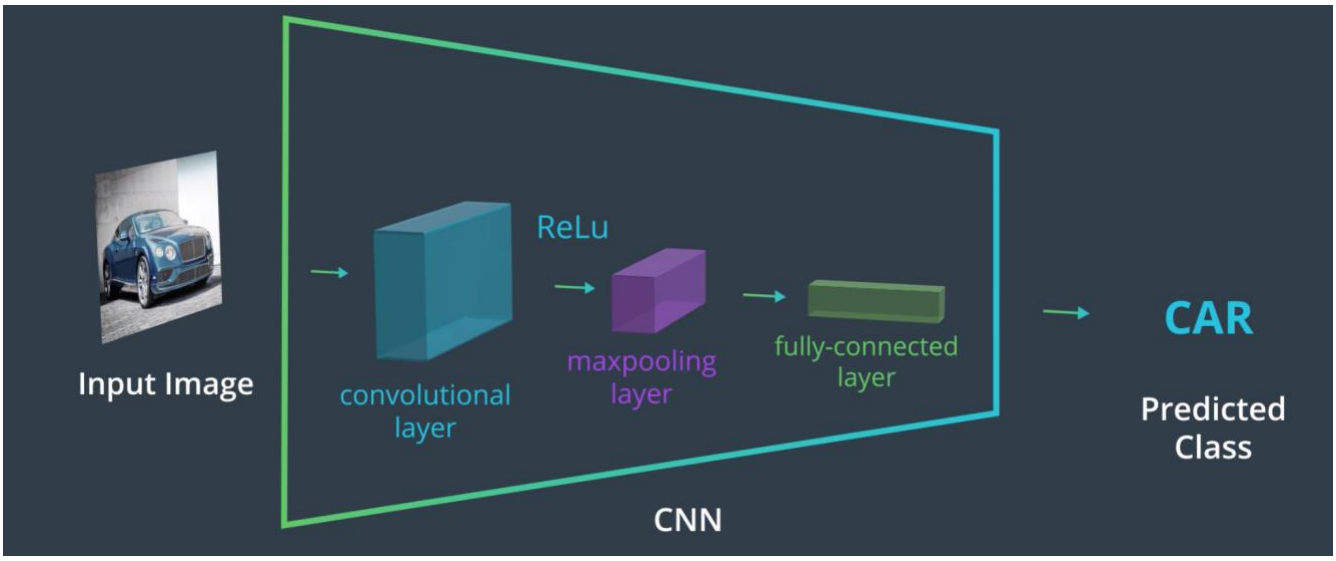
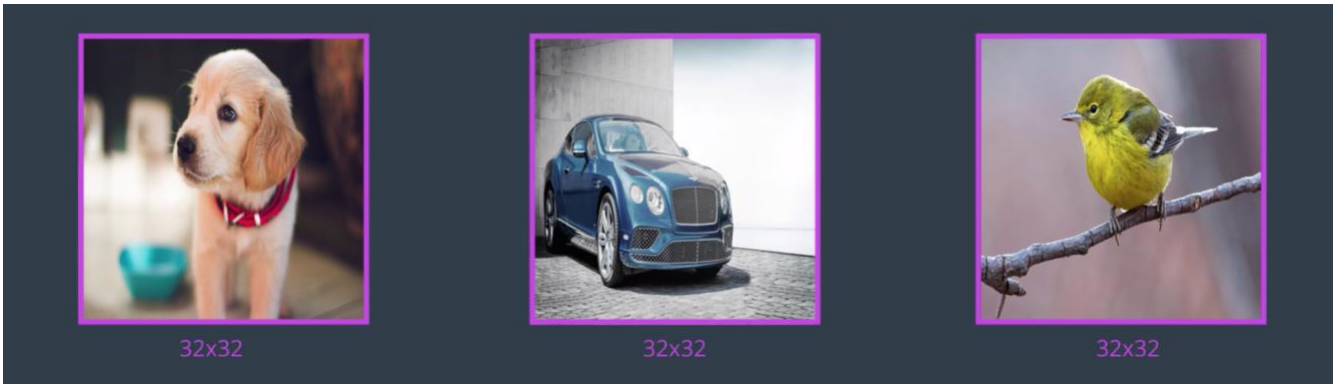
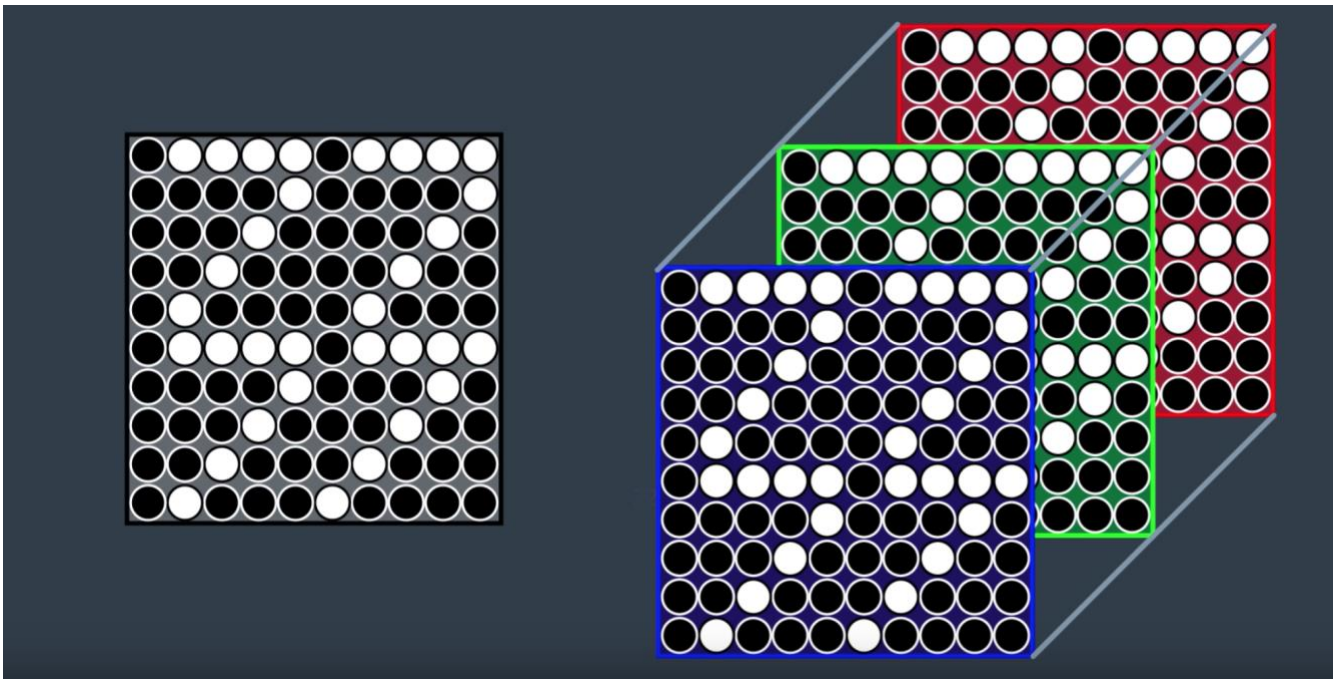



116. In this lesson, we've investigated two new types of layers for neural networks. We began with convolutional layers which detect regional patterns in an image using a series of image filters. We've seen how typically a ReLu activation function is applied to the output of these filters to standardize their output values. Then, you learned about max pooling layers, which appear after convolutional layers to reduce the dimensionality of our input arrays. These new layers, along with fully-connected layers, that should be familiar, are often the only layers that you'll find in a CNN. In this video, we'll discuss how to arrange these layers to design a complete CNN architecture. We'll focus again on CNNs for image classification. In this case, our CNN must accept an image array as input. Now, if we're going to work with messy real-world images, there's a complication that we haven't yet discussed.

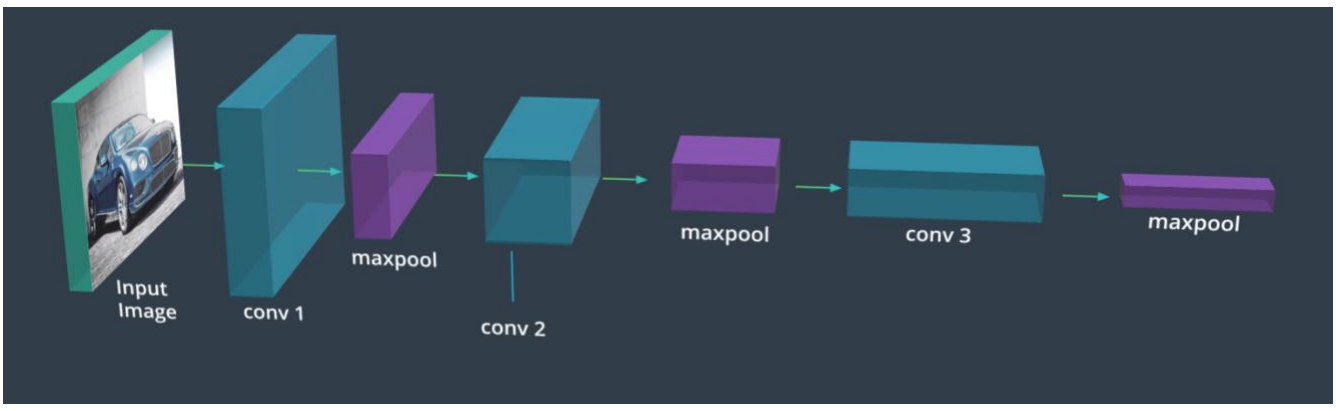



If I go online and collect thousands or millions of images, well, it's pretty much guaranteed that they'll all be different sizes. Similar to MLPs, the CNNs we'll discuss will require a fixed size input. So, we'll have to pick an image size and resize all of our images to that same size before doing anything else. This is considered to be another pre-processing step, alongside normalization and conversion to a tensor datatype. It's very common to resize each image to be a square, with the spatial dimensions equal to a power of two, or else a number that's divisible by a large power of

two. In the next few videos, we'll work with a dataset composed of images that have all been resized to 32 by 32 pixels.

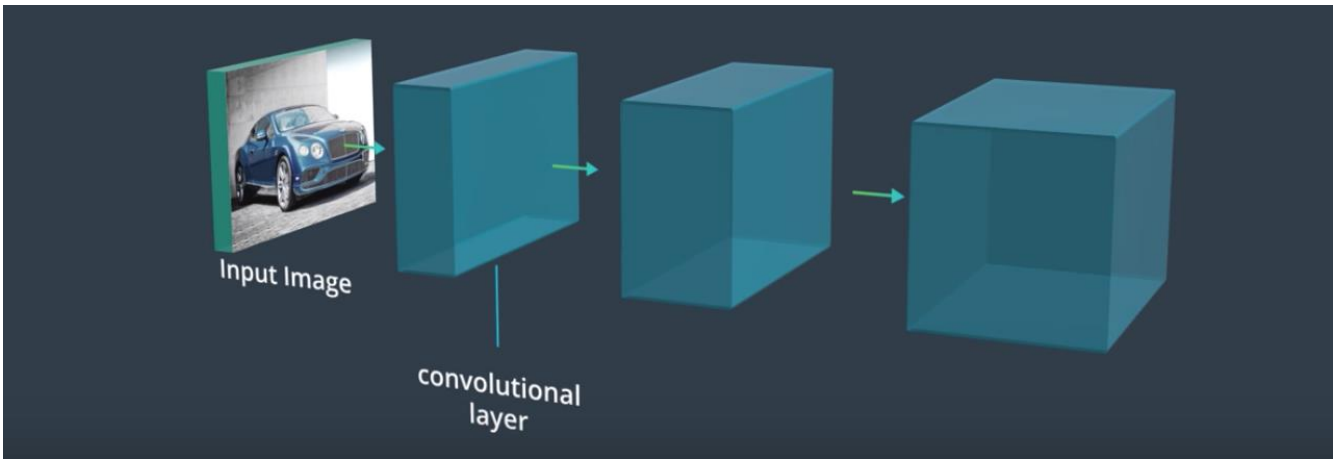


Recall that any image is interpreted by the computer as a three-dimensional array. Color images had some height and width in pixels along with red, green, and blue color channels corresponding to a depth of three. Grayscale images, while technically two-dimensional, can also be thought of as having their own width and height and a depth of one. For both of these cases, with color or grayscale images, the input array will always be much taller and wider than it is deep.

