CSE 676 Deep Learning

DNN and CNN

Jue Guo[1]

University at Buffalo

[1] The material is adapted from *Dive into Deep Learning*

Course Content

Generalization in Classification: Quick Review

Multilayer Perceptrons
- Activation Functions
- Numerical Stability and Initialization

Convolution Neural Networks

Generalization in Classification: Quick Review

- Empirical Error

$$\epsilon_{\mathcal{D}}(f) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\left(f\left(\mathbf{x}^{(i)}\right) \neq y^{(i)}\right)$$

- Population Error

$$\epsilon(f) = E_{(\mathbf{x},y)\sim P}\mathbf{1}(f(\mathbf{x}) \neq y) = \iint \mathbf{1}(f(\mathbf{x}) \neq y)p(\mathbf{x}, y)d\mathbf{x}dy$$

Generalization in Classification: Empirical Error

- Let's focus on fixed classifier f, without worrying about how it was obtained.
- Suppose that we possess a *fresh* dataset of examples $\mathcal{D} = \left(\mathbf{x}^{(i)}, y^{(i)}\right)_{i=1}^{n}$ that were not used to train the classifier f.
- The *empirical error* of our classifier f on $\mathcal{D}$ is simply the fraction of instances for which the prediction $f\left(\mathbf{x}^{(i)}\right)$ disagrees with the true label $y^{(i)}$

$$\epsilon_{\mathcal{D}}(f) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\left(f\left(\mathbf{x}^{(i)}\right) \neq y^{(i)}\right)$$

Generalization in Classification: Population Error

- By contrast, the *population error* is the *expected* fraction of examples in the underlying population (some distribution $P(X, Y)$ characterized by probability density function $p(\mathbf{x}, y)$ for which our classifier disagree with the true label:

$$\epsilon(f) = E_{(\mathbf{x},y)\sim P}\mathbf{1}(f(\mathbf{x}) \neq y) = \iint \mathbf{1}(f(\mathbf{x}) \neq y)p(\mathbf{x}, y)d\mathbf{x}dy$$

- This quantity is the one that we actually care about, we cannot observe it directly, just as we can not directly observe the average height in a large population without measuring every single person.

Test Set: Central limit Theorem

- The *central limit theorem* guarantees that whenever we posses n random samples $a_1, \ldots, a_n$ drawn from any distribution with the mean μ and standard deviation σ, as the number of sample n approaches infinity, the sample average $\hat{\mu}$ approximately tends towards a normal distribution centered at the true mean and with standard deviation $\frac{\delta}{\sqrt{n}}$.
- This tells us something **important**: as the number of examples grows large, our test error $\epsilon_{\mathcal{D}}(f)$ should approach the true error $\epsilon(f)$ at a rate of $\mathcal{O}(1/\sqrt{n})$. Thus, *to estimate our test error twice as precisely*, we must collect four times as large a test set. To reduce our test error by a factor of one hundred, we must collect ten thousand times as large a test set. In general, such a rate of $\mathcal{O}(1/\sqrt{n})$ is often the best we can hope for in statistics.

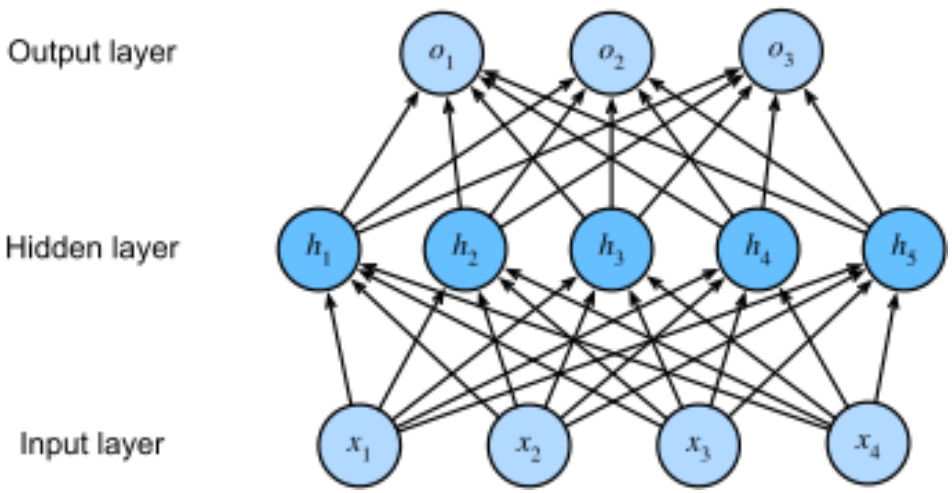
Multilayer Perceptrons

Simplest Deep Networks: Multilayer Perceptrons

- multiple layer of neurons
- fully connected to those in the layer below and those above

Limitation of Linear Models

- Weaker assumption of monotonicity
- Good with task has linear property



Hidden Layers

Linear to Non-Linear

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$$

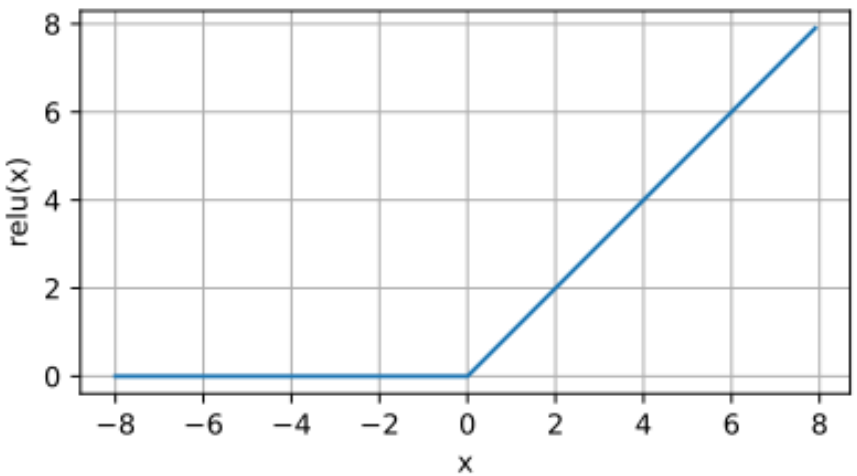
- **Non-Linear Activation Function**

$$\mathbf{H} = \sigma\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right)$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$$

Activation Functions: ReLU Function

- **ReLU Function**: $\text{ReLU}(x) = \max(x, 0)$

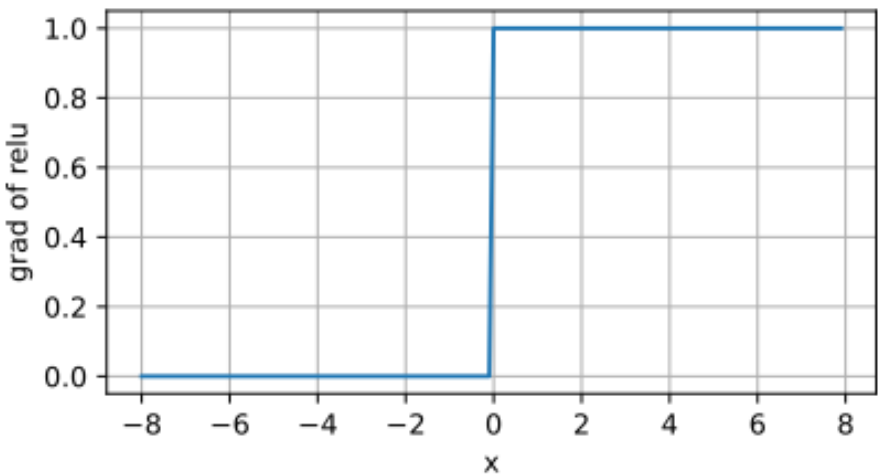
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', ↩
    figsize=(5, 2.5))
```



The Derivative of ReLU Function

- The well behaving ReLU:

```
y.backward(torch.ones_like(x), retain_graph=True)
```

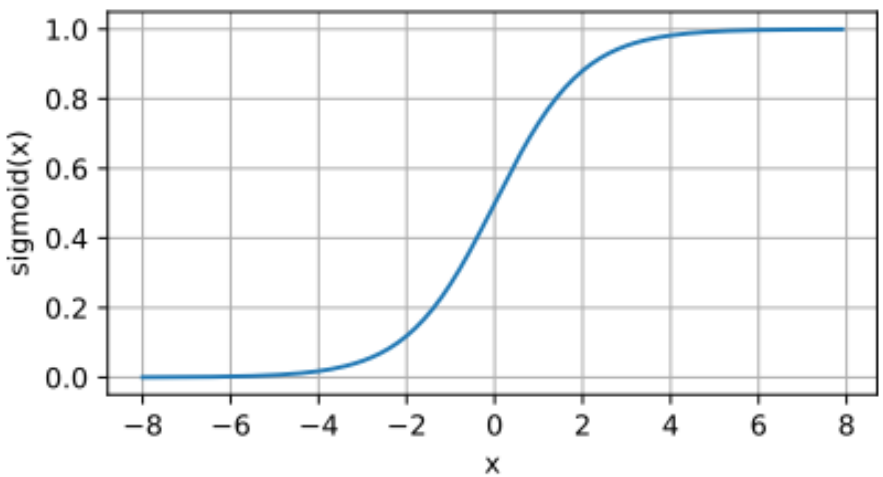


Pros of ReLU Function

- They vanish or they just let the argument through.
- This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradient problems (more on this later)

Activation Function: Sigmoid Function

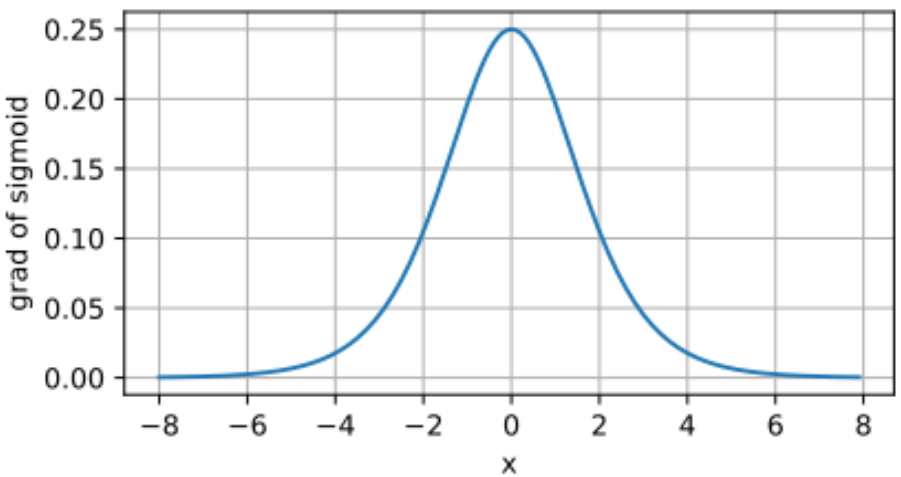
- **Sigmoid Function**: $\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}$
- *squashing function*: it squashes any input in the range (-inf,inf) to some value in the range (0,1).



The Derivative of Sigmoid Function

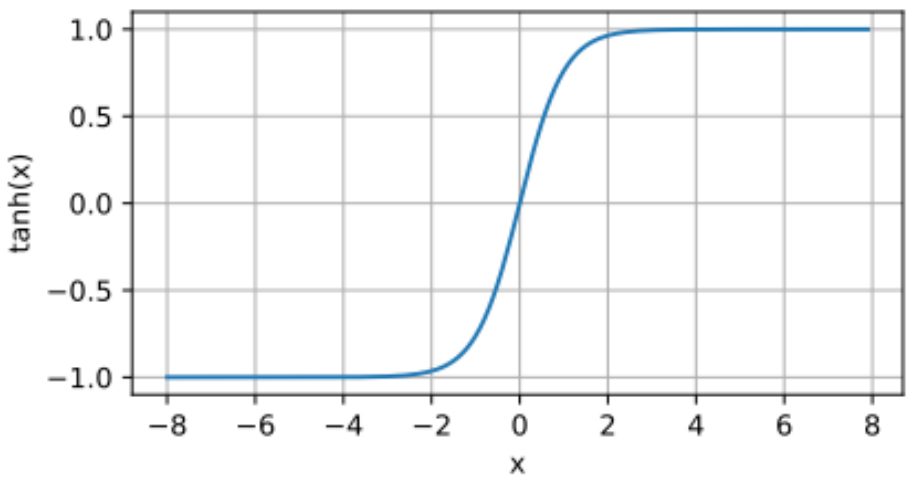
- The derivative of the sigmoid function is given by the following equation:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)(1-\text{sigmoid}(x))$$



Activation Function: Tanh

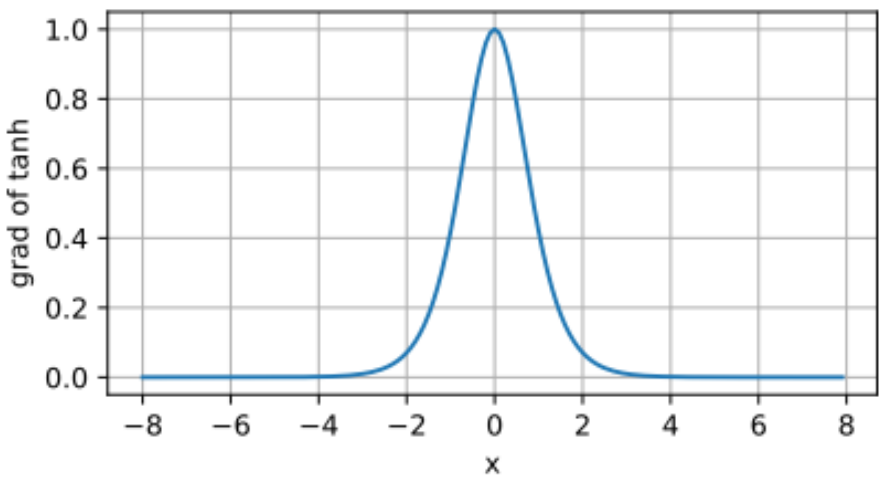
- Tanh Function: $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$



The Derivative of Tanh Function

- The derivative of the tanh function is:

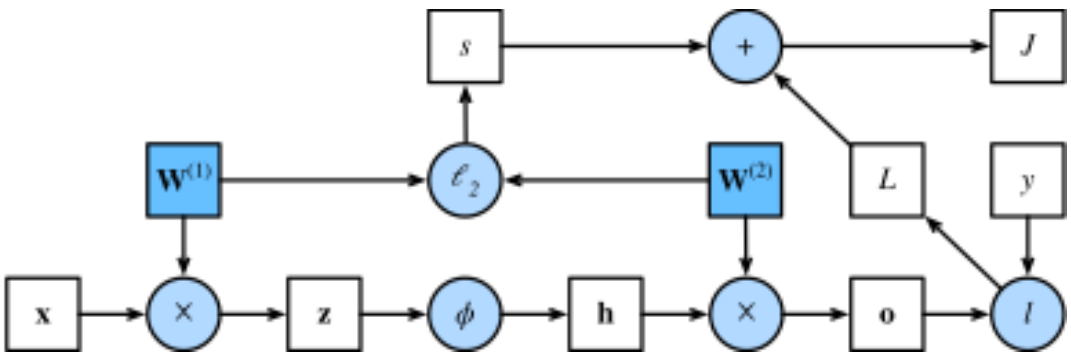
$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$



Forward Propagation

Pay the cost to be the boss

1. $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$
2. $\mathbf{h} = \phi(\mathbf{z})$
3. $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$
4. $L = l(\mathbf{o}, y)$
5. $s = \frac{\lambda}{2}\left(\left\|\mathbf{W}^{(1)}\right\|_F^2 + \left\|\mathbf{W}^{(2)}\right\|_F^2\right)$
6. $J = L + s$



Backpropagation

Partial Derivative $\frac{\partial Z}{\partial X} = \text{prod}\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right)$, $\mathrm{Y} = f(\mathrm{X})$ and $\mathrm{Z} = g(\mathrm{Y})$

- $\frac{\partial J}{\partial L} = 1$ and $\frac{\partial J}{\partial s} = 1$
- $\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$
- $\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$ and $\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$
- $\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$
- $\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}$
- $\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$
- $\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}$

Training Neural Network

- Relationship between forward and backpropagation ($\mathbf{W}^{(1)}$,$\mathbf{W}^{(2)}$)
  - $s = \frac{\lambda}{2}\left(\left\|\mathbf{W}^{(1)}\right\|_F^2 + \left\|\mathbf{W}^{(2)}\right\|_F^2\right)$
  - $\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$
- After parameter initialization,
  - **forward propagation** calculate and stores intermediate variables
  - **backpropagation** sequentially calculates and stores gradients of intermediate variables and parameters within the neural network in reversed order

Numerical Stability and Initialization I

Consider a deep network with L layers, input $\mathbf{x}$ and output $\mathbf{o}$

$$\mathbf{h}^{(l)} = f_l\left(\mathbf{h}^{(l-1)}\right) \text{ and thus } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x})$$

gradient of $\mathbf{o}$ with respect to any set of parameter $\mathbf{W}^{(l)}$ as follows:

$$\partial_{\mathbf{W}^{(l)}}\mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}}\mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}}\mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}}\mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}$$

Numerical Stability and Initialization II

Spot any problems?

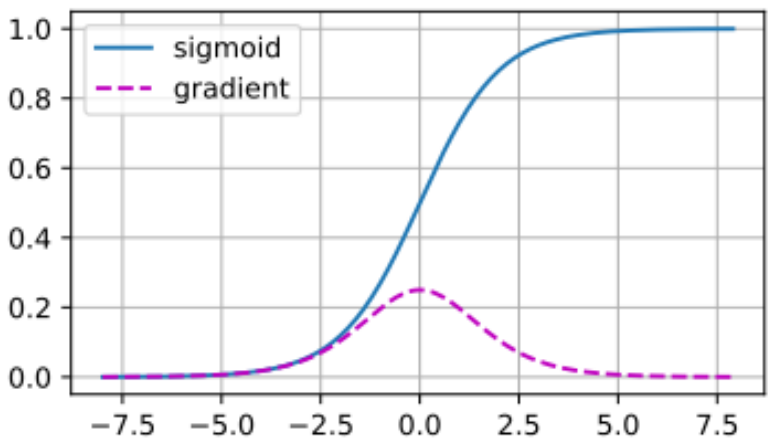
$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}$$

- Excessively large, destroy our model (the *exploding gradient problem*)
- Excessively small, rendering learning impossible as parameters hardly move on each update. (the *vanishing gradient problem*)

Vanishing Gradients

- Cause? The choice of the activation function σ

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.sigmoid(x)
y.backward(torch.ones_like(x))
```



