Convolutional Neural Networks

20210413

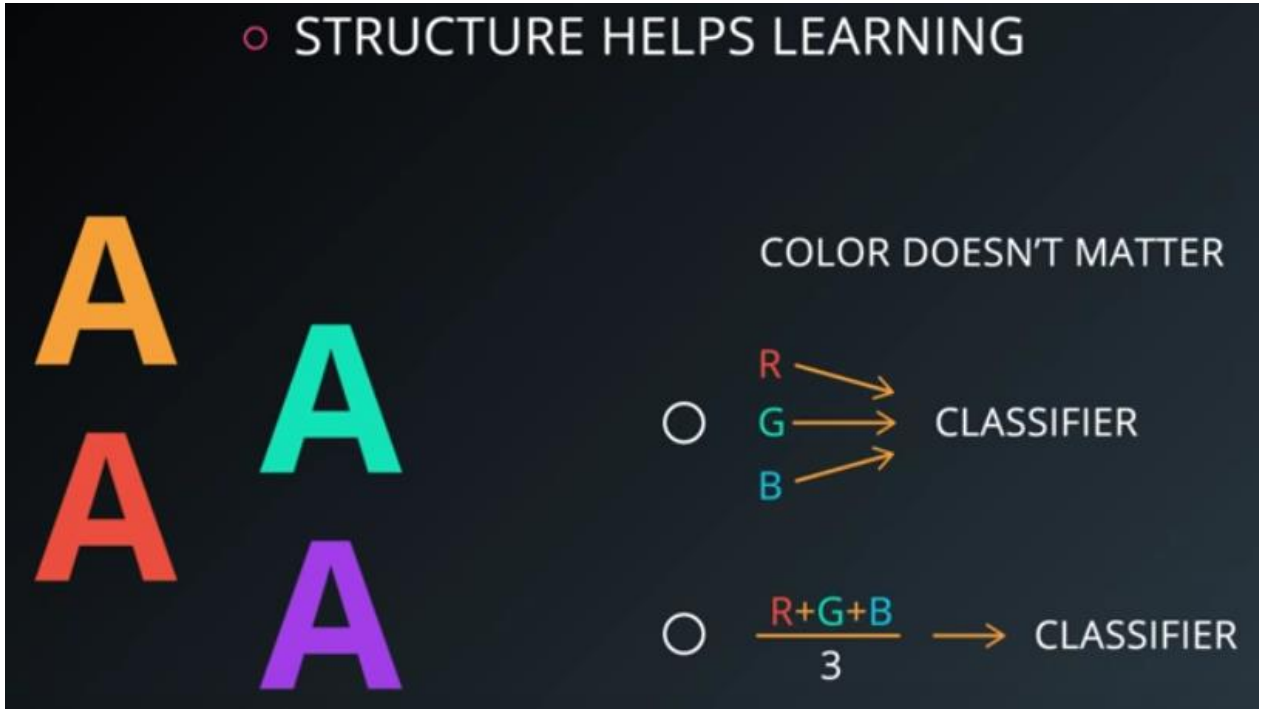
Intro To CNNs

Buzzword : CNN

Convolutional neural networks (CNN, ConvNet) is a class of deep, feed-forward (not recurrent) artificial neural networks that are applied to analyzing visual imagery.

It was invented by Yann LeCun about 30 years ago

Color

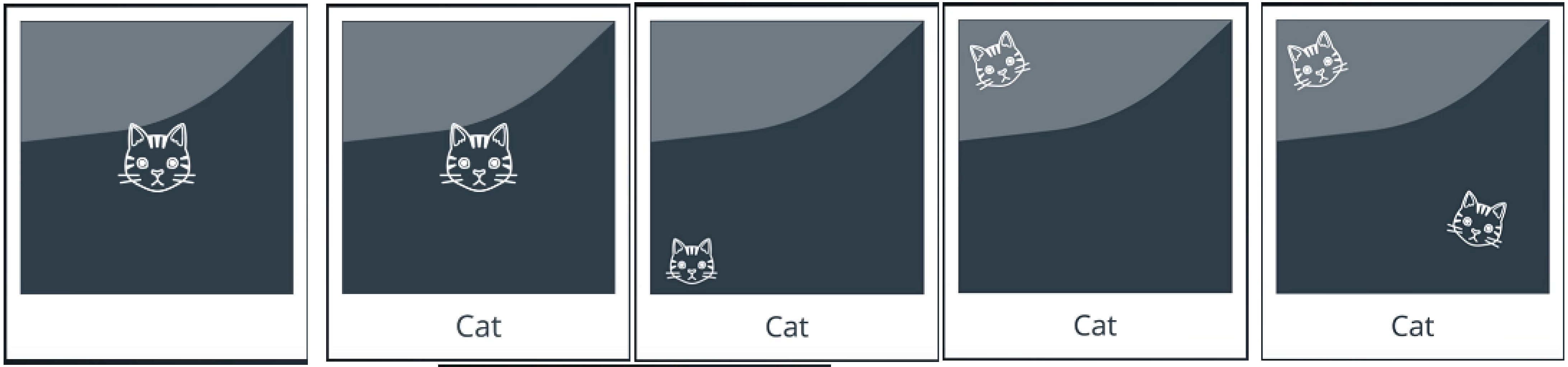


- STRUCTURE HELPS LEARNING

COLOR DOESN'T MATTER

A model that uses the color image, or a model that only looks at the gray scale?

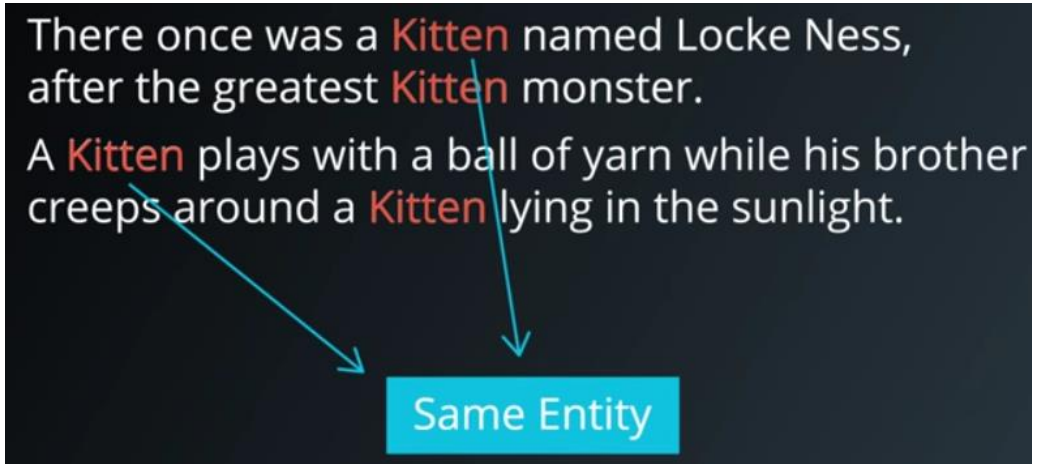
Statistical Invariance



Translation Invariance
Cat

Explicitly, that objects and images are largely the same whether they are on the left or on the right of the picture. That’ s what’ s called translation invariance.

Statistical Invariance



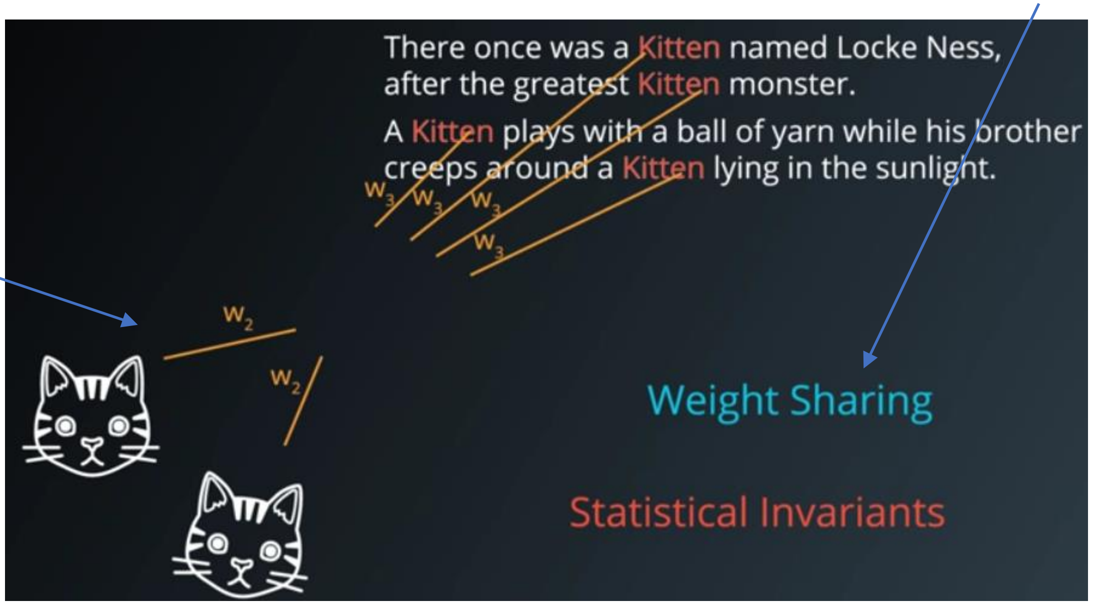
There once was a Kitten named Locke Ness, after the greatest Kitten monster.

A Kitten plays with a ball of yarn while his brother creeps around a Kitten lying in the sunlight.

If 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 relearn it everytime.

The way you achieve this in your own network is using what’s called “weight sharing”.

When you know that two inputs can contain the same kind of information, the you share the weights and train the weight jointly for these inputs.

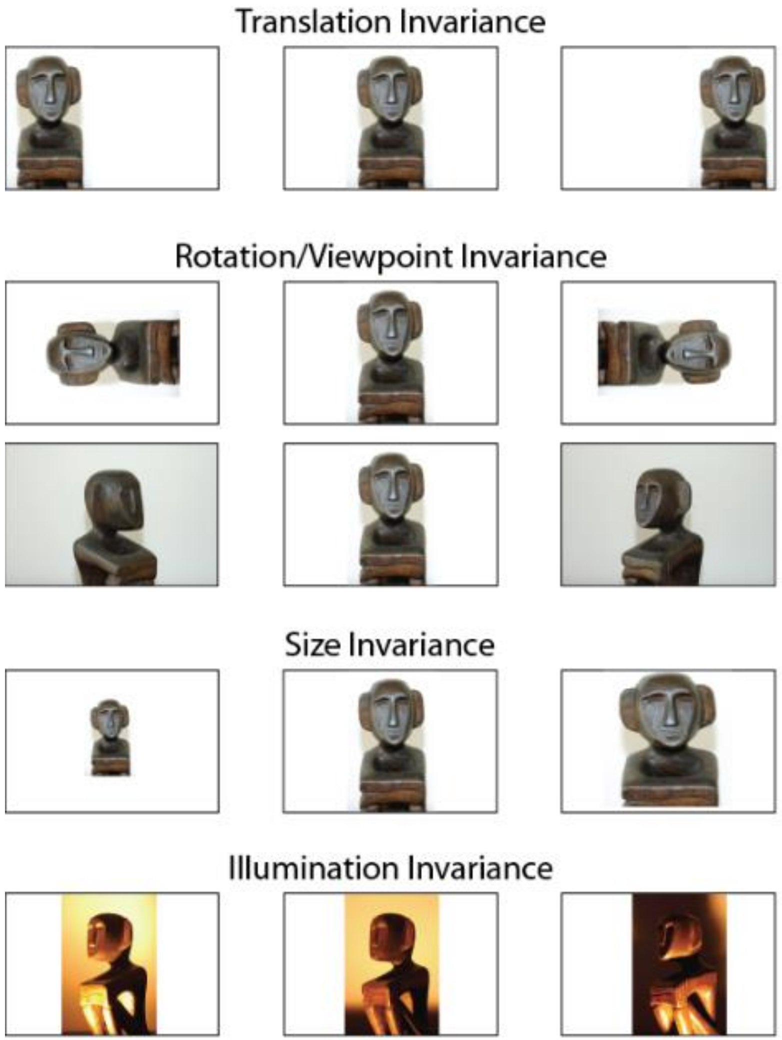


There once was a Kitten named Locke Ness, after the greatest Kitten monster.

A Kitten plays with a ball of yarn while his brother creeps around a Kitten lying in the sunlight.

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

Complement



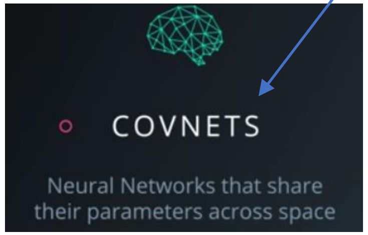
- Invariance means that you can recognize an object as an object, even when its appearance varies in some way. This is generally a good thing, because it allows to abstract an object's identity or category from the specifics of the visual input, like relative positions of the viewer/camera and the object.
- The image left contains many views of the same statue. You (and well-trained neural networks) can recognize that the same object appears in every picture, even though the actual pixel values are quite different.
- Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations.

reference : http://www.deeplearningbook.org/contents/convnets.html

Why is CNN used for image classification, and why not other algorithms?

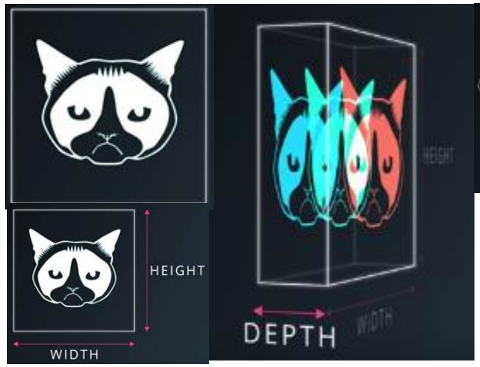
- The main reason behind CNN is feature engineering not required. Before CNN, we need to spend so much time on feature selection (algorithm for features extraction).
- Deep learning there is transfer learning happens. then it will learning more, and also less error will occurred.
- The network can be simplified by considering the properties of images.
  - Some pattern are much smaller than the whole image.
  - The same patterns appear in different regions.
  - Subsampling the pixels will not change the object.

Convolutional Networks

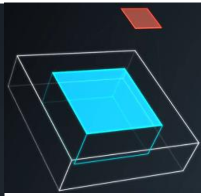


COVNETS

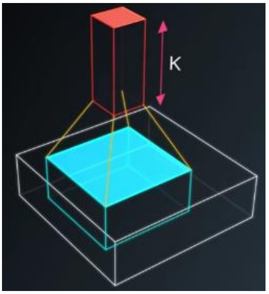
Neural Networks that share their parameters across space

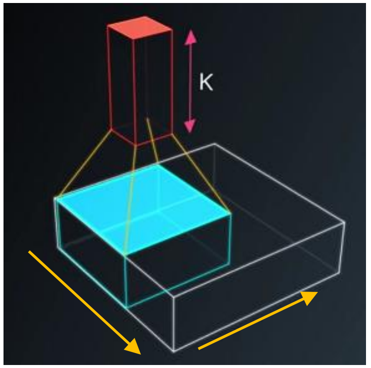


Take a small patch of this image



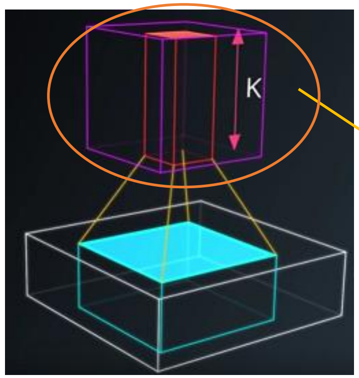
K output



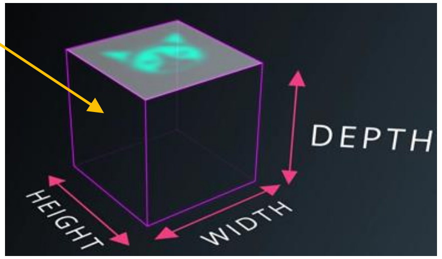


Slide across the image

K output image

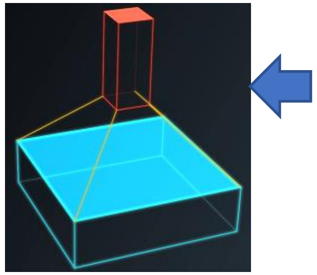


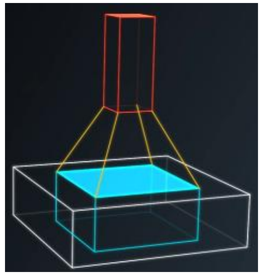
On the output, we've drawn another image.



