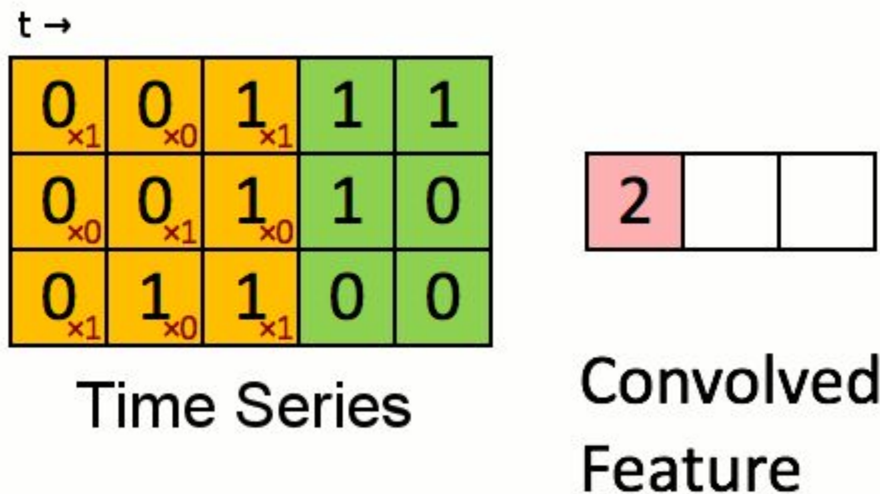
Convolutional Networks

Dhruv Shah, Soham Ghosh

How to compute a convolution?

1. Given an input, overlay a small window (known as the convolution kernel) on the input. Find the dot product between the kernel and the segment of the input that was covered by the kernel.
2. Slide the kernel across the input and perform the previous step for all the different kernel locations.
3. Perform steps 1 and 2 for all output channels, each with a different kernel.

How to compute a convolution? (3D time series)



How to compute a convolution? (Image, one channel)

1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

How to compute a convolution? (Image, 3 channels)

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #1

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #2

0	1	1
0	1	0
1	-1	1

Kernel Channel #3

$$308 \quad + \quad -498 \quad + \quad 164 \quad + 1 = -25$$

Bias = 1

Output

-25				...
				...
				...
				...
...	...	...	...	...

How to compute a convolution?

Good questions to ask:

- Does the kernel have to completely fit inside the input?
    - Can apply **padding** to the input
- Do we have to place the kernel at all possible locations? Or can we just place them at regular intervals?
    - Can adjust the **stride** of the convolution

Animations of padding and striding, as well as other sometimes useful concepts such as transposition and dilation:

https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

A technicality

Technically, the kernel needs to be mirror reversed across each of its dimensions before we slide it across the input (“flip and drag”).

- Without the mirroring step, the operation is called a correlation instead.
- Convolution has nicer mathematical properties than correlation.

But really, in deep learning it doesn’t matter since our kernels are randomly initialized and learned during the training process. So we mostly just ignore the mirror reversing step and use the two terms interchangeably.

What does convolution really do?

Sounds a little arbitrary to slide a window across an input, right?

What do we even get out of this?

1. Convolution as a way to extract local features
2. Convolution as a way to reduce model complexity

Convolution as a feature detector

How do you find the vertical edges in an image?

How do you find the horizontal edges?

How can we combine the two to find the locations of all the edges?

Convolution as a feature detector

-1	0	1
-2	0	2
-1	0	1

How do you find the vertical edges in an image?

1	2	1
0	0	0
-1	-2	-1

How do you find the horizontal edges?

How can we combine the two to find the locations of all the edges?

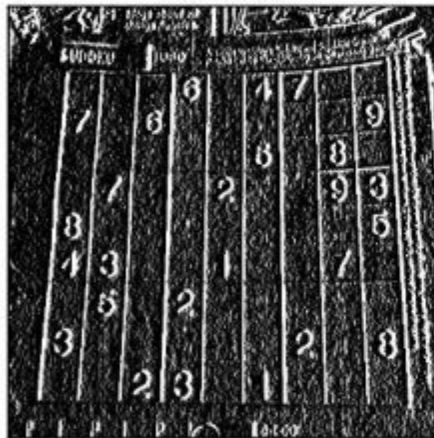
|*horizontal edge intensity*| + |*vertical edge intensity*|

