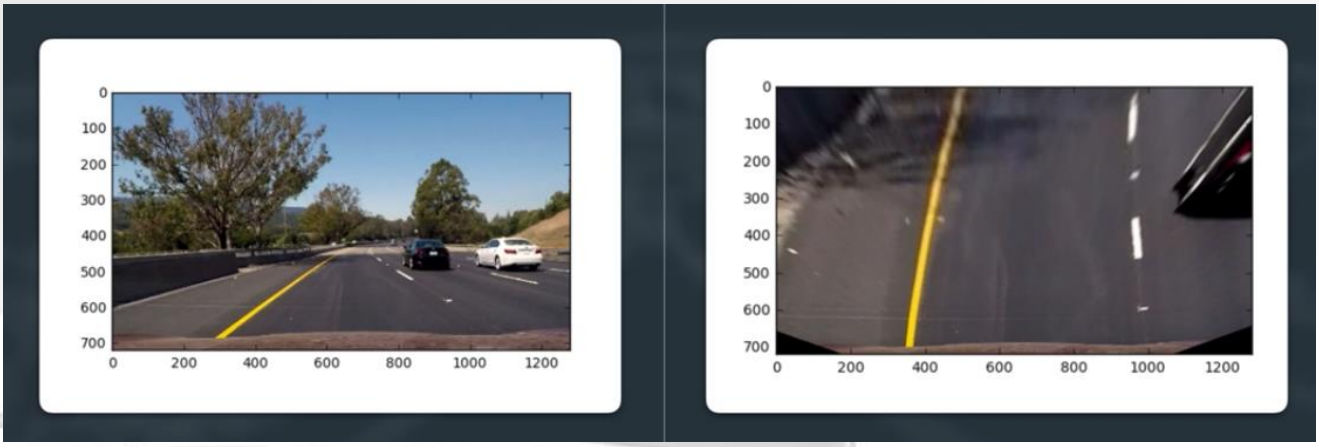


- Mathematically, we can characterize perspective by saying that in real world coordinates x, y, and z, the greater the magnitude of an objects z coordinate, or distance from the camera, the smaller it will appear in a 2D image.

- A perspective transform uses this information to transform an image. It essentially transforms the apparent z coordinate of the object points, which in turn changes that object's 2D image representation.

- A perspective transforms warps the image and effectively drags points towards or pushes them away from the camera to change the apparent perspective.

- For example, to change this into a bird's eye view scene, we can apply a perspective transform that zooms in on the farther away objects.