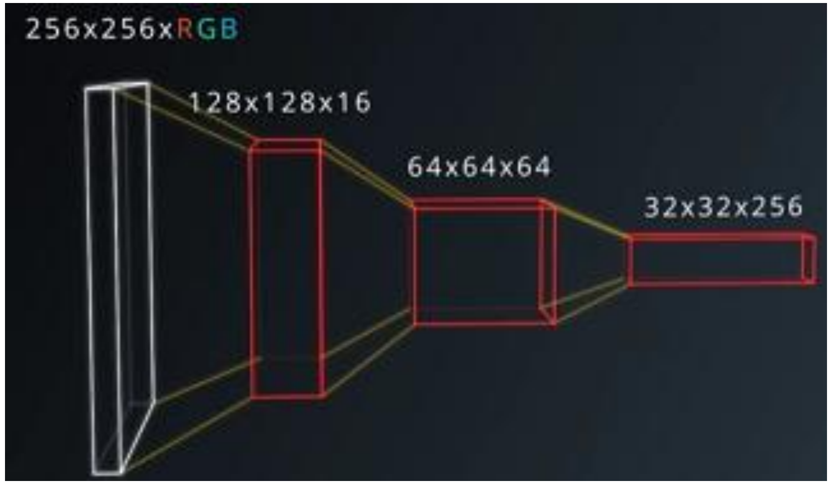
K channels

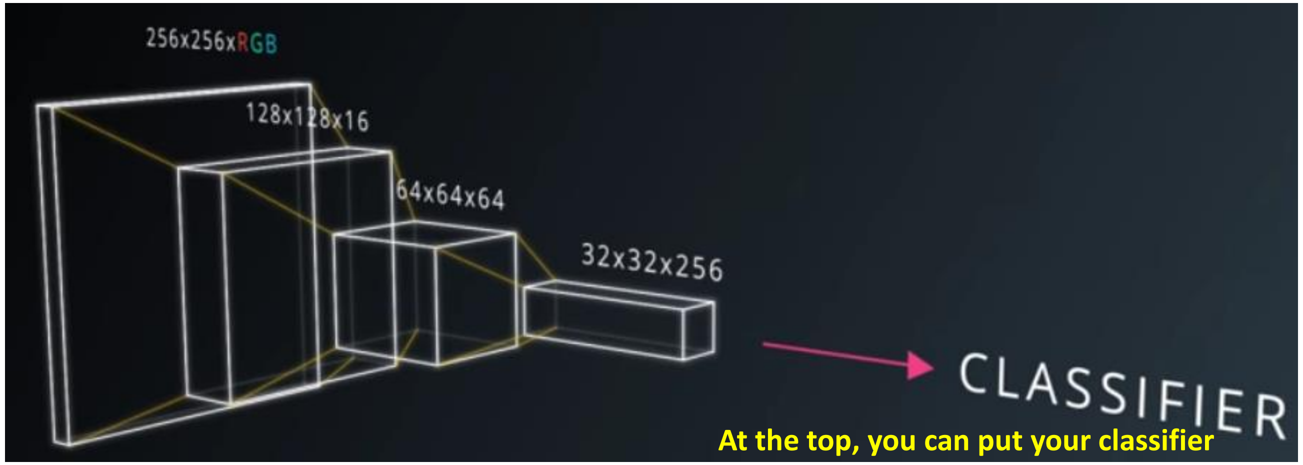
This operation is called the convolution.





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

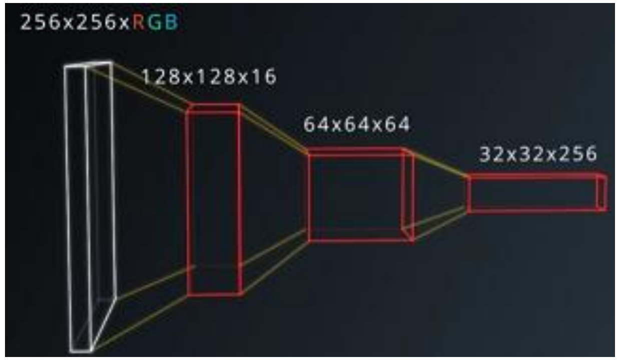


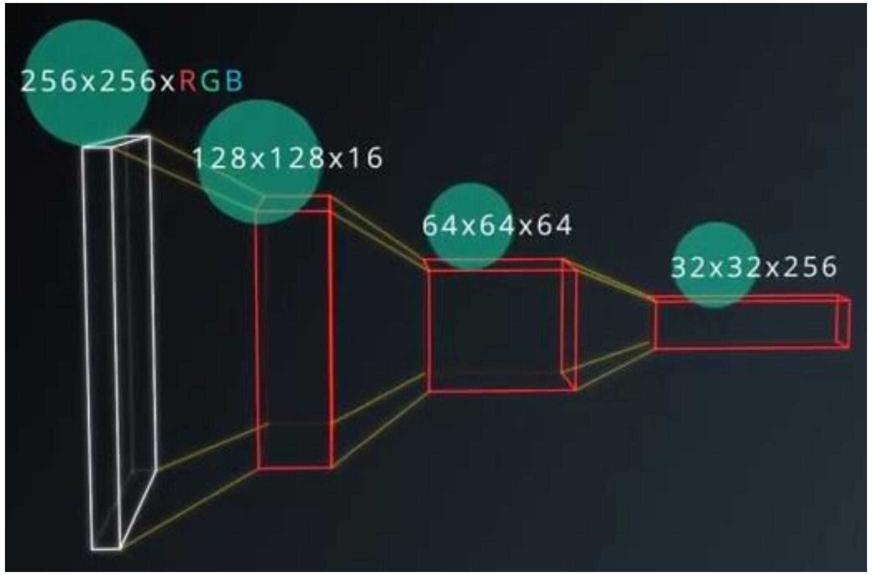


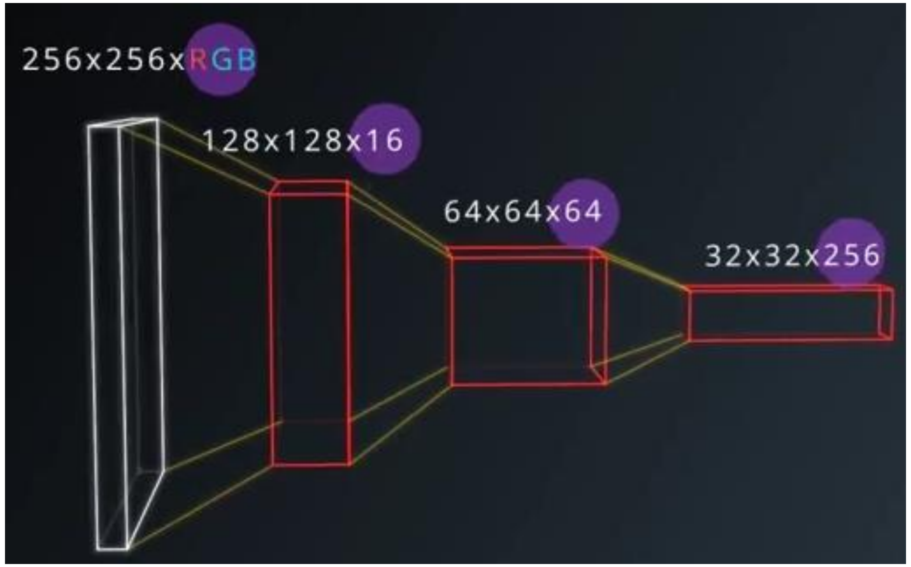
At the top, you can put your classifier

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

Convolutions operation



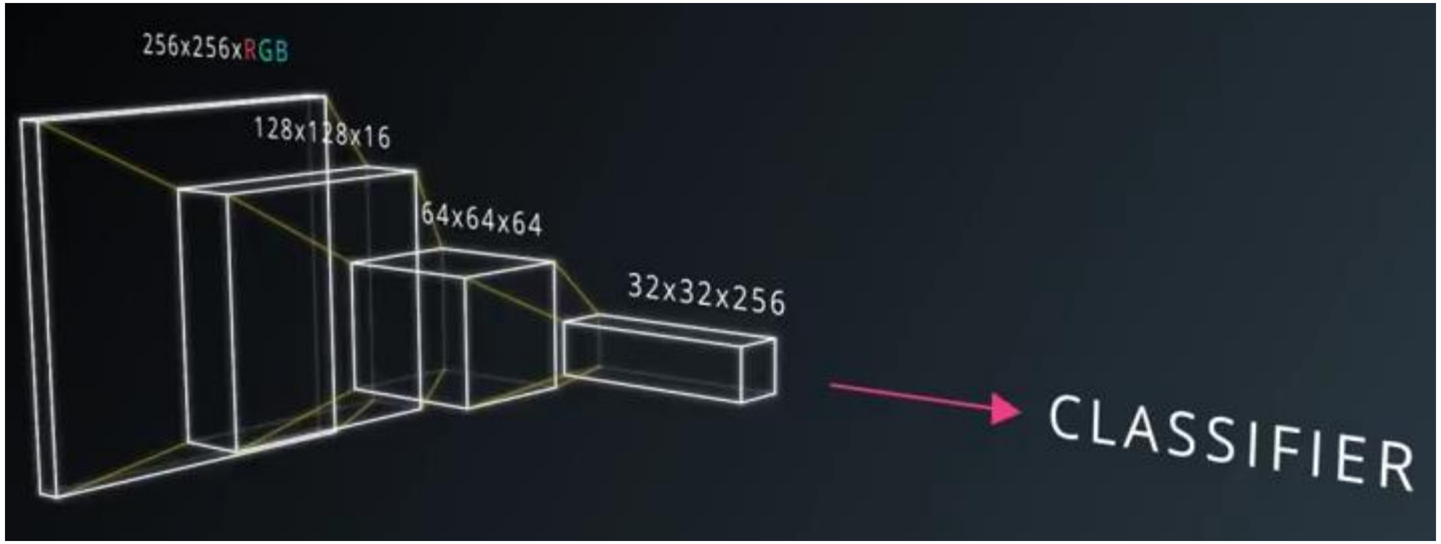




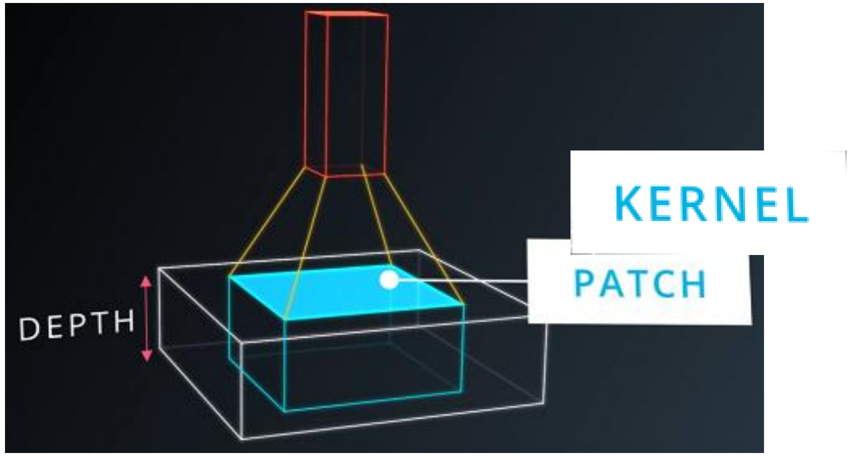
Stack of Convolutions:

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

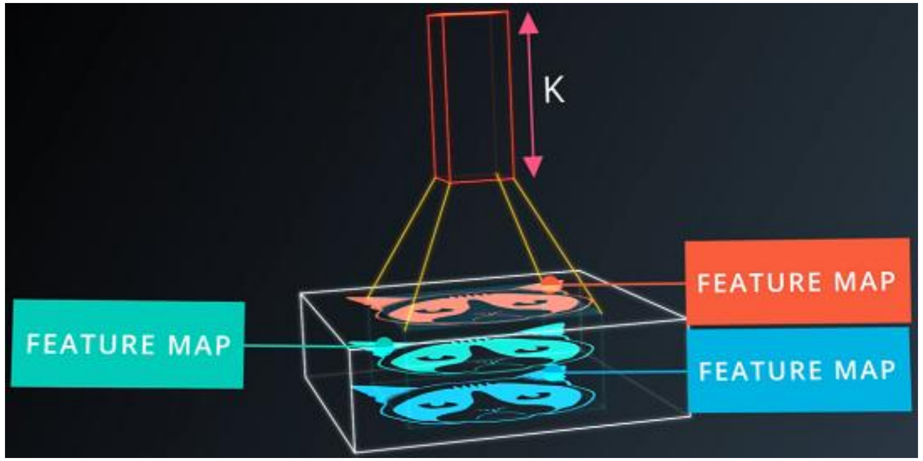
..apply convolutions to progressively squeeze the spacial dimension while increasing the depth which corresponds roughly to the semantic complexity of your representation.

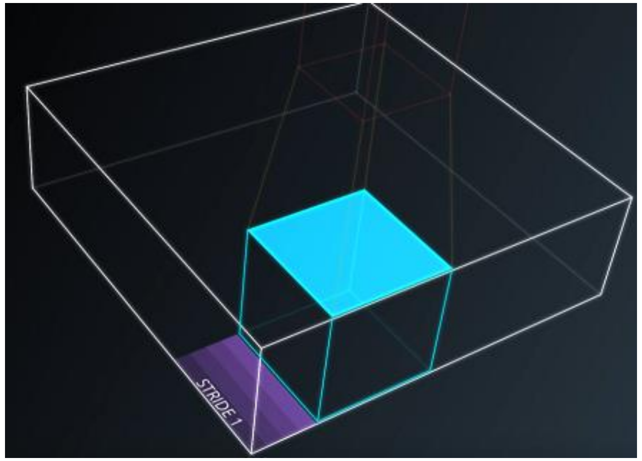


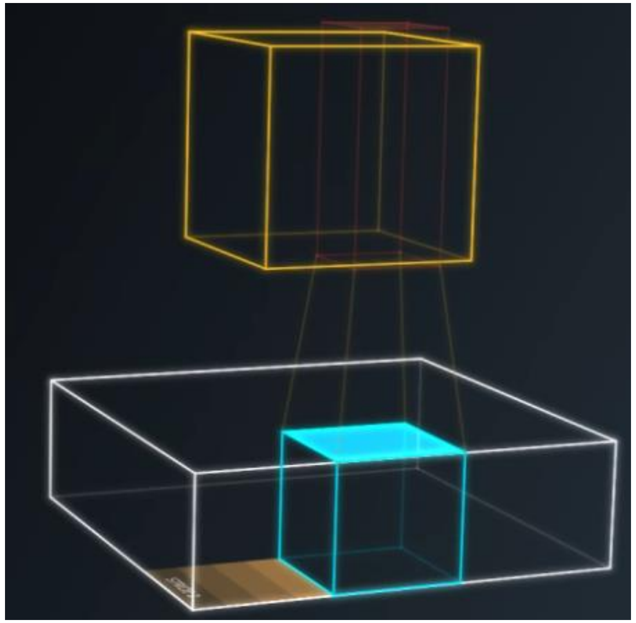
Convolutional Networks

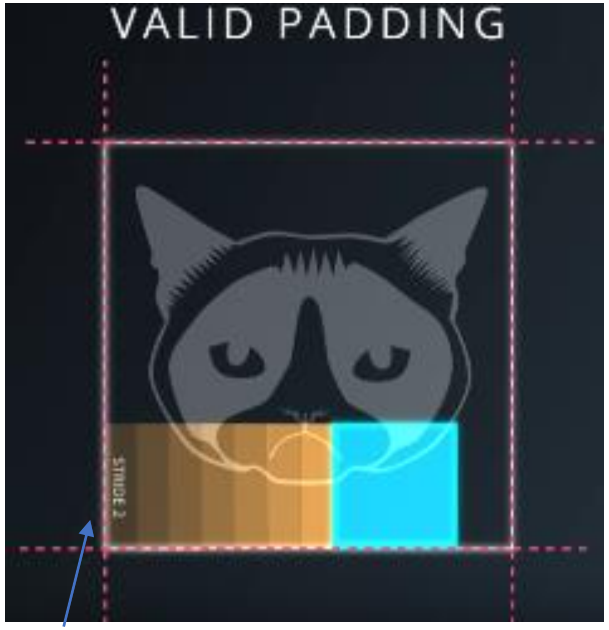


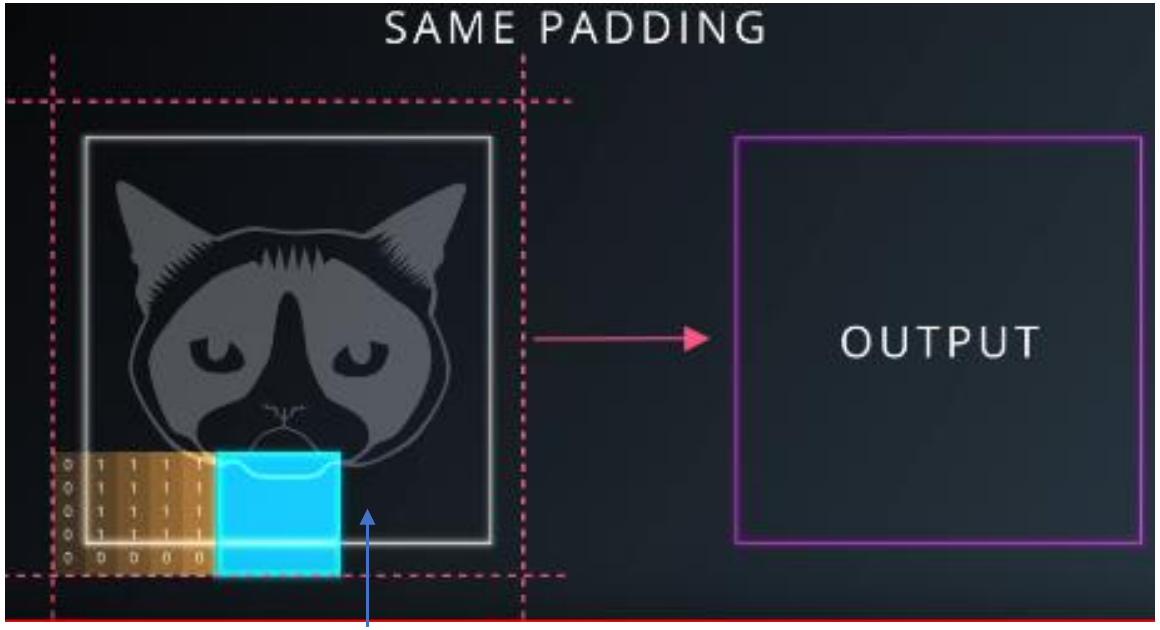
Mapping three feature maps to K feature maps











means no padding

The output map size is exactly the same size as the input map.

Intuition

Intuition

Let's develop better intuition for how Convolutional Neural Networks (CNN) work. We'll examine how humans classify images, and then see how CNNs use similar approaches.

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

An image that we'd like to classify 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 the nose, the eyes, and the fur. We essentially break up the image into smaller pieces, recognize the smaller pieces, and then combine those pieces to get an idea of the overall dog.

In this case, we might break down the image into a combination of the following:

- A nose
- Two eyes
- Golden fur

These pieces can be seen below:

The eye of the dog.

The nose of the dog.

The fur of the 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 two black holes inside it. Thus, one way of classifying a Retriever’s nose is to to break it up into smaller pieces and look for black holes (nostrils) and curves that define an oval as shown below.

A curve that we can use to determine a nose.

A nostril that we can use to classify a nose of the dog.