√ (*horizontal edge intensity* ^{2} + *vertical edge intensity* ^{2})

Convolution as a feature detector

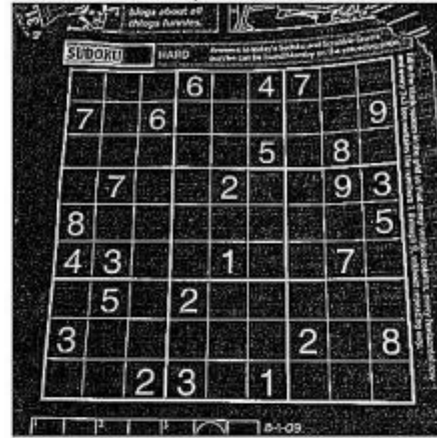
Original

Vertical edges



Horizontal edges

All edges



Convolution as a feature detector

You are given a 2D image with 2 channels:

- Denote the first channel as H and the second channel as V
- Large values in H represent horizontal edges and large values in V represent vertical edges. The values are signed, and were computed using the kernels shown on the previous slide.

How can we find solid 5x5 boxes by convolving on H and V?

Convolution as a feature detector

Kernel for the H channel:

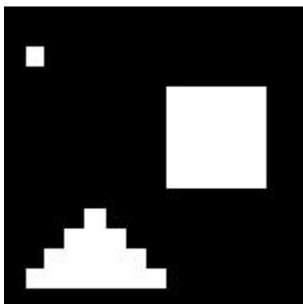
-1	-1	-1	-1	-1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1

Kernel for the V channel:

1	0	0	0	-1
1	0	0	0	-1
1	0	0	0	-1
1	0	0	0	-1
1	0	0	0	-1

Convolution as a feature detector

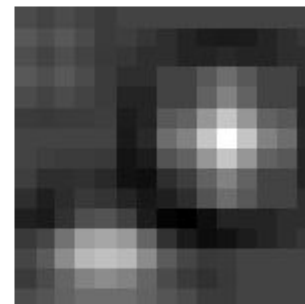
Original



Horizontal edges

Vertical edges

5x5 boxes



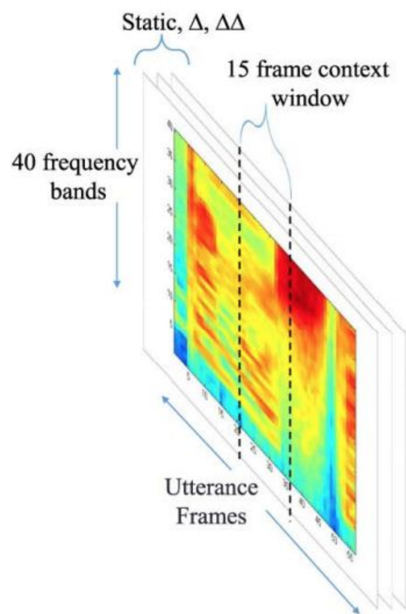
Convolution as a feature detector

A convolution is used to extract simple features in the local neighborhood of each position in the input.

- An image ➜ Edges, corners, dots...
- Words ➜ Prepositions, short phrases, compound words...
- Sound intensity waveform ➜ Phonemes, tones, inflection...

Perform convolutions on convolutions (with a nonlinear function in between) to find larger and more complex features in the input.

Convolution - what does it mean for Audio?



HW 1 data!

- Use MFSC/MFCC features (40 frequency bands)
- Why does it make sense to have convolutions?
- Because CNNs are translation invariant.
- Translation Invariant means that the model can recognize features irrespective of its place in the input.
- Also compared to MLP, CNNs have lower complexity in terms of number of parameters to learn.