- As you can see, the sigmoid gradient vanishes both when its inputs are large and when they are small.
- More stable choice: ReLU

Exploding Gradients I

Now, the opposite problem, when gradients explode

```
M = torch.normal(0, 1, size=(4, 4))
print('a single matrix \n',M)
for i in range(100):
    M = M @ torch.normal(0, 1, size=(4, 4))
print('after multiplying 100 matrices\n', M)
```

Exploding Gradients II

```
a single matrix
tensor([[ 0.0837, -0.9784, -0.5752, -0.0418],
        [ 2.0032,  2.0948, -1.4284, -1.5950],
        [-0.9720, -2.1672, -0.2809,  0.2282],
        [-0.7581,  0.0328, -0.2364, -0.5804]])
after multiplying 100 matrices
tensor([[ 7.5119e+24, -9.2313e+24, -2.1761e+24,
     7.0456e+23],
        [-1.3462e+24,  1.6544e+24,  3.8999e+23, -1.2627e+23],
        [ 1.4648e+25, -1.8001e+25, -4.2433e+24,  1.3739e+24],
        [ 8.9242e+24, -1.0967e+25, -2.5852e+24,
          8.3702e+23]])
```

This happens usually due to the initialization of a deep network, we have no chance of getting a gradient descent optimizer to converge.

Parameter Initialization

One solution to aforementioned problems

- **Default Initialization**
  - normal distribution to initialize the value of weights
  - framework (pytorch and tensorflow) will use a default random initialization methods
  - works well in practice for moderate problem size

Xavier Initialization I

The **goal**, *" maintain the variance of the input and output activations throughout the network, making training deep neural networks more efficient."*

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j$$

- The weights w_{ij} are all drawn independently from the same distribution.
  - distribution has zero mean and variance σ^2

Let's pull out some equations and definitions:

$$\mu = E[X] = \sum_{i=1}^{n} p_i x_i$$

$$\sigma^2 = E\left[(X-\mu)^2\right] = E\left[X^2\right] - (E[X])^2$$

Xavier Initialization II

For now,

- let's assume that the inputs to the layer x_j have zero mean and variance γ^2
- the inputs are independent of w_{ij} and independent of each other

$$\begin{aligned} E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\ &= 0 \end{aligned}$$

$$\begin{aligned} \text{Var}[o_i] &= E\left[o_i^2\right] - (E[o_i])^2 \\ &= \sum_{j=1}^{n_{\text{in}}} E\left[w_{ij}^2 x_j^2\right] - 0 \\ &= \sum_{j=1}^{n_{\text{in}}} E\left[w_{ij}^2\right] E\left[x_j^2\right] \\ &= n_{\text{in}}\, \sigma^2 \gamma^2 \end{aligned}$$

Xavier Initialization III

To keep variance fixed:

- forward propagation: $n_{\text{in}}\,\sigma^2 = 1$
- backpropagation: $n_{\text{out}}\sigma^2 = 1$

to balance, **Xavier initialization** is used:

$$\frac{1}{2}\left(n_{\text{in}} + n_{\text{out}}\right)\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

Typically,

- samples weights from a Gaussian distribution with zero mean and variance $\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$
- the methods works well in practice

Breaking the Symmetry

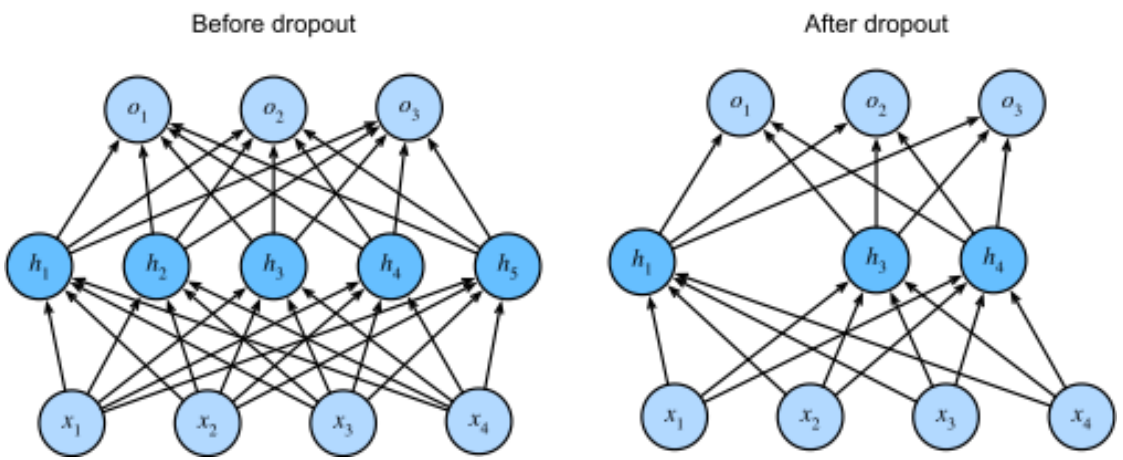
Consider one-hidden-layer MLP with just two hidden units.

- If we initialize all of the parameters of the hidden layer as $\mathbf{W}^{(1)} = c$ for some costant c
- Same input, parameters, same activation $\rightarrow$ same gradient

Dropout I

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

- Disable dropout at test time.
- Some researchers use dropout at test time as a heuristic for estimating the uncertainty of neural network predictions:
  - if the predictions agree across many different dropout masks, then we might say that the network is more confident.



Dropout II

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1 # raise assertion error if not true
    if dropout == 1: return torch.zeros_like(X) # drop all
    mask = (torch.rand(X.shape) > dropout).float() # match the rate of drop, greater set to 1
    return mask * X / (1.0 - dropout)
```