Intuition

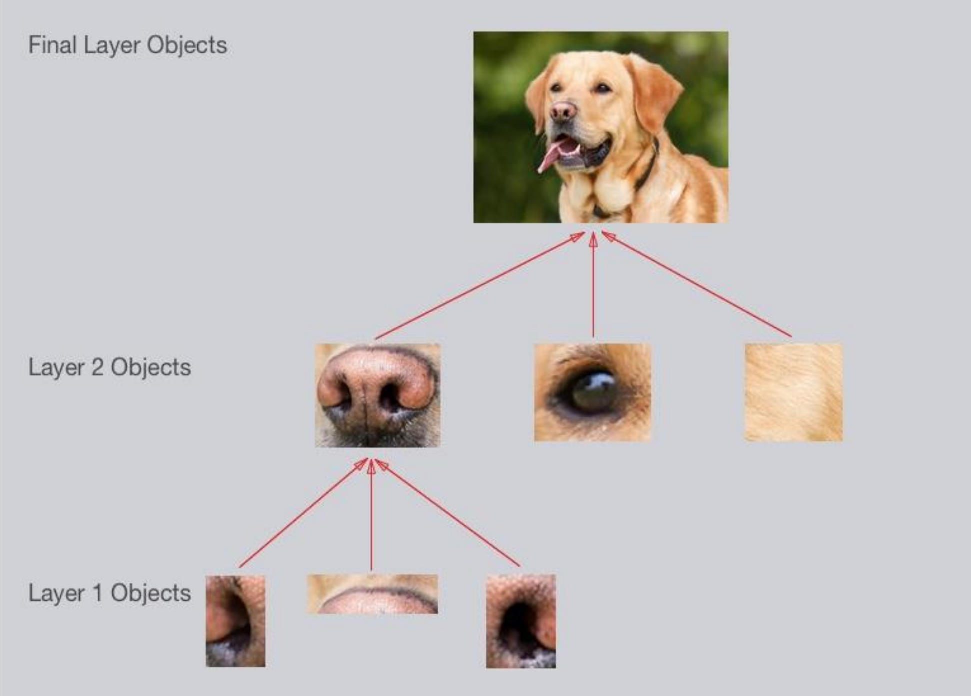
Broadly speaking, this is what a CNN learns to do. It learns to recognize basic lines and curves, then shapes and blobs, and then increasingly complex objects within the image. Finally, the CNN classifies the image by combining the larger, more complex objects.

In our case, the levels in the hierarchy are:

- Simple shapes, like ovals and dark circles
- Complex objects (combinations of simple shapes), like eyes, nose, and fur
- The dog as a whole (a combination of complex objects)

With deep learning, we don't actually program the CNN to recognize these specific features. Rather, the CNN learns on its own to recognize such objects through forward propagation and backpropagation!

It's amazing how well a CNN can learn to classify images, even though we never program the CNN with information about specific features to look for.



An example of what each layer in a CNN might recognize when classifying a picture of a dog

A CNN might have several layers, and each layer might capture a different level in the hierarchy of objects. The first layer is the lowest level in the hierarchy, where the CNN generally classifies small parts of the image into simple shapes like horizontal and vertical lines and simple blobs of colors. The subsequent layers tend to be higher levels in the hierarchy and generally classify more complex ideas like shapes (combinations of lines), and eventually full objects like dogs.

Once again, the CNN ***learns all of this on its own***. We don‘t ever have to tell the CNN to go looking for lines or curves or noses or fur. The CNN just learns from the training set and discovers which characteristics of a Golden Retriever are worth looking for. That’s a good start! Hopefully you‘ve developed some intuition about how CNNs work. Next, let’s look at some implementation details.

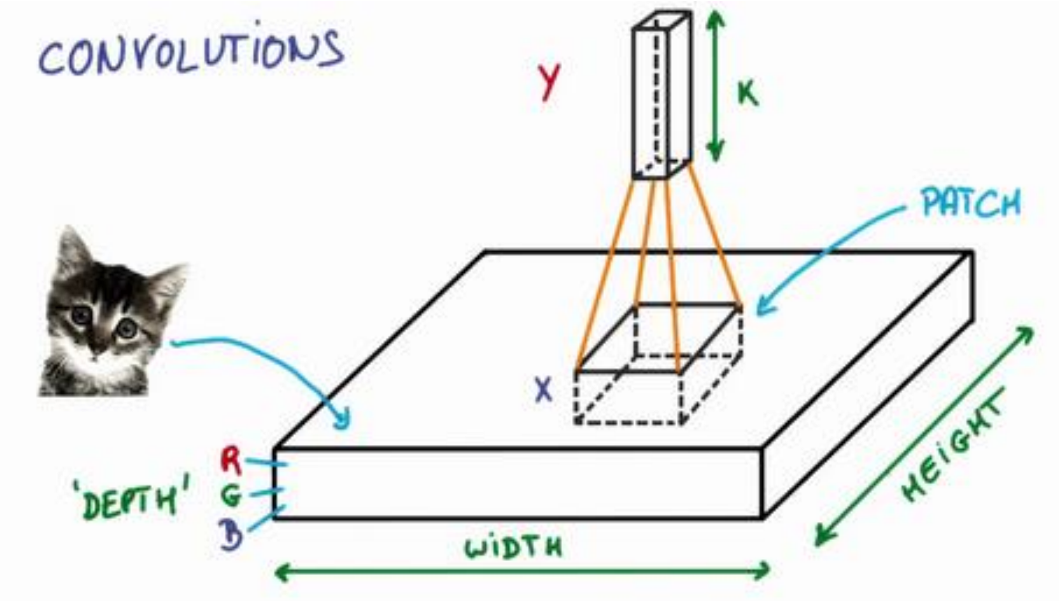
Filters

The filter (or kernel or feature detector)

Breaking up an Image

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.

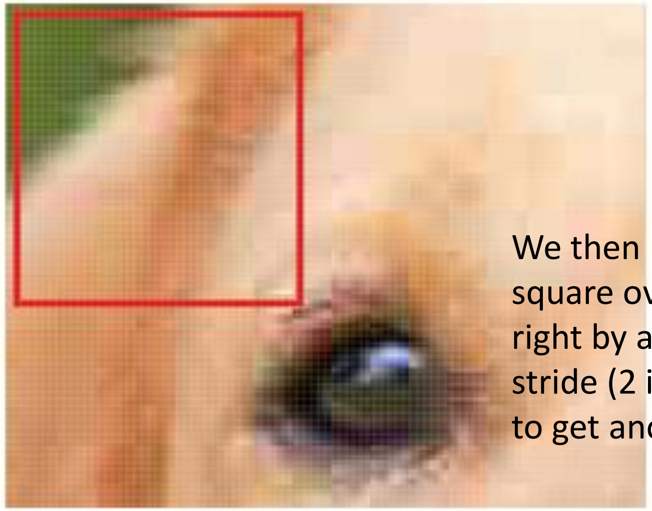


As shown in the previous video, a CNN uses filters to split an image into smaller patches. The size of these patches matches the filter size.

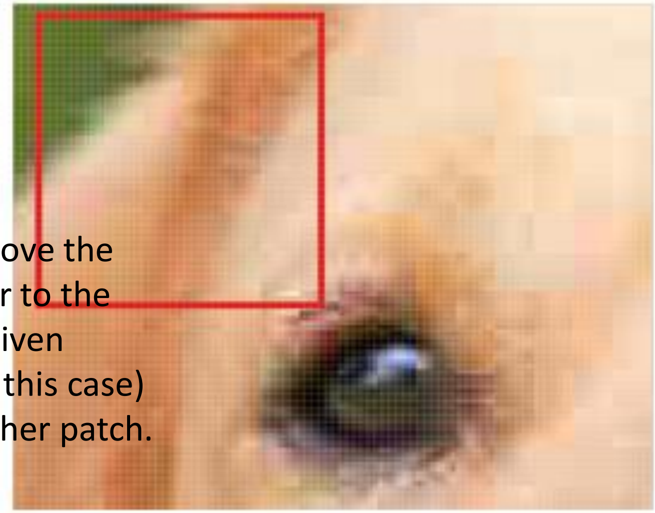
We then simply slide this filter horizontally or vertically to focus on a different piece of the image.

The amount by which the filter slides is referred to as the 'stride'. The stride is a hyperparameter which you, the engineer, can 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 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.



One patch of the Golden Retriever image.



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

We move our square to the right by two pixels to create another patch.

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

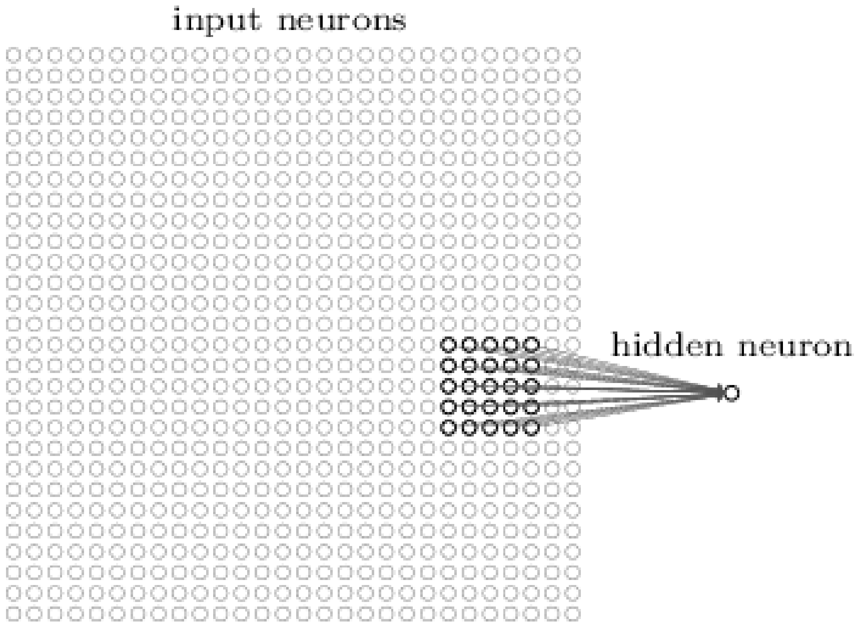
In a normal, non-convolutional neural network, we would have ignored this adjacency. In a normal network, we would have connected every pixel in the input image to a neuron in the next layer. In doing so, we would not have taken advantage of the fact that pixels in an image are close together for a reason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

Filters

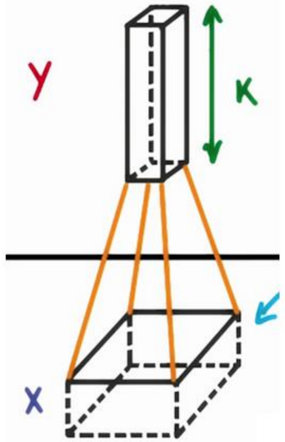
Filter Depth

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*.



In the above example, a patch is connected to a neuron in the next layer. Source: Michael Nielsen.

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 right. In practice, k is a hyperparameter we tune, and most CNNs tend to pick the same starting values.



Choosing a filter depth of k connects each patch to k neurons in the next layer.

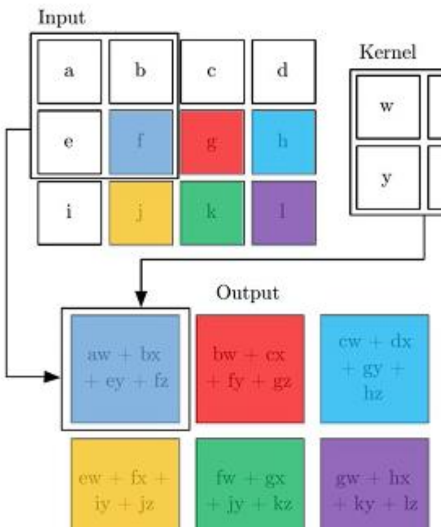
But why connect a single patch to multiple neurons in the next layer? Isn't one neuron good enough?

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

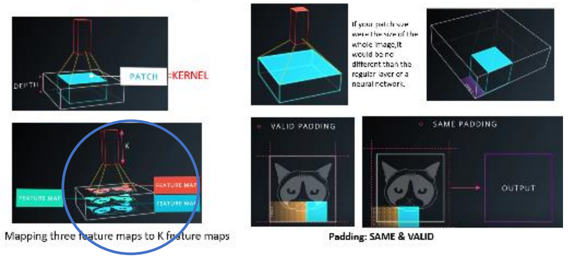
For example, one patch might include some white teeth, some blonde whiskers, and part of a red tongue. In that case, we might want a filter depth of at least three - one for each of teeth, whiskers, and tongue.

This patch of the dog has many interesting features we may want to capture. These include the presence of teeth, the presence of whiskers, and the pink color of the tongue.

Having multiple neurons for a given patch ensures that our CNN can learn to capture whatever characteristics the CNN learns are important. the CNN isn't "programmed" to look for certain characteristics. Rather, it learns **on its own** which characteristics to notice Remember that.

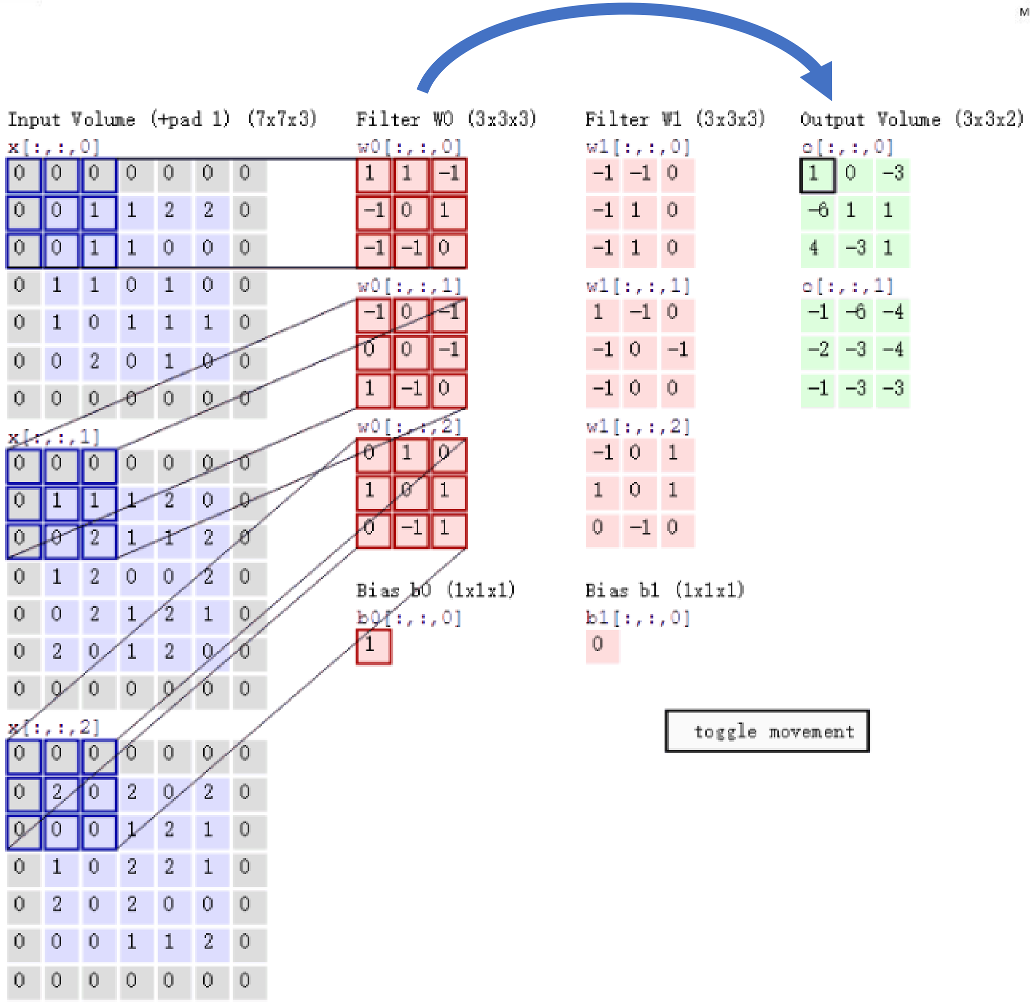


Convolutions operation

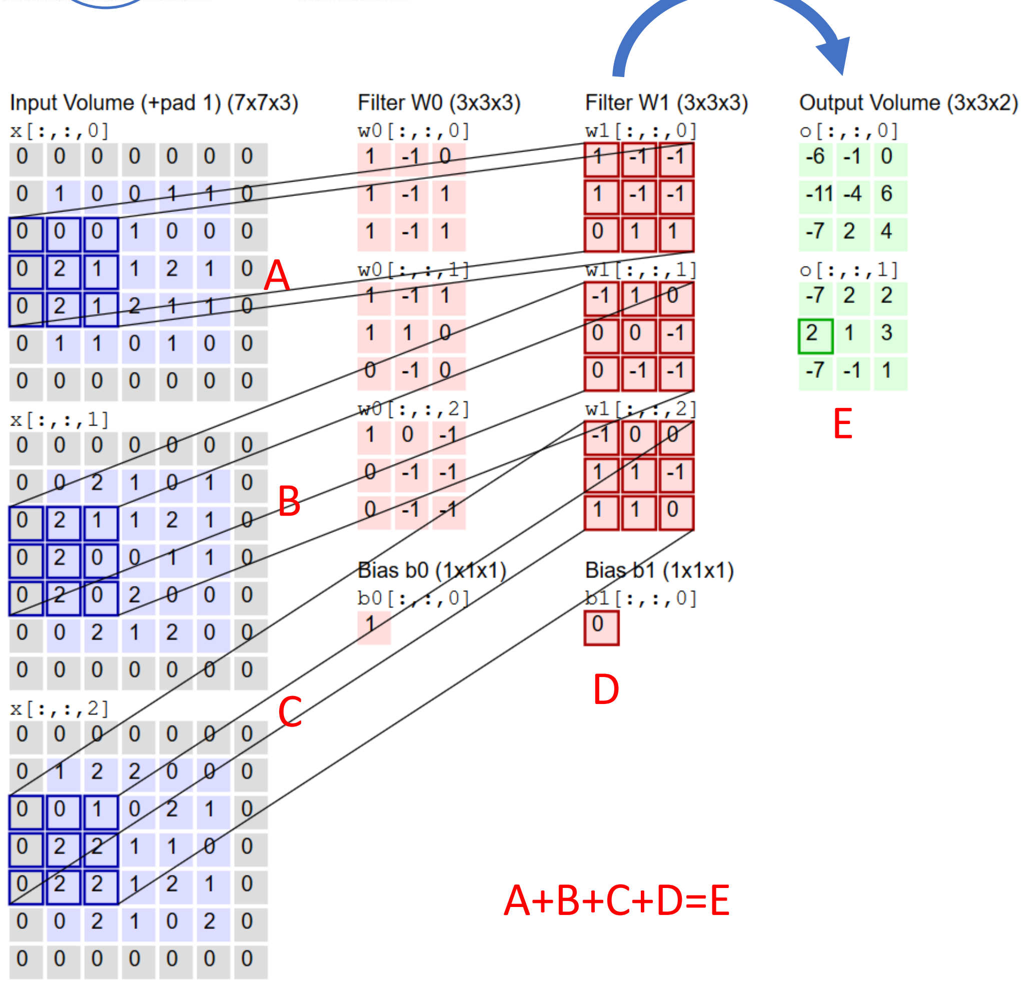

DEPTH
PATCH
KERNEL
If your patch size were the size of the whole image, it would be no different than the regular layer of a neural network.
FEATURE MAP
VALID PADDING
SAME PADDING
OUTPUT