Convolution to reduce complexity

Let’s consider MLP and CNN for MFCC. Input dimension of a frame is 40

Let the context size be 10. Hence, one input to the NN is (21, 40). The output dimension is 138

Number of parameters to be learned for a 3 layer MLP with hidden layer size of 128 is

Convolution to reduce complexity

Let's consider MLP and CNN for MFCC. Input dimension of a frame is 40

Let the context size be 10. Hence, one input to the NN is (21, 40). The output dimension is 138

Number of parameters to be learned for a 3 layer MLP with hidden layer size of 128 is

$$(21*40*128 + 128*128 + 128*128 + 128*138) \sim \mathbf{160k}$$

Convolution to reduce complexity

Let's consider MLP and CNN for MFCC. Input dimension of a frame is 40

Let the context size be 10. Hence, one input to the NN is (21, 40). The output dimension is 138

Number of parameters to learn for a 10 layer CNN with with 3 filters each of size 3x3, stride=1 and no padding

Convolution to reduce complexity

Let's consider MLP and CNN for MFCC. Input dimension of a frame is 40

Let the context size be 10. Hence, one input to the NN is (21, 40). The output dimension is 138

Number of parameters to learn for a 10 layer CNN with with 3 filters each of size 3x3, stride=1 and no padding. Fully connected layer in the end.

((3*3*1*3) + (9 * (3*3*3*3)) + (19*38*138)) ~ **100k**

Convolution to reduce complexity

The input is a sequence of 1000 words. Each word is represented as a 100-dimensional vector.

The output is another 1000x100 matrix.

If we do this with a normal NN layer, how many parameters do we need?

*Weight matrix: (1000x100) ** 2 = 10,000,000,000*

Biases: 1000x100 = 100,000

We need 10,000,100,000 parameters

Convolution to reduce complexity

The input is a sequence of 1000 words. Each word is represented as a 100-dimensional vector.

The output is another 1000x100 matrix.

If we use a convolutional layer with a kernel size of 5 (plus a bias term for each of the 100 output channels), how many parameters do we need?

Convolution to reduce complexity

The input is a sequence of 1000 words. Each word is represented as a 100-dimensional vector.

The output is another 1000x100 matrix.

If we use a convolutional layer with a kernel size of 5 (plus a bias term for each of the 100 output channels), how many parameters do we need?

Convolution kernels: 100 x 100 x 5 = 50,000

Biases: 100

We need 50,100 parameters

Convolution to reduce complexity

Convolution assumes translational invariance in the inputs.

- E.g. The first 5 words can be analyzed in the mostly same way as the last 5 words (seems reasonable)

And that only local / nearby information is important.

- E.g. The meaning of a word doesn’t strongly depend on anything more than two words away (probably not true!)

Convolutional layers use fewer parameters at the expense of making these assumptions. A linear layer will basically assume nothing, and therefore would require many more parameters to define.

Keeping the dimensionality under control

Suppose we're trying to perform a classification task on 200x200 images. We need to find a way to systematically process a 40,000 dimensional input to a low-dimensional classification vector.

But the dimensionality of an input vector doesn't change very much after passing through a convolutional layer (assuming a small kernel size and a stride of 1).

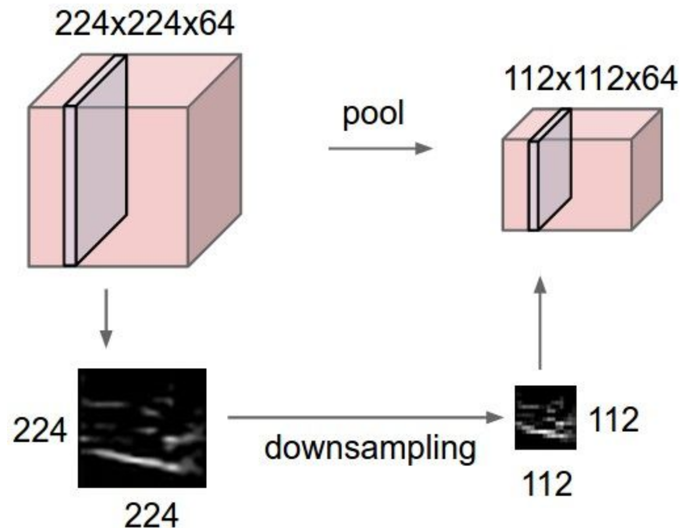
How can we reduce the dimensionality of the feature maps without losing useful information?