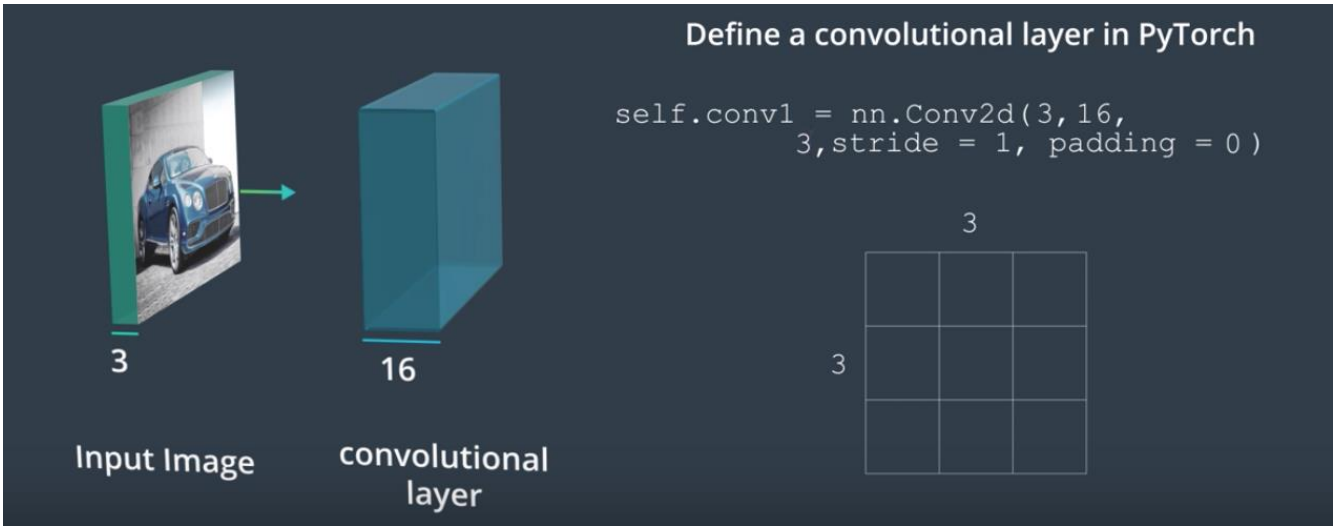



Our CNN architecture will be designed with the goal of taking that array and gradually making it much deeper than it is tall or wide. Convolutional layers will be used to make the array deeper as it passes through the network, and max pooling layers will be used to decrease the XY dimensions. As the network gets deeper, it's actually extracting more and more complex patterns and features that help identify the content and the objects in an image, and it's actually discarding some spatial

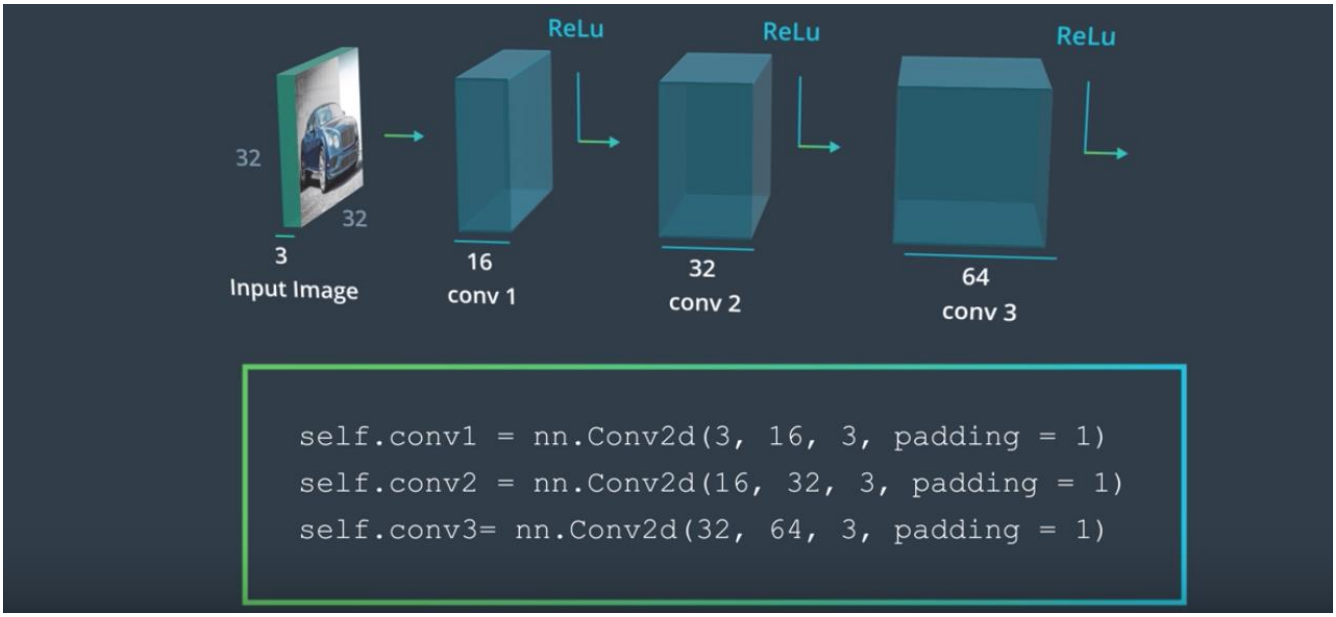
information about features like a smooth background and so on that do not help identify the image. To see how this works, next, let's go over a complete image classification CNN in detail.



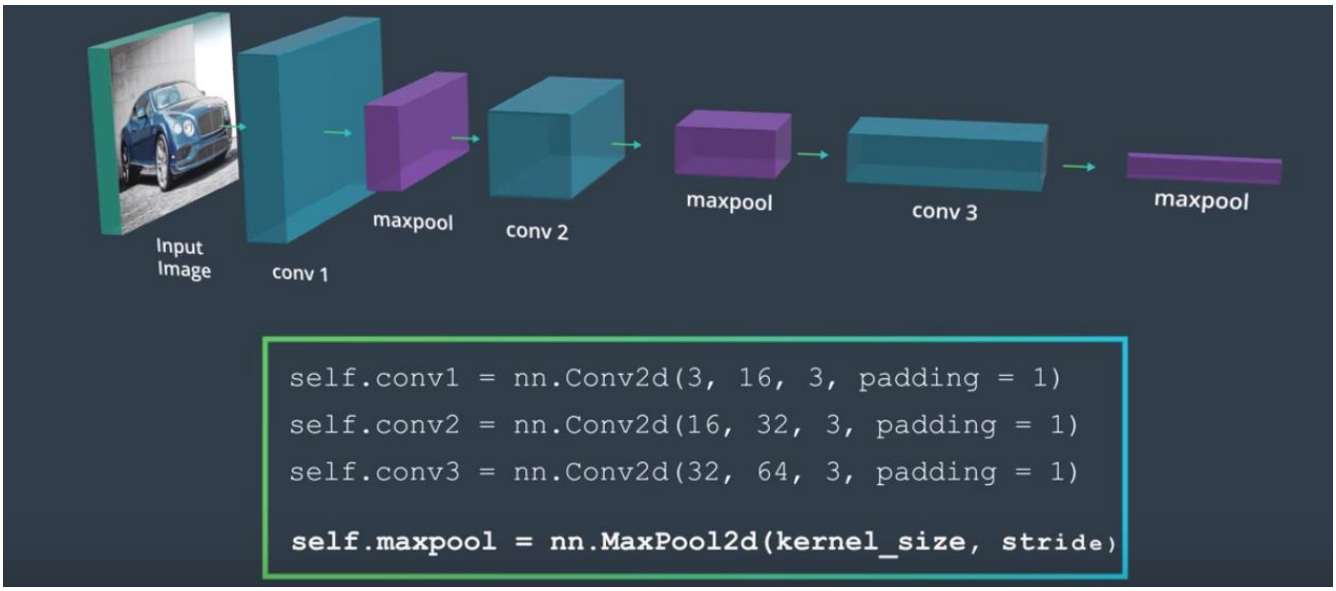

117. Say we want to classify an input image. There are a few ways we could go about this using a deep learning architecture. Consider following the input layer with a sequence of convolutional layers. This stack will discover hierarchies of spatial patterns in the image. The first layer of filters looks at patterns in the input image, the second looks at patterns in the previous convolutional layer, and so on. Each of the convolutional layers requires us to specify a number of hyperparameters.



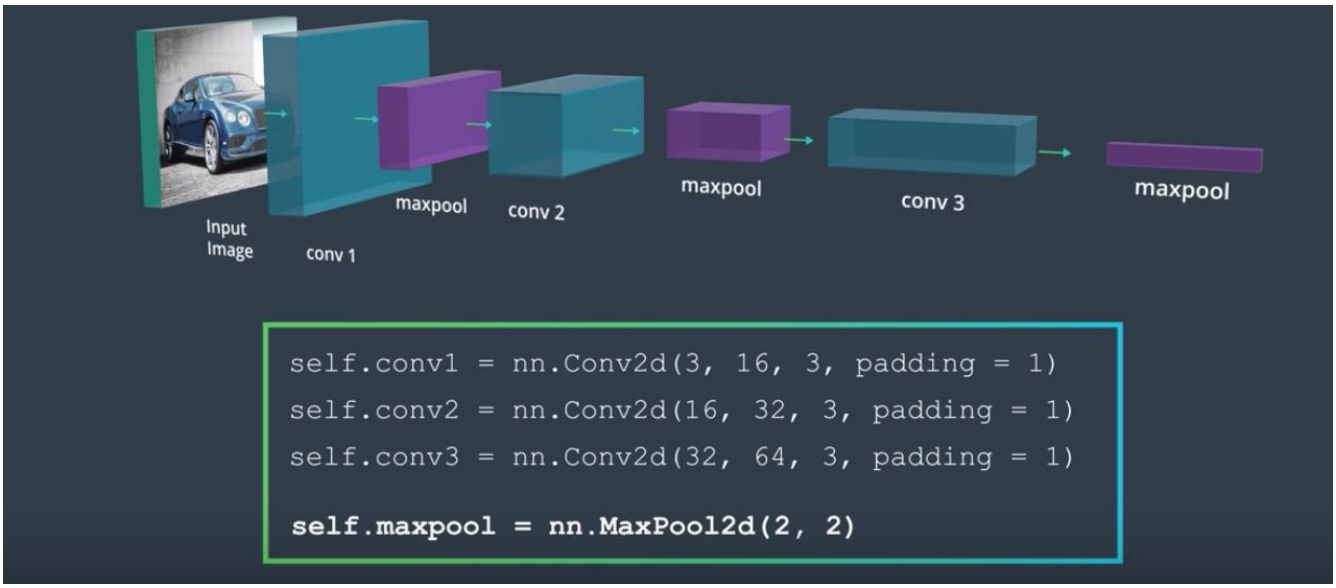
The first and second inputs to define a convolutional layer are simply the depth of the input and the desired depth of the output. For example, the input depth of a color image will be three for the RGB channels, and we might want to produce 16 different filtered images in this convolutional layer. Next, we define the size of the filters that define a convolutional layer. These are often square and range from the size of two-by-two at the smallest to up to a seven-by-seven or so for very large images. Here, I'll choose to use three-by-three filters. The stride is generally set to one and many frameworks will have this as the default value, so you may need to input this value. As for padding, you may get better results if you set your padding such that a convolutional layer will have the same width and height as its input from the previous layer. In the case of a three-by-three filter, which can almost center itself perfectly on an image but misses the border pixels by one, this padding will be equal to one. You can read a bit more about different cases for padding below. When deciding the depth or number of filters in a convolutional layer, often we'll have a number of filters increase in sequence. So, the first convolutional layer might have 16 filters. The second second will see that depth as input and produce a layer with a depth of 32. The third will have a depth of 64 and so on. After each convolutional layer, we'll apply a ReLU activation function.




If we follow this process, we have a method for gradually increasing the depth of our array without modifying the height and width. The input, just like all of the layers in this sequence, has a height and width of 32. But the depth increases from an input layers depth of three to 16 to 32 to 64. We call that, yes we wanted to increase the depth, but we also wanted to decrease the height and width and discard some spatial information. This is where max pooling layers will come in. They generally follow every one or two convolutional layers in the sequence.




Here's one such example with a max pooling layer after each convolutional layer. To define a max pooling layers, you'll only need to define the filter size and stride.




The most common setting will use filters of size two with a stride of two. This has the effect of making the XY dimensions half of what they were from the previous layer. In this way, the combination of convolutional and max pooling layers accomplishes our goal of attaining an array that's quite deep but small in the X and Y dimensions. Next, let's talk about finally connecting this output to a fully-connected layer, and see what exactly is happening to an input as it moves through these layers.