Mapping three feature maps to K feature maps

Padding: SAME & VALID



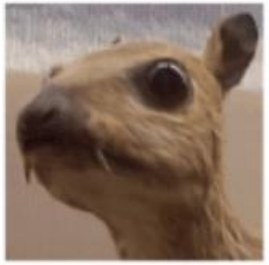
```
Input Volume (+pad 1) (7x7x3)   Filter W0 (3x3x3)   Filter W1 (3x3x3)   Output Volume (3x3x2)
x[:,:,0]                        w0[:,:,0]           w1[:,:,0]           o[:,:,0]
0  0  0  0  0  0  0             1  1  -1            -1 -1  0            1  0  -3
0  0  1  1  2  2  0             -1 0  1             -1  1  0            -6 1  1
0  0  1  1  0  0  0             -1 -1 0             -1  1  0            4  -3 1
0  1  1  0  1  0  0             w0[:,:,1]           w1[:,:,1]           o[:,:,1]
0  1  0  1  1  1  0             -1 0  -1            1  -1  0            -1 -6 -4
0  0  2  0  1  0  0             0  0  -1            -1  0 -1            -2 -3 -4
0  0  0  0  0  0  0             1  -1 0             -1  0  0            -1 -3 -3
x[:,:,1]                        w0[:,:,2]           w1[:,:,2]
0  0  0  0  0  0  0             0  1  0             -1  0  1
0  1  1  1  2  0  0             1  0  1             1   0  1
0  0  2  1  1  2  0             0  -1 1             0  -1  0
0  1  2  0  0  2  0
0  0  2  1  2  1  0             Bias b0 (1x1x1)     Bias b1 (1x1x1)
0  2  0  1  2  0  0             b0[:,:,0]           b1[:,:,0]
0  0  0  0  0  0  0             1                   0
x[:,:,2]
0  0  0  0  0  0  0                                toggle movement
0  2  0  2  0  2  0
0  0  0  1  2  1  0
0  1  0  2  2  1  0
0  2  0  2  0  0  0
0  0  0  1  1  2  0
0  0  0  0  0  0  0
```



```
Input Volume (+pad 1) (7x7x3)   Filter W0 (3x3x3)   Filter W1 (3x3x3)   Output Volume (3x3x2)
x[:,:,0]                        w0[:,:,0]           w1[:,:,0]           o[:,:,0]
0  0  0  0  0  0  0             1  -1  0            1  -1  -1           -6  -1  0
0  1  0  0  1  1  0             1  -1  1            1  -1  -1           -11 -4  6
0  0  0  1  0  0  0             1  -1  1            0   1   1           -7   2  4
0  2  1  1  2  1  0    A        w0[:,:,1]           w1[:,:,1]           o[:,:,1]
0  2  1  2  1  1  0             1  -1  1            -1  1   0           -7   2  2
0  1  1  0  1  0  0             1   1  0            0   0  -1           2    1  3
0  0  0  0  0  0  0             0  -1  0            0  -1  -1           -7  -1  1
x[:,:,1]                        w0[:,:,2]           w1[:,:,2]
0  0  0  0  0  0  0             1   0  -1           -1  0   0            E
0  0  2  1  0  1  0             0  -1  -1           1   1  -1
0  2  1  1  2  1  0    B        0  -1  -1           1   1   0
0  2  0  0  1  1  0
0  2  0  2  0  0  0             Bias b0 (1x1x1)     Bias b1 (1x1x1)
0  0  2  1  2  0  0             b0[:,:,0]           b1[:,:,0]
0  0  0  0  0  0  0             1                   0
x[:,:,2]                                            D
0  0  0  0  0  0  0    C
0  1  2  2  0  0  0
0  0  1  0  2  1  0
0  2  2  1  1  0  0
0  2  2  1  2  1  0                          A+B+C+D=E
0  0  2  1  0  2  0
0  0  0  0  0  0  0
```

Fei-Fei Li Stanford CNN for Visual Recognition

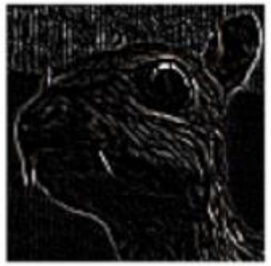
Input image



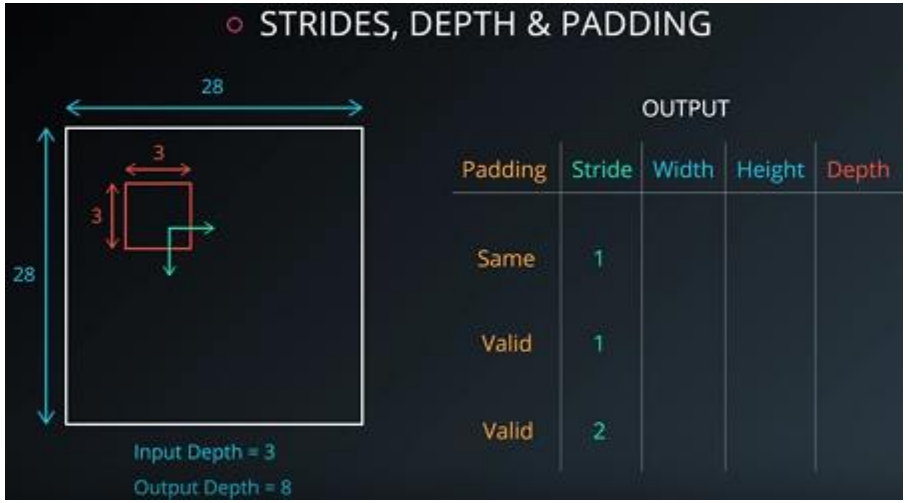
Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map



Feature Map Sizes



○ STRIDES, DEPTH & PADDING

OUTPUT

Padding	Stride	Width	Height	Depth
Same	1			
Valid	1			
Valid	2			

- What are the width, height and depth for padding = 'same', stride = 1?

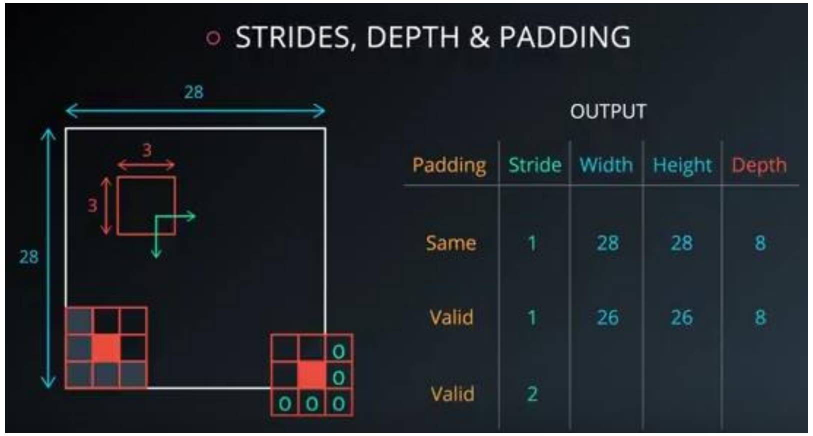
Enter your answers in the format "width, height, depth"

- What are the width, height and depth for padding = 'valid', stride = 1?

Enter your answers in the format "width, height, depth"

- What are the width, height and depth for padding = 'valid', stride = 2?

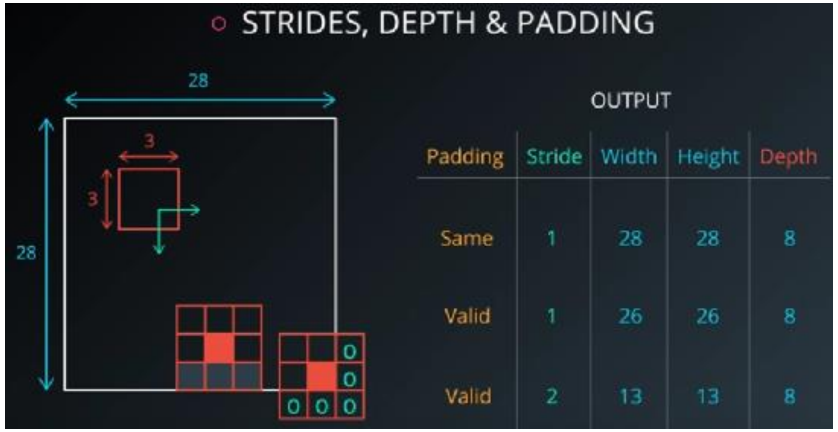
Enter your answers in the format "width, height, depth"



○ STRIDES, DEPTH & PADDING

OUTPUT

Padding	Stride	Width	Height	Depth
Same	1	28	28	8
Valid	1	26	26	8
Valid	2			



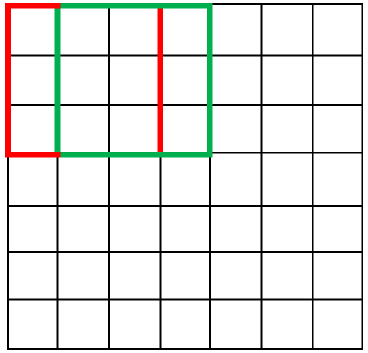
○ STRIDES, DEPTH & PADDING

OUTPUT

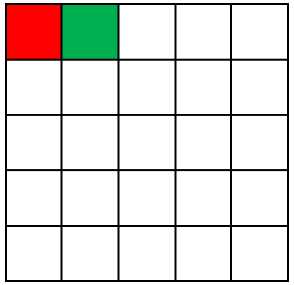
Padding	Stride	Width	Height	Depth
Same	1	28	28	8
Valid	1	26	26	8
Valid	2	13	13	8

Complement

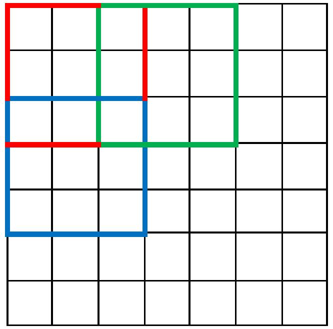
7 x 7 Input Volume



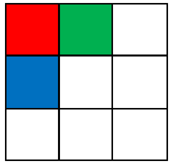
5 x 5 Output Volume

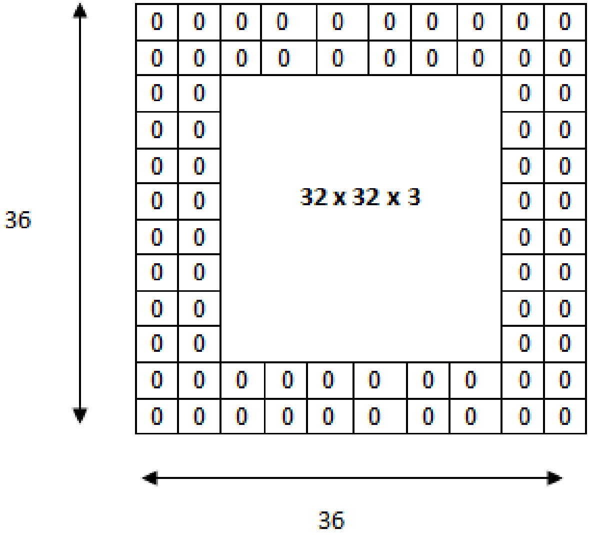


7 x 7 Input Volume



3 x 3 Output Volume





The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

If you have a stride of 1 and if you set the size of zero padding to

$$Zero\ Padding = \frac{(K-1)}{2}$$

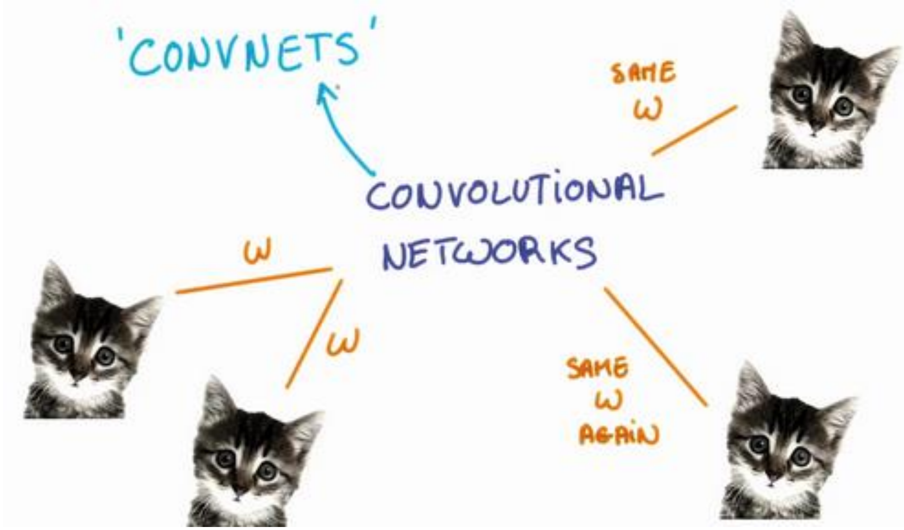
The formula for calculating the output size for any given conv layer is

$$O = \frac{(W-K+2P)}{S} + 1$$

where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

Parameters

Parameter Sharing



The weights, w, are shared across patches for a given layer in a CNN to detect the cat above regardless of where in the image it is located.

When we are trying to classify a picture of a cat, we don't care where in the image a cat is. If it's in the top left or the bottom right, 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?

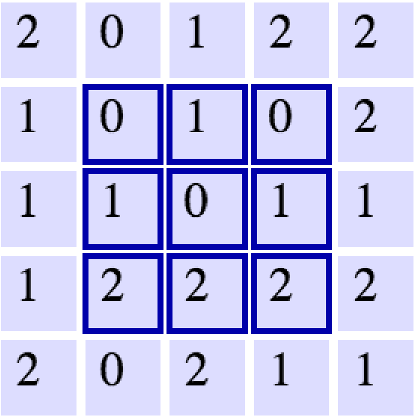
As we saw earlier, the classification of a given patch in an image is determined by the weights and biases corresponding to that patch.

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.

Padding



2	0	1	2	2
1	0	1	0	2
1	1	0	1	1
1	2	2	2	2
2	0	2	1	1

A 5x5 grid with a 3x3 filter. Source: Andrej Karpathy.

Quiz Question

Let's say we have a 5x5 grid (as shown above) and a filter of size 3x3 with a stride of 1. What's the width and height of the next layer?

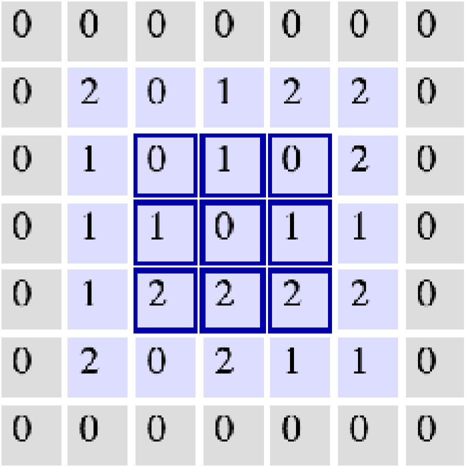
○5x5

○2x2

○3x3

Parameters

As we can see, the width and height of each subsequent layer decreases in the above scheme. In an ideal world, we'd be able to maintain the same width and height across layers so that we can continue to add layers without worrying about the dimensionality shrinking and so that we have consistency. How might we achieve this? One way is to simply add a border of 0s to our original 5x5 image. You can see what this looks like in the below image.



0	0	0	0	0	0	0
0	2	0	1	2	2	0
0	1	0	1	0	2	0
0	1	1	0	1	1	0
0	1	2	2	2	2	0
0	2	0	2	1	1	0
0	0	0	0	0	0	0

The same grid with 0 padding. Source: Andrej Karpathy.

This would expand our original image to a 7x7. With this, we now see how our next layer's size is again a 5x5, keeping our dimensionality consistent.

$$Zero\ Padding = \frac{(K-1)}{2}$$

$$O = \frac{(W - K + 2P)}{S} + 1$$

Dimensionality

From what we've learned so far, 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 (k output depth)

The following formula gives us 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.
And the output depth would be equal to the number of filters D_out = K.
The output volume would be W_out * H_out * D_out.
Knowing the dimensionality of each additional layer helps us understand how large our model is and how our decisions around filter size and stride affect the size of our network.

Quiz: Convolution Output Shape

Introduction

For the next few quizzes we'll test your understanding of the dimensions in CNNs. Understanding dimensions will help you make accurate tradeoffs between model size and performance. As you'll see, some parameters have a much bigger impact on model size than others.