```
X = torch.arange(16, dtype = torch.float32).reshape((2, 8))
print('dropout_p = 0:', dropout_layer(X, 0))
print('dropout_p = 0.5:', dropout_layer(X, 0.5))
print('dropout_p = 1:', dropout_layer(X, 1))
```

Dropout III

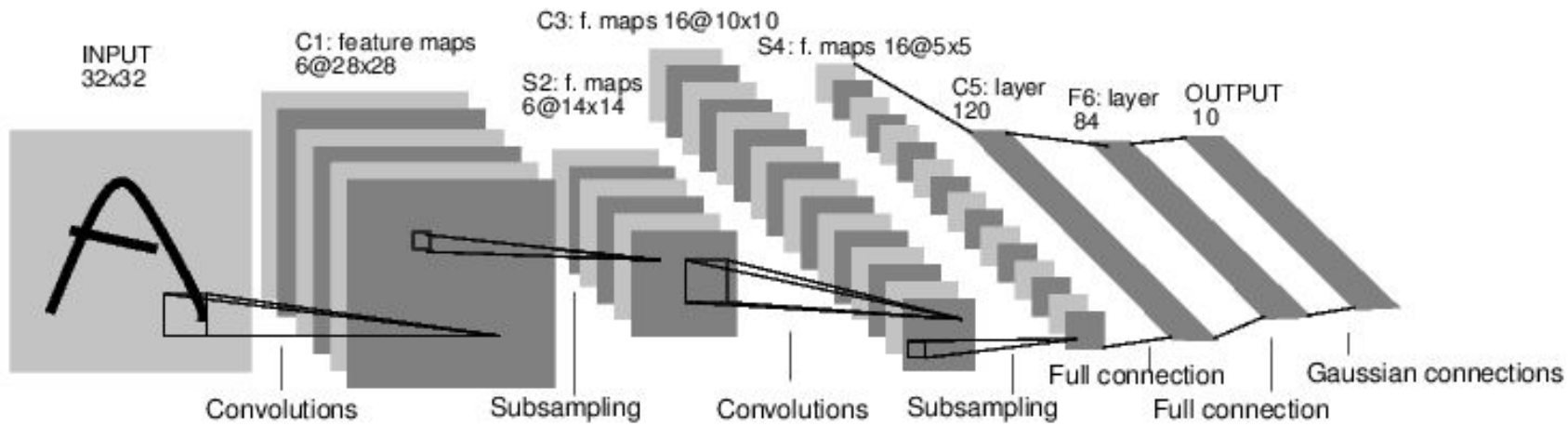
```
dropout_p = 0: tensor([[ 0., 1., 2., 3., 4., 5., 6., 7.],
[ 8., 9., 10., 11., 12., 13., 14., 15.]])
dropout_p = 0.5: tensor([[ 0., 2., 4., 0., 8., 0., 0.,
    0.],
       [16., 18., 0., 0., 24., 26., 28., 0.]])
dropout_p = 1: tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])
```

Introduction and History

1. Convolution as a specialized type of linear operation:
   - Sparse interaction
   - Parameter sharing
2. Convolutional neural networks (CNN) are simply neural networks that use convolution, instead of general matrix multiplication, in at least one of its layers.

Introduction and History

1. CNN has been introduced decades ago (1980's) by Lecun:
   - First applied on classifying handwritten digits, obtained state-of-the-art performance.
   - Fast development due to the success of computational power acceleration and neural network training algorithms.



Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

Convolutional Layers[1]

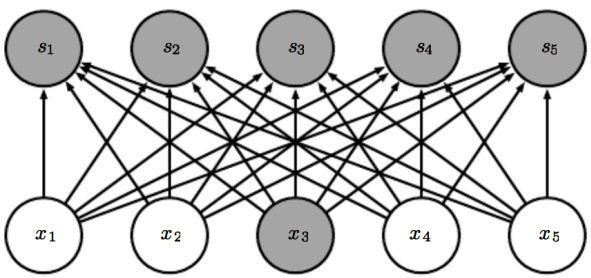
[1] Partially adapted from
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf

Convolutional Neural Networks

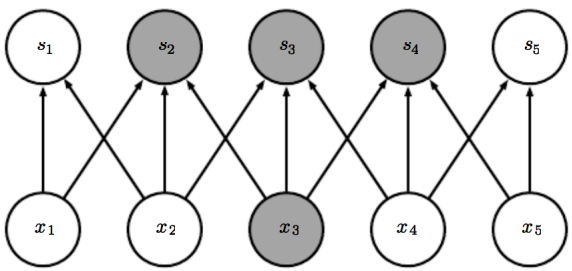
1. Scale up neural networks to process very large images/video sequences.
   - Sparse connections.
   - Parameter sharing.
2. Automatically generalize across spatial translations of inputs.
3. Applicable to any input that is laid out on a grid:
   - 1-D: signals
   - 2-D: gray images
   - 3-D: gray videos
   - ...

Key Idea

1. Replace matrix multiplication in neural nets with convolution:
   - MLP uses matrix multiplication: with m inputs and n outputs, $m \times n$ parameters and $O(m \times n)$ runtime per example.
   - CNN uses convolutional with sparse interactions and parameter sharing: with k connections for each input, $k \times n$ parameters and $O(k \times n)$ runtime per example.



MLP



CNN

2. Everything else stays the same:
   - Maximum likelihood.
   - Back-propagation.
   - ...

Convolution on Continuous Domains

1-D convolution:

For two functions $f : \mathbb{R} \mapsto \mathbb{R}$ and $g : \mathbb{R} \mapsto \mathbb{R}$

$$(f * g)(t) \triangleq \int f(\tau) g(t - \tau) \mathrm{d}\tau$$

Extension to 2-D convolution:

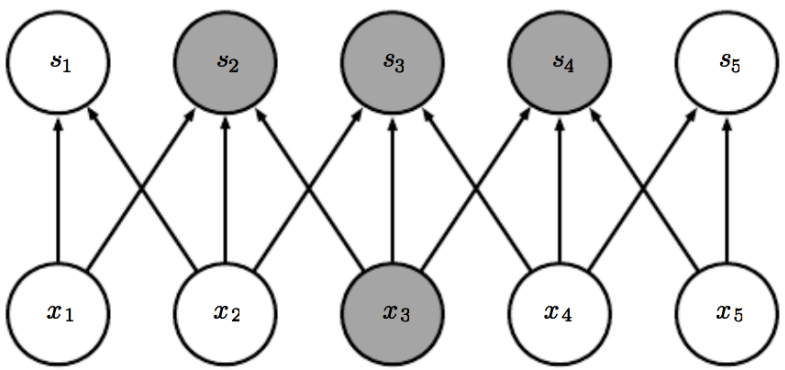
For two functions $f : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R} \times \mathbb{R}$ and $g : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R} \times \mathbb{R}$

$$(f * g)(t_1, t_2) \triangleq \int \int f(\tau_1, \tau_2) g(t_1 - \tau_1, t_2 - \tau_2) \mathrm{d}\tau_1 \mathrm{d}\tau_2$$