Not_read_begin

**Convolutional Layers in PyTorch**

To create a convolutional layer in PyTorch, you must first import the necessary module:

import torch.nn as nn

Then, there is a two part process to defining a convolutional layer and defining the feedforward behavior of a model (how an input moves through the layers of a network). First, you must define a Model class and fill in two functions.

**init**

You can define a convolutional layer in the __init__ function of by using the following format:

self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)

**forward**

Then, you refer to that layer in the forward function! Here, I am passing in an input image x and applying a ReLU function to the output of this layer.

x = F.relu(self.conv1(x))

**Arguments**

You must pass the following arguments:

in_channels - The number of inputs (in depth), 3 for an RGB image, for example.
out_channels - The number of output channels, i.e. the number of filtered "images" a convolutional layer is made of or the number of unique, convolutional kernels that will be applied to an input.
kernel_size - Number specifying both the height and width of the (square) convolutional kernel.

There are some additional, optional arguments that you might like to tune:

stride - The stride of the convolution. If you don't specify anything, stride is set to 1.
padding - The border of 0's around an input array. If you don't specify anything, padding is set to 0.

NOTE: It is possible to represent both kernel_size and stride as either a number or a tuple.

There are many other tunable arguments that you can set to change the behavior of your convolutional layers. To read more about these, we recommend perusing the official documentation.

**Pooling Layers**

Pooling layers take in a kernel_size and a stride. Typically the same value as is the down-sampling factor. For example, the following code will down-sample an input's x-y dimensions, by a factor of 2:

self.pool = nn.MaxPool2d(2,2)

**forward**

Here, we see that poling layer being applied in the forward function.

x = F.relu(self.conv1(x))
x = self.pool(x)

**Convolutional Example #1**

Say I'm constructing a CNN, and my input layer accepts grayscale images that are 200 by 200 pixels (corresponding to a 3D array with height 200, width 200, and depth 1). Then, say I'd like the next layer to be a convolutional layer with 16 filters, each filter having a width and height of 2. When performing the convolution, I'd like the filter to jump two pixels at a time. I also don't want the filter to extend outside of the image boundaries; in other words, I don't want to pad the image with zeros. Then, to construct this convolutional layer, I would use the following line of code:

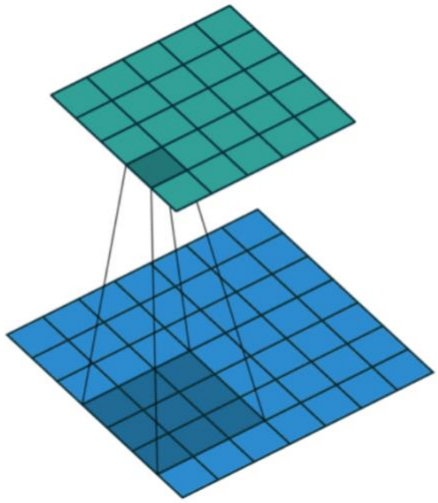
self.conv1 = nn.Conv2d(1, 16, 2, stride=2)

**Convolutional Example #2**

Say I'd like the next layer in my CNN to be a convolutional layer that takes the layer constructed in Example 1 as input. Say I'd like my new layer to have 32 filters, each with a height and width of 3. When performing the convolution, I'd like the filter to jump 1 pixel at a time. I want this layer to have the same width and height as the input layer, and so I will pad accordingly. Then, to construct this convolutional layer, I would use the following line of code:

self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

Convolution with 3x3 window and stride 1



**Sequential Models**

We can also create a CNN in PyTorch by using a Sequential wrapper in the __init__ function. Sequential allows us to stack different types of layers, specifying activation functions in between!

```
def __init__(self):
    super(ModelName, self).__init__()
    self.features = nn.Sequential(
        nn.Conv2d(1, 16, 2, stride=2),
        nn.MaxPool2d(2, 2),
        nn.ReLU(True),

        nn.Conv2d(16, 32, 3, padding=1),
        nn.MaxPool2d(2, 2),
        nn.ReLU(True)
     )
```

**Formula: Number of Parameters in a Convolutional Layer**

The number of parameters in a convolutional layer depends on the supplied values of filters/out_channels, kernel_size, and input_shape. Let's define a few variables:

K - the number of filters in the convolutional layer
F - the height and width of the convolutional filters
D_in - the depth of the previous layer

Notice that K = out_channels, and F = kernel_size. Likewise, D_in is the last value in the input_shape tuple, typically 1 or 3 (RGB and grayscale, respectively).

Since there are F*F*D_in weights per filter, and the convolutional layer is composed of K filters, the total number of weights in the convolutional layer is K*F*F*D_in. Since there is one bias term per filter, the convolutional layer has K biases. Thus, the number of parameters in the convolutional layer is given by K*F*F*D_in + K.

**Formula: Shape of a Convolutional Layer**

The shape of a convolutional layer depends on the supplied values of kernel_size, input_shape, padding, and stride. Let's define a few variables:

K - the number of filters in the convolutional layer
F - the height and width of the convolutional filters
S - the stride of the convolution
P - the padding
W_in - the width/height (square) of the previous layer
Notice that K = out_channels, F = kernel_size, and S = stride. Likewise, W_in is the first and second value of the input_shape tuple.

The depth of the convolutional layer will always equal the number of filters K.

The spatial dimensions of a convolutional layer can be calculated as: (W_in−F+2P)/S+1

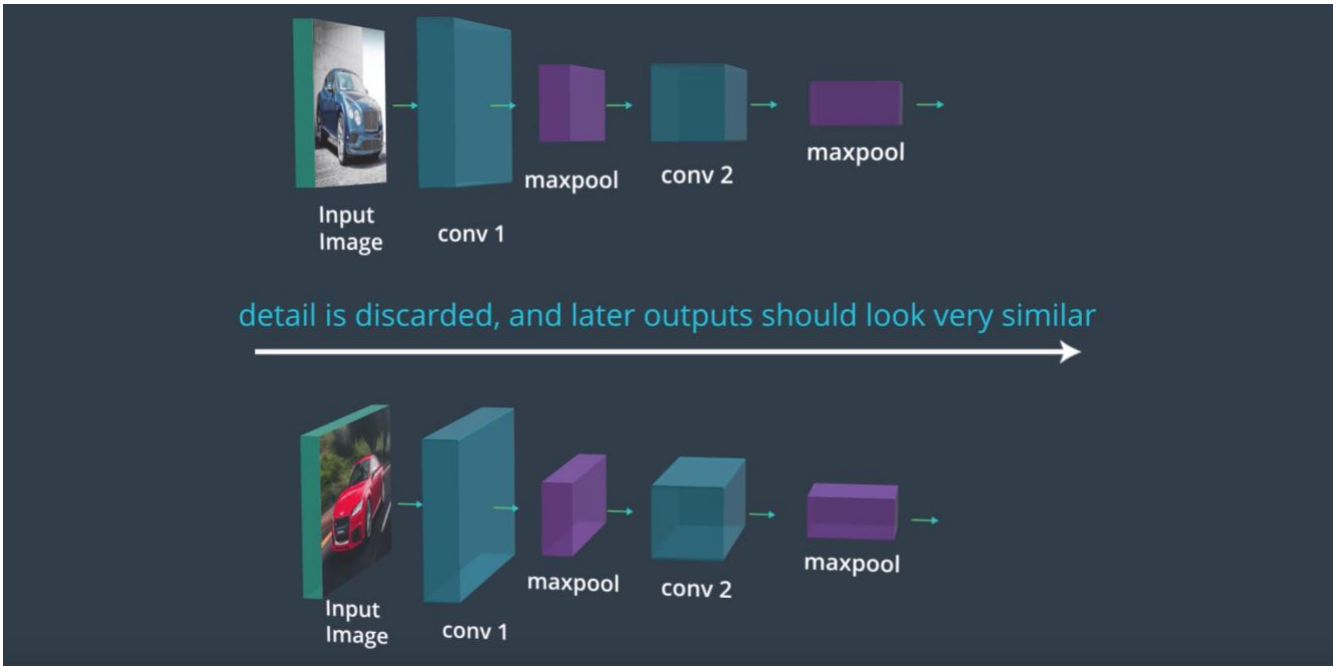
**Flattening**

Part of completing a CNN architecture, is to flatten the eventual output of a series of convolutional and pooling layers, so that all parameters can be seen (as a vector) by a linear classification layer. At this step, it is imperative that you know exactly how many parameters are output by a layer.

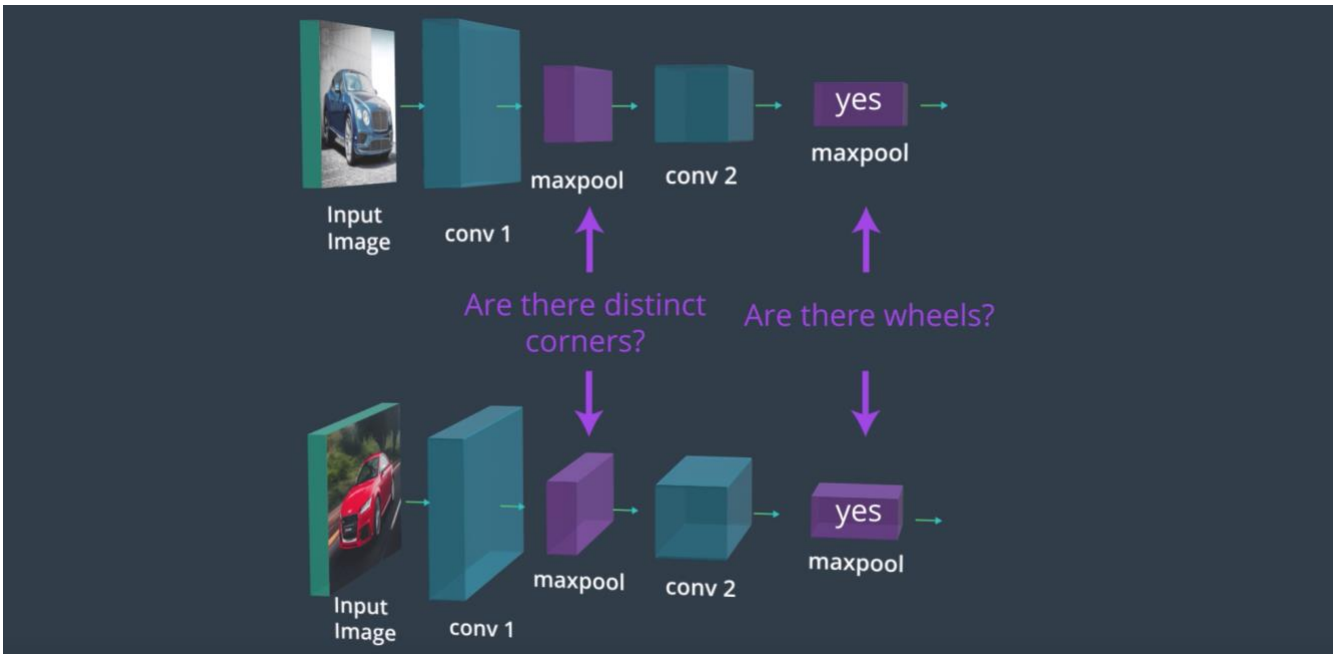
For the following quiz questions, consider an input image that is 130x130 (x, y) and 3 in depth (RGB). Say, this image goes through the following layers in order:

nn.Conv2d(3, 10, 3)
nn.MaxPool2d(4, 4)
nn.Conv2d(10, 20, 5, padding=2)
nn.MaxPool2d(2, 2)