Setup

H = height, W = width, D = depth

- We have an input of shape 32x32x3 (HxWxD)
- 20 filters of shape 8x8x3 (HxWxD)
- A stride of 2 for both the height and width (S)
- With padding of size 1 (P)

$$\boxed{Zero\ Padding = \frac{(K-1)}{2}}$$

$$\boxed{O = \frac{(W-K+2P)}{S} + 1}$$

Recall the formula for calculating the new height or width:

```
new_height = (input_height - filter_height + 2 * P)/S + 1
new_width = (input_width - filter_width + 2 * P)/S + 1
```

Convolutional Layer Output Shape

What's the shape of the output?

The answer format is **HxWxD**, so if you think the new height is 9, new width is 9, and new depth is 5, then type 9x9x5.

Solution: Convolution Output Shape

Solution

The answer is **14x14x20**.

We can get the new height and width with the formula resulting in:

$$Zero\ Padding = \frac{(K-1)}{2}$$

$$O = \frac{(W-K+2P)}{S} + 1$$

(32 - 8 + 2 * 1)/2 + 1 = 14

(32 - 8 + 2 * 1)/2 + 1 = 14

The new depth is equal to the number of filters, which is 20.

This would correspond to the following code:

```
input = tf.placeholder(tf.float32, (None, 32, 32, 3))
filter_weights = tf.Variable(tf.truncated_normal((8, 8, 3, 20)))
                                   # (height, width, input_depth, output_depth)
filter_bias = tf.Variable(tf.zeros(20))
strides = [1, 2, 2, 1]  # (batch, height, width, depth)
padding = 'SAME’
conv = tf.nn.conv2d(input, filter_weights, strides, padding) +
filter_bias
```

Note the output shape of conv will be [1, 16, 16, 20]. It's 4D to account for batch size, but more importantly, it's not [1, 14, 14, 20]. This is because the padding algorithm TensorFlow uses is not exactly the same as the one above. An alternative algorithm is to switch padding from 'SAME' to 'VALID' which would result in an output shape of [1, 13, 13, 20]. If you're curious how padding works in TensorFlow, read this document.

In summary TensorFlow uses the following equation for 'SAME' vs 'VALID'

SAME Padding, the output height and width are computed as:

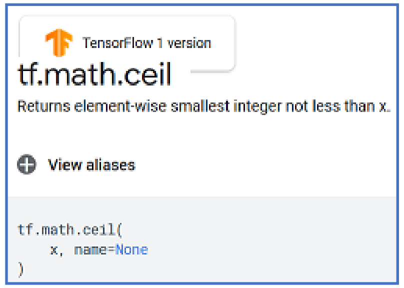
out_height = ceil(float(in_height) / float(strides[1]))

out_width = ceil(float(in_width) / float(strides[2]))

VALID Padding, the output height and width are computed as:

out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))

out_width = ceil(float(in_width - filter_width + 1) / float(strides[2]))



TensorFlow 1 version

tf.math.ceil

Returns element-wise smallest integer not less than x.

⊕ View aliases

```
tf.math.ceil(
    x, name=None
)
```

Quiz: Number of Parameters

We're now going to calculate the number of parameters of the convolutional layer. The answer from the last quiz will come into play here!

Being able to calculate the number of parameters in a neural network is useful since we want to have control over how much memory a neural network uses.

Setup

H = height, W = width, D = depth

- We have an input of shape 32x32x3 (HxWxD)
- 20 filters of shape 8x8x3 (HxWxD)
- A stride of 2 for both the height and width (S)
- Zero padding of size 1 (P)

$$\boldsymbol{Zero\ Padding = \frac{(K-1)}{2}}$$

$$\boldsymbol{O = \frac{(W-K+2P)}{S} + 1}$$

Output Layer

- 14x14x20 (HxWxD)

Hint

Without parameter sharing, each neuron in the output layer must connect to each neuron in the filter. In addition, each neuron in the output layer must also connect to a single bias neuron.

Convolution Layer Parameters 1

How many parameters does the convolutional layer have (without parameter sharing)?

Solution: Number of Parameters

Solution

There are 756560 total parameters. That's a HUGE amount! Here's how we calculate it:

(8 * 8 * 3 + 1) * (14 * 14 * 20) = 756560

8 * 8 * 3 is the number of weights, we add 1 for the bias. Remember, each weight is assigned to every single part of the output (14 * 14 * 20). So we multiply these two numbers together and we get the final answer.

Quiz: Parameter Sharing

Now we'd like you to calculate the number of parameters in the convolutional layer, if every neuron in the output layer shares its parameters with every other neuron in its same channel.
This is the number of parameters actually used in a convolution layer (tf.nn.conv2d()).

Setup

H = height, W = width, D = depth

- We have an input of shape 32x32x3 (HxWxD)
- 20 filters of shape 8x8x3 (HxWxD)
- A stride of 2 for both the height and width (S)
- Zero padding of size 1 (P)

Output Layer

- 14x14x20 (HxWxD)

Hint

With parameter sharing, each neuron in an output channel shares its weights with every other neuron in that channel. So the number of parameters is equal to the number of neurons in the filter, plus a bias neuron, all multiplied by the number of channels in the output layer.

Convolution Layer Parameters 2

How many parameters does the convolution layer have (with parameter sharing)?

Solution: Parameter Sharing

Solution

There are 3860 total parameters. That's 196 times fewer parameters! Here's how the answer is calculated:

(8 * 8 * 3 + 1) * 20 = 3840 + 20 = 3860

That's 3840 weights and 20 biases. This should look similar to the answer from the previous quiz. The difference being it's just 20 instead of (14 * 14 * 20). Remember, with weight sharing we use the same filter for an entire depth slice. Because of this we can get rid of 14 * 14 and be left with only 20.

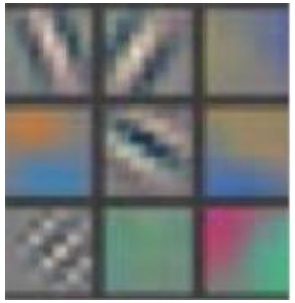
Visualizing CNNs

Visualizing CNNs

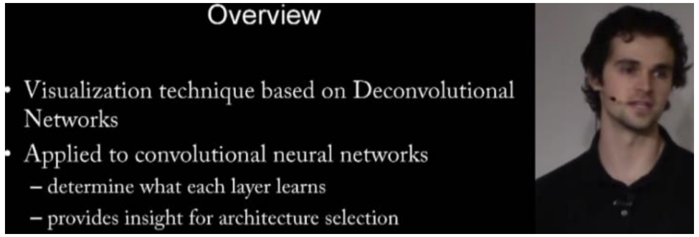
Let’s look at an example CNN to see how it works in action.

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.

Layer 1



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).

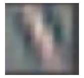


Overview

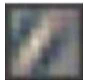
- Visualization technique based on Deconvolutional Networks
- Applied to convolutional neural networks
  - determine what each layer learns
  - provides insight for architecture selection

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.

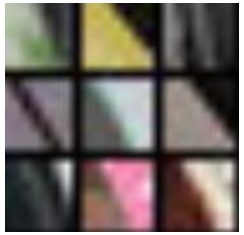


As visualized here, the first layer of the CNN can recognize -45 degree lines.



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.

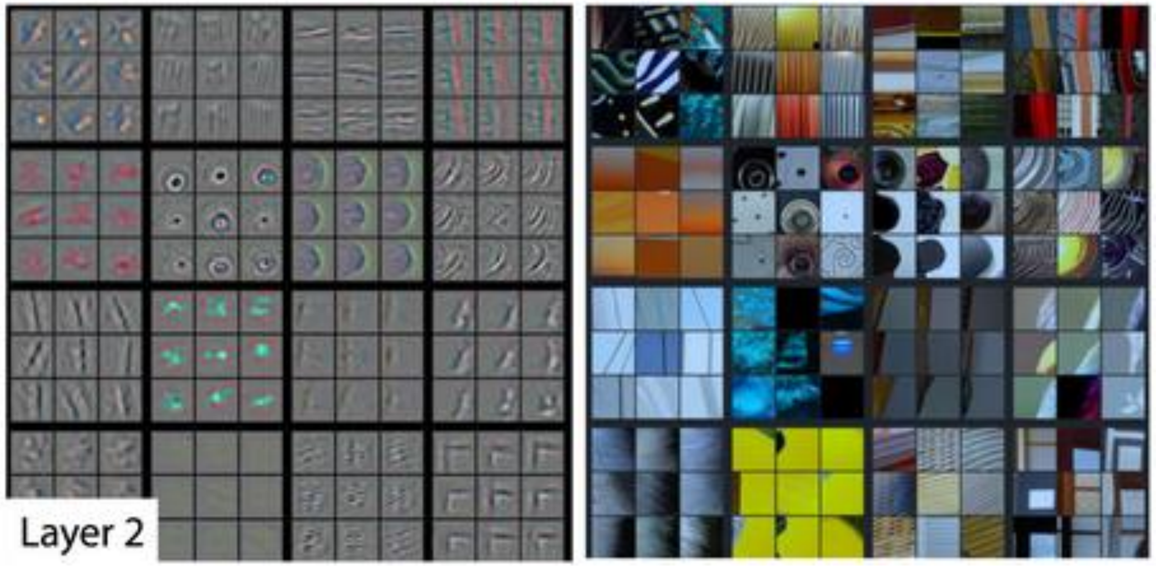


Example patches that activate the -45 degree line detector in the first layer.

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

Visualizing CNNs

Layer 2

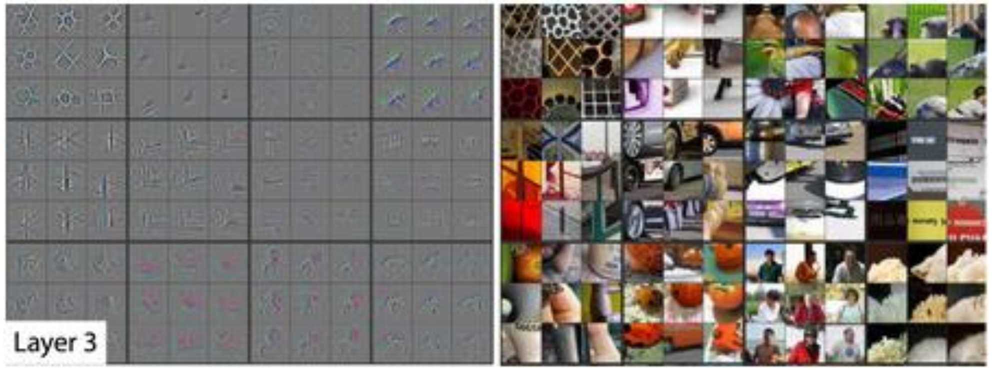


A visualization of the second layer in the CNN. Notice how we are picking up more complex ideas like circles and stripes. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

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.

Visualizing CNNs

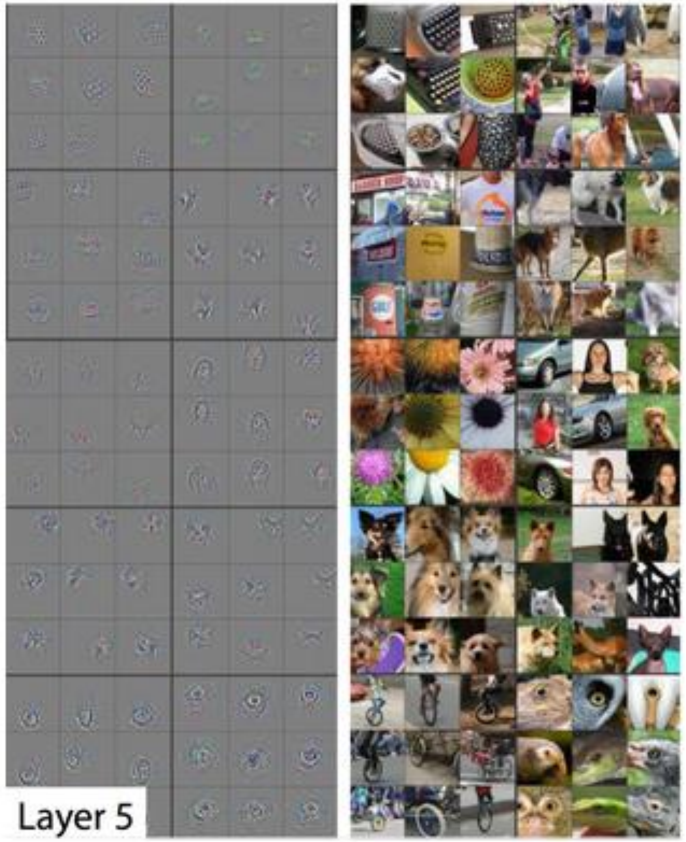
Layer 3



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.

On to TensorFlow

This concludes our high-level discussion of Convolutional Neural Networks. Next you'll practice actually building these networks in TensorFlow.

TensorFlow Convolution Layer

TensorFlow Convolution Layer

Let's examine how to implement a CNN in TensorFlow. TensorFlow provides the tf.nn.conv2d() and tf.nn.bias_add() functions to create your own convolutional layers.

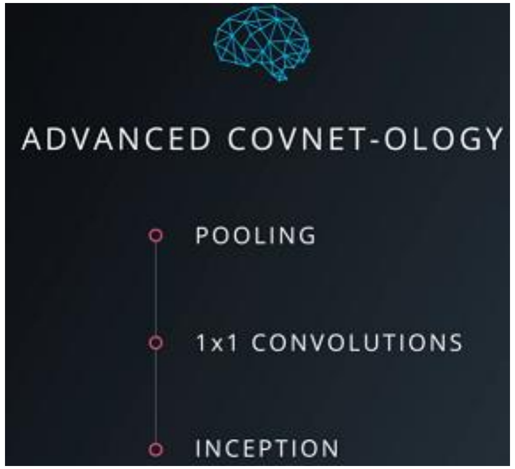
The code right uses the **tf.nn.conv2d()** function to compute the convolution with weight as the filter and [1, 2, 2, 1] for the strides. TensorFlow uses a stride for each input dimension, [batch, input_height, input_width, input_channels]. We are generally always going to set the stride for batch and input_channels (i.e. the first and fourth element in the strides array) to be 1.

You'll focus on changing input_height and input_width while setting batch and input_channels to 1. The input_height and input_width strides are for striding the filter over input. This example code uses a stride of 2 with 5x5 filter over input.

The **tf.nn.bias_add()** function adds a 1-d bias to the last dimension in a matrix.

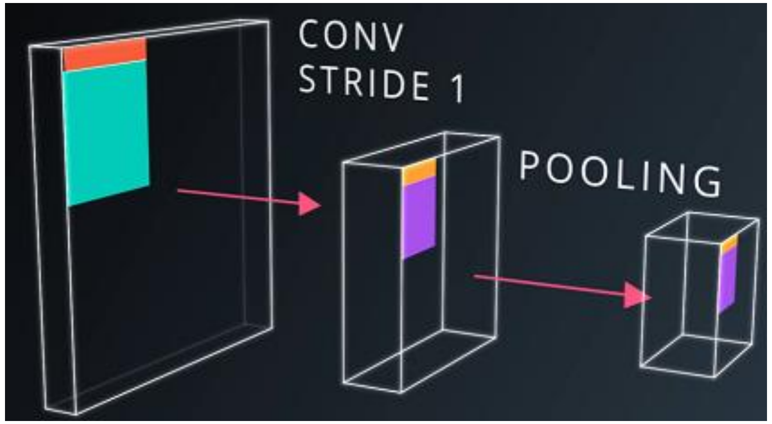
```
# Output depth
k_output = 64

# Image Properties
image_width = 10
image_height = 10
color_channels = 3

# Convolution filter
filter_size_width = 5
filter_size_height = 5

# Input/Image
input = tf.placeholder(
    tf.float32,
    shape=[None, image_height, image_width, color_channels])

# Weight and bias
weight = tf.Variable(tf.truncated_normal(
    [filter_size_height, filter_size_width, color_channels, k_output]))
bias = tf.Variable(tf.zeros(k_output))

# Apply Convolution
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1],
padding='SAME')
# Add bias
conv_layer = tf.nn.bias_add(conv_layer, bias)
# Apply activation function
conv_layer = tf.nn.relu(conv_layer)
```

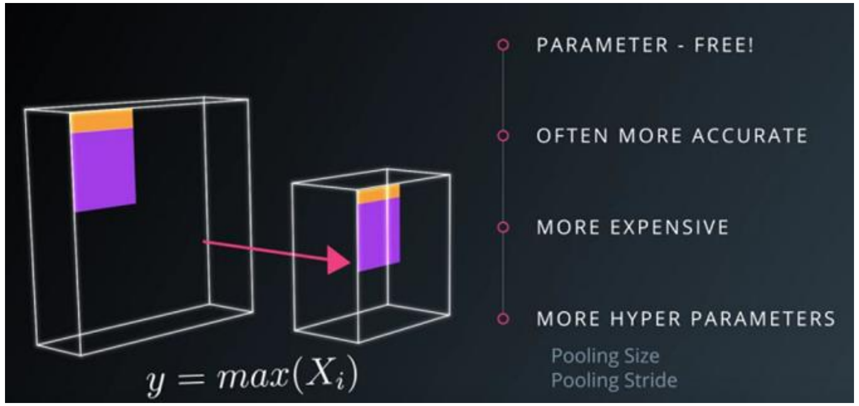
Explore The Design Space



ADVANCED COVNET-OLOGY

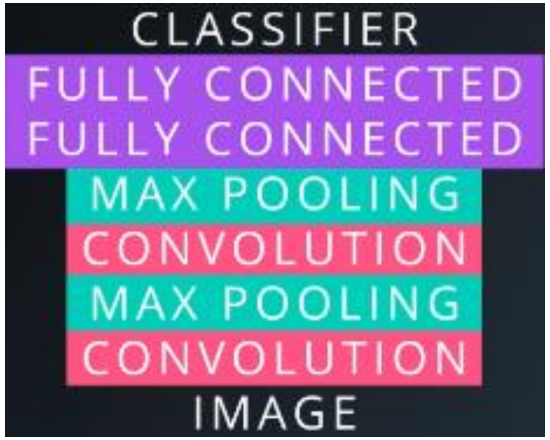
- POOLING
- 1x1 CONVOLUTIONS
- INCEPTION