Convolution with Discrete Variables

1. Real data is usually presented in discrete domain.
2. Let **x** be a 1-dimensional input, **w** be a 1-dimensional kernel (filter), the output **s** is defined via the 1-D convolution:

$$\mathbf{s}(t) \triangleq (\mathbf{x} * \mathbf{w})(t) = \sum_{i=-\infty}^{\infty} \mathbf{x}(i)\,\mathbf{w}(t-i)$$



3. In ML applications,input is a multidimensional array of data, and the kernel is a multidimensional array of parameters to be learned.

Two-dimensional Convolution

- If we use a 2-D image $\mathbf{I}$ as input and use a 2-D kernel $\mathbf{K}$, we have[2]

$$\mathbf{S}(i,j) \triangleq (\mathbf{I} * \mathbf{K})(i,j) = \sum_m \sum_n \mathbf{I}(m,n)\,\mathbf{K}(i-m, j-n)$$

0	1	3
1	2	2
0	0	0

$*$

0	**1**	**2**
2	2	0
0	1	2

$=$?

[2] Take the out-of-bound elements to be zero.

Two-dimensional Convolution

- If we use a 2-D image $\mathbf{I}$ as input and use a 2-D kernel $\mathbf{K}$, we have[2]

$$\mathbf{S}(i,j) \triangleq (\mathbf{I} * \mathbf{K})(i,j) = \sum_m \sum_n \mathbf{I}(m,n)\,\mathbf{K}(i-m, j-n)$$

0	1	3
1	2	2
0	0	0

$*$

0	1	2
2	2	0
0	1	2

$=$

0	0	1
0	3	10
2	6	9

[2] Take the out-of-bound elements to be zero.

Commutativity of Convolution

1. Convolution is commutative:

$$\mathbf{S}(i,j) = \sum_m \sum_n \mathbf{I}(m,n)\,\mathbf{K}(i-m, j-n) = \sum_m \sum_n \mathbf{I}(i-m, j-n)\,\mathbf{K}(m,n)$$

2. Commutativity arises because we have flipped the kernel relative to the input:
   - As m increases, index to the input increases, but index to the kernel decreases.

Cross-Correlation

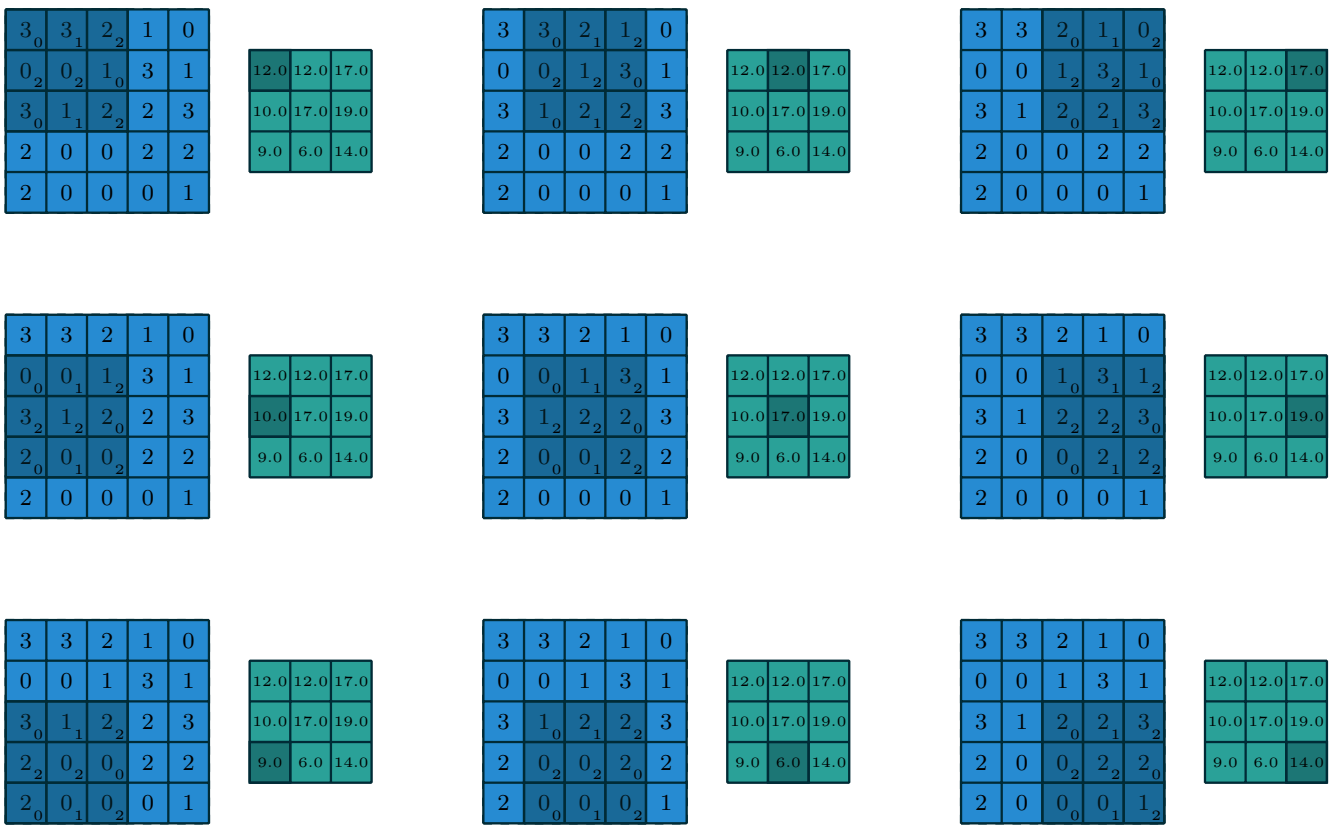
1. Cross-Correlation: same as convolution, but without flipping the kernel

$$\mathbf{S}(i,j) = \sum_m \sum_n \mathbf{I}(i+m, j+n)\, \mathbf{K}(m,n)$$

2. Both referred to as convolution, whether kernel is flipped or not.
3. In ML, cross-correlation is preferable because of computational convenience.
4. Generally, the two representations are equivalent if the kernel is learned.

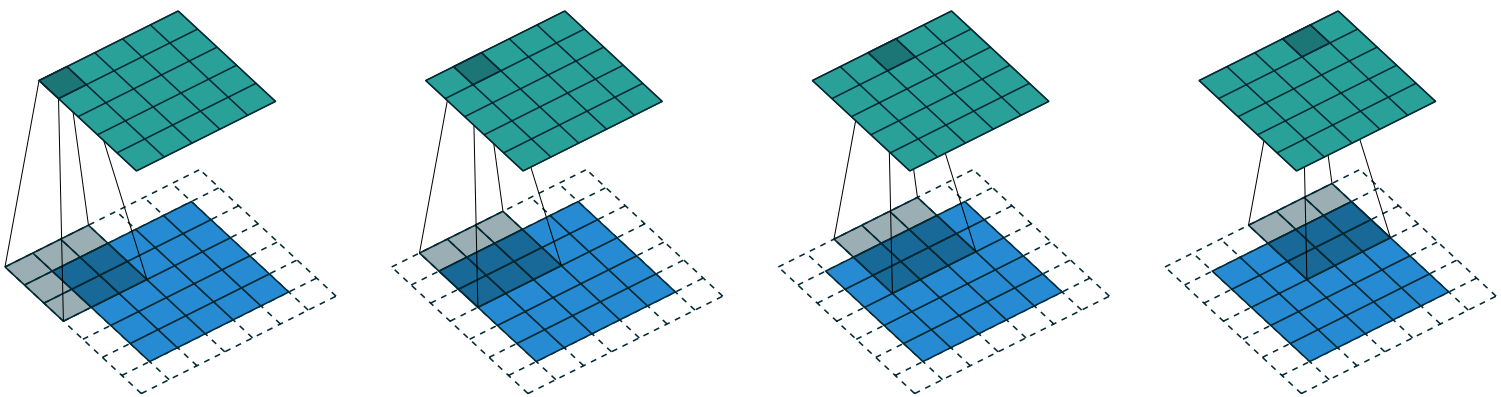
Cross-Correlation Convolution Examples

$$\mathbf{S}(i,j) = \sum_m \sum_n \mathbf{I}(i+m, j+n)\,\mathbf{K}(m,n)$$



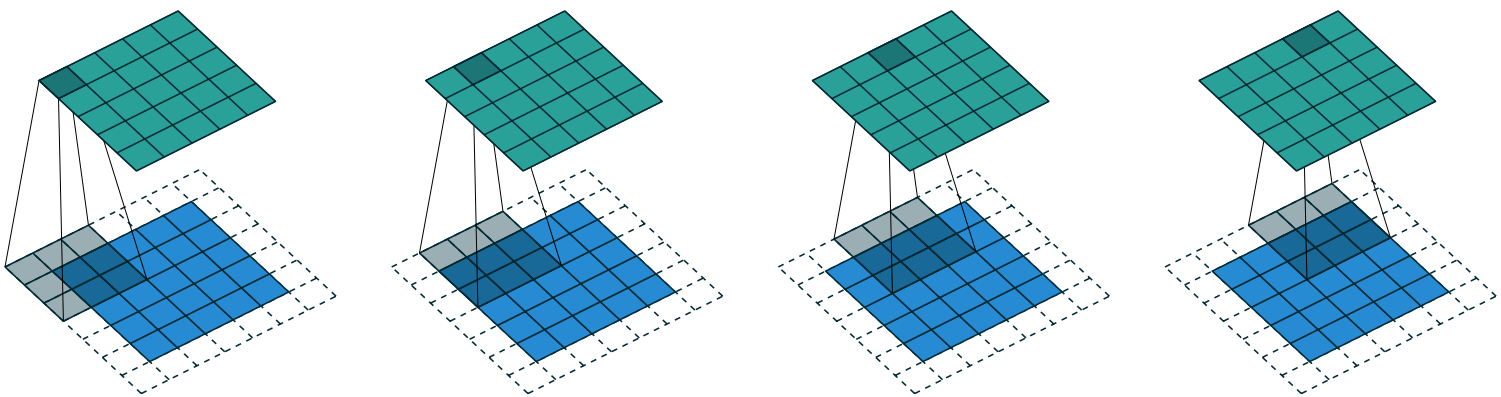
Padding

- Augment the input with zeros on the boundary.
- Padding can make the size of output equal to the size of input.



Padding

- Augment the input with zeros on the boundary.
- Padding can make the size of output equal to the size of input.



General rules for the output sizes and padding? $\Rightarrow$ later

Convolution with Strides

- Convolution with skipped indexes.
- Convolution with stride can be viewed as a way of doing dimensional reduction.

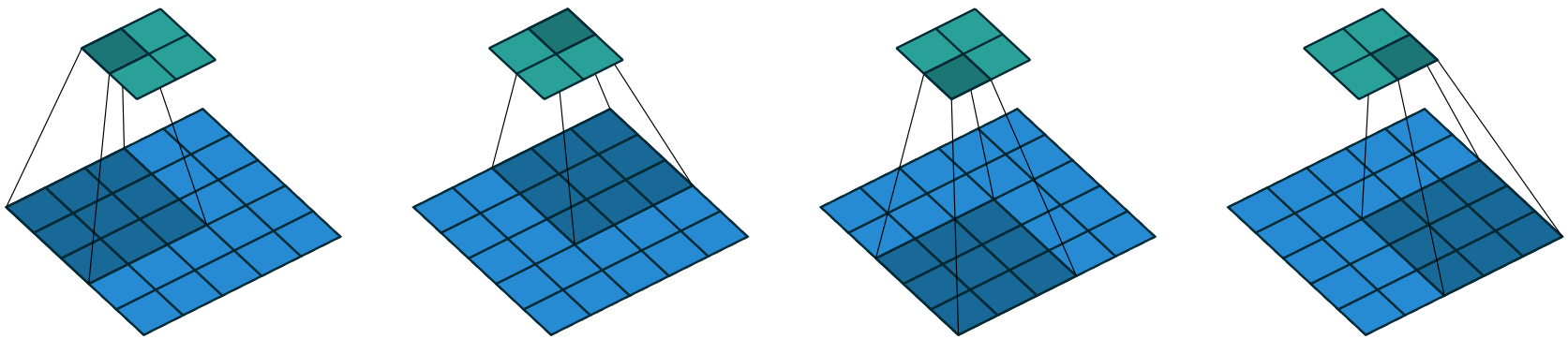