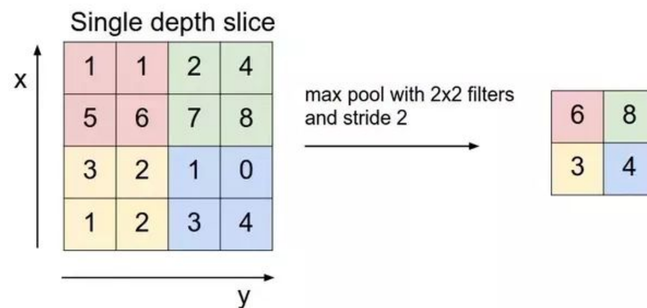
Pooling

A feature map essentially tell us:

- Whether a feature exists in the image (high activation)
- If so, approximately where in the image was it found

We can preserve both pieces of information pretty well with max-pooling





Pooling

Max-pooling splits a feature map into fixed sized pools (e.g. 2x2 segments for images) and computes the maximum value within each pool.

- If a feature was expressed strongly at some location in the input, then the output for the pool containing this location will also have a high value.
- The pools are usually quite small, so we would still know fairly accurately where the features are located in the input.
- It’s still a differentiable operation
- Also, we learn to be a little more robust to translation: let’s say we moved a picture just one pixel to the right, we would not want our prediction to change much (and it won’t, thanks to pooling!)

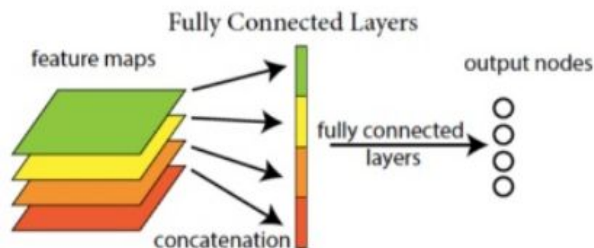
Stride greater than 1

Why make your model more complex by creating pooling layers? It's possible to use a stride size larger than 1, which would also reduce the dimensionality of the output feature map considerably.

This comes at the cost of being more sloppy at determining whether a feature exists in an image, since a layer with stride larger than 1 might skip over the location where the feature is expressed most strongly.

Fully-connected layers

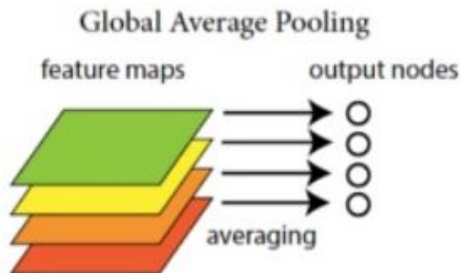
Simply treat the final feature map as a vector, and use a fully-connected neural network that outputs the desired number of dimensions.



This can only be done after all convolutions, because the output of the fully-connected layer does not store any location information. Performing a convolution on the output of a fully-connected layer will be meaningless!

Global average pooling

If you don’t want to use a fully-connected layer, you can also take the average activation for each channel of the final feature map. The number of channels in the final feature map equals the number of output dimensions.



Again, this can only be done after all convolutions.

Deep residual networks

A deep convolutional neural network would perform convolutions of convolutions of convolutions...

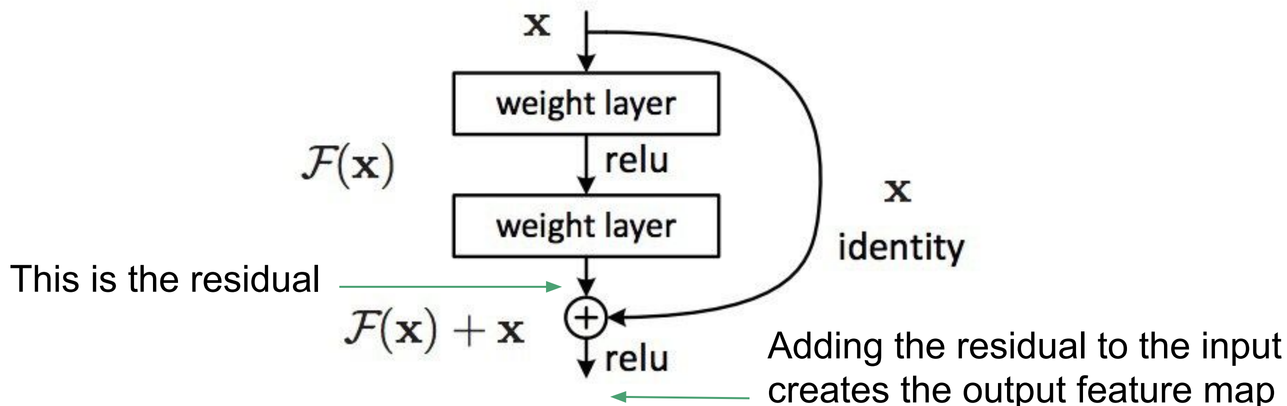
The output of the final convolutional layer has the potential to find very complex features, but what if we really don’t need that much complexity?