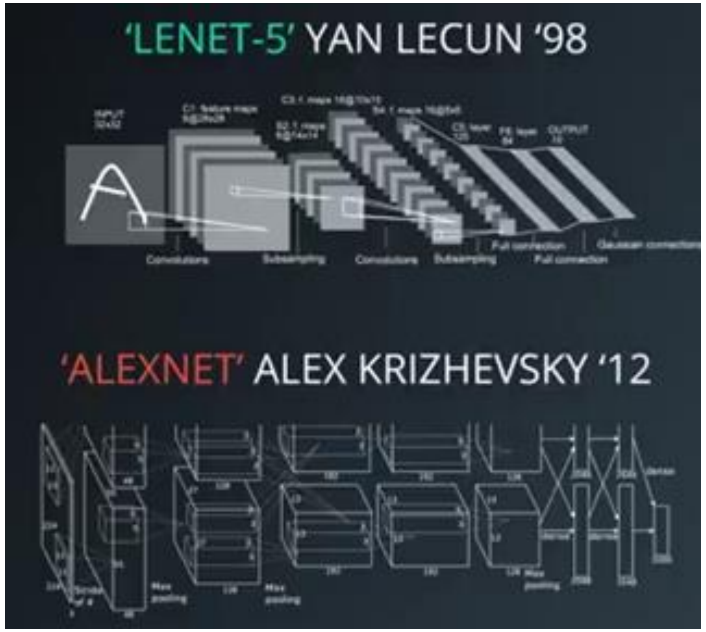


$y = max(X_i)$

- PARAMETER - FREE!
- OFTEN MORE ACCURATE
- MORE EXPENSIVE
- MORE HYPER PARAMETERS
  Pooling Size
  Pooling Stride

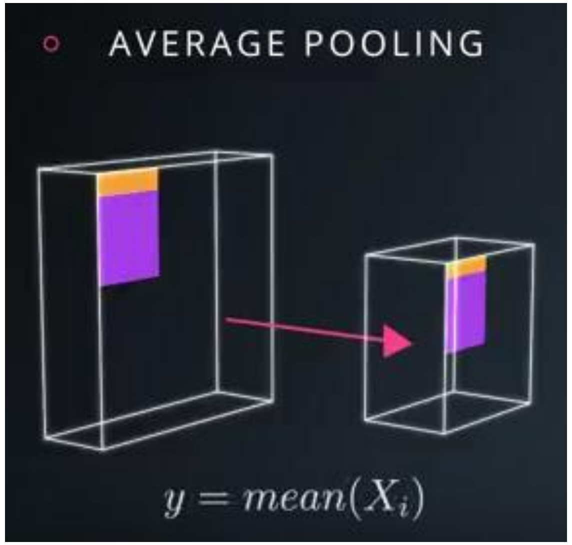


CLASSIFIER
FULLY CONNECTED
FULLY CONNECTED
MAX POOLING
CONVOLUTION
MAX POOLING
CONVOLUTION
IMAGE



'LENET-5' YAN LECUN '98

'ALEXNET' ALEX KRIZHEVSKY '12

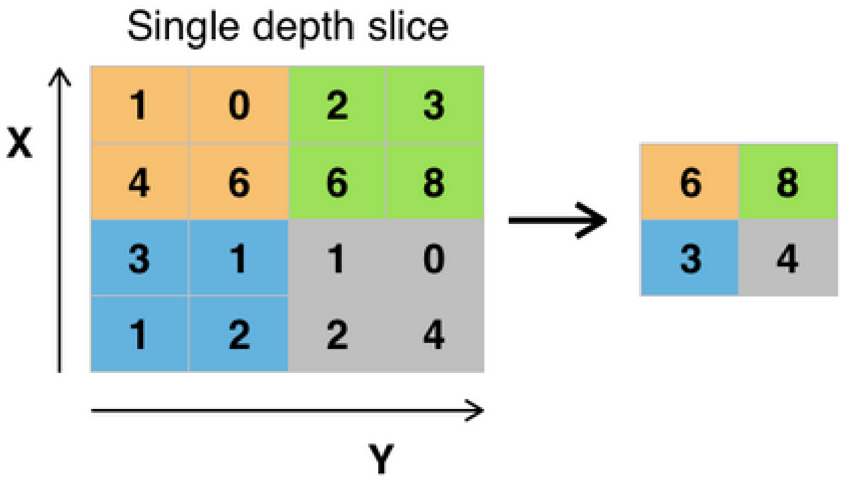


- AVERAGE POOLING

$y = mean(X_i)$

TensorFlow Max Pooling

TensorFlow Max Pooling



1	0	2	3
4	6	6	8
3	1	1	0
1	2	2	4

6	8
3	4

By Aphex34 (Own work) [CC BY-SA 4.0 (http://creativecommons.org/licenses/by-sa/4.0)], via Wikimedia Commons

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.

For example, [[1, 0], [4, 6]] becomes 6, because 6 is the maximum value in this set. Similarly, [[2, 3], [6, 8]] becomes 8.

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.

TensorFlow provides the tf.nn.max_pool() function to apply max pooling to your convolutional layers.

```
...
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1], padding='SAME')
conv_layer = tf.nn.bias_add(conv_layer, bias)
conv_layer = tf.nn.relu(conv_layer)
# Apply Max Pooling
conv_layer = tf.nn.max_pool(
    conv_layer,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')
```

The tf.nn.max_pool() function performs max pooling with the ksize parameter as the size of the filter and the strides parameter as the length of the stride. 2x2 filters with a stride of 2x2 are common in practice.

The ksize and strides parameters are structured as 4-element lists, with each element corresponding to a dimension of the input tensor ([batch, height, width, channels]). For both ksize and strides, the batch and channel dimensions are typically set to 1.

Quiz: Pooling Intuition

The next few quizzes will test your understanding of **pooling layers**.

Quiz Question
A pooling layer is generally used to ...
□Increase the size of the output
□ Decrease the size of the output
□ Prevent overfitting
□ Gain information

Solution: Pooling Intuition

Solution

The correct answer is **decrease the size of the output** and **prevent overfitting**. Preventing overfitting is a consequence of reducing the output size, which in turn, reduces the number of parameters in future layers.

Recently, pooling layers have fallen out of favor. Some reasons are:

- Recent datasets are so big and complex we're more concerned about underfitting.
- Dropout is a much better regularizer.
- Pooling results in a loss of information. Think about the max pooling operation as an example. We only keep the largest of *n* numbers, thereby disregarding *n-1* numbers completely.

Quiz: Pooling Mechanics

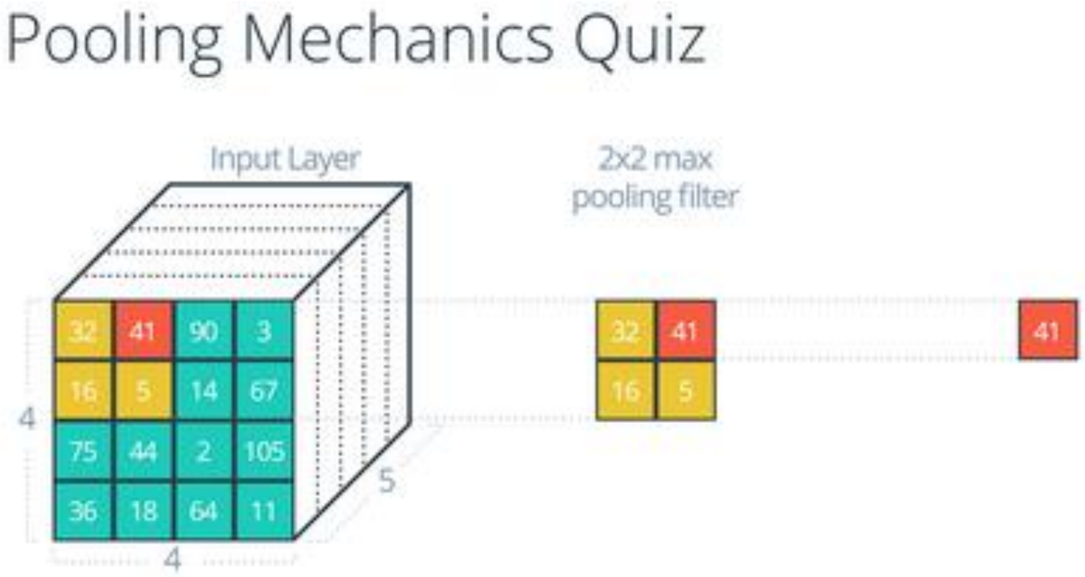
Setup

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
```



Pooling Layer Output Shape
What's the shape of the output? Format is **HxWxD**.

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 image right gives an example of how a max pooling layer works. In this case, the max pooling filter has a shape of 2x2. As the max pooling filter slides across the input layer, the filter will output the maximum value of the 2x2 square.

Solution: Pooling Mechanics

Solution

The answer is **2x2x5**. Here's how it's calculated using the formula:

```
(4 - 2)/2 + 1 = 2
(4 - 2)/2 + 1 = 2
```

The depth stays the same.
Here's the corresponding code:

```
input = tf.placeholder(tf.float32, (None, 4, 4, 5))
filter_shape = [1, 2, 2, 1]
strides = [1, 2, 2, 1]
padding = 'VALID'
pool = tf.nn.max_pool(input, filter_shape, strides, padding)
```

The output shape of pool will be [1, 2, 2, 5], even if padding is changed to 'SAME'.

Quiz: Pooling Practice

Great, now let's practice doing some pooling operations manually.

Max Pooling

What's the result of a **max pooling** operation on the input:

```
[[[0, 1, 0.5, 10],
   [2, 2.5, 1, -8],
   [4, 0, 5, 6],
   [15, 1, 2, 3]]]
```

Assume the filter is 2x2 and the stride is 2 for both height and width. The output shape is 2x2x1.

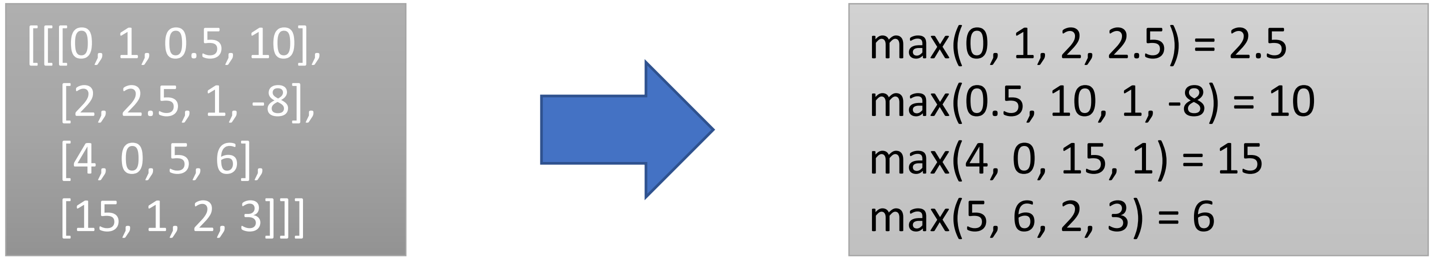
The answering format will be 4 numbers, each separated by a comma, such as: 1,2,3,4.

Work from the top left to the bottom right

Solution: Pooling Practice

Solution

The correct answer is 2.5,10,15,6. We start with the four numbers in the top left corner. Then we work left-to-right and top-to-bottom, moving 2 units each time.