Figure: Convolution with stride of size 2×2 (sometimes simply say "stride 2"). 5×5 input, 2×2 output.

Convolution with Strides

- Convolution with skipped indexes.
- Convolution with stride can be viewed as a way of doing dimensional reduction.

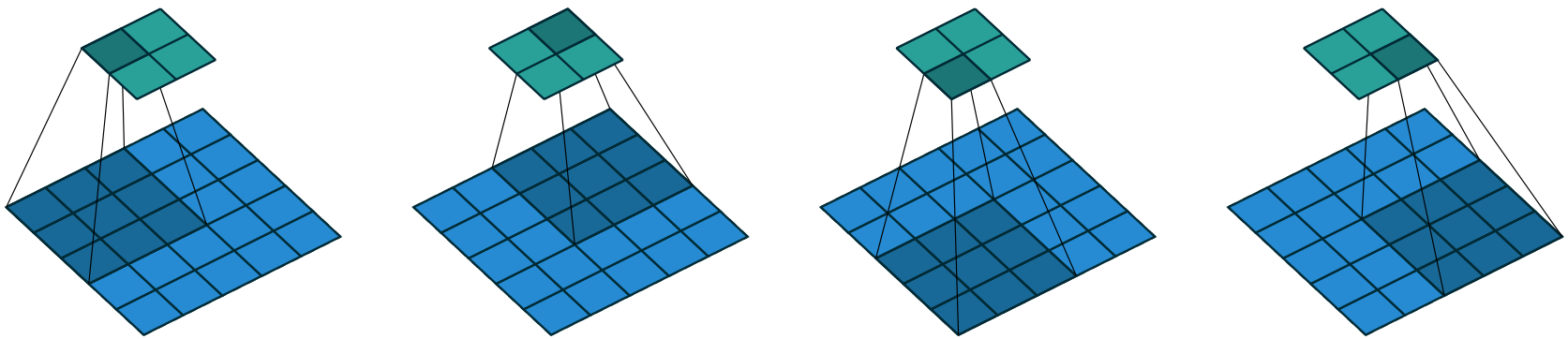


Figure: Convolution with stride of size 2×2 (sometimes simply say "stride 2"). 5×5 input, 2×2 output.

Relation of output size w.r.t. stride?

Convolution with Strides

- Convolution with skipped indexes.
- Convolution with stride can be viewed as a way of doing dimensional reduction.

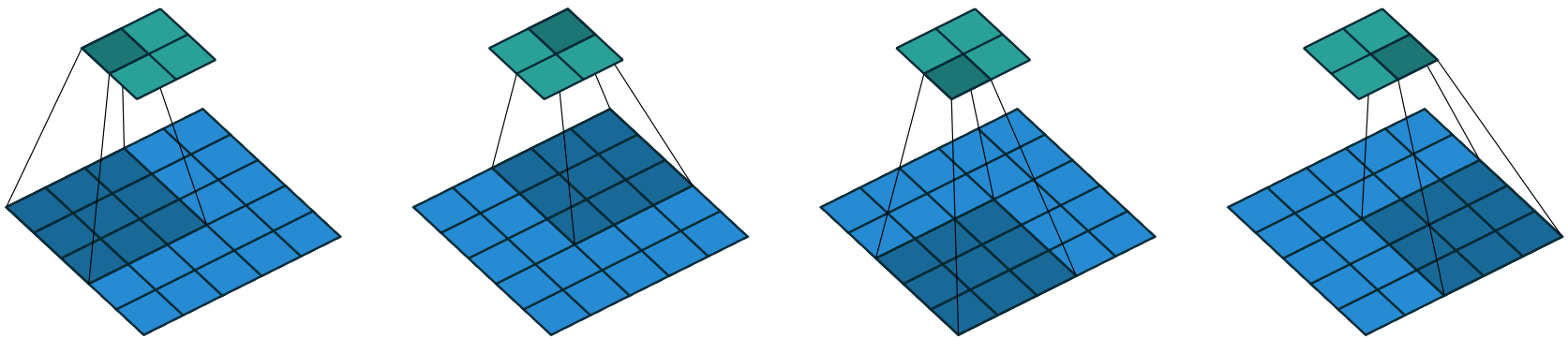
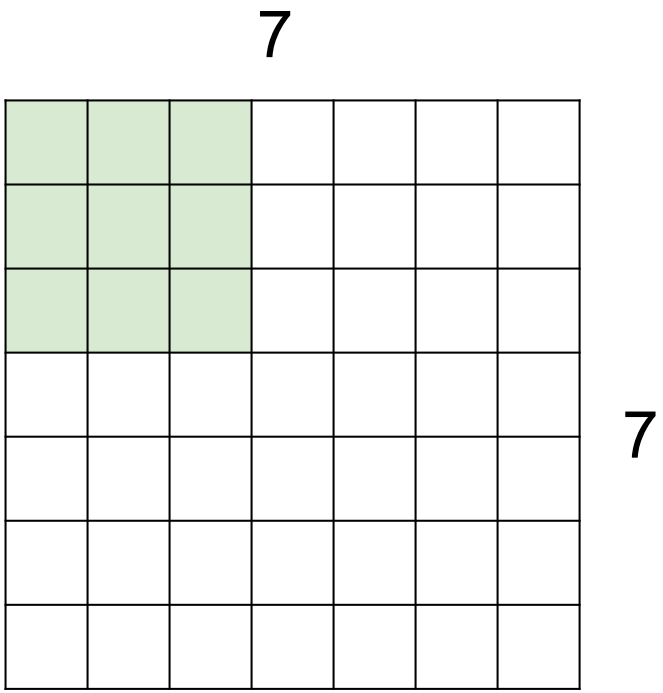


Figure: Convolution with stride of size 2×2 (sometimes simply say "stride 2"). 5×5 input, 2×2 output.

Relation of output size w.r.t. stride

$$\text{output size} = (\text{input size - filter size}) \; / \; \text{stride} + 1$$

A Closer Look at Strides

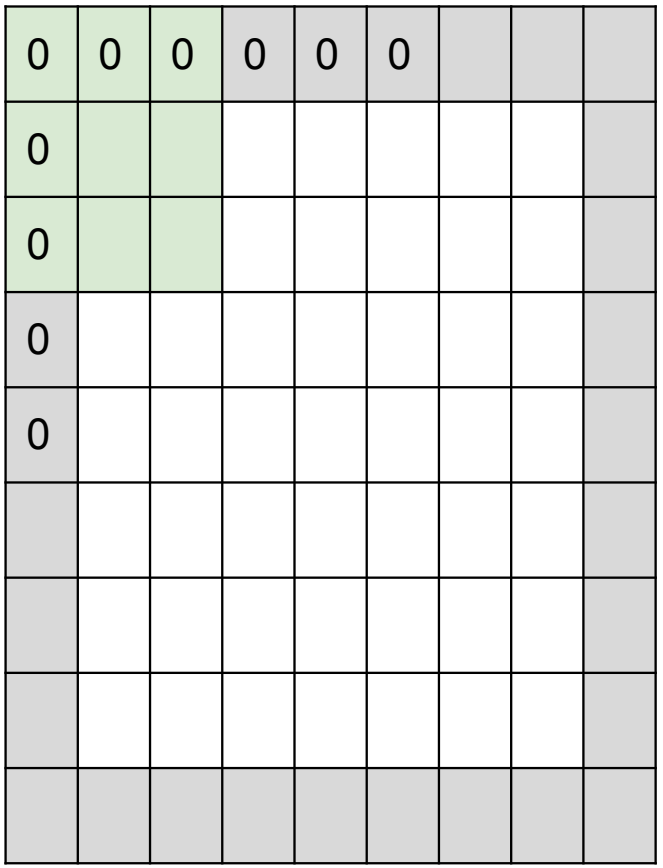


7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.

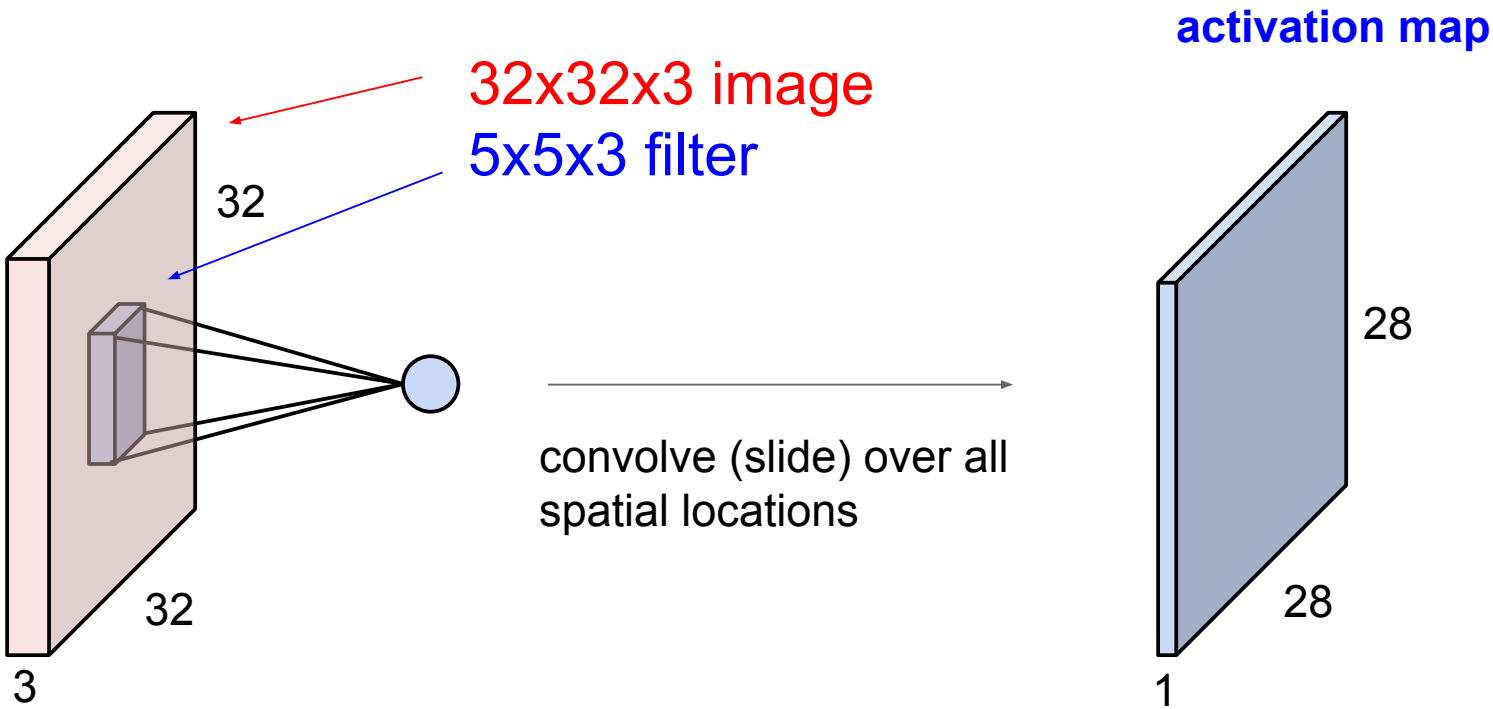
Practical Setting

- In practice, common to set convolutional layers with stride 1, filters of size $F \times F$, and zero-padding with $(F-1)/2$:
  - will preserve input size



Extension to n-dimensional tensors

Convolution Layer

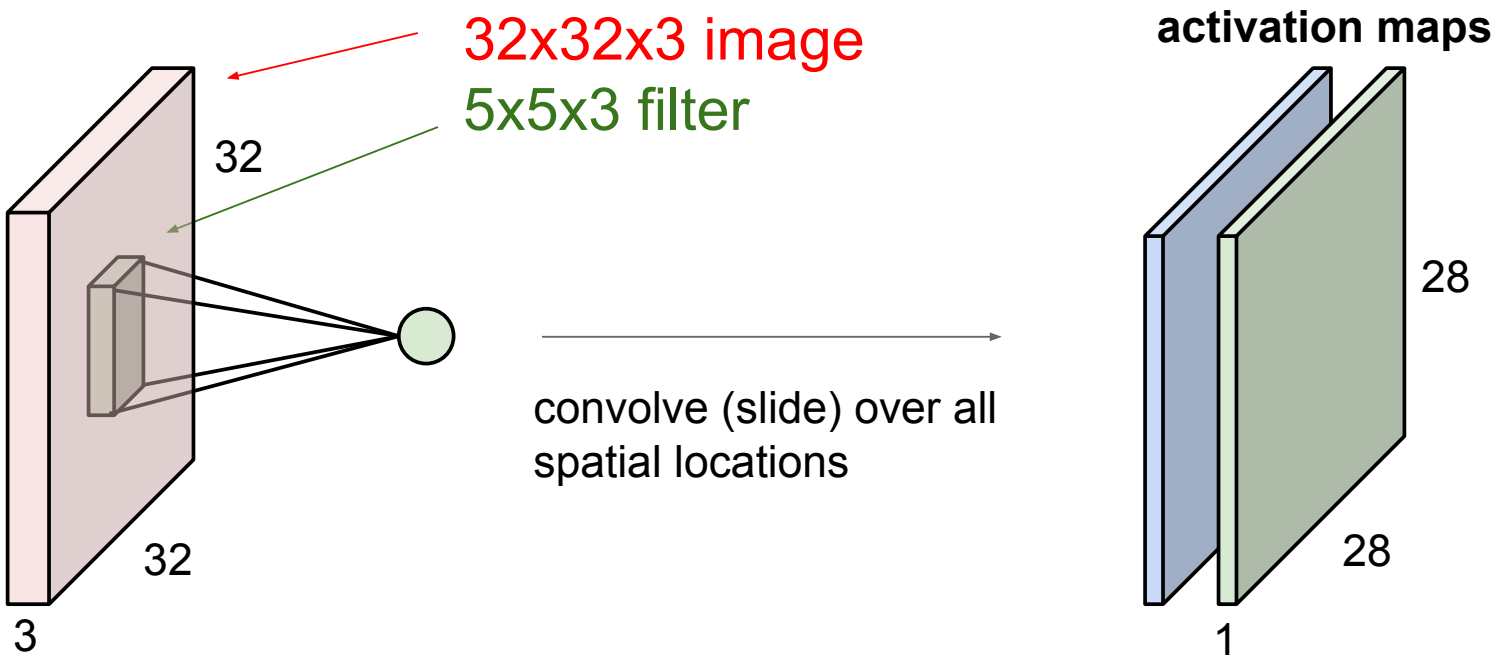


- Convolution and sum over the third dimension.

Extension to *n*-dimensional tensors

Convolution Layer

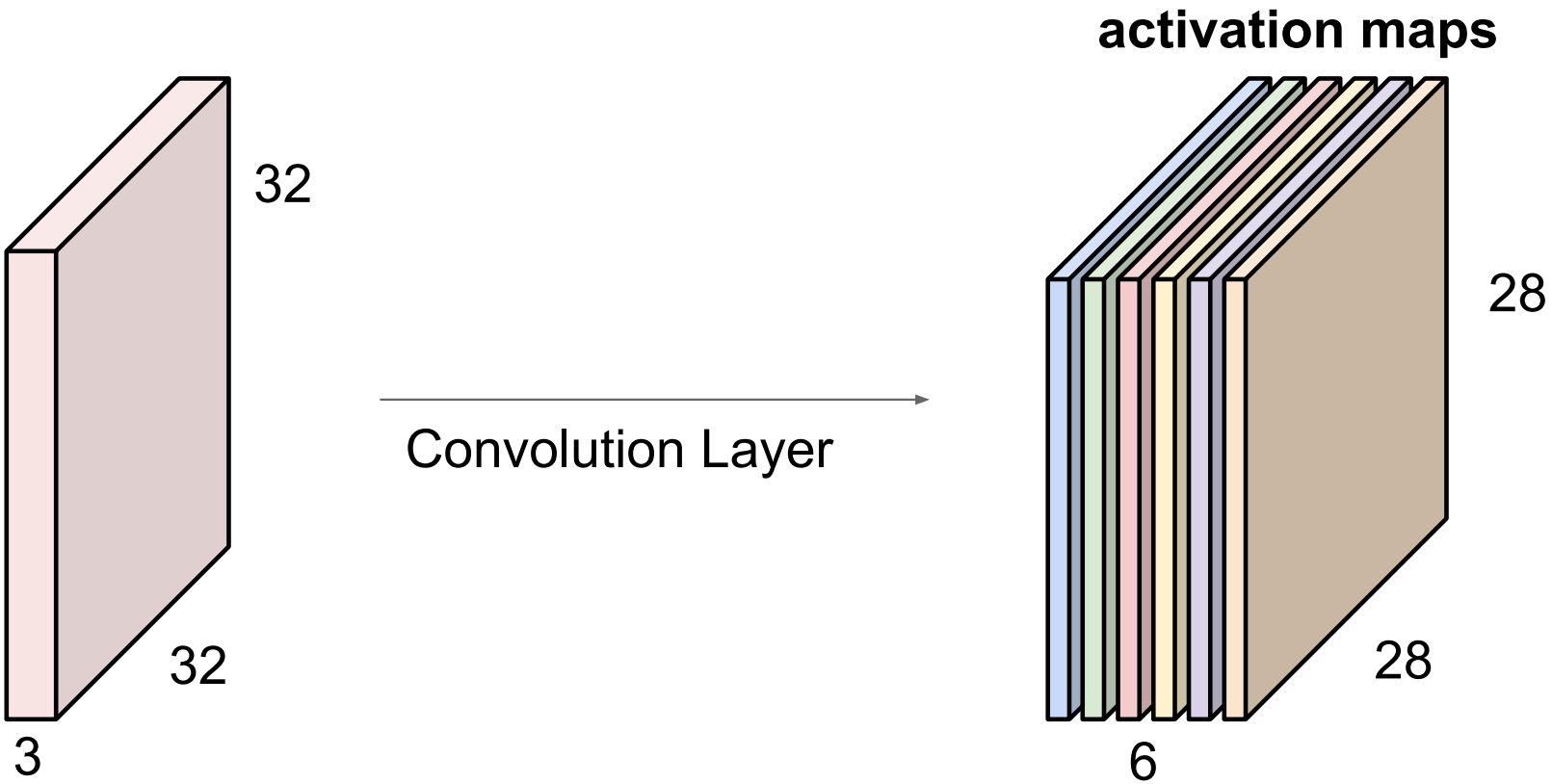
consider a second, green filter



- Convolution and sum over the third dimension.

Extension to *n*-dimensional tensors

If we had six $5 \times 5 \times 3$ filters[a], we'll get 6 separate activation maps.



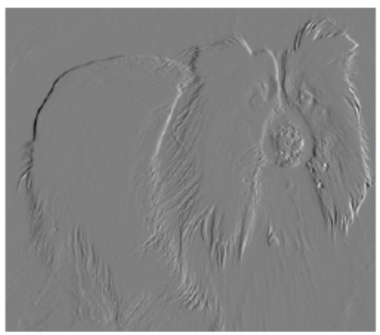
We stack these up to get a "new image" of size 28x28x6!

[a]We sometimes ignore the last dimension for simplicity, written as filter size of 5×5.

Example: Efficiency of Convolution for Edge Detection

1. Image on right formed by taking each pixel of input image and subtracting the value of its neighboring pixel on the left:
   - this is a measure of all the vertically oriented edges in input image, useful for object detection.
   - can be implemented as a convolution with a 1×2 kernel: $K = (1, -1)$.

Input image

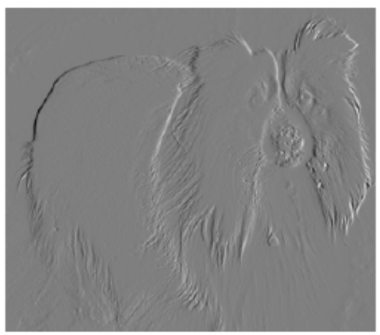


Both images are 280 pixels tall
Input image is 320 pixels wide
Output image is 319 pixels wide

Example: Efficiency of Convolution for Edge Detection

1. Image on right formed by taking each pixel of input image and subtracting the value of its neighboring pixel on the left:
   - this is a measure of all the vertically oriented edges in input image, useful for object detection.
   - can be implemented as a convolution with a 1×2 kernel: $K = (1, -1)$.

Input image



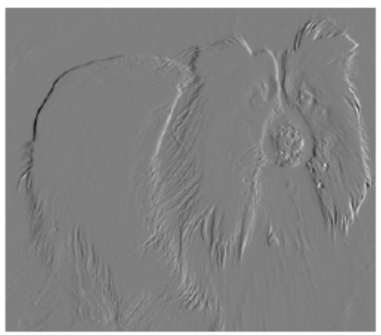