- We end up forcing many of the layers to “do nothing” to the input.
- Until then we will suffer from the vanishing gradient problem:
    - The output of an early layer gets garbled by (untrained) later layers
    - A small change in the output of an early layer will probably not change the output meaningfully
    - Even if it did, the effects of the small charge are likely quite chaotic and unpredictable

Deep residual networks

Instead of learning a new feature map from the input, we can instead learn the residual, or the difference between the desired feature map and the original.

Start with an image, compute a residual, add the residual to the image, end with an image, repeat. Can produce a very deep convolutional network from this.



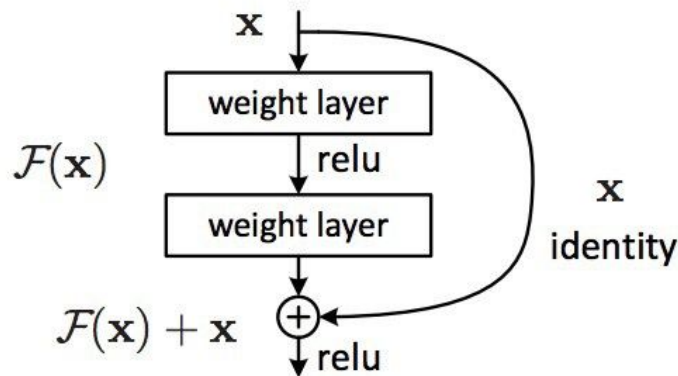
Deep residual networks

It’s easy to make a layer do nothing: just set the residual to zero.

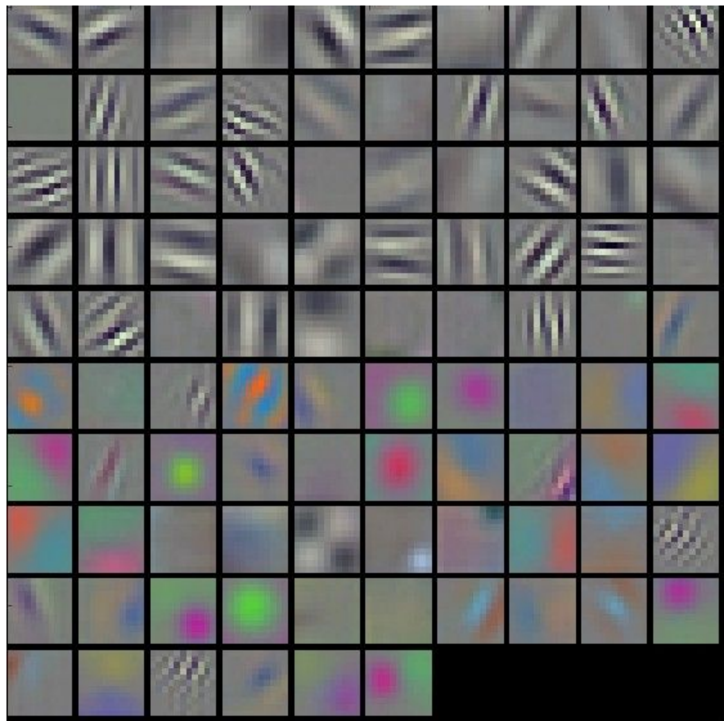
The vanishing gradient problem is also alleviated. The gradients at the output layer propagate backward unchanged through the “skip connections”, in addition to propagating along the original path through all the convolutional layers.

Why 2 layers per block?

2-layer CNNs with enough channels can define arbitrarily complex filters, much like how vanilla networks with one hidden layer can approximate any function.