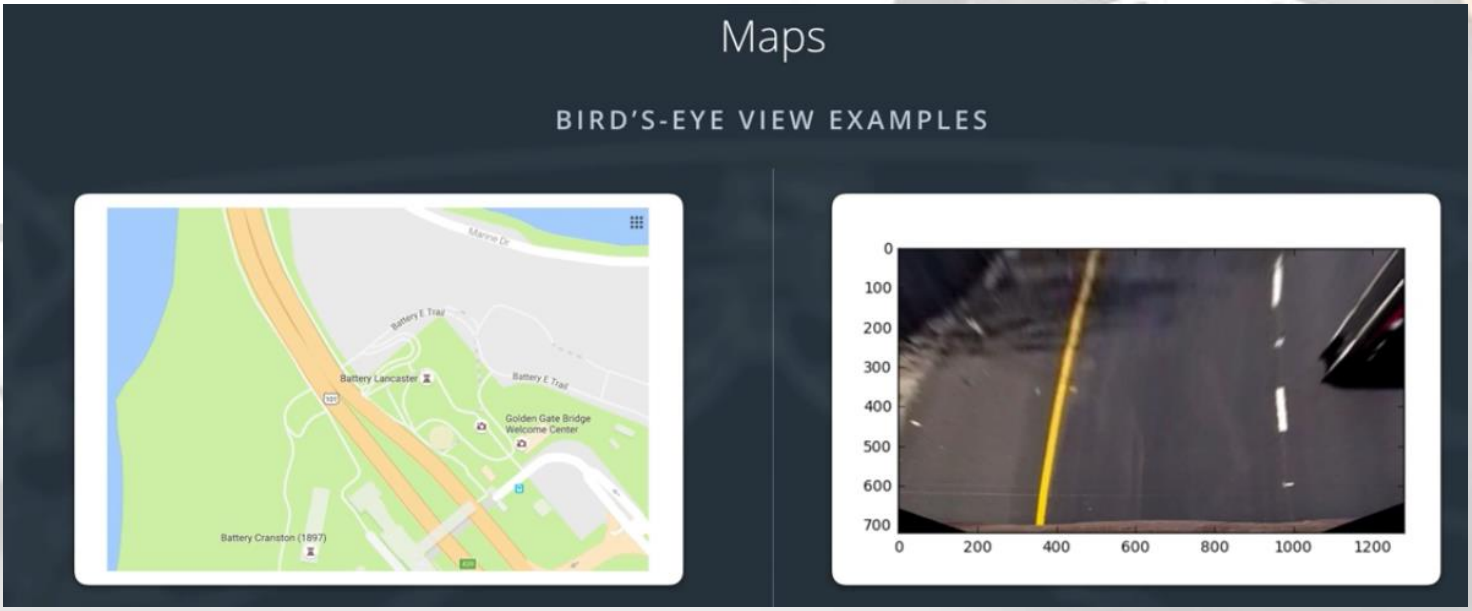


- This is really useful, because some tasks, like find the curvature of a lane, are easier to perform on a bird's eye view of an image.

- Take this image of a road, if we just looked at these lane edges, the left line looks like it's leaning and curving a little to the right, but the right lane line looks like it's leaning to the left without much of a curve.

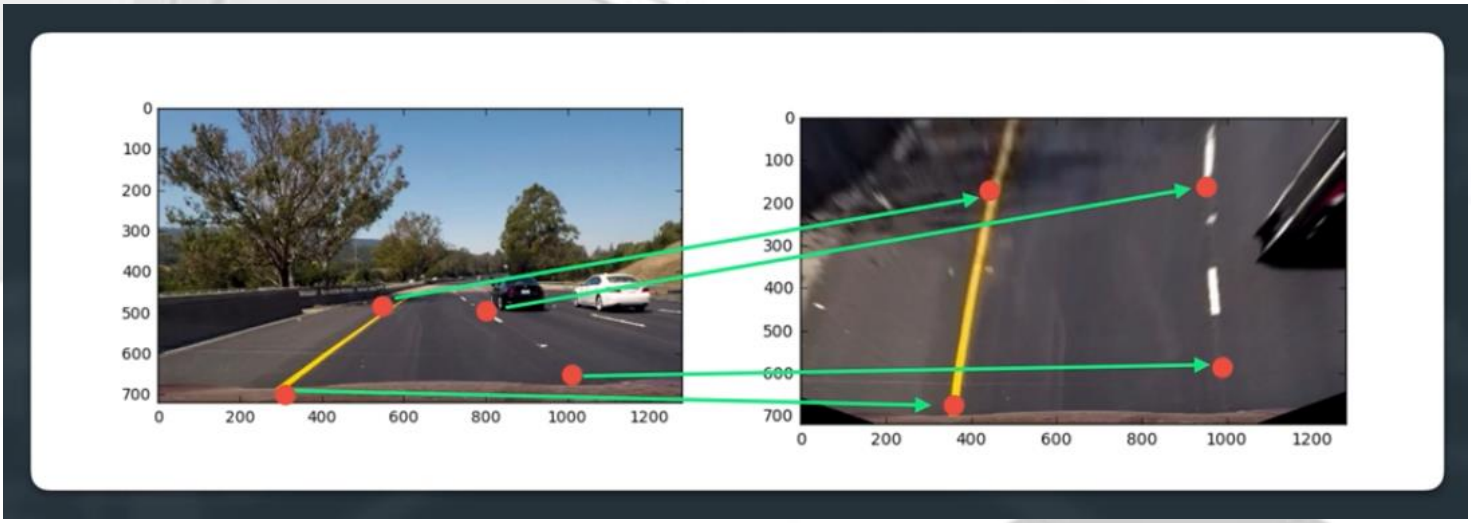
- By doing a perspective transform and viewing this same image from above, we can see that the lanes are parallel and both curve about the same amount to the right.