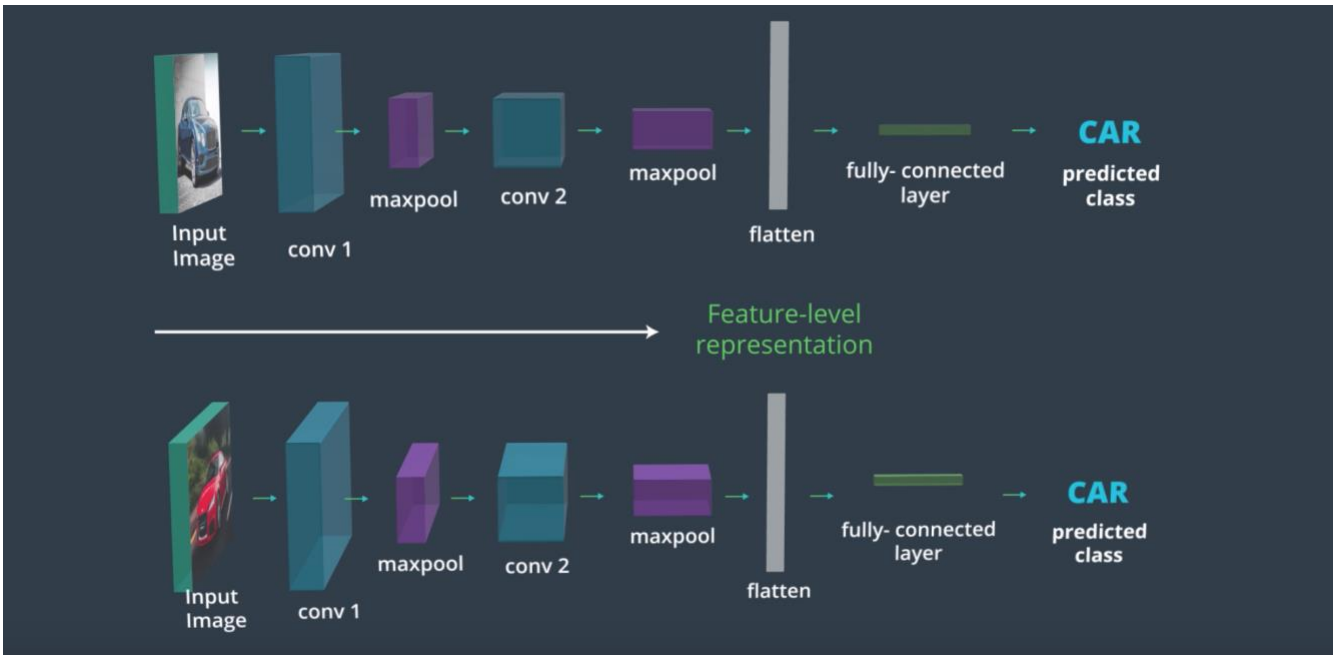
Not_read_end




118. The job of a convolutional neural network is to discover patterns contained in an image. A sequence of layers is responsible for this discovery. The layers in a CNN convert an input image array into a representation that encodes only the content of the image. This is often called a feature level representation of an image or a feature vector. It may be helpful to think about it like this, take two input images that are both images of a car. Both are very different, and if I was to frame these pictures, the detail would be what makes them stylistically interesting. But for an image classifier, it only wants to know that these are both images of cars. When you look at how these images are transformed as they move through several layers of a CNN, the exact original pixel values matter less and less. The two transformed outputs should start to look much more similar to one another, moving closer towards the idea that both are a car rather than the details about what the cars look like. Later layers in a CNN have discarded information about style and texture,



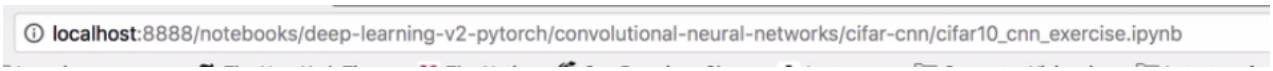

and instead are pushed towards answering questions about general shape, and about the presence of unique patterns, like, "Are there wheels in the image? Are there eyes in the image? What about three legs or tails?"




Once we get to a representation where the content of an image has been distilled like this, we can flatten the array into a feature vector and feed it to one or more fully connected layers to determine what object is contained in the image. For instance, if wheels were found after the last max pooling layer, the feature vector will be able to reflect that. The fully connected layer will transform that information to predict that a car is present in the image with higher probability. If there were eyes, three legs, and a tail, then the output layer would take that information and deduce that a dog is likely present in the image. But of course, to emphasize, all this understanding in the model, is not pre-specified bias. It is learned by the model during training and through back-propagation that updates the weights that define the filters and the weights in the fully connected layers. This architecture that we're specifying here just gives the model a structure that will allow it to train better. So, it has the potential to classify objects with greater accuracy. Next, I'll show you how to start defining a CNN architecture for image classification, and you'll get to practice coding on your own.

Not_read_begin

The following paragraph is about this notebook:

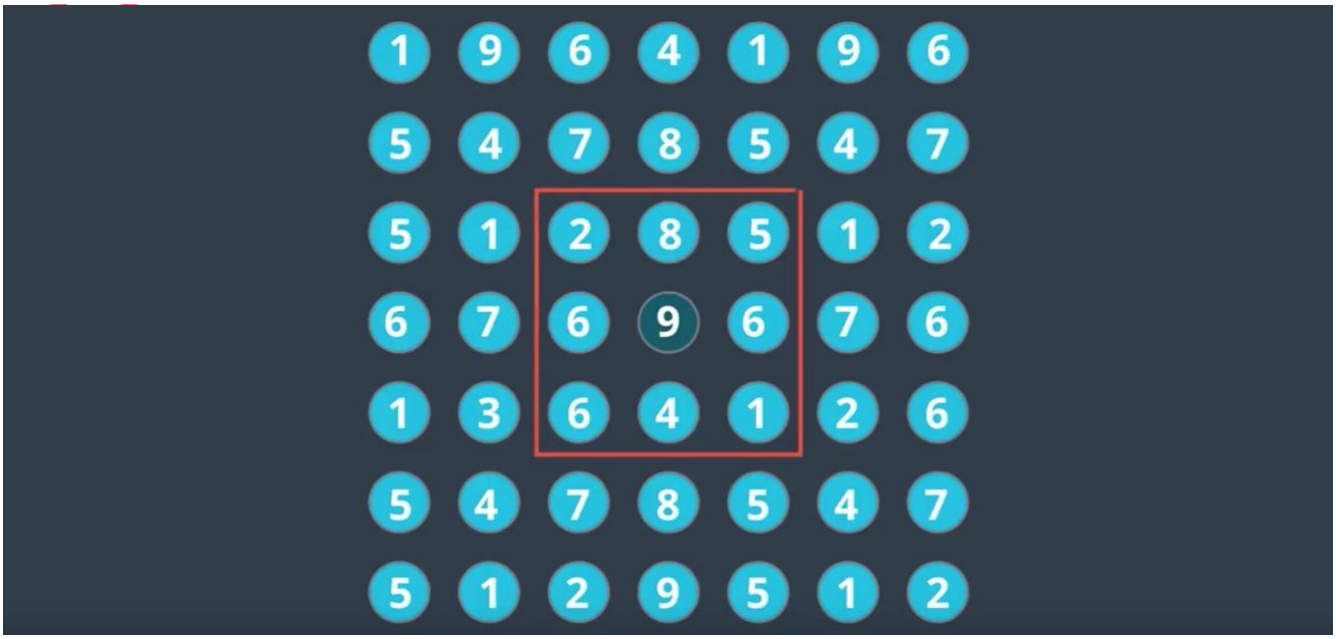


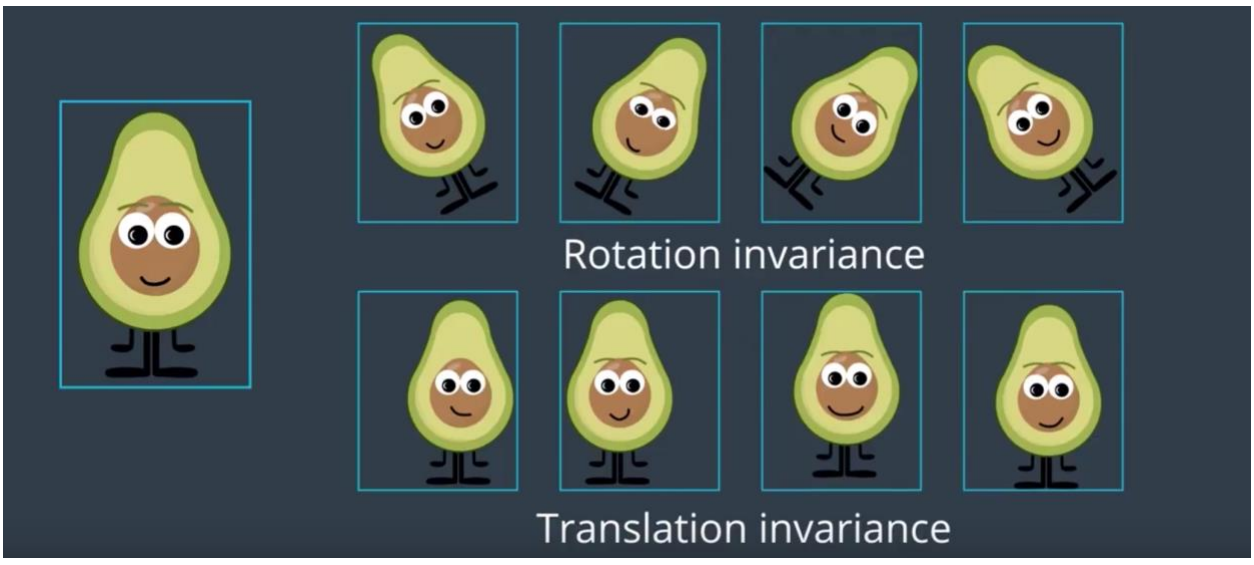

119. CIFAR-10 is a popular dataset of 60,000 tiny images, each depicting an object from one of 10 classes. You can see that each image is truly tiny. Only 32 pixels high and wide. These are color images so they're interpreted by the computer as arrays with a depth of three for RGB color channels. Because these are color images and there are a lot of them, the first thing I'm going to show you is an option to use a GPU for training. A GPU is essentially something that allows you to do data processing in parallel and so it may prove useful to speed up your training time. Here, you can see I'm loading in my usual libraries and then PyTorch gives us a way to check if a GPU

device is available. Torch.cuda is available will return a Boolean variable true or false whether or not a GPU is available. We'll store this and then later if we do have a GPU available will be able to move our data and our model to that device for faster training. For now, I'm just going to visualize some data and present this problem and so I'm using my laptop and when I run this cell you can see that I'm training on CPU. Later, when I do try to develop a solution and train a model, I'll switch to GPU and in general this is a good thing to know how to use. Next, I'm going to load in my data as usual. CIFAR, much like is Amnest is available in the torchvision datasets library. I'm going to define my training and my test data by calling datasets.CIFAR10. I'll also transform this data into a tensor datatype and normalize the RGB values so that the pixel values are in a range from about zero to one. Finally, I can load in my transform data in batches using PyTorch's data loader class. These lines of code should look very similar to the Amnest code in which we loaded in data, transformed it into a tensor datatype, and separated the data into training, validation, and test sets. Also down here, I'm specifying the 10 image classes that I'm reading in. All CIFAR-10 images will fall into one of these categories. The data may take a moment to load in but once you load it in you can visualize a batch of that data. Here, I've defined a helper function which will basically unnormalize the images and convert them from a tensor image type to a non-py image type for visualization, then I'm just going to load in and display a batch of images and here we can verify that these images look pretty much how you would expect. We have images of cats, frogs, deer, and automobiles. We can even choose to view an image in more detail. Here, I'm displaying the red, green, and blue color channels as grayscale values for one single image and we should again see that the brightest pixel values are close to the value one and darker ones closer to zero. Next, you'll want to define and train a CNN to classify these images. You're also welcome to try out an MLP approach, see how both work, and compare the results. I'll be assuming that you want to define a CNN architecture so I provided links to the documentation for convolutional and maxpooling layers in PyTorch. Here's also a simple diagram showing how an input image might pass through a couple of these layers. Then if you scroll down, you can see that I've defined one example convolutional layer for you. I've defined it in the init function of our network. To define a convolutional layer, I use nn.Conv2d and I pass in some parameters. For our first convolutional layer, this will be the number of inputs seen as the depth of the input image. Recall that our inputs are 32 by 32 images with a depth of three for RGB color channels. So, I've defined the input and I've specified that this should output a convolutional layer with a depth of 16. That means this layer should take in our image and produce 16 filtered images as output. I've also specified that I'm using filters that are 3 by 3 in size and to keep my output layer the same x y size as my input image, I'll add one pixel of padding on the border. I've also defined one maxpooling layer named pool. This has a kernel size and stride size of two which means that any input it's applied to it will downsample it x y dimensions by a factor of two. Then in the forward function, we can see how all of these things fit together. For an input image x, I first pass it into our first convolutional layer and I apply a relu activation function. This output will be passed to a final pooling layer resulting in a downsampled transformed x which I'll return. To complete this model, it will be up to you to add multiple convolutional and pooling layers and finally flatten and apply a fully-connected layer for producing the desired number of class score outputs. After defining a complete CNN, you can instantiate it and even move it to GPU if it's available. Next, it'll be up to you to specify an appropriate loss and optimization function for this training task and finally, I've provided a training loop for you. You may want to increase the number of epochs you train your final model for. But this loop will keep track of the training and validation loss as you go. If you're validation loss decreases over an epoch, your model will be saved. Then you'll be able to test your train network

and see how it performs on each type of test image. The biggest challenge will be defining a complete CNN and as always I encourage you to do your research before you define your model so that you can make an informed decision as you go. Try to solve this challenge on your own and if you want to check your work next I'll go over one possible solution.