Understanding CNNs (and what they have learned)

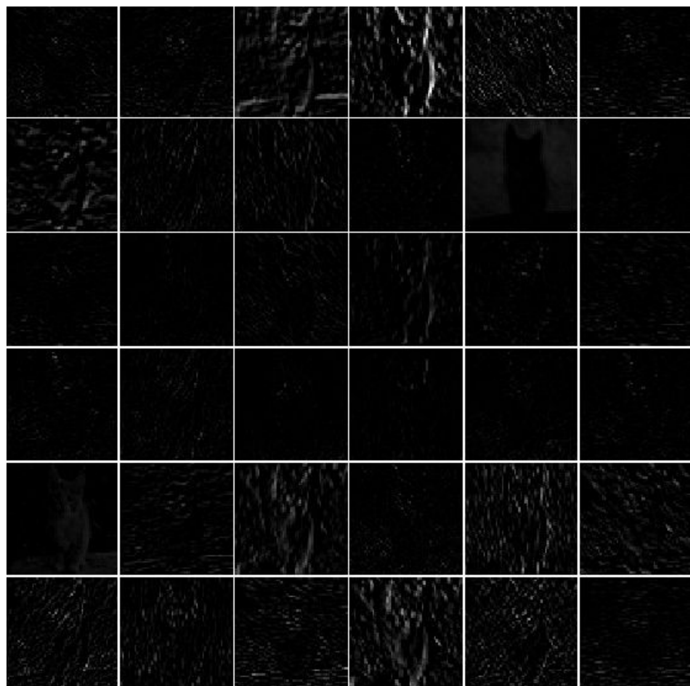


Typical filters learned in **AlexNet**

- Notice there are two streams of processing: ~50% of the filters are grayscale edge detectors, ~50% are color features

http://cs231n.github.io/understanding-cnn/

Understanding CNNs (and what they have learned)

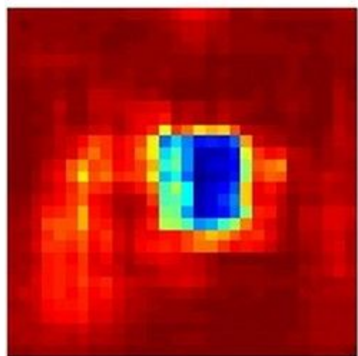


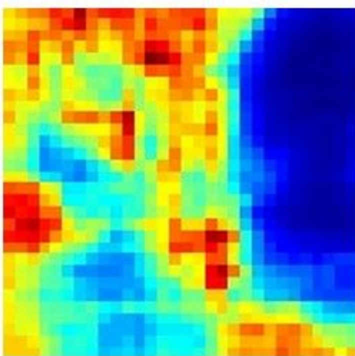
It also helps to look at the **activation maps** produced by a pass through one convolution layer (followed by non-linearity)

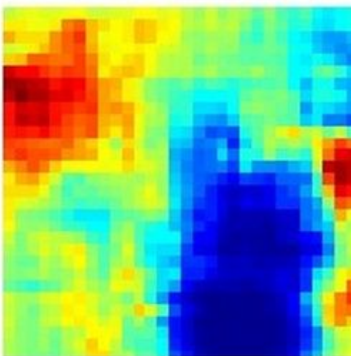
- If this is what you see for a lot of the images, that means there are *dead* filters
- *Dead* filters can happen because of **high learning rates** - all the outputs from convolution are negative, so post ReLU we get 0s.
- Feature maps should be uncorrelated

http://cs231n.github.io/understanding-cnn/

Understanding CNNs (and what they have learned)







- Cover up part of the picture
- At different parts that you cover up, record the probability of the true class
- **Blue** parts - probability drops sharply → that part of the image is important for making correct prediction

Dealing with CNN's in Pytorch

This is all useful conceptual knowledge, but how do you make CNNs in code?

We have a demo on Jupyter Notebook.

Convolutional Neural Networks in Pytorch

This tutorial expects a basic understanding of pytorch and convolutional networks and focuses on pytorch-specific implementation concerns.

- `torch.nn` documentation [http://pytorch.org/docs/master/nn.html]
- Convolution animations. Make sure to understand these animations so you can visualize stride, padding, etc. [https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md]

In [240]:

```
import torch
import torch.nn.functional as F
from torch import autograd, nn
import numpy as np
import matplotlib.pyplot as plt
from torchvision import transforms, datasets
```

Data Format

Dimension ordering is very important. For pytorch:

- 2D data like images should be `(samples, channels, height, width)` sometimes called "NCHW" or "channels first"
- 1D data like stock prices should be `(samples, channels, time)`
- 2d filters should be `(out_channels, in_channels, height, width)`
- 1d filters should be `(out_channels, in_channels, time)`