Both images are 280 pixels tall
Input image is 320 pixels wide
Output image is 319 pixels wide

2. Number of operations:
   - Convolution with one kernel of size 1×2:
   - MLP:

Example: Efficiency of Convolution for Edge Detection

1. Image on right formed by taking each pixel of input image and subtracting the value of its neighboring pixel on the left:
   - this is a measure of all the vertically oriented edges in input image, useful for object detection.
   - can be implemented as a convolution with a 1×2 kernel: $K = (1, -1)$.

Input image



Both images are 280 pixels tall
Input image is 320 pixels wide
Output image is 319 pixels wide

2. Number of operations:
   - Convolution with one kernel of size 1×2: $319 \times 280 \times 1 \times 2$.
   - MLP: $320 \times 280 \times 319 \times 280$.

Output Volumn Size w.r.t. Padding and Stride

- Input volume: 1×1, 5×5 filter with stride 1, pad 2.
- Output volume size?

Output Volumn Size w.r.t. Padding and Stride

- Input volume: 1×1, 5×5 filter with stride 1, pad 2.
- Output volume size?
  - 1×1

Output Volumn Size w.r.t. Padding and Stride

- Input volume: 2×2, 5×5 filter with stride 1, pad 2.
- Output volume size?

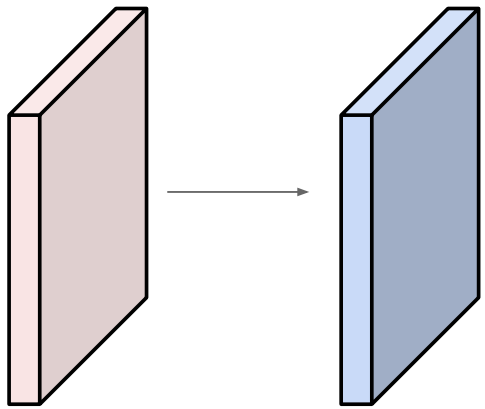
Output Volumn Size w.r.t. Padding and Stride

- Input volume: 2×2, 5×5 filter with stride 1, pad 2.
- Output volume size?
  - 2×2

Output Volumn Size w.r.t. Padding and Stride

Examples

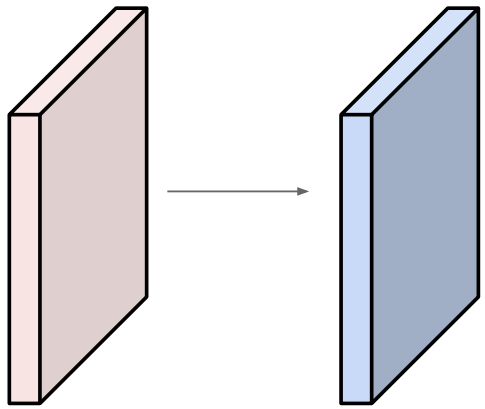
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2



Output volume size: ?

Output Volumn Size w.r.t. Padding and Stride

Examples

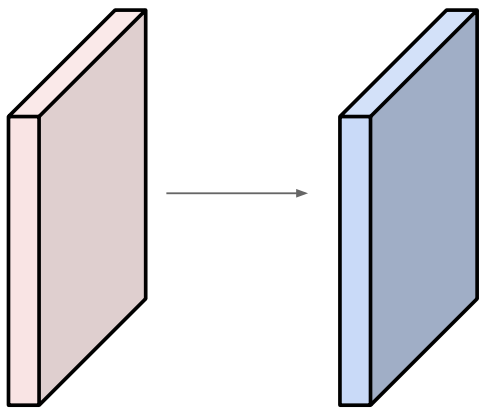


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

Output Volumn Size w.r.t. Padding and Stride

Examples

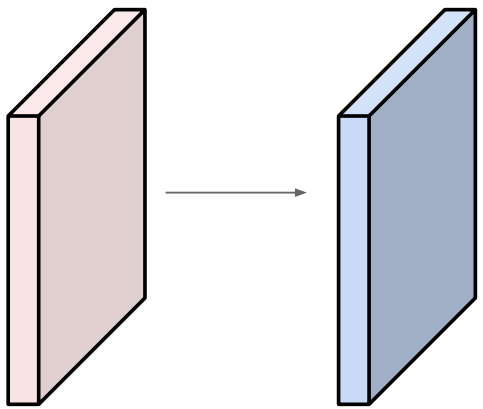


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

Output Volumn Size w.r.t. Padding and Stride

Examples



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Summary

To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$.
- Requires four hyperparameters:
  - Number of filters K.
  - Filter size F.
  - Stride S.
  - Amount of zero-padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$
- With parameter sharing, it introduces $F^2 D_1$ weights per filter, for a total of $F^2 D_1 K$ weights and K biases.
- In the output volume, the d-th depth slices (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input with a stride of S, and then offset by d-th bias.

Summary

To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$.
- Requires four hyperparameters:
  - Number of filters K.
  - Filter size F.
  - Stride S.
  - Amount of zero-padding P.

Common settings:

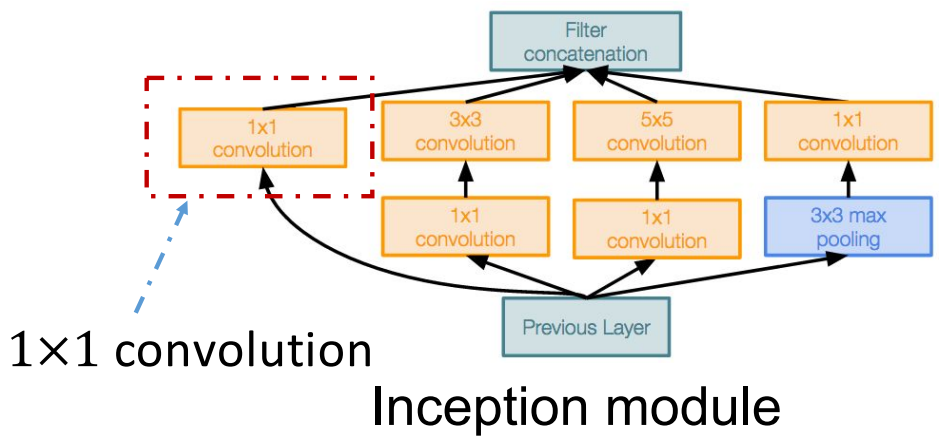
K = (powers of 2, e.g. 32, 64, 128, 512)
- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

The Power of 1×1 Convolution[3]

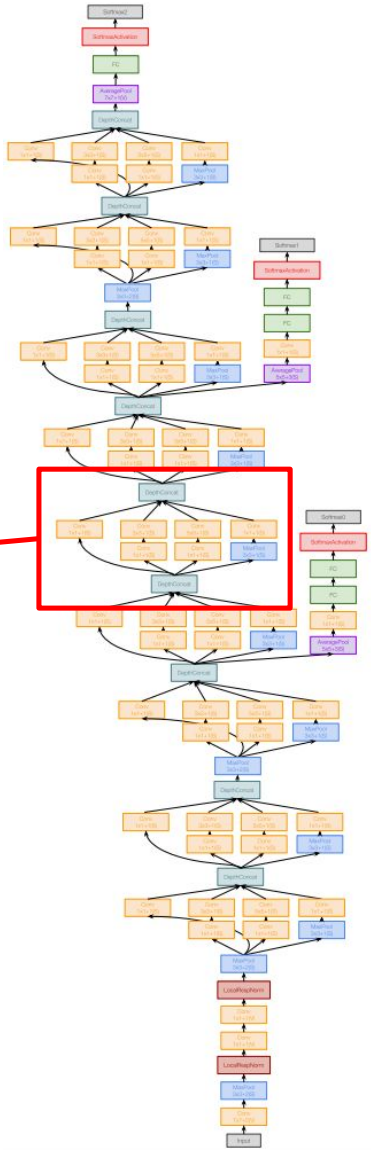
[3] Partially adapted from
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

GoogleNet

Stack the module on top of each other!



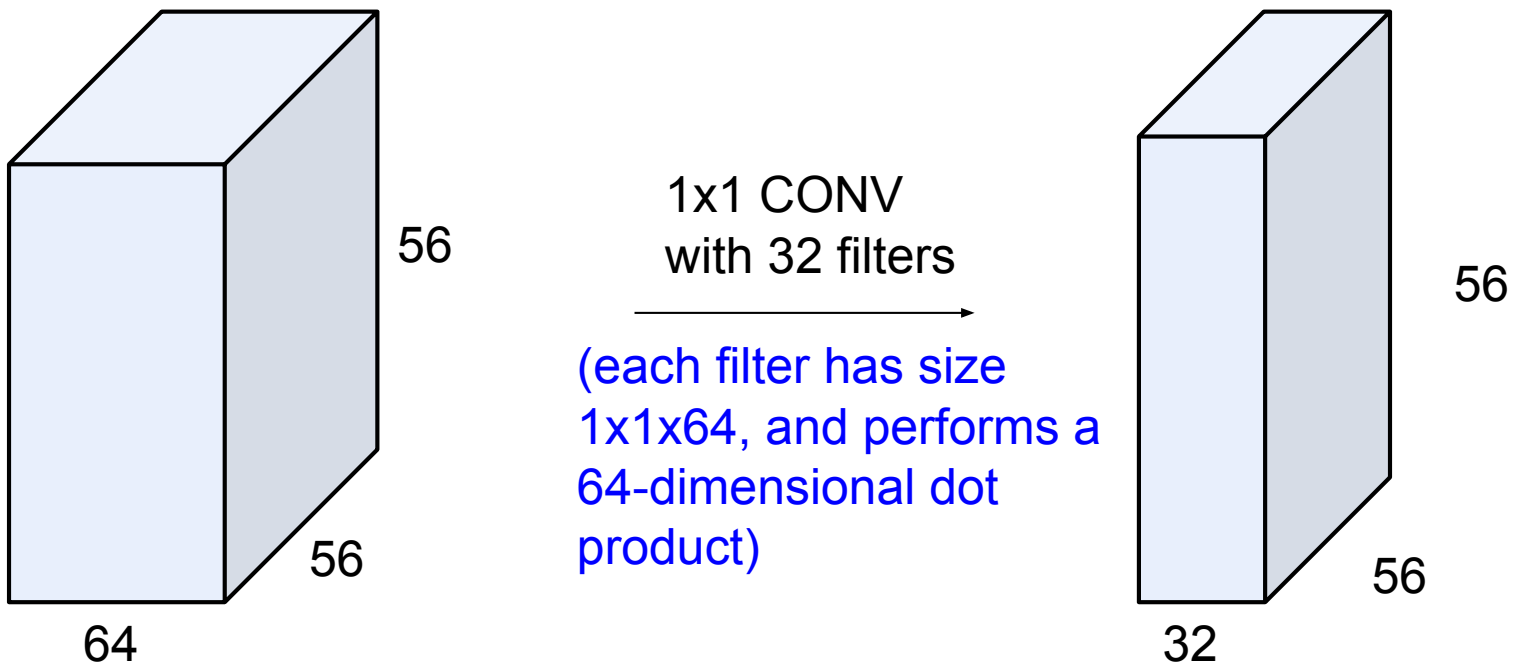
Inception module



1x1 Convolution Layers Make Perfect Sense

Used in GoogleNet (inception model)

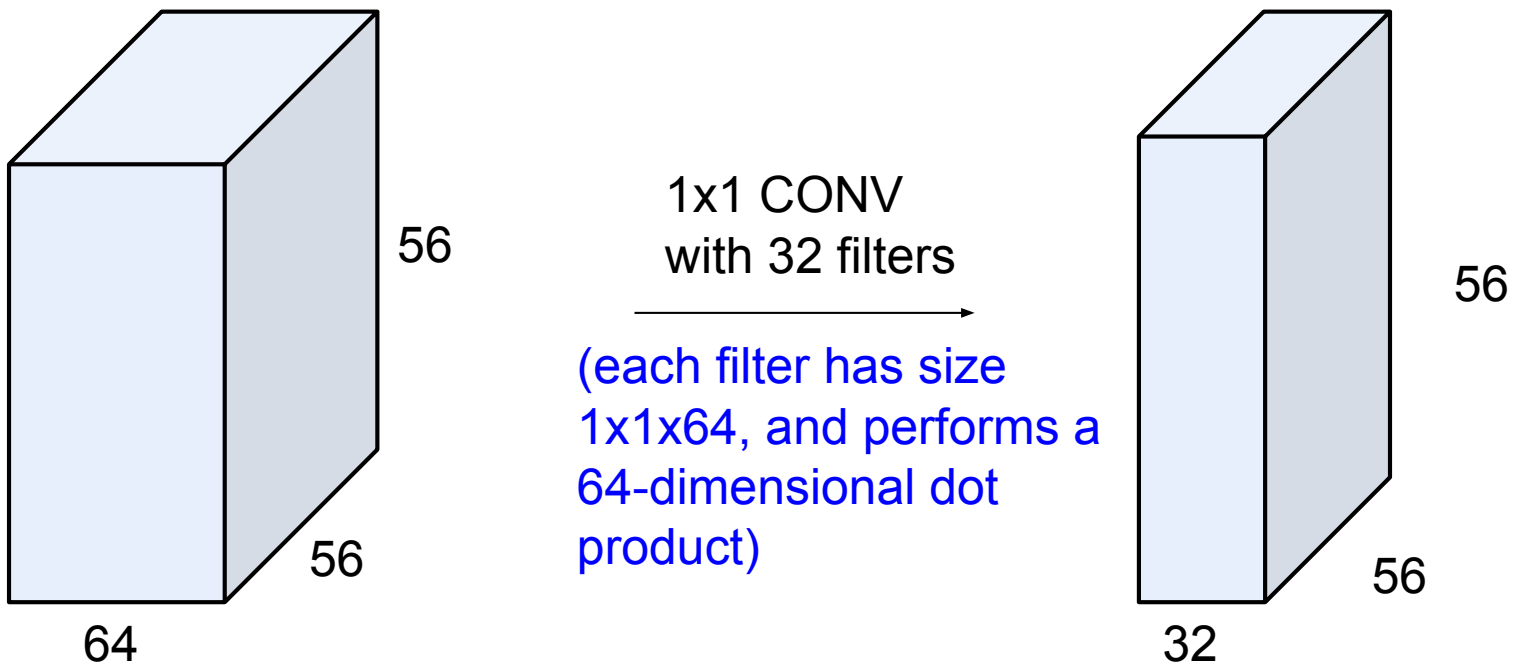
Perform like dimension reduction/extension on the third dimension.



1x1 Convolution Layers Make Perfect Sense

Used in GoogleNet (inception model)

Perform like dimension reduction/extension on the third dimension.



What happens with the 1×1 convolution here?

The Power of Small Filters

Why do we need small filters?

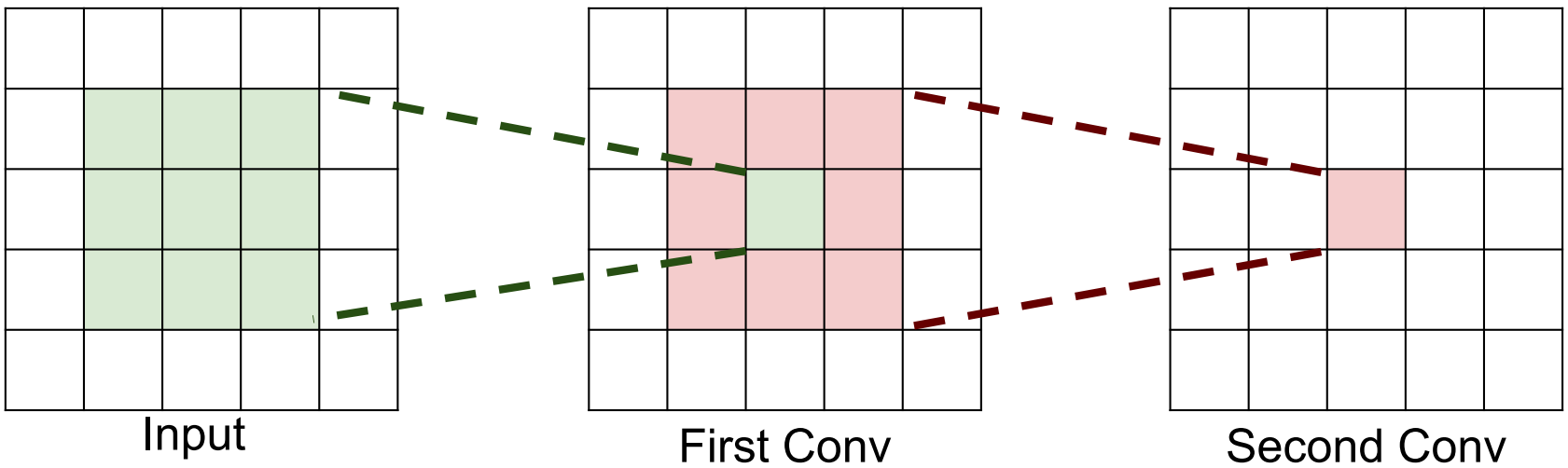
Proof Idea

- Define two kinds of CNNs which have different filter sizes such that they have the same representation power.
- Show the CNN with smaller filter size has less parameters.

The Power of Small Filters

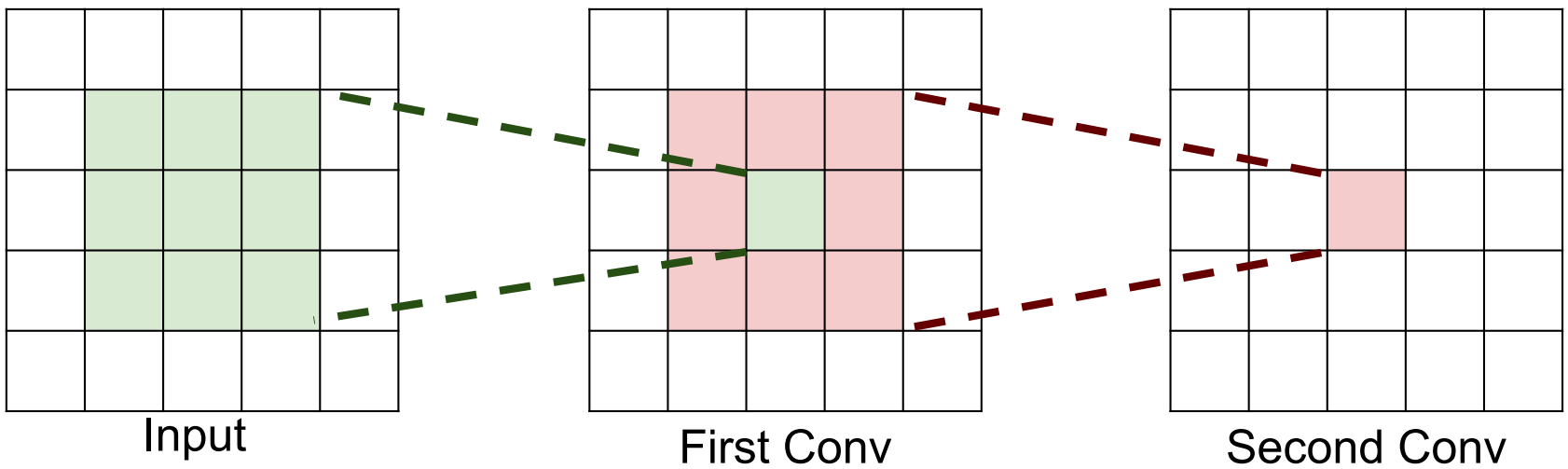
Suppose we stack two 3x3 (filter size) conv layers (stride 1):

- each neuron sees 3x3 region of previous activation map.



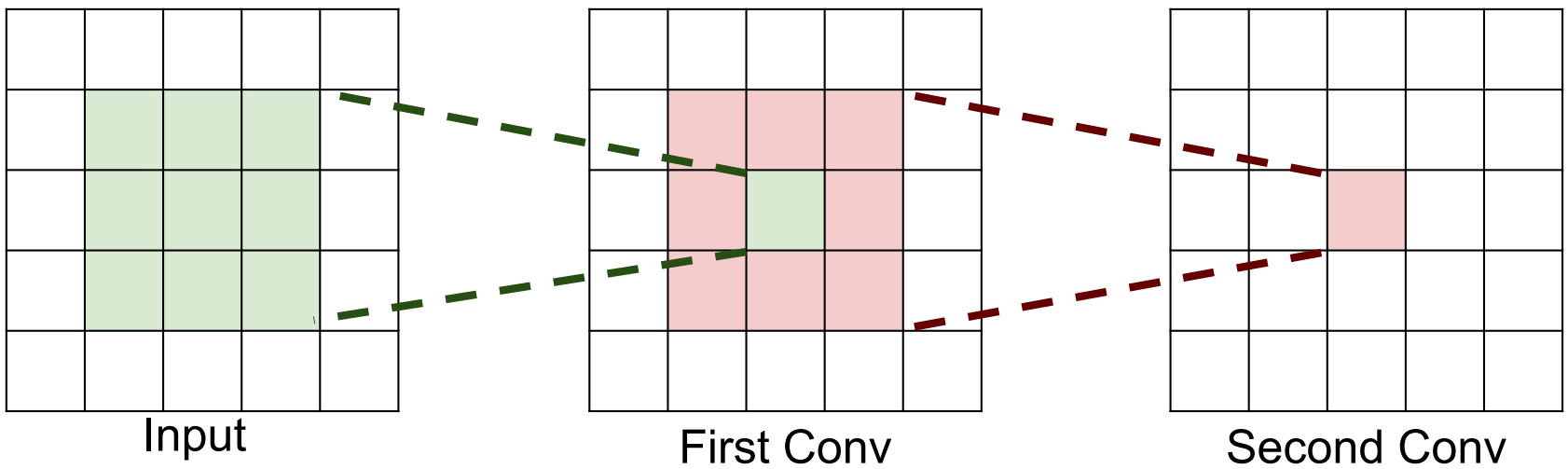
The Power of Small Filters

Question: How big of a region in the input does a neuron on the second conv layer see?



The Power of Small Filters

Question: How big of a region in the input does a neuron on the second conv layer see?
Answer: 5×5

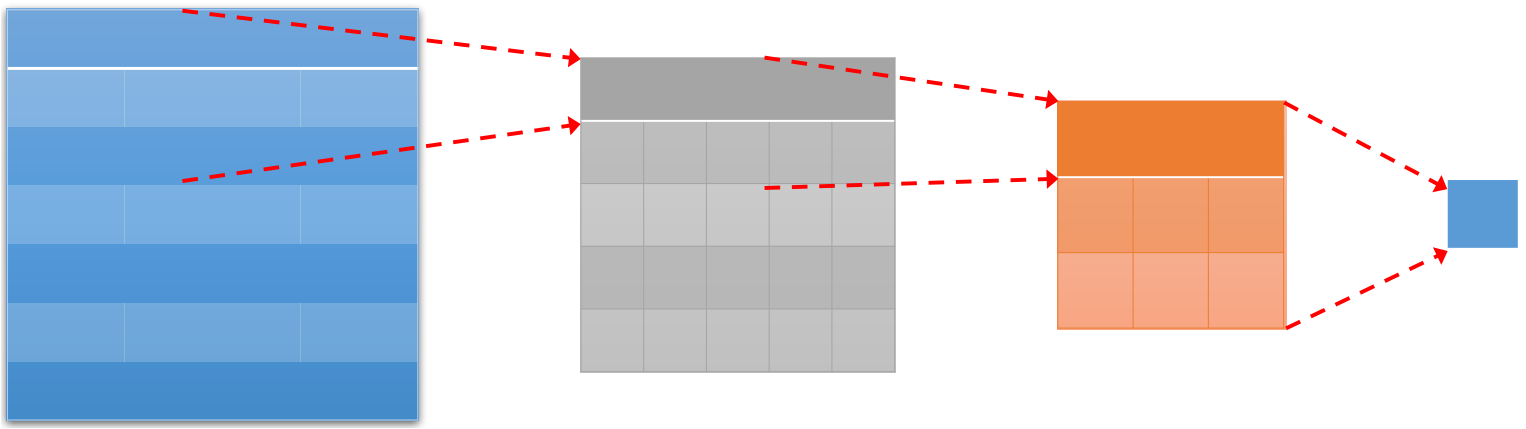


The Power of Small Filters

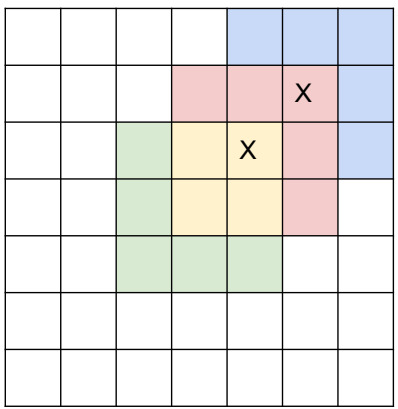
Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?

The Power of Small Filters

Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?



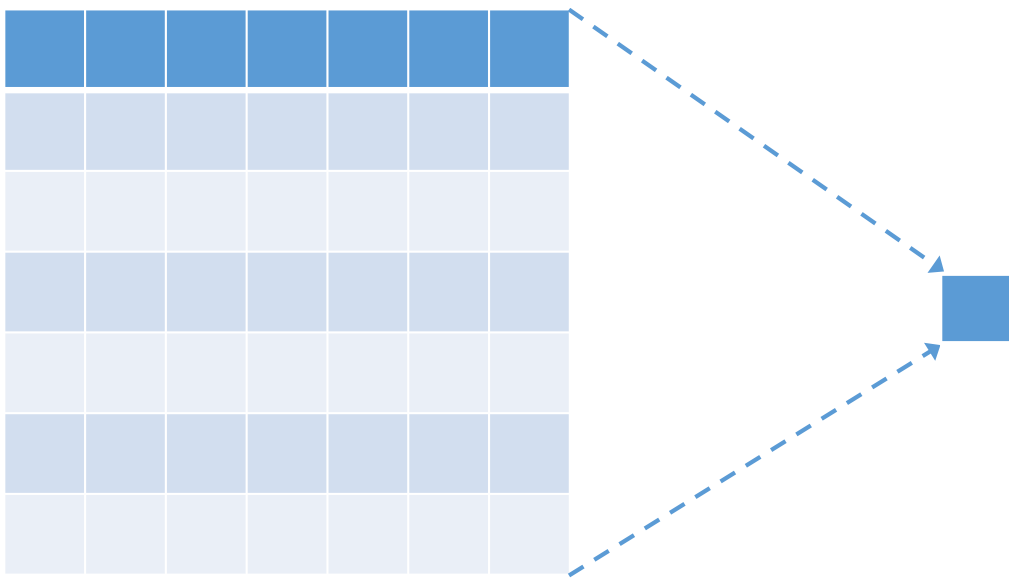
Answer: 7 x 7



The Power of Small Filters

Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?

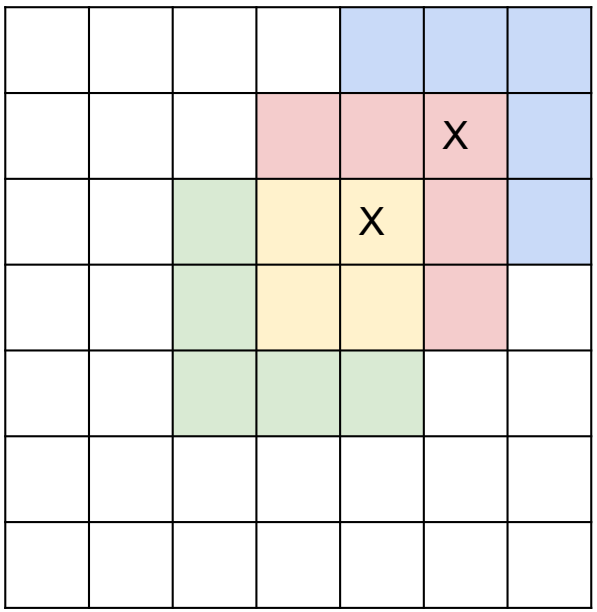
- If we use 1 convolution layer, we need to use a kernel of 7×7 in order to let a neuron in the output layer see a 7×7 region in the input.



The Power of Small Filters

Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?

Answer: 7 x 7



Three 3 x 3 conv gives similar representational power as a single 7 x 7 convolution

While it has less parameters!

The Power of Small Filters

Suppose input is $H \times W \times C$ and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W).

- One CONV with 7×7 filters.
- Number of weights:

- Three CONV with 3×3 filters.
- Number of weights:

The Power of Small Filters

Suppose input is $H \times W \times C$ and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W).

- One CONV with 7×7 filters.
- Number of weights:
  - $= C \times (7 \times 7 \times C) = 49C^2$
  -

- Three CONV with 3 x 3 filters.