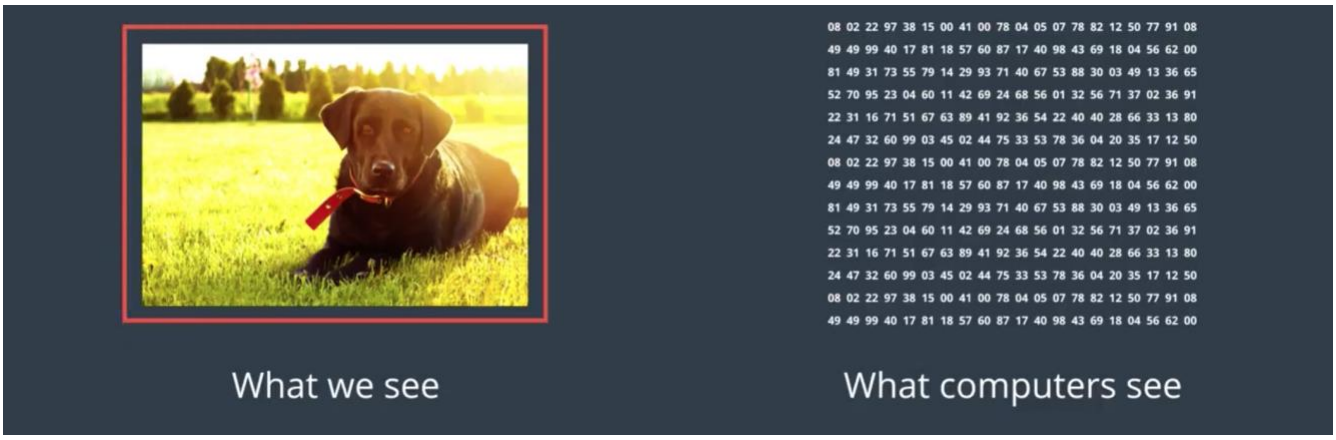
Not_read_end



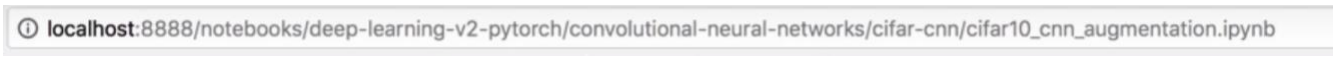

120. When we design an algorithm to classify objects in images, we have to deal with a lot of irrelevant information. We really only want our algorithm to determine if an object is present in the image or not. The size of the object doesn't matter, neither does the angle, or if I move it all the way to the right side of the image. It's still an image with an avocado(鱷梨). In other words, we can say that we want our algorithm to learn an invariant representation of the image. We don't want our model to change its prediction based on the size of the object. This is called scale invariance.

Likewise, we don't want the angle of the object to matter. This is called rotation invariance. If I shift the image a little to the left or to the right, well, it's still an image with an avocado, and this is called translation invariance. CNN's do have some built-in translation invariance. To see this, you'll need to first recall how we calculate max-pooling layers. Remember that at each window location, we took the maximum of the pixels contained in the window. This maximum value can occur anywhere within the window. The value of the max-pooling node would be the same if we translated the image a little to the left, to the right, up, down, as long as the maximum value stays within the window. The effect of applying many max-pooling layers in a sequence each one following a convolutional layer, is that we could translate the object quite far to the left, to the top of the image, to the bottom of the image, and still our network will be able to make sense of it all.




This is truly a non-trivial problem. Recall that the computer only sees a matrix of pixels. Transforming an object's scale, rotation, or position in the image has a huge effect on the pixel values. We as humans can see the difference in images quite clearly, but how do you think you'd do if you were just given the corresponding array of numbers? Thankfully, there's a technique that works well for making our algorithms more statistically invariant, but it will feel a little bit like cheating. The idea is this, if you want your CNN to be rotation invariant, well, then you can just add some images to your training set created by doing random rotations on your training images. If you want more translation invariance, you can also just add new images created by doing random translations of your training images. When we do this, we say that we have expanded the training set by augmenting the data. Data augmentation will also help us to avoid overfitting. This is because the model is seeing many new images. Thus, it should be better at generalizing and we should get better performance on the test dataset. Let's augment the training data and the CFR 10 dataset from the previous video, and see if we can improve our test accuracy. We'll be using a Jupiter notebook that you can download below.

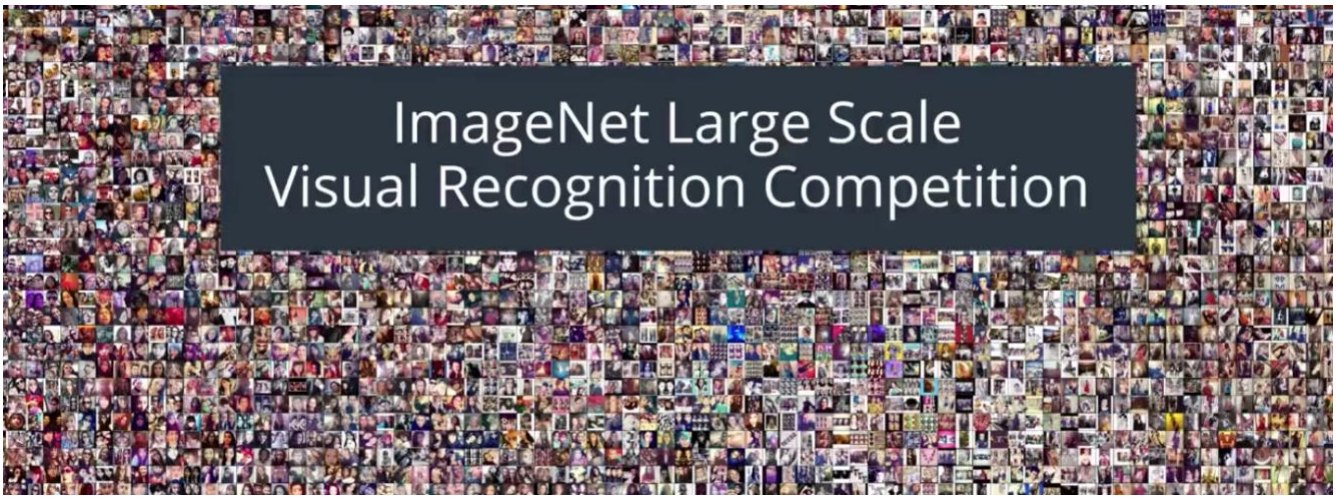
Not_read_begin



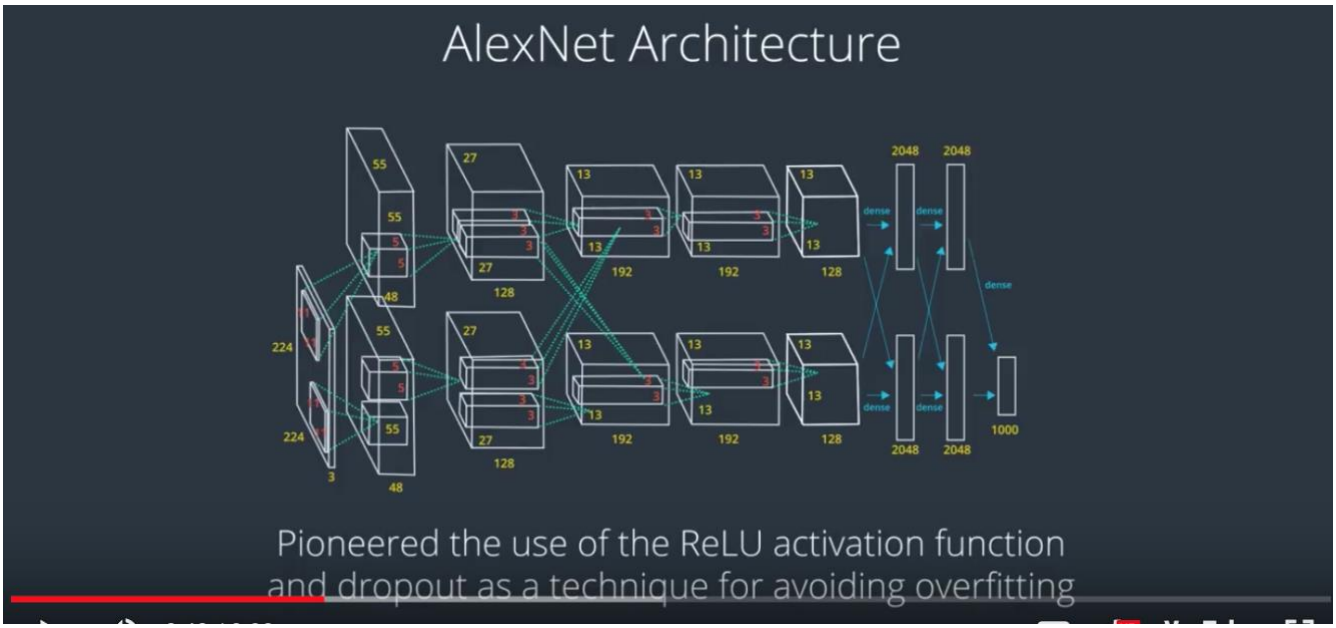

121. To perform image augmentation in PyTorch, you can use the help of a built-in image transformation library. Here you can scroll through all the common transforms available in the transforms library. You can see transforms that shift around the image from left to right or that crop

the image randomly. In our case, we want our dataset to be a little more rotation and scale invariant and so we can choose random transforms that change the scale and how much an image is rotated. I'm going to briefly show you how to do this in code and then you can choose whether or not to use it in your own work. I'm going to augment our data in the same step that I'm loading it in. Basically, I want to add to our composite transform. Where you've previously just seen conversion to tensors and normalization you can put other types of transforms as well. This transformation randomly flips an image along its horizontal axis and this code randomly rotates an image by 10 degrees. I apply this transform as usual when I form my training and test data. And that's all you need to do to give your images some geometric variation. After transforming my data, I've gone through the same visualization steps and we can see by looking at the borders of some of these images that some of our images have been rotated slightly right or left within 10 degrees. Then to see if data augmentation actually improve the performance of our model I use the same convolutional neural network and loop that I did before. I trained the model over 30 epics using our augmented training data. In this case it seemed that the training and validation mass took a little longer to decrease but may have reached a smaller minimum. I saved this augmented model and when I tested it, I got one percent more accuracy in general. This is a minor increase in test accuracy but the small performance improvements can add up and there's something worthwhile if you're really trying to find the best model for a task. Generally in this course, your job will be to design a model so that it can reach a high accuracy but in addition to that, choices about data augmentation and loss and optimization functions can take that even further. So this is a useful skill to have.