- So, a perspective transform, let's us change our perspective to view the same scene from different viewpoints and angles.
- This could be viewing a scene from a side of a camera, from below the camera, or looking down on the road from above.
- Doing a bird's-eye view transform is especially helpful for road images because it will also allow us to match a car's location directly with a map, since map's display roads and scenery from a top down view.



- The process of applying a perspective transform will be kind of similar to how we applied un-distortion.

- But this time, instead of mapping object points to image points, we want to map the points in a given image to different desired image points with a new perspective.
- And again, this perspective can be done for all kinds of different viewpoints, whether that's from above like with a map or from different camera angles.



4.1.12: Curvature and Perspective:

Question:

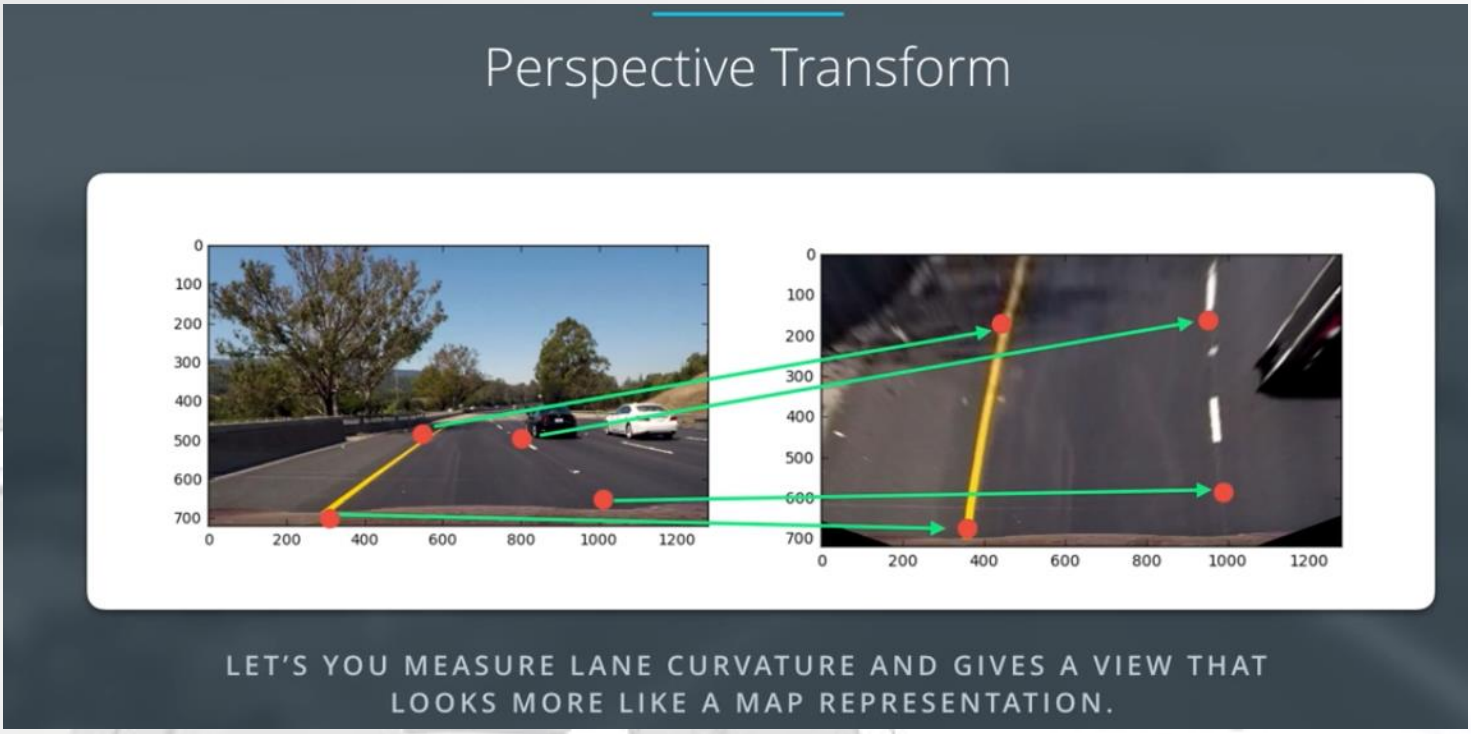
- There are various reasons you might want to perform a perspective transform on an image, but in the case of road images taken from our front-facing camera on a car, why are we interested in doing a perspective transform?