```
[[[0, 1, 0.5, 10],
  [2, 2.5, 1, -8],
  [4, 0, 5, 6],
  [15, 1, 2, 3]]]
```

max(0, 1, 2, 2.5) = 2.5
max(0.5, 10, 1, -8) = 10
max(4, 0, 15, 1) = 15
max(5, 6, 2, 3) = 6

Quiz: Average Pooling

Mean Pooling

What's the result of a **average (or mean) pooling**?

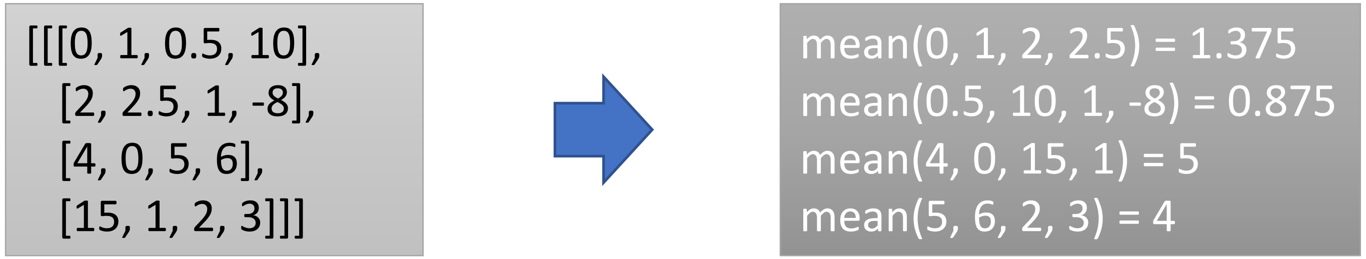
```
[[[0, 1, 0.5, 10],
   [2, 2.5, 1, -8],
   [4, 0, 5, 6],
   [15, 1, 2, 3]]]
```

Assume the filter is 2x2 and the stride is 2 for both height and width. The output shape is 2x2x1.
The answering format will be 4 numbers, each separated by a comma, such as: 1,2,3,4.
Answer to 3 decimal places. Work from the top left to the bottom right

Solution: Average Pooling

Solution

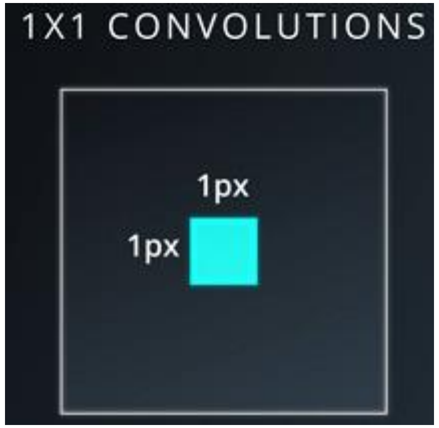
The correct answer is 1.375,0.875,5,4. We start with the four numbers in the top left corner. Then we work left-to-right and top-to-bottom, moving 2 units each time.

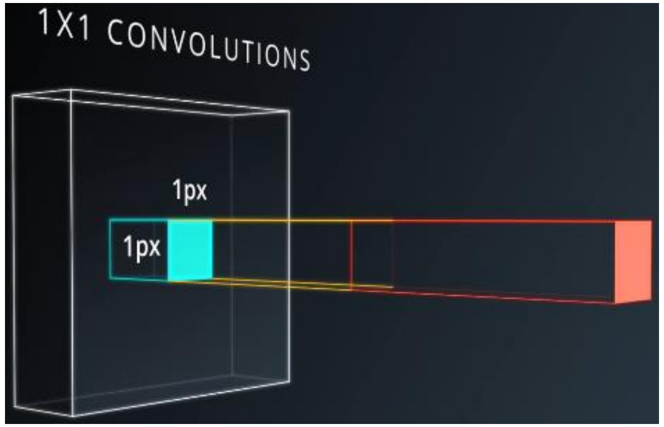


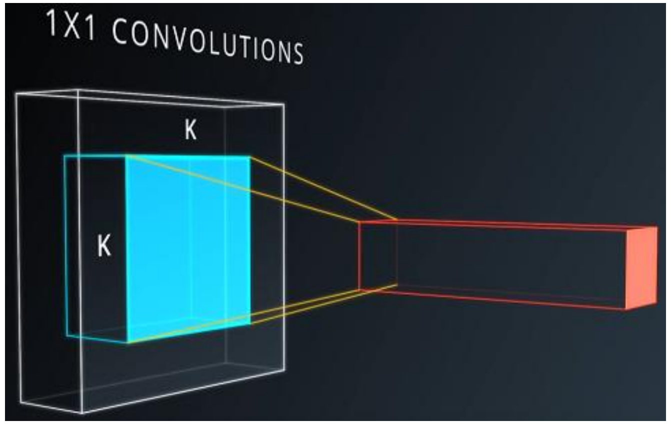
```
[[[0, 1, 0.5, 10],
  [2, 2.5, 1, -8],
  [4, 0, 5, 6],
  [15, 1, 2, 3]]]
```

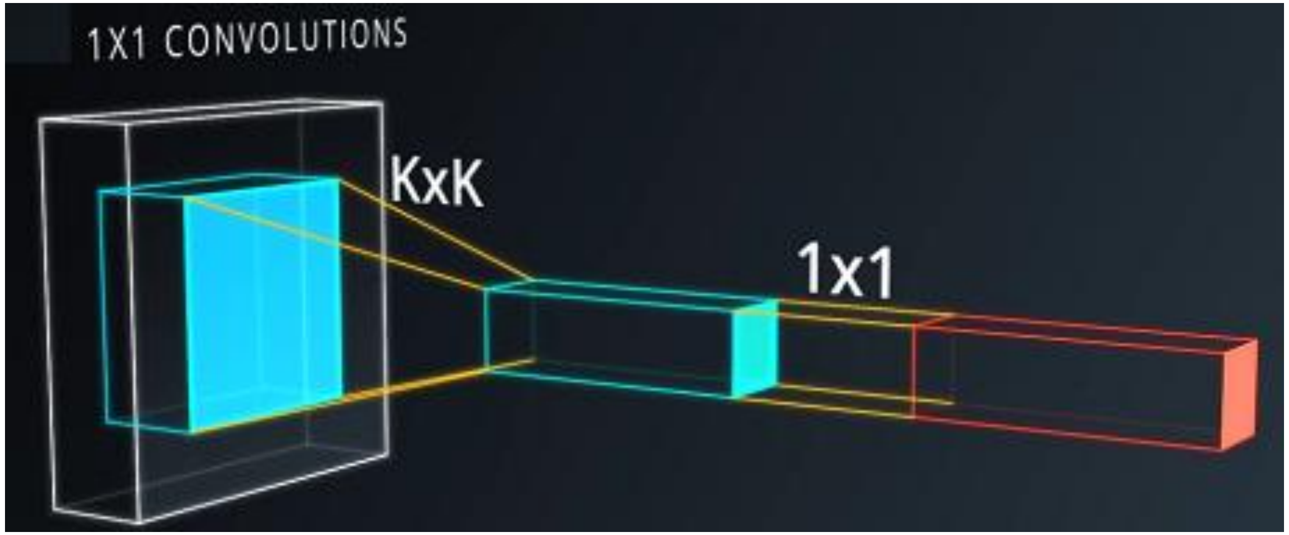
mean(0, 1, 2, 2.5) = 1.375
mean(0.5, 10, 1, -8) = 0.875
mean(4, 0, 15, 1) = 5
mean(5, 6, 2, 3) = 4

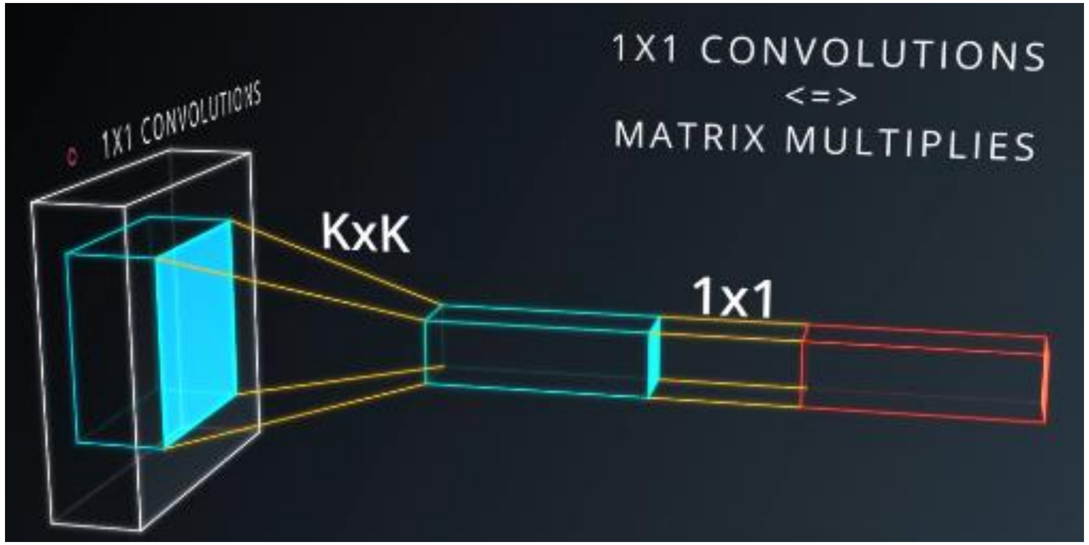
1x1 Convolutions



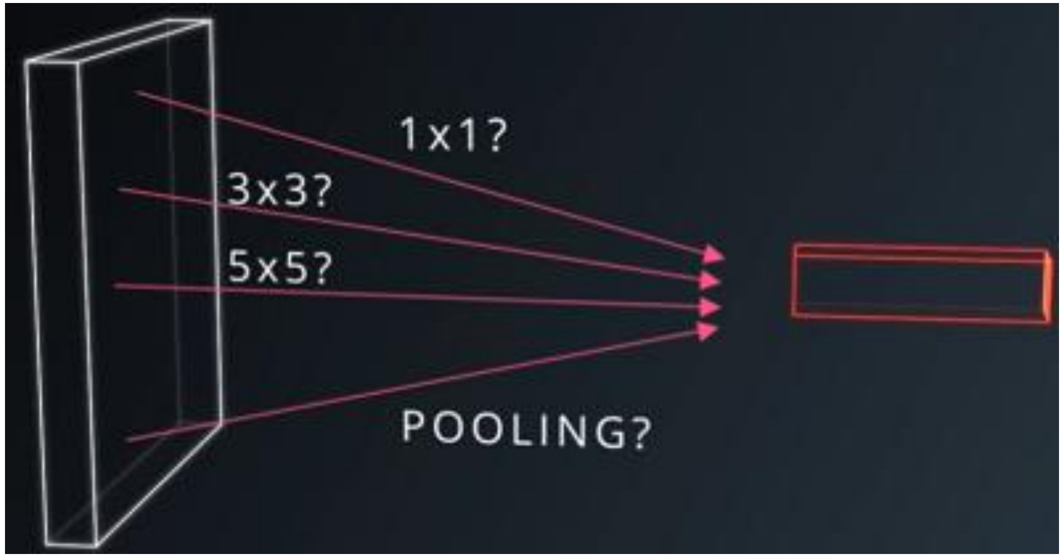


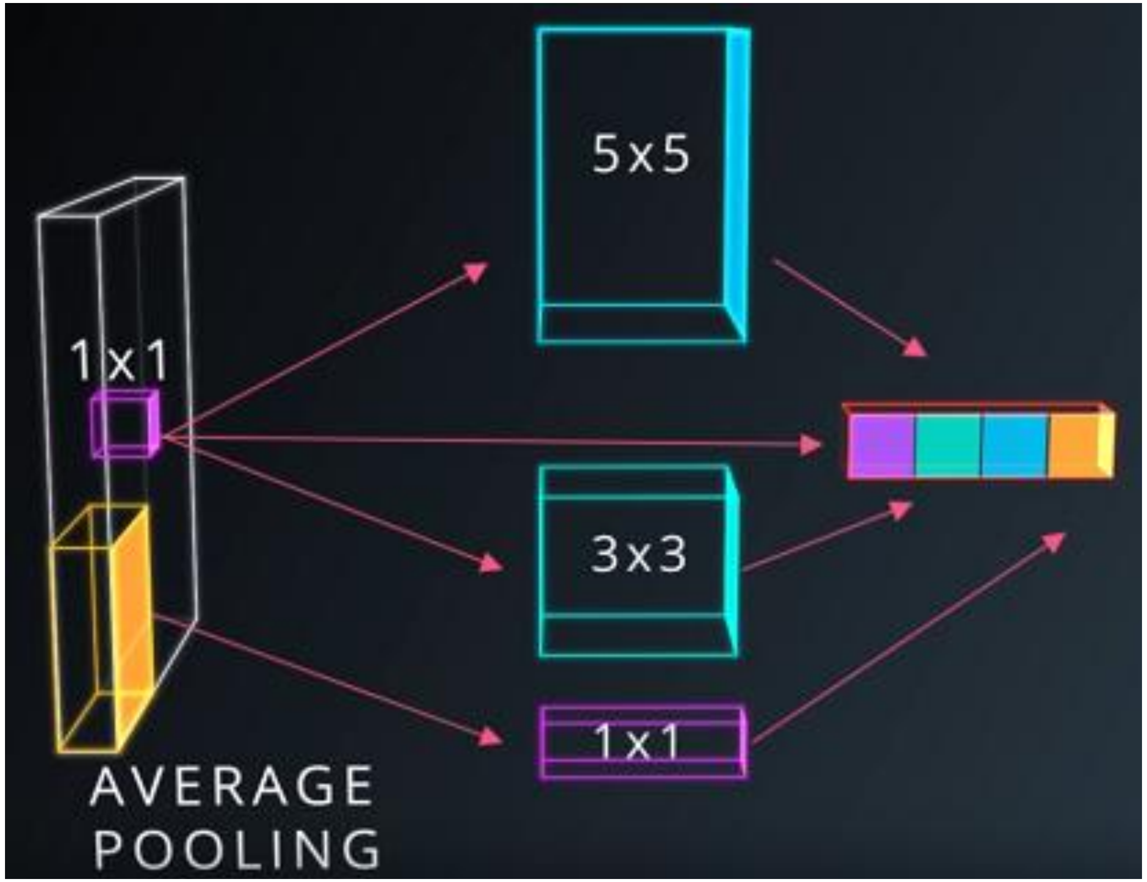




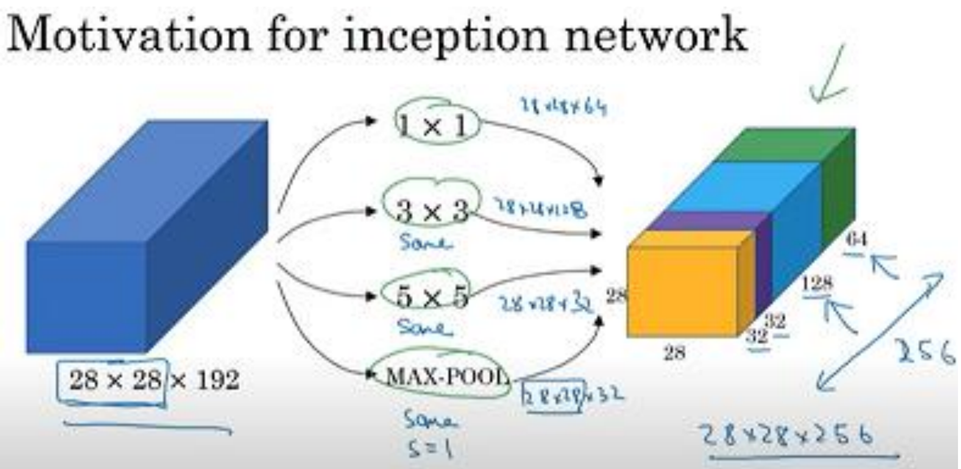


Inception Module

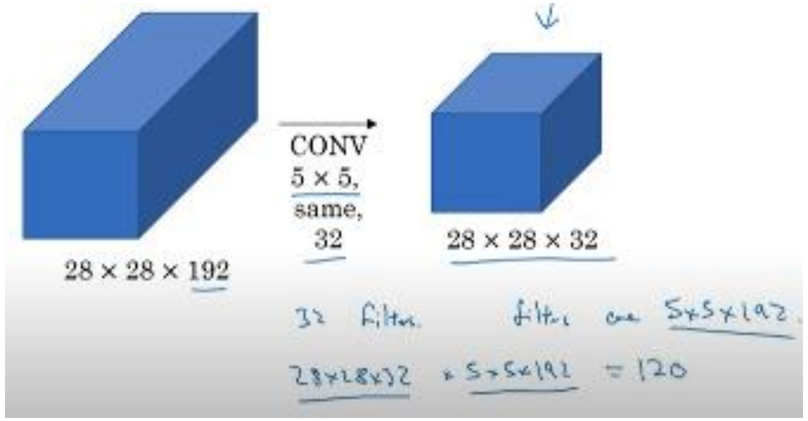


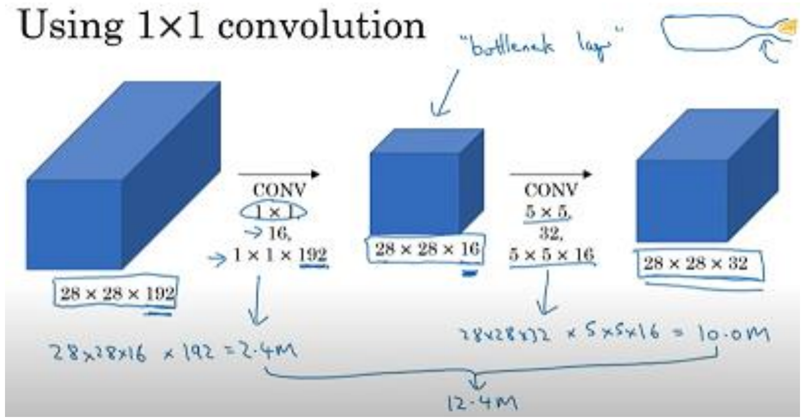


Complement
The problem of computational cost



Motivation for inception network





Using 1×1 convolution

Andrew Ng [Networks in Networks and 1x1 Convolutions](#)

[Inception Network Motivation](#)

StackExchange **[What does 1x1 convolution mean in a neural network?](#)**

Convolutional Network in TensorFlow

Convolutional Network in TensorFlow

It's time to walk through an example Convolutional Neural Network (CNN) in TensorFlow.

The structure of this network follows the classic structure of CNNs, which is a mix of convolutional layers and max pooling, followed by fully-connected layers.

The code you'll be looking at is similar to what you saw in the segment on **Deep Neural Network in TensorFlow** in the previous lesson, except we restructured the architecture of this network as a CNN.

Just like in that segment, here you'll study the line-by-line breakdown of the code. If you want, you can even download the code and run it yourself.

Thanks to Aymeric Damien for providing the original TensorFlow model on which this segment is based.

Time to dive in!

Dataset

You've seen this section of code from previous lessons. Here we're importing the MNIST dataset and using a convenient TensorFlow function to batch, scale, and One-Hot encode the data.

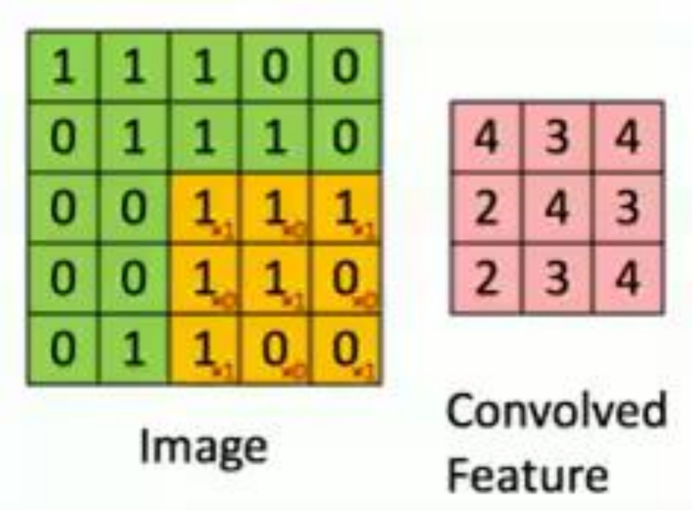
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(".", one_hot=True, reshape=False)
import tensorflow as tf
# Parameters
learning_rate = 0.00001
epochs = 10
batch_size = 128
# Number of samples to calculate validation and accuracy
# Decrease this if you're running out of memory to calculate accuracy
test_valid_size = 256
# Network Parameters
n_classes = 10  # MNIST total classes (0-9 digits)
dropout = 0.75  # Dropout, probability to keep units
```

Weights and Biases

```
# Store layers weight & bias
weights = {
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
    'out': tf.Variable(tf.random_normal([1024, n_classes]))}
biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))}
```

Convolutional Network in TensorFlow

Convolutions



1	1	1	0	0
0	1	1	1	0
0	0	$1_{\times1}$	$1_{\times0}$	$1_{\times1}$
0	0	$1_{\times0}$	$1_{\times1}$	$0_{\times0}$
0	1	$1_{\times1}$	$0_{\times0}$	$0_{\times1}$

4	3	4
2	4	3
2	3	4

Convolution with 3x3 Filter. Source: http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

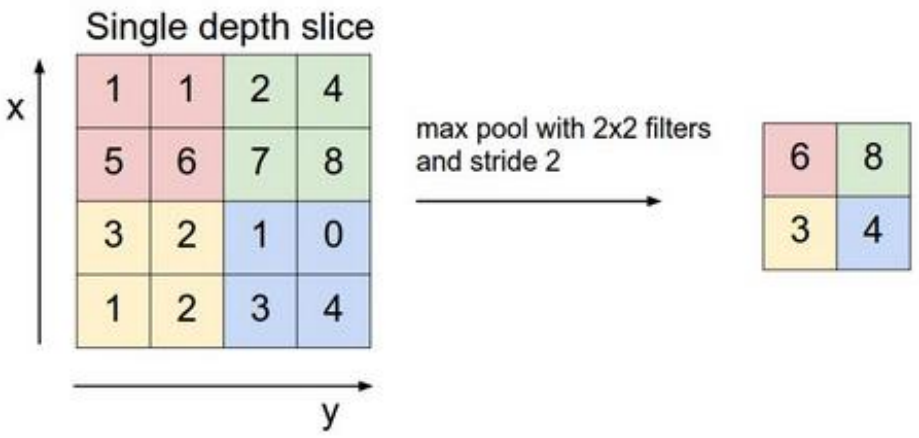
The above is an example of a convolution with a 3x3 filter and a stride of 1 being applied to data with a range of 0 to 1. The convolution for each 3x3 section is calculated against the weight, [[1, 0, 1], [0, 1, 0], [1, 0, 1]], then a bias is added to create the convolved feature on the right. In this case, the bias is zero. In TensorFlow, this is all done using tf.nn.conv2d() and tf.nn.bias_add().

```
def conv2d(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
```

The tf.nn.conv2d() function computes the convolution against weight W as shown above.
In TensorFlow, strides is an array of 4 elements; the first element in this array indicates the stride for batch and last element indicates stride for features. It's good practice to remove the batches or features you want to skip from the data set rather than use a stride to skip them. You can always set the first and last element to 1 in strides in order to use all batches and features.
The middle two elements are the strides for height and width respectively. I've mentioned stride as one number because you usually have a square stride where height = width.
When someone says they are using a stride of 3, they usually mean tf.nn.conv2d(x, W, strides=[1, 3, 3, 1]).
To make life easier, the code is using tf.nn.bias_add() to add the bias. Using tf.add() doesn't work when the tensors aren't the same shape.

Convolutional Network in TensorFlow

Max Pooling



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

Max Pooling with 2x2 filter and stride of 2. Source: http://cs231n.github.io/convolutional-networks/

The above is an example of max pooling with a 2x2 filter and stride of 2. The left square is the input and the right square is the output. The four 2x2 colors in input represents each time the filter was applied to create the max on the right side. For example, [[1, 1], [5, 6]] becomes 6 and [[3, 2], [1, 2]] becomes 3.

```
def maxpool2d(x, k=2):
    return tf.nn.max_pool(
        x,
        ksize=[1, k, k, 1],
        strides=[1, k, k, 1],
        padding='SAME')
```

The tf.nn.max_pool() function does exactly what you would expect, it performs max pooling with the ksize parameter as the size of the filter.