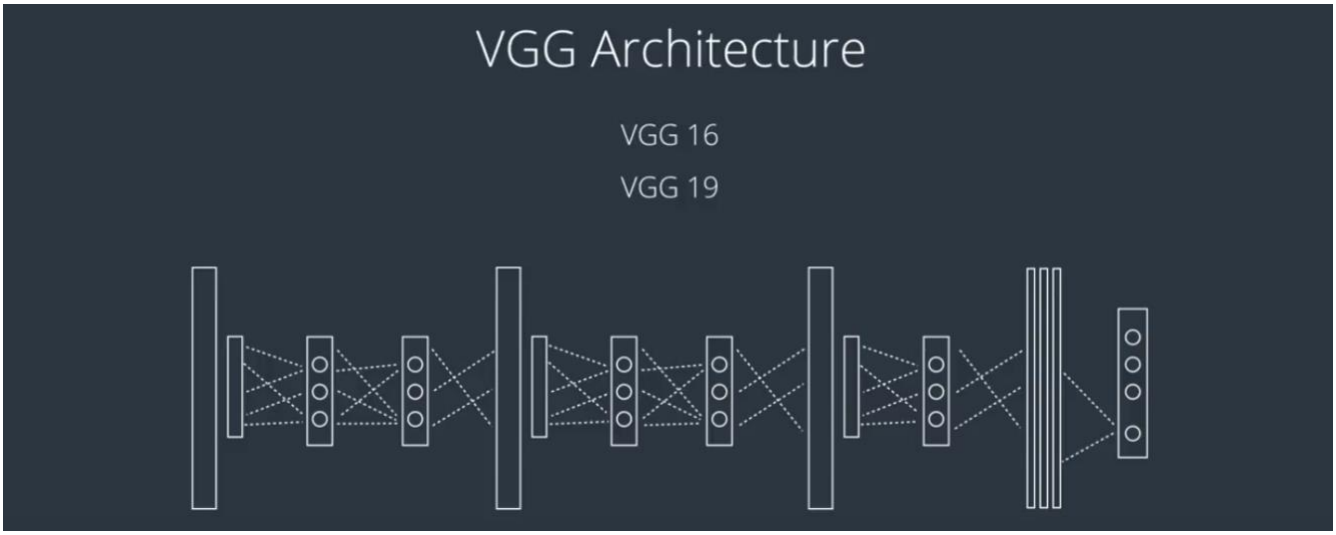
Not_read_end



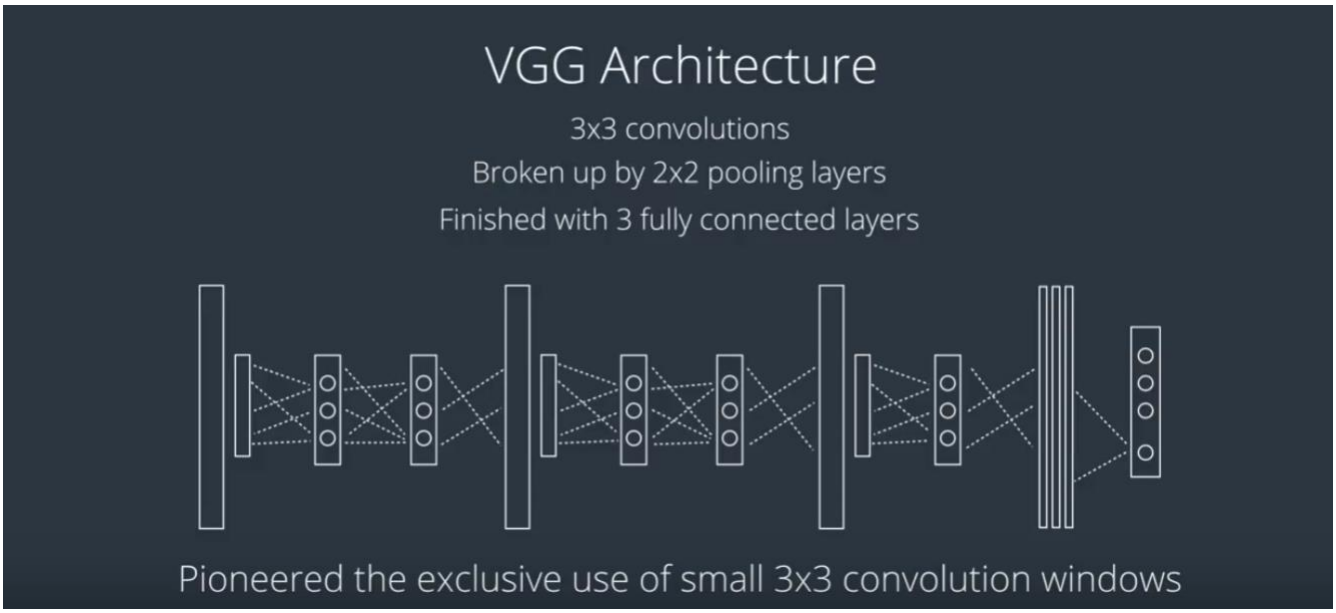

122. ImageNet is a database of over 10 million hand labeled images, drawn from 1,000 different image categories. Since 2010, the ImageNet project has held the ImageNet Large Scale Visual Recognition Competition, an annual competition where teams try to build the best CNN for object recognition and classification.



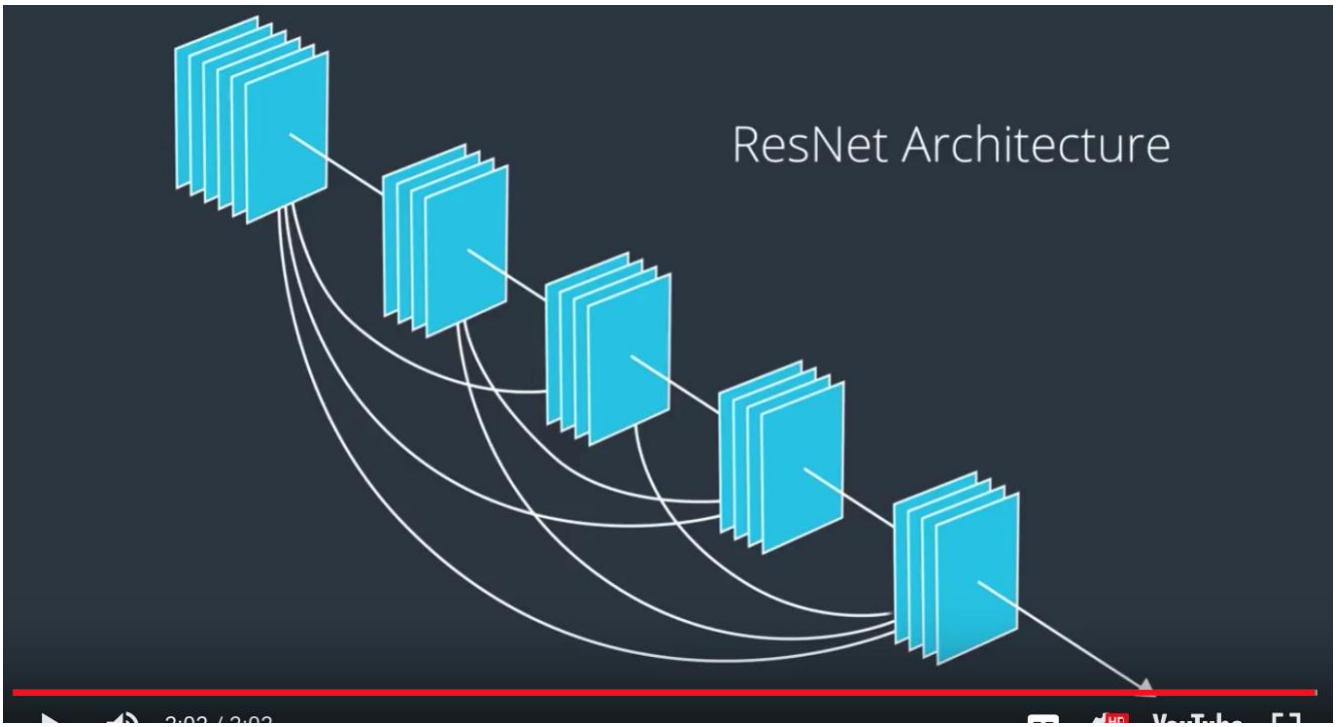

The first breakthrough was in 2012. The network called AlexNet, was developed by a team at the University of Toronto. Using the best GPUs available in 2012, the AlexNet team trained the network in about a week. AlexNet pioneered the use of the ReLU activation function, and dropout as a technique for avoiding overfitting.



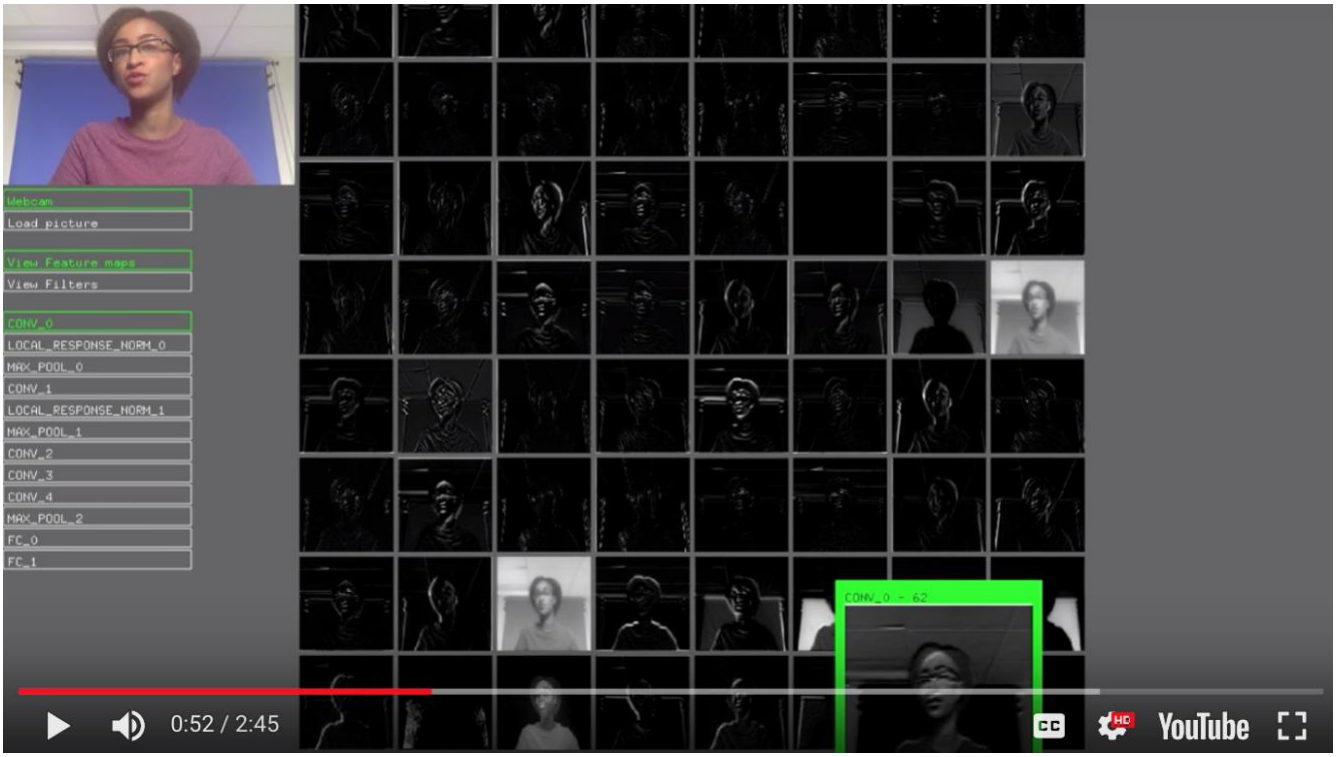

In 2014, two different groups nearly tied in the ImageNet competition. One of those networks was called VGGNet, often referred to as just VGG, and it came from the Visual Geometry Group at Oxford University. VGG has two versions termed VGG16, and VGG 19, with 16 and 19 total layers respectively.



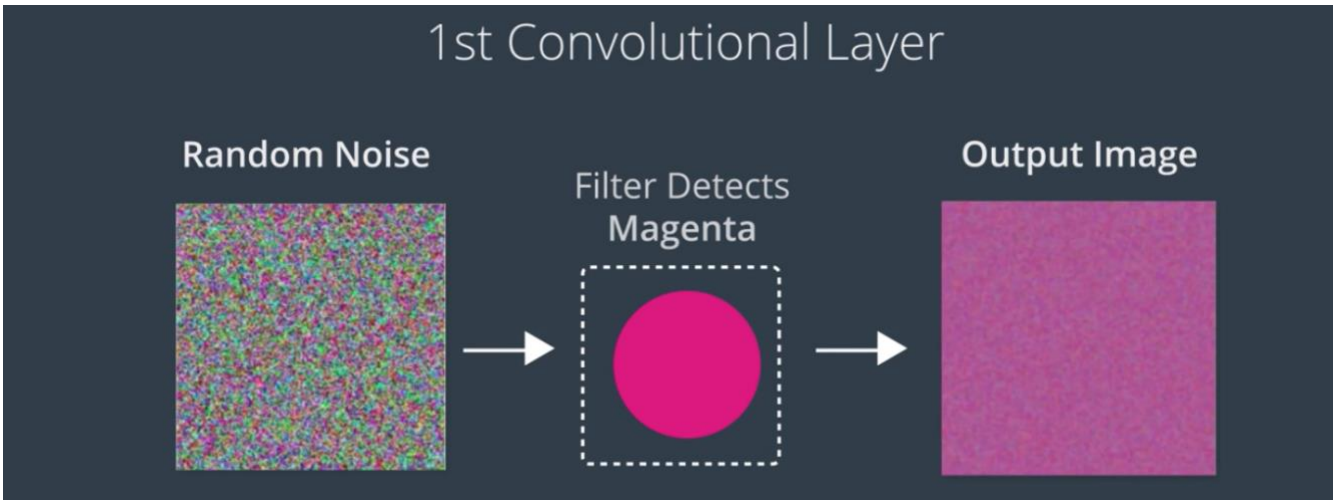

Both versions have a simple and elegant architecture, which is just a long sequence of three by three convolutions, broken up by two by two pooling layers, and finished with three fully-connected layers. VGG pioneered the exclusive use of small three by three convolution windows, to contrast AlexNets much larger 11 by 11 windows.