Answer:

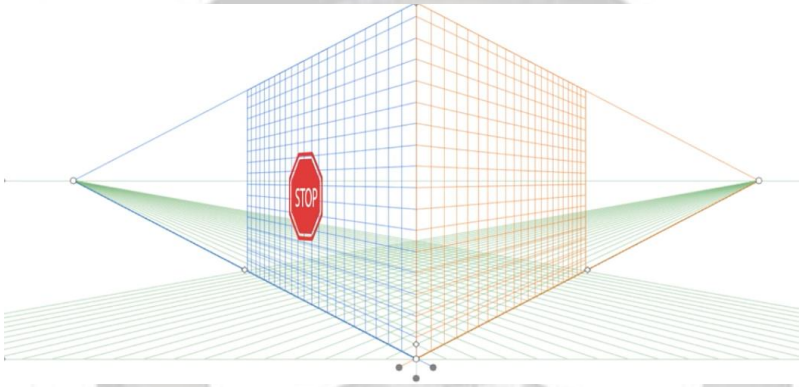
- Because ultimately we want to measure the curvature of the lines, and to do that, we need to transform to a top-down view.

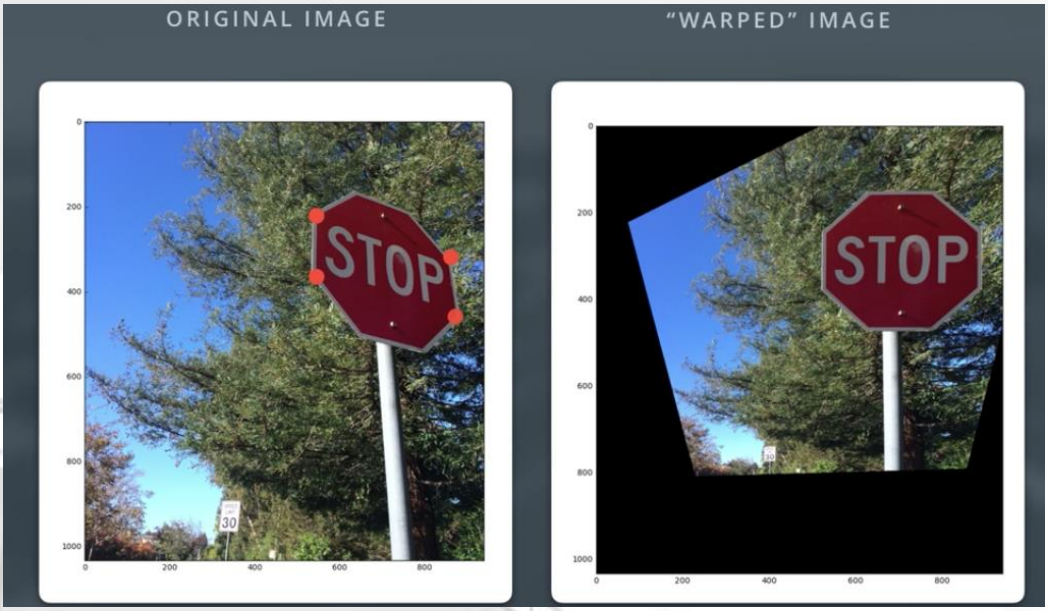
4.1.13: Transform a Stop Sign:

- Here's an image of a STOP sign that's viewed from an angle. And I'll show you how to create and apply a perspective transform so that it looks we're viewing this sign from the front.



- Remember, we care about the perspective transform because the software we written will eventually do this to images of a road, so that we can find the lane curvature.
- We're using a traffic sign as an example here mainly because it's easiest to see that you've performed a perspective transform correctly on images with text or other distinct reference points.
- To create a perspective transform, we'll first select four points that define a rectangle on a plane in this image.
- Four points are enough to define a linear transformation from one perspective to another.





- And in this case, we're selecting point that define a plane on the surface of the STOP sign.
- And we'll also select where we want those same four points to appear and are transformed, often called "wraped image".

- Then, we can use OpenCV functions to calculate the transform that maps the points in the original image to the warped image with the difference perspective.