Model

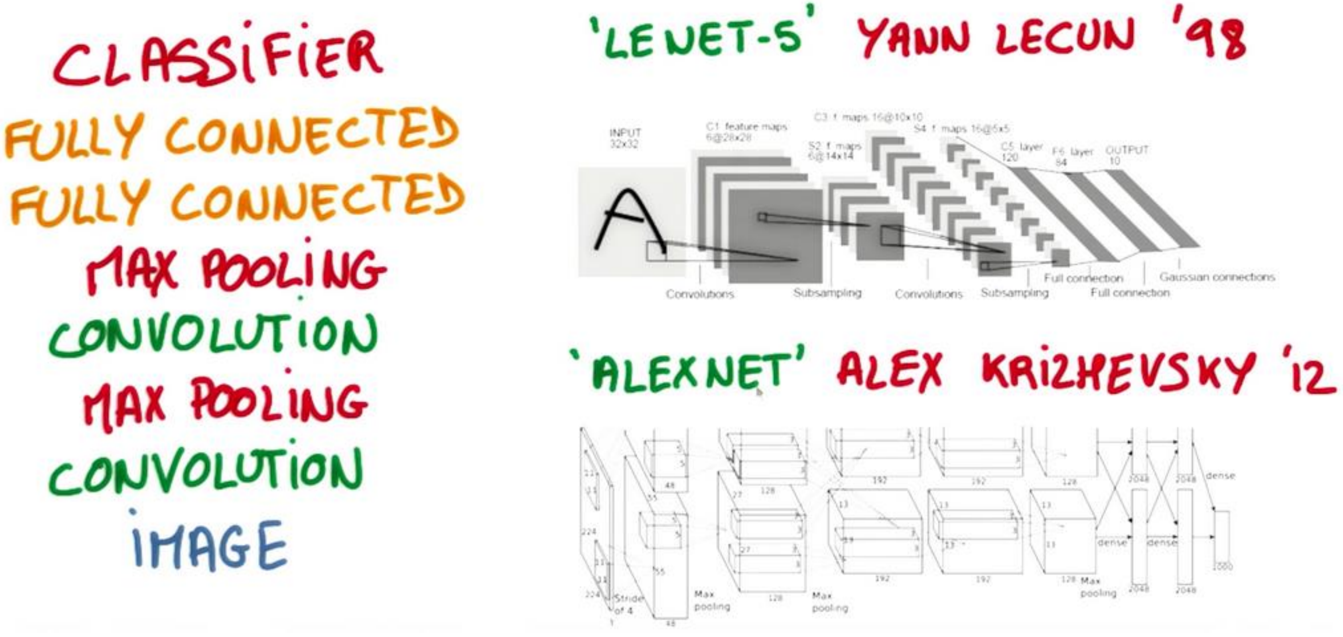


Image from Explore The Design Space video

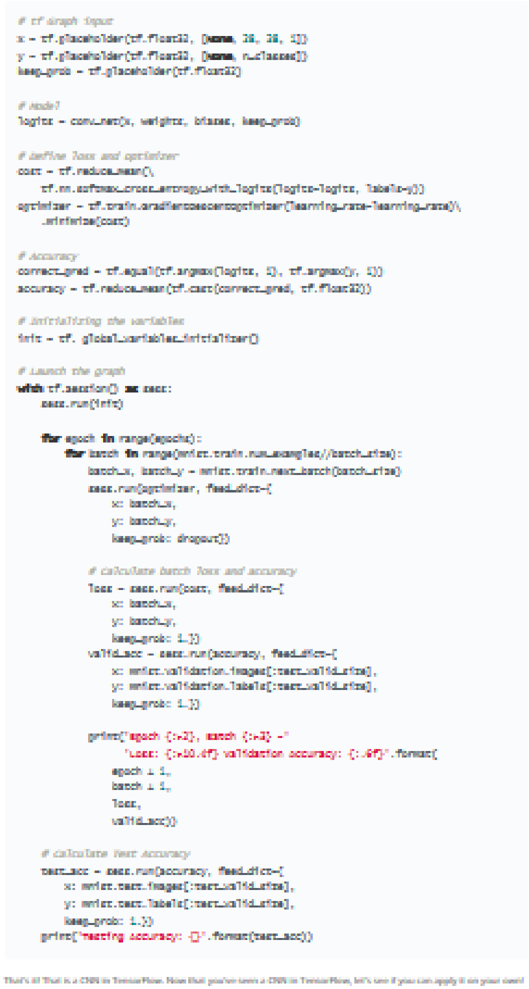
Convolutional Network in TensorFlow

In the code below, we're creating 3 layers alternating between convolutions and max pooling followed by a fully connected and output layer. The transformation of each layer to new dimensions are shown in the comments. For example, the first layer shapes the images from 28x28x1 to 28x28x32 in the convolution step. Then next step applies max pooling, turning each sample into 14x14x32. All the layers are applied from conv1 to output, producing 10 class predictions.

```
def conv_net(x, weights, biases, dropout):
    # Layer 1 - 28*28*1 to 14*14*32
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    conv1 = maxpool2d(conv1, k=2)
    # Layer 2 - 14*14*32 to 7*7*64
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    conv2 = maxpool2d(conv2, k=2)
    # Fully connected layer - 7*7*64 to 1024
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    fc1 = tf.nn.dropout(fc1, dropout)
    # Output Layer - class prediction - 1024 to 10
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

Session

Now let's run it!



```
# tf Graph input
x = tf.placeholder(tf.float32, [None, 28, 28, 1])
y = tf.placeholder(tf.float32, [None, n_classes])
keep_prob = tf.placeholder(tf.float32)

# Model
logits = conv_net(x, weights, biases, keep_prob)

# Define loss and optimizer
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)

# Accuracy
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
init = tf. global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(epochs):
        for batch in range(mnist.train.num_examples//batch_size):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            sess.run(optimizer, feed_dict={
                x: batch_x,
                y: batch_y,
                keep_prob: dropout})

            # Calculate batch loss and accuracy
            loss = sess.run(cost, feed_dict={
                x: batch_x,
                y: batch_y,
                keep_prob: 1.})
            valid_acc = sess.run(accuracy, feed_dict={
                x: mnist.validation.images[:test_valid_size],
                y: mnist.validation.labels[:test_valid_size],
                keep_prob: 1.})

            print('Epoch {:>2}, Batch {:>3} -'
                  'Loss: {:>10.4f} Validation Accuracy: {:.6f}'.format(
                epoch + 1,
                batch + 1,
                loss,
                valid_acc))

    # Calculate Test Accuracy
    test_acc = sess.run(accuracy, feed_dict={
        x: mnist.test.images[:test_valid_size],
        y: mnist.test.labels[:test_valid_size],
        keep_prob: 1.})
    print('Testing Accuracy: {}'.format(test_acc))
```

That's it! That is a CNN in TensorFlow. Now that you've seen a CNN in TensorFlow, let's see if you can apply it on your own!

That's it! That is a CNN in TensorFlow. Now that you've seen a CNN in TensorFlow, let's see if you can apply it on your own!

TensorFlow Convolution Layer

Using Convolution Layers in TensorFlow

Let's now apply what we've learned to build real CNNs in TensorFlow. In the below exercise, you'll be asked to set up the dimensions of the Convolution filters, the weights, the biases. This is in many ways the trickiest part to using CNNs in TensorFlow. Once you have a sense of how to set up the dimensions of these attributes, applying CNNs will be far more straight forward.

Review

You should go over the TensorFlow documentation for 2D convolutions. Most of the documentation is straightforward, except perhaps the padding argument. The padding might differ depending on whether you pass 'VALID' or 'SAME'.

Here are a few more things worth reviewing:

1. Introduction to TensorFlow -> TensorFlow Variables.
2. How to determine the dimensions of the output based on the input size and the filter size (shown below). You'll use this to determine what the size of your filter should be.

```
new_height = (input_height - filter_height + 2 * P)/S + 1
new_width = (input_width - filter_width + 2 * P)/S + 1
```

Instructions

1. Finish off each TODO in the conv2d function.
2. Setup the strides, padding and filter weight/bias (F_w and F_b) such that the output shape is (1, 2, 2, 3). Note that all of these except strides should be TensorFlow variables.

03TensorFlow Convolution Layer

Solution: TensorFlow Convolution Layer

Solution

Here's how I did it. **NOTE**: there's more than 1 way to get the correct output shape. Your answer might differ from mine.

```
def conv2d(input):
   # Filter (weights and bias)
   F_W = tf.Variable(tf.truncated_normal((2, 2, 1, 3)))
   F_b = tf.Variable(tf.zeros(3))
   strides = [1, 2, 2, 1]
   padding = 'VALID'
   return tf.nn.conv2d(input, F_W, strides, padding) + F_b
```

I want to transform the input shape (1, 4, 4, 1) to (1, 2, 2, 3). I choose 'VALID' for the padding algorithm. I find it simpler to understand and it achieves the result I'm looking for.

```
out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))
```

Plugging in the values:

```
out_height = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
out_width = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
```

In order to change the depth from 1 to 3, I have to set the output depth of my filter appropriately:

```
F_W = tf.Variable(tf.truncated_normal((2, 2, 1, 3)))
# (height, width, input_depth, output_depth)
F_b = tf.Variable(tf.zeros(3)) # (output_depth)
```

The input has a depth of 1, so I set that as the input_depth of the filter.

TensorFlow Pooling Layer

Using Pooling Layers in TensorFlow

In the below exercise, you'll be asked to set up the dimensions of the pooling filters, strides, as well as the appropriate padding. You should go over the TensorFlow documentation for tf.nn.max_pool(). Padding works the same as it does for a convolution.

Instructions

1. Finish off each TODO in the maxpool function.
2. Setup the strides, padding and ksize such that the output shape after pooling is (1, 2, 2, 1).

04TensorFlow Pooling Layer

Solution: TensorFlow Pooling Layer

Solution

Here's how I did it. **NOTE**: there's more than 1 way to get the correct output shape. Your answer might differ from mine.

```
def maxpool(input):
   ksize = [1, 2, 2, 1]
   strides = [1, 2, 2, 1]
   padding = 'VALID'
   return tf.nn.max_pool(input, ksize, strides, padding)
```

I want to transform the input shape (1, 4, 4, 1) to (1, 2, 2, 1). I choose 'VALID' for the padding algorithm. I find it simpler to understand and it achieves the result I'm looking for.

```
out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))
```

Plugging in the values:

```
out_height = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
out_width  = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
```

The depth doesn't change during a pooling operation so I don't have to worry about that.

Lab: LeNet In TensorFlow

Lab: LeNet in TensorFlow

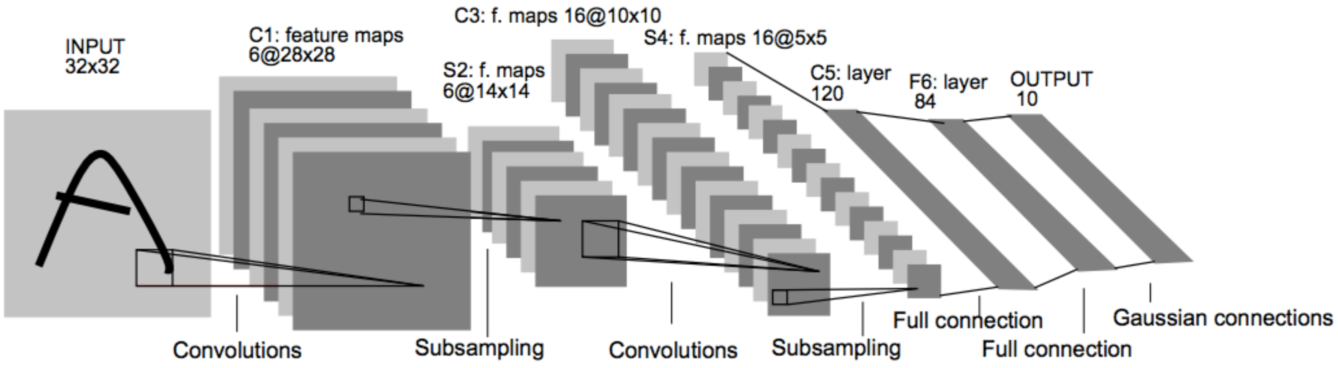


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet. Source: Yann Lecun.

You're now going to put together everything you've learned and implement the LeNet architecture using TensorFlow.
When you get to your next project, remember that LeNet can be a great starting point for your network architecture!

Instructions:

1.Set up your development environment with the CarND Starter Kit
2.git clone https://github.com/udacity/CarND-LeNet-Lab.git
3.cd CarND-LeNet-Lab
4.jupyter notebook
5.Finish off the architecture implementation in the LeNet function. That's the only piece that's missing.

Preprocessing

An MNIST image is initially 784 features (1D). If the data is not normalized from [0, 255] to [0, 1], normalize it. We reshape this to (28, 28, 1) (3D), and pad the image with 0s such that the height and width are 32 (centers digit further). Thus, the input shape going into the first convolutional layer is 32x32x1.

Lab: LeNet In TensorFlow

Specs

Convolution layer 1. The output shape should be 28x28x6.

Activation 1. Your choice of activation function.

Pooling layer 1. The output shape should be 14x14x6.

Convolution layer 2. The output shape should be 10x10x16.

Activation 2. Your choice of activation function.

Pooling layer 2. The output shape should be 5x5x16.

Flatten layer. Flatten the output shape of the final pooling layer such that it's 1D instead of 3D. The easiest way to do is by using tf.contrib.layers.flatten, which is already imported for you.

Fully connected layer 1. This should have **120 outputs**.

Activation 3. Your choice of activation function.

Fully connected layer 2. This should have **84 outputs**.

Activation 4. Your choice of activation function.

Fully connected layer 3. This should have **10 outputs**.

You'll return the result of the final fully connected layer from the LeNet function.

If implemented correctly you should see output similar to the following:

```
EPOCH 1 ...
Validation loss = 52.809
Validation accuracy = 0.864
EPOCH 2 ...
Validation loss = 24.749
Validation accuracy = 0.915
EPOCH 3 ...
Validation loss = 17.719
Validation accuracy = 0.930
EPOCH 4 ...
Validation loss = 12.188
Validation accuracy = 0.943
EPOCH 5 ...
Validation loss = 8.935
Validation accuracy = 0.954
EPOCH 6 ...
Validation loss = 7.674
Validation accuracy = 0.956
EPOCH 7 ...
Validation loss = 6.822
Validation accuracy = 0.956
EPOCH 8 ...
Validation loss = 5.451
Validation accuracy = 0.961
EPOCH 9 ...
Validation loss = 4.881
Validation accuracy = 0.964
EPOCH 10 ...
Validation loss = 4.623
Validation accuracy = 0.964
Test loss = 4.726
Test accuracy = 0.962
```

Parameters Galore

As an additional fun exercise calculate the total number of parameters used by the network. Note, the convolutional layers use weight sharing!

Supporting Materials

lenet.py

Solution: LeNet In TensorFlow

Solution

You can see one implementation of the solution in the GitHub repo.

Here is the LeNet function:

LeNet function

Walkthrough

Let's go through this solution layer by layer.

```
# Hyperparameters
mu = 0
sigma = 0.1
```

This solution uses the tf.truncated_normal() function to initialize the weights and bias Variables. Using the default mean and standard deviation from tf.truncated_normal() is fine. However, tuning these hyperparameters can result in better performance.

```
# SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma))
conv1_b = tf.Variable(tf.zeros(6))
conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b
```

This layer transforms the Tensor **32x32x1** to **28x28x6**.
Use a filter with the shape (5, 5, 1, 6) with VALID padding.
Recall the shape has dimensions: (height, width, input_depth, output_depth).
With VALID padding, the formula for the new height and width is:

```
out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))
```

Plugging in values:

```
out_height = ceil(float(32 - 5 + 1) / float(1)) = 28
out_width = ceil(float(32 - 5 + 1) / float(1)) = 28
```

Clearly, the strides must equal 1, or the output would be too small.

```
# SOLUTION: Activation.
conv1 = tf.nn.relu(conv1)
```

A standard ReLU activation. You might have chosen another activation.

```
# SOLUTION: Pooling. Input = 10x10x16. Output = 5x5x16.
conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')
```

Solution: LeNet In TensorFlow

The formula to calculate the new height and width for a pooling layer mirrors the formula for a convolutional layer.

```
new_height = ceil(float(28 - 2 + 1) / float(2)) = ceil(13.5) = 14
new_width = ceil(float(28 - 2 + 1) / float(2)) = ceil(13.5) = 14
```

The next round of convolution -> activation -> pooling uses an identical methodology.

```
# SOLUTION: Flatten Layer.
fc0 = flatten(conv2)
```

The flatten function flattens a Tensor into two dimensions: (batches, length). The batch size remains unaltered, so all of the other dimensions of the input Tensor are flattened into the second dimension of the output Tensor.

In this model, the the output shape of Pooling Layer 2 should be **5x5x16** (ignoring batch size). Applying flatten will multiply the length of each dimension together, which equals **400**.

Now that the Tensor is 2D, it's ready to be used in fully connected layers.

```
# SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 120.
fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))
fc1_b = tf.Variable(tf.zeros(120))
fc1   = tf.matmul(fc0, fc1_W) + fc1_b
# SOLUTION: Activation.
fc1    = tf.nn.relu(fc1)
# SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
fc2_W  = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev = sigma))
fc2_b  = tf.Variable(tf.zeros(84))
fc2    = tf.matmul(fc1, fc2_W) + fc2_b
# SOLUTION: Activation.
fc2    = tf.nn.relu(fc2)
# SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 10.
fc3_W  = tf.Variable(tf.truncated_normal(shape=(84, 10), mean = mu, stddev = sigma))
fc3_b  = tf.Variable(tf.zeros(10))
logits = tf.matmul(fc2, fc3_W) + fc3_b
```

You're already familiar with fully connected layers so I won't go into much detail. Note the output sizes: **120**, **84**, and **10**.

Congratulations! You're now a convolution and pooling expert!

CNNs - Additional Resources

Additional Resources

There are many wonderful free resources that allow you to go into more depth around Convolutional Neural Networks. In this course, our goal is to give you just enough intuition to start applying this concept on real world problems so you have enough of an exposure to explore more on your own. We strongly encourage you to explore some of these resources more to reinforce your intuition and explore different ideas.

These are the resources we recommend in particular:

- Andrej Karpathy's CS231n Stanford course on Convolutional Neural Networks.
- Michael Nielsen's free book on Deep Learning.
- Goodfellow, Bengio, and Courville's more advanced free book on Deep Learning.