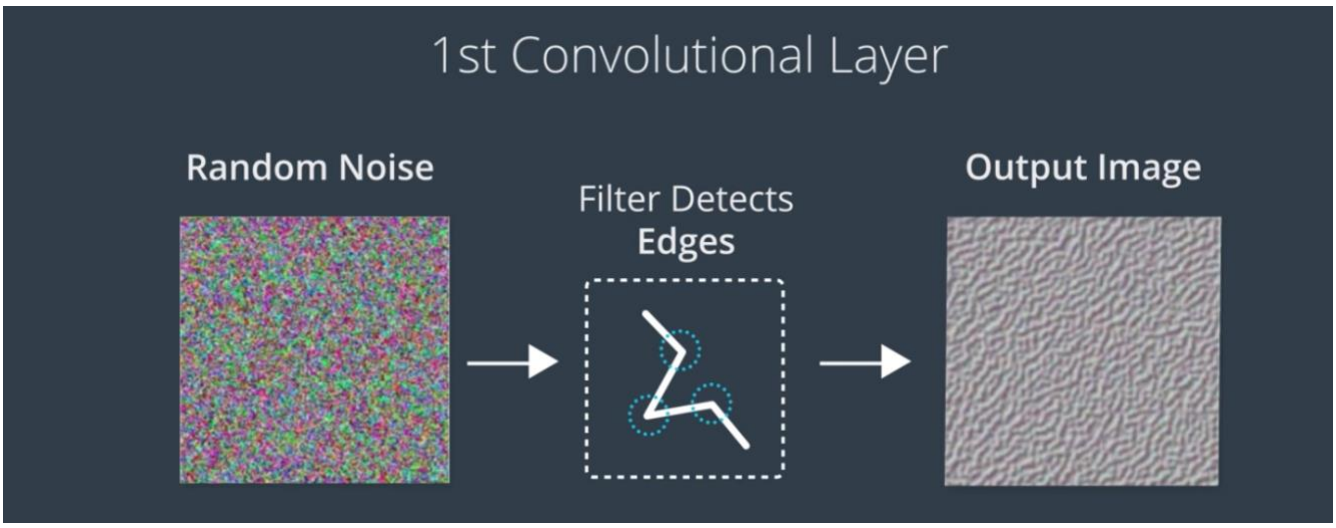
In 2015, the ImageNet winner was a network from Microsoft Research called ResNet. ResNet is like VGG and not the same structure is repeated again and again, for layer after layer. Also like VGG, ResNet has different versions that vary in their number of layers. The largest having a groundbreaking 152 layers. Previous researchers tried to make their CNNs this deep, but they ran into a problem where as they were adding layers, performance increased up to a point, after which performance quickly declined. This is partially due to what's known as the vanishing gradient's problem, which arises when we go to train the network through backpropagation. The main idea is that the gradient signal has to be pushed through the entire network. The deeper the network becomes, the more likely that the signal gets weakened before it gets where it needs to go. The ResNet team added connections to their very deep CNN that skip layers. So, the gradient signal has a shorter route to travel. ResNet achieves superhuman performance in classifying images in the ImageNet database.



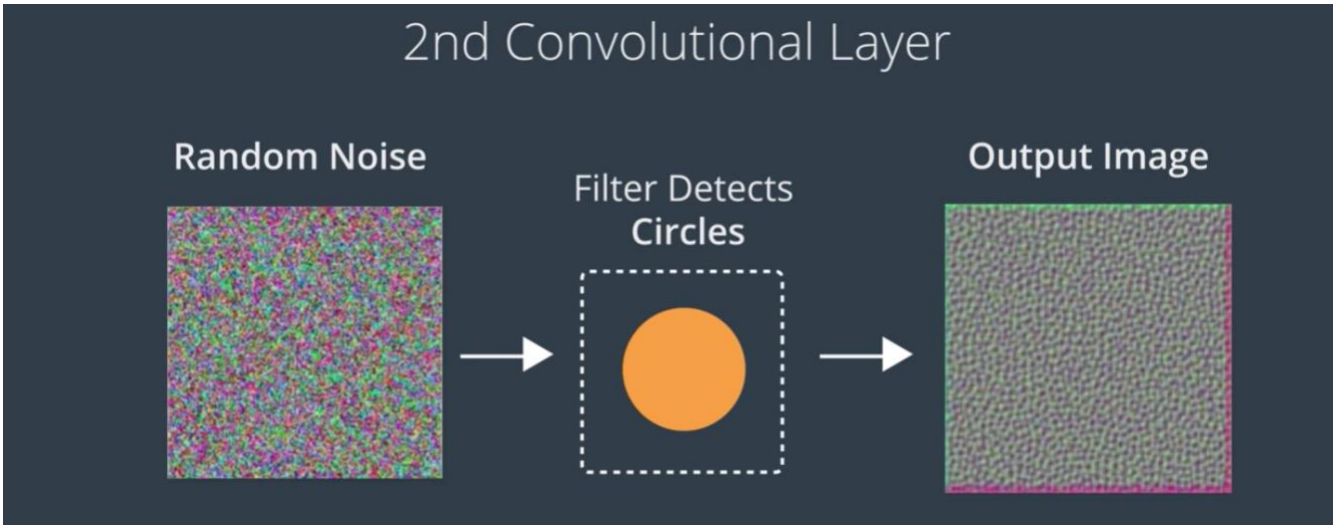

123. We've seen that CNNs achieve state-of-the-art and sometimes superhuman performance in object classification tasks. We've explored some techniques for building CNN architectures, and we've gotten some good performance on toy datasets. But at the end of the day, we don't really have a strong understanding of how these CNNs discover patterns in raw image pixels. If you've already taken the time to train your own CNNs, you've noticed that some architectures work while some just don't. And if it's not clear to you why that's the case, it's probably not truly clear to the experts either. Earlier in this lesson, I mentioned visualizing the activation maps and convolutional layers as one technique for digging deeper into understanding how a CNN is working. There are many implementations of this online. This is one of them, and as you can see, it's designed to pass your webcam's video through a trained CNN in real time. If you'd like to play around with this, you're encouraged to check out the links below.

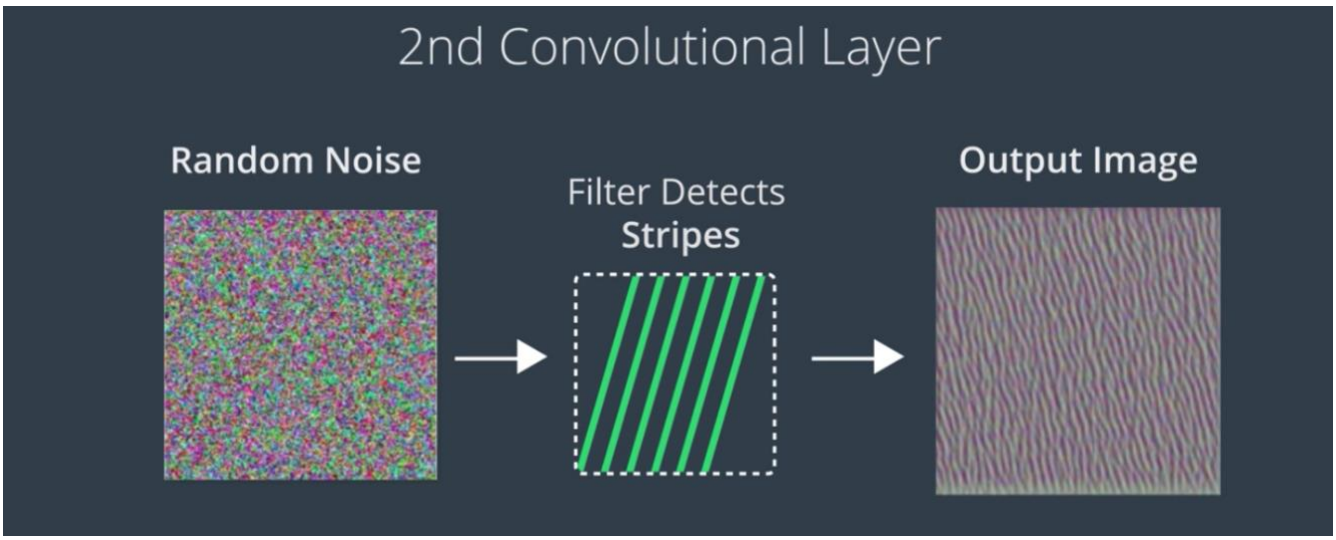



Another technique designed for understanding CNNs involves taking the filters from our convolutional layers and constructing images that maximize their activations. You can start with an image containing random noise and then gradually amend the pixels, and each step changing them to values that make the filter more highly activated.

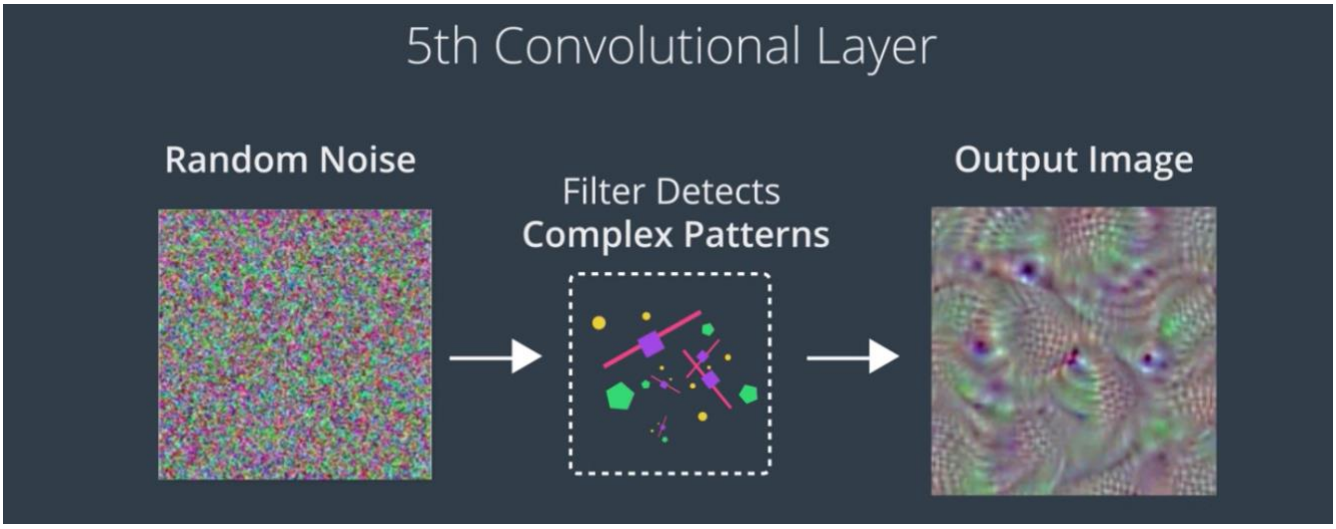


When you do this, you'll notice that the first three layers are pretty general. The first layer might include color or edge detectors,

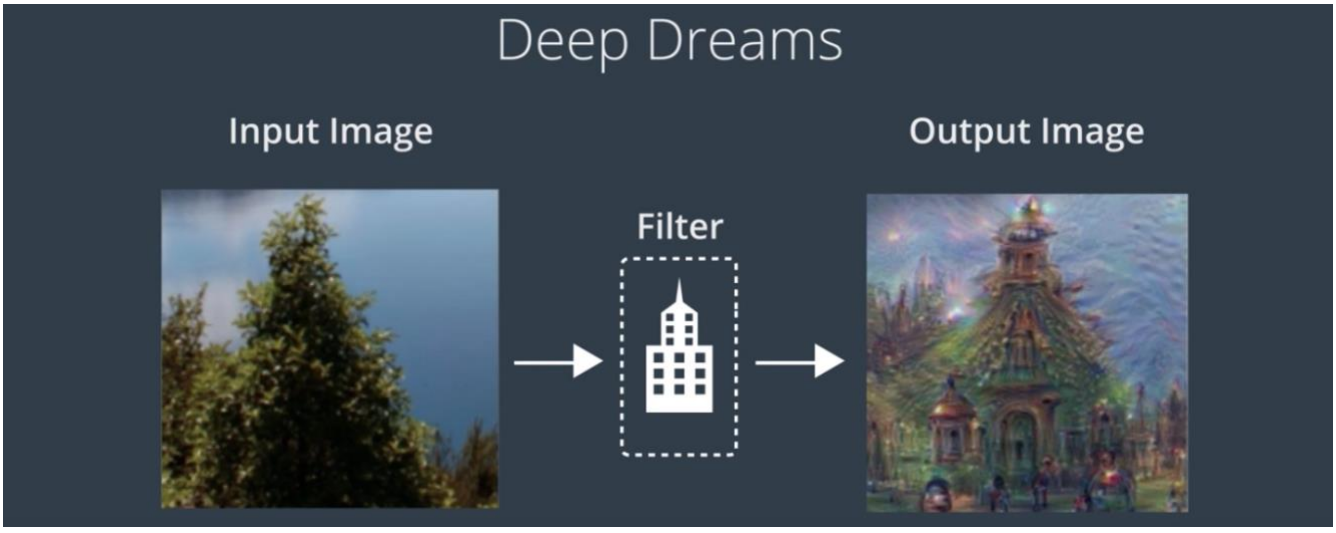



where the second layer detects circles or stripes.