- Number of weights:
  - $= 3 \times C \times (3 \times 3 \times C) = 27C^2$
  - fewer parameters = GOOD

The Power of Small Filters

Suppose input is $H \times W \times C$ and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W).

- One CONV with 7×7 filters.
- Number of weights:
  - $= C \times (7 \times 7 \times C) = 49C^2$
  -
- Number of multiply-adds:

- Three CONV with 3 x 3 filters.
- Number of weights:
  - $= 3 \times C \times (3 \times 3 \times C) = 27C^2$
  - fewer parameters = GOOD
- Number of multiply-adds:

The Power of Small Filters

Suppose input is $H \times W \times C$ and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W).

- One CONV with 7×7 filters.
- Number of weights:
  - $= C \times (7 \times 7 \times C) = 49C^2$
  -
- Number of multiply-adds:
  - $= (H \times W \times C) \times (7 \times 7 \times C) = 49HWC^2$
  -

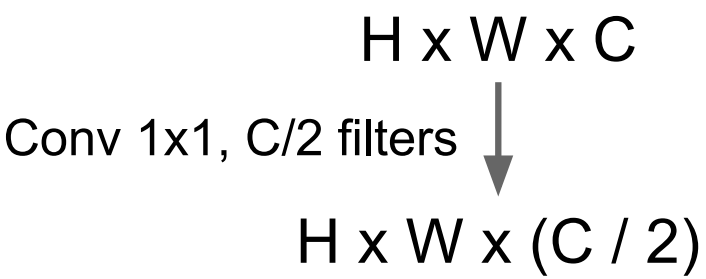
- Three CONV with 3 x 3 filters.
- Number of weights:
  - $= 3 \times C \times (3 \times 3 \times C) = 27C^2$
  - fewer parameters = GOOD
- Number of multiply-adds:
  - $= 3 \times (H \times W \times C) \times (3 \times 3 \times C) = 27HWC^2$
  - less compute = GOOD

The Power of Small Filters

Why stop at 3×3 filters? Why not try 1×1?

The Power of Small Filters

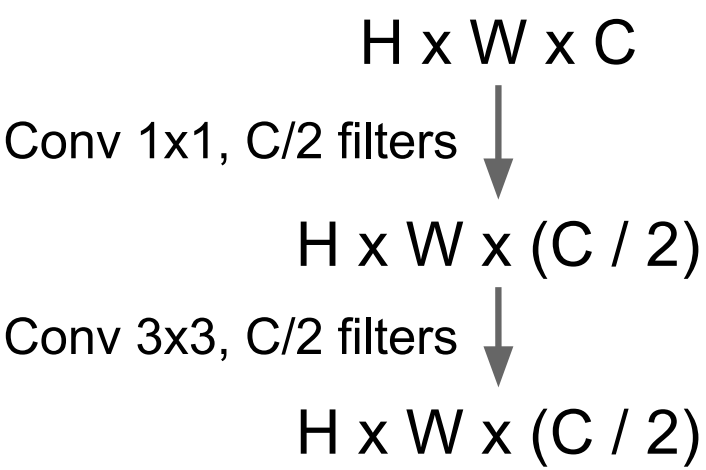
Why stop at 3×3 filters? Why not try 1×1?



1. "bottleneck" 1 x 1 conv to reduce dimension

The Power of Small Filters

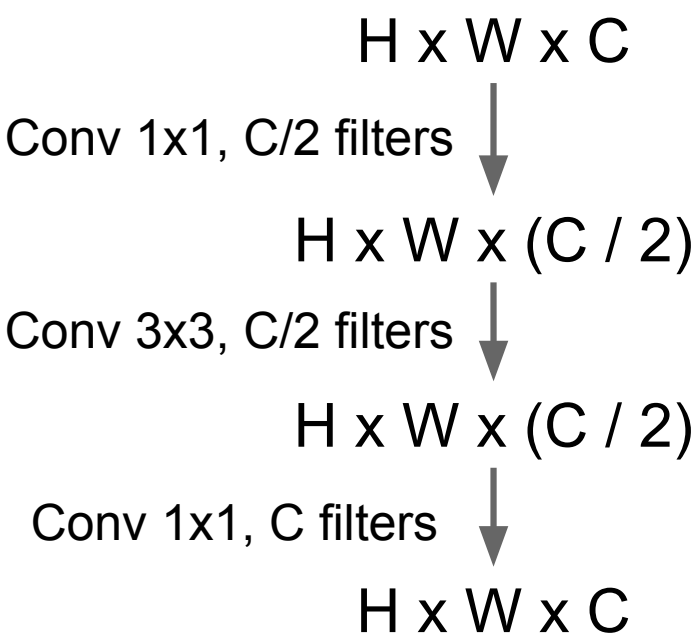
Why stop at 3×3 filters? Why not try 1×1?



1. "bottleneck" 1 x 1 conv to reduce dimension
2. 3 x 3 conv at reduced dimension

The Power of Small Filters

Why stop at 3×3 filters? Why not try 1×1?

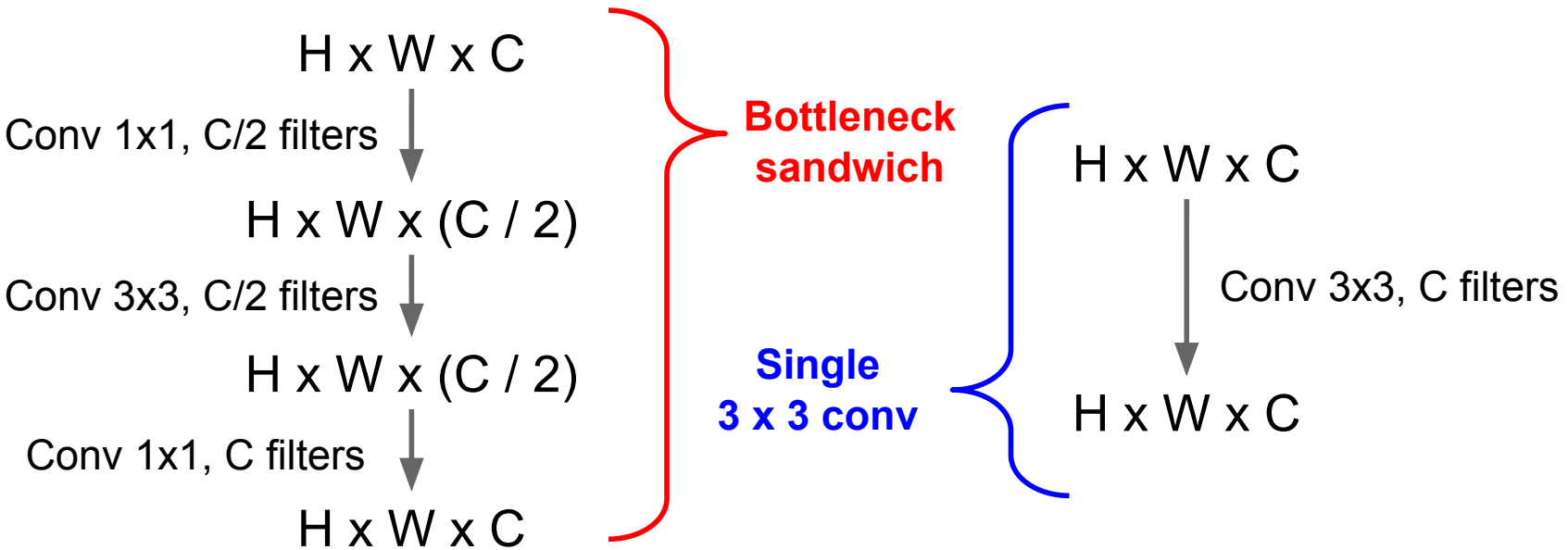


1. "bottleneck" 1 x 1 conv to reduce dimension
2. 3 x 3 conv at reduced dimension
3. Restore dimension with another 1 x 1 conv

[Seen in Lin et al, "Network in Network", GoogLeNet, ResNet]

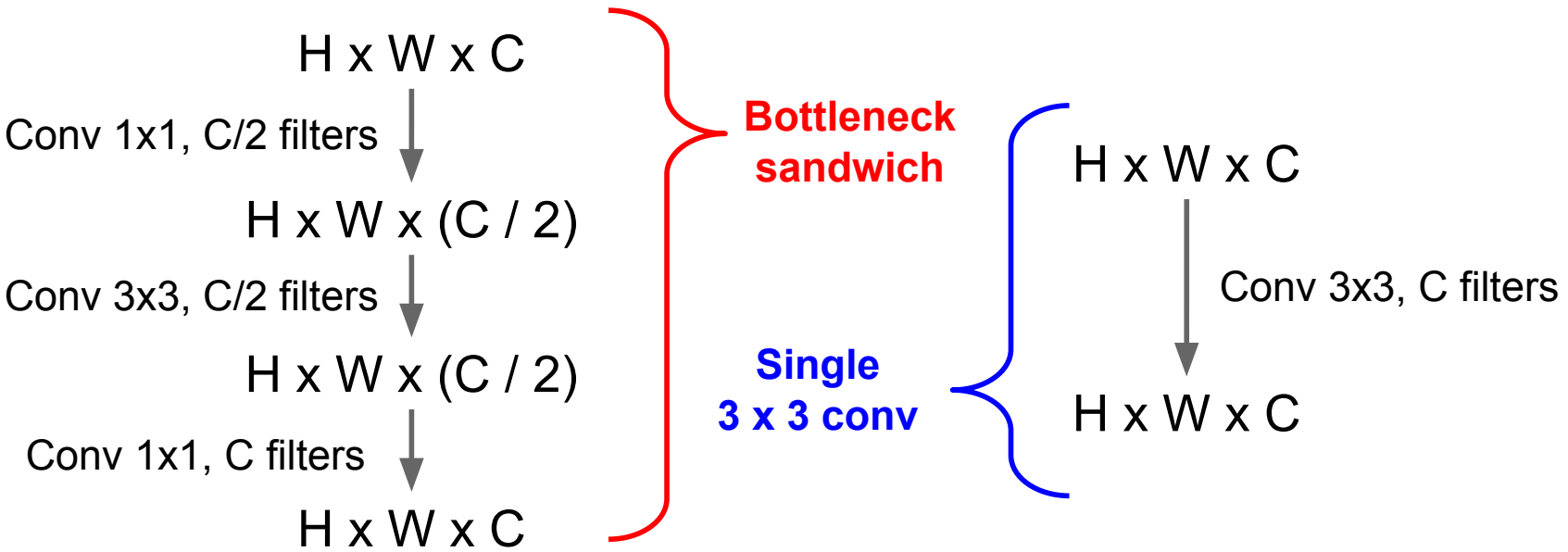
The Power of Small Filters

Why stop at 3×3 filters? Why not try 1×1?



The Power of Small Filters

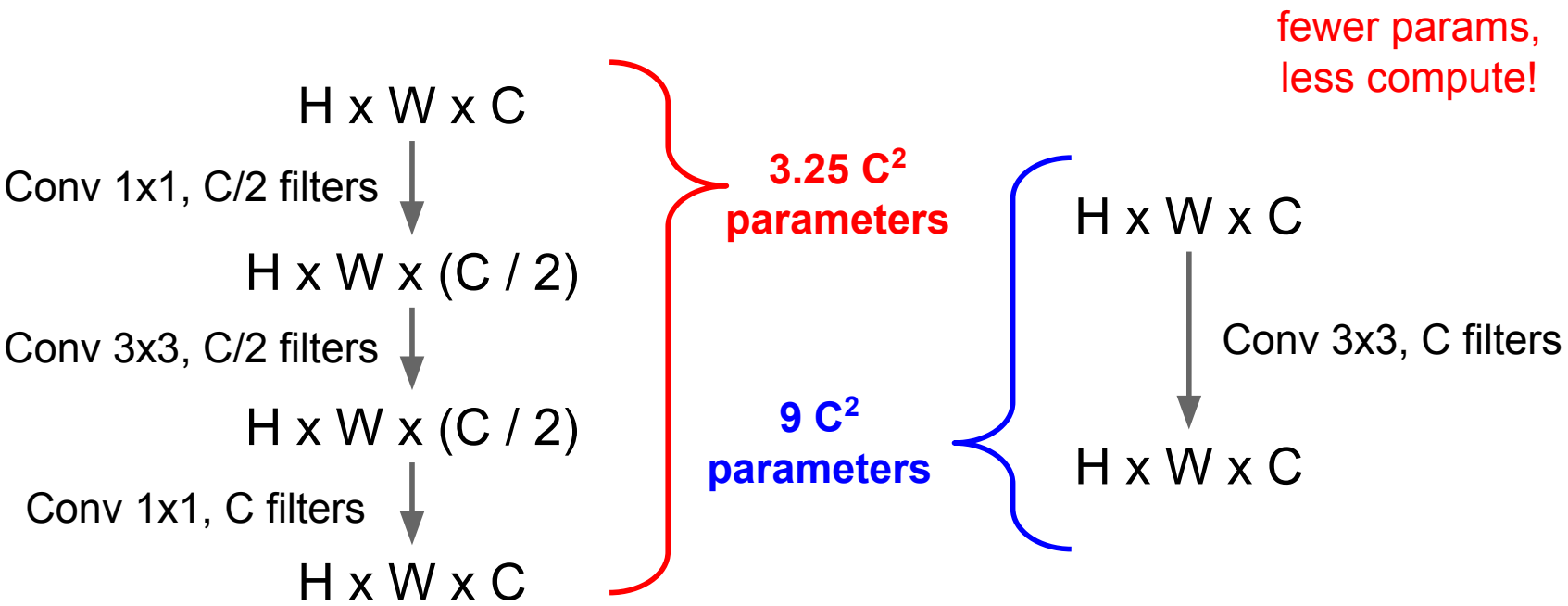
Why stop at 3×3 filters? Why not try 1×1?



How many parameters?

The Power of Small Filters

Why stop at 3×3 filters? Why not try 1×1?

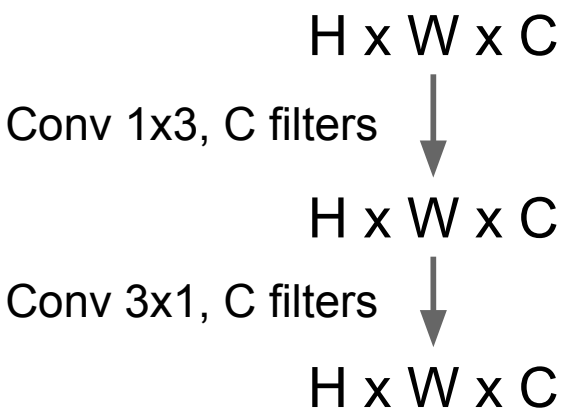


The Power of Small Filters

Still using 3×3 filters $\cdots$ can we break it up?

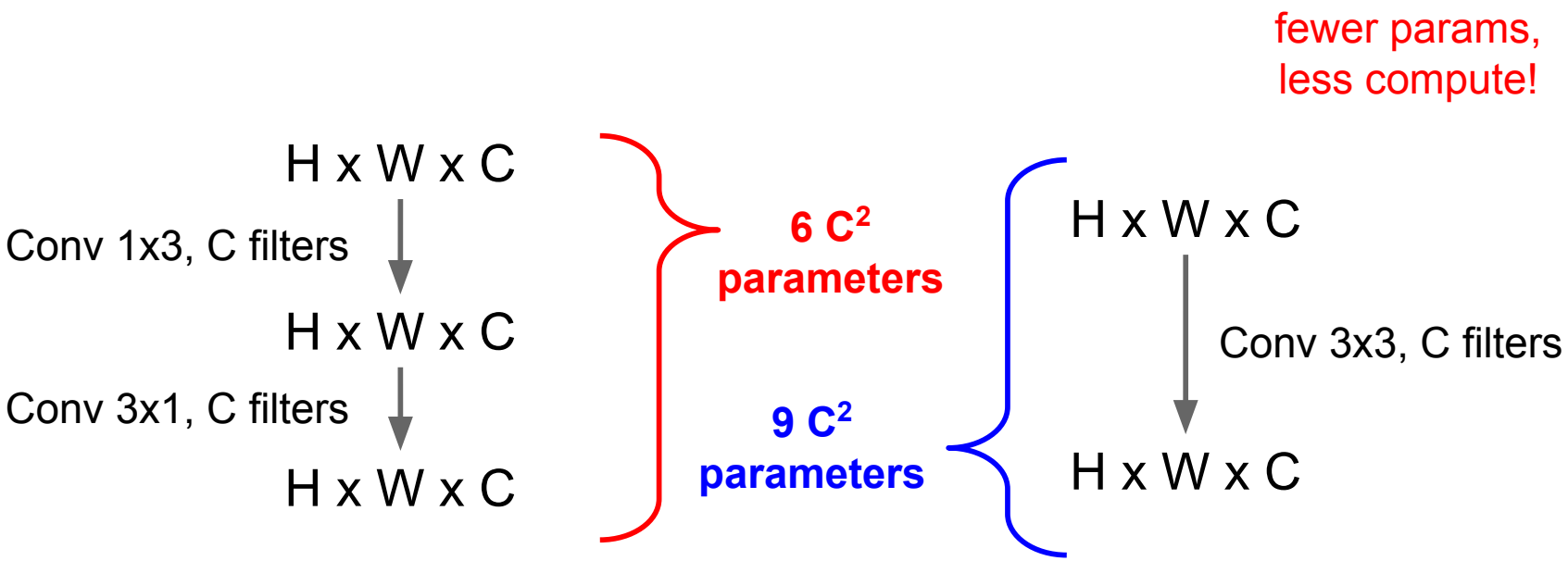
The Power of Small Filters

Still using 3×3 filters $\cdots$ can we break it up?



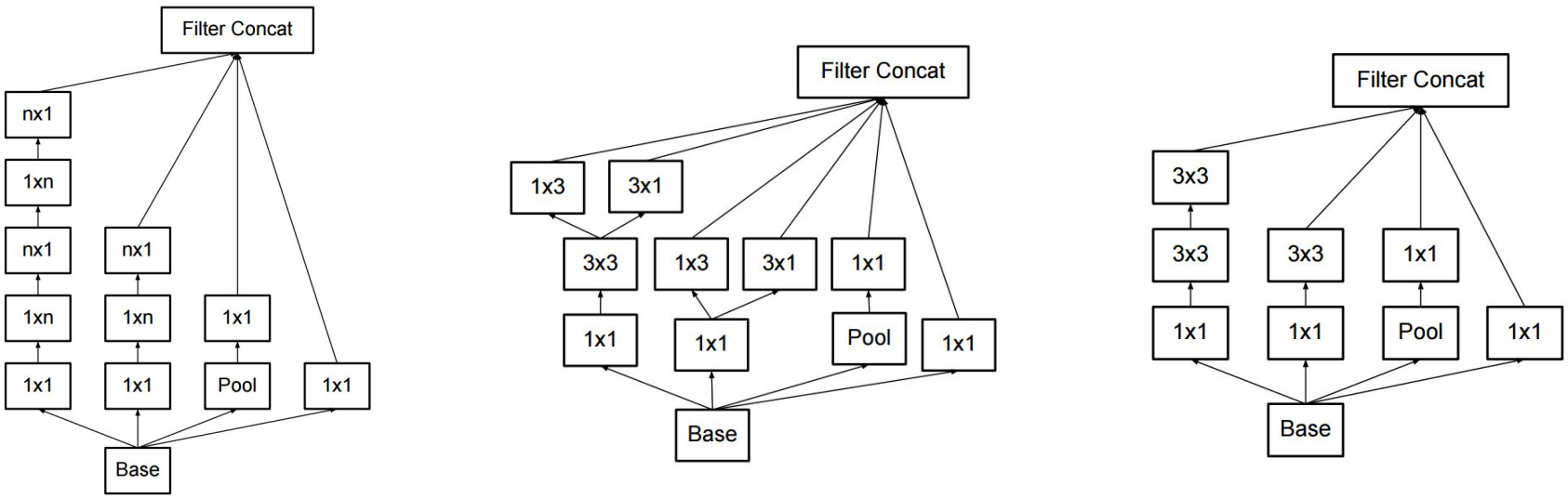
The Power of Small Filters

Still using 3×3 filters $\cdots$ can we break it up?



The Power of Small Filters

Latest version of GoogLeNet incorporates all these ideas



Szegedy et al, “Rethinking the Inception Architecture for Computer Vision”