The filters later in the network are activated by much more complicated patterns.

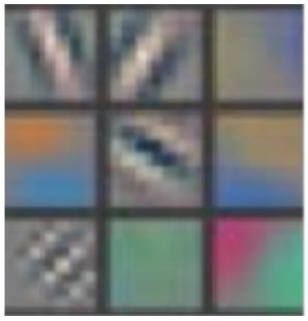



Researchers at Google got creative with this. They designed a technique called deep dreams where they replace the starting image with a picture. Say we have a picture of a tree, but you can use a picture of anything here. Then, you could investigate one of the more interesting filters, maybe one that looks like it's used for detecting a building. When you follow the same technique where the only thing you've changed is the starting array of pixels, you end up creating an image that looks like some sort of tree or building hybrid. If you'd like to code this yourself in Keras, you're encouraged to pursue this in the links below. There are many other techniques for visualizing what a CNN learns. We've only scratched the surface with this, but you're definitely encouraged to explore more in this area. The idea is that if we can understand more of what a CNN learns, we can help it to perform even better.

**Visualizing CNNs**

Let’s look at an example CNN to see how it works in action.

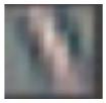
The CNN we will look at is trained on ImageNet as described in this paper by Zeiler and Fergus. In the images below (from the same paper), we’ll see what each layer in this network detects and see how each layer detects more and more complex ideas.



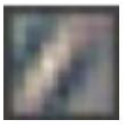
Example patterns that cause activations in the first layer of the network. These range from simple diagonal lines (top left) to green blobs (bottom middle).

The images above are from Matthew Zeiler and Rob Fergus' deep visualization toolbox, which lets us visualize what each layer in a CNN focuses on.

Each image in the above grid represents a pattern that causes the neurons in the first layer to activate - in other words, they are patterns that the first layer recognizes. The top left image shows a -45 degree line, while the middle top square shows a +45 degree line. These squares are shown below again for reference.

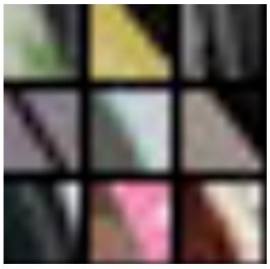


As visualized here, the first layer of the CNN can recognize -45 degree lines.



The first layer of the CNN is also able to recognize +45 degree lines, like the one above.

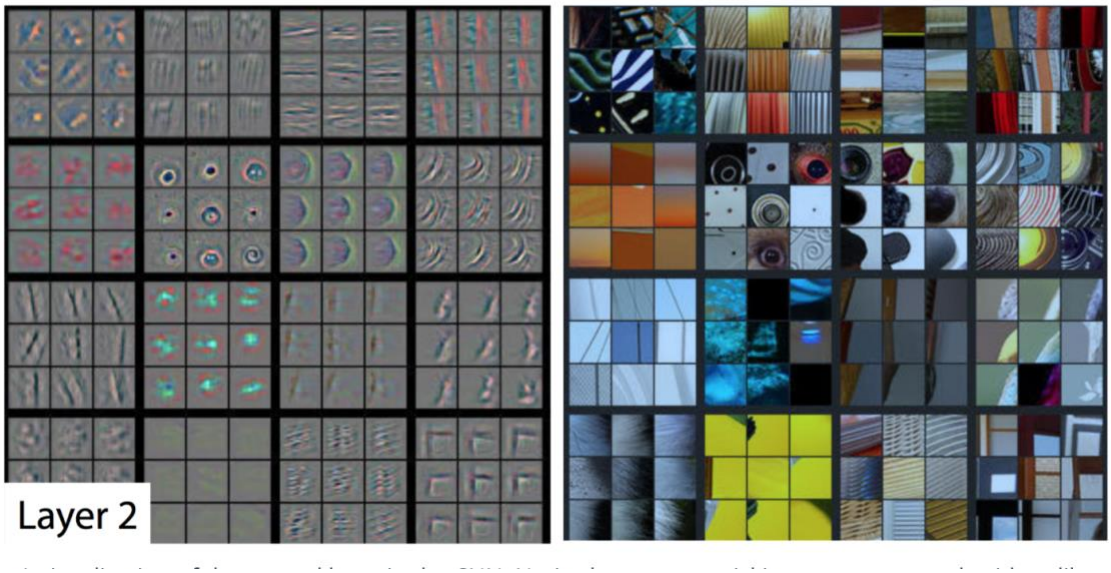
Let's now see some example images that cause such activations. The below grid of images all activated the -45 degree line. Notice how they are all selected despite the fact that they have different colors, gradients, and patterns.



Example patches that activate the -45 degree line detector in the first layer.

So, the first layer of our CNN clearly picks out very simple shapes and patterns like lines and blobs.

**Layer 2**



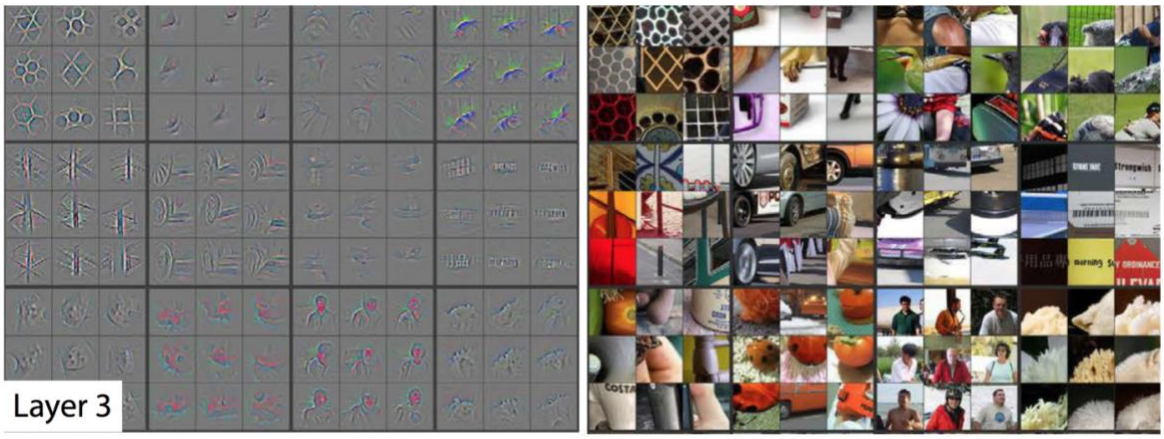

A visualization of the second layer in the CNN. Notice how we are picking up more complex ideas like circles and stripes. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The second layer of the CNN captures complex ideas.

As you see in the image above, the second layer of the CNN recognizes circles (second row, second column), stripes (first row, second column), and rectangles (bottom right).

The CNN learns to do this on its own. There is no special instruction for the CNN to focus on more complex objects in deeper layers. That's just how it normally works out when you feed training data into a CNN.

**Layer 3**

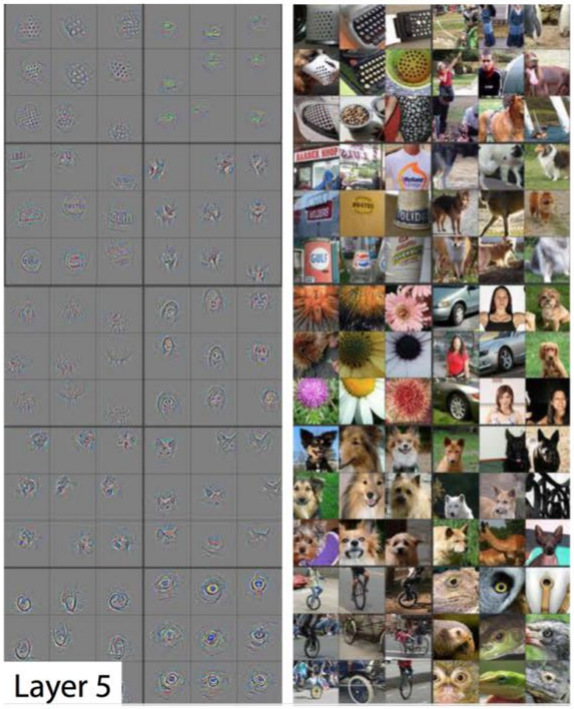



A visualization of the third layer in the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The third layer picks out complex combinations of features from the second layer. These include things like grids, and honeycombs (top left), wheels (second row, second column), and even faces (third row, third column).

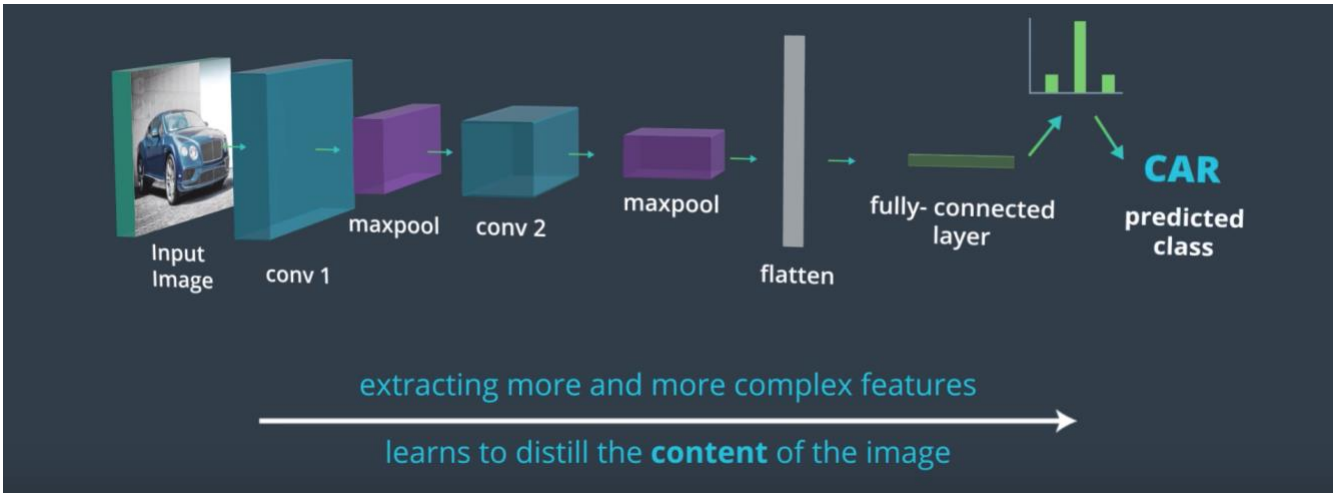
We'll skip layer 4, which continues this progression, and jump right to the fifth and final layer of this CNN.

**Layer 5**




A visualization of the fifth and final layer of the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The last layer picks out the highest order ideas that we care about for classification, like dog faces, bird faces, and bicycles.




124. So far you've seen how to create two kinds of neural networks for image classification, and you've seen how convolutional neural networks follow a series of steps to classify an image. Just to recap, a CNN first takes in an input image, then puts that image through several convolutional and pooling layers. The result is a set of feature maps reduced in size from the original image that

through a training process have learned to distill information about the content in the original image. We then flatten these maps, creating a feature vector that we can then pass to a series of fully-connected linear layers to produce a probability distribution of class course. From this, we can extract the predicted class for the input image.


In short, an image comes in and a predicted class label comes out. You've learned the foundational concepts behind CNN's and in later lessons you'll learn about techniques that leverage existing architectures and even more techniques for improving the performance of a model. You should also know that CNN's are not restricted to the task of image classification. They can be applied to any task with a fixed number of outputs such as regression tasks that look at points on a face or detect human poses. The applications of CNN's are very far-reaching and after seeing just a few, I hope you're excited to learn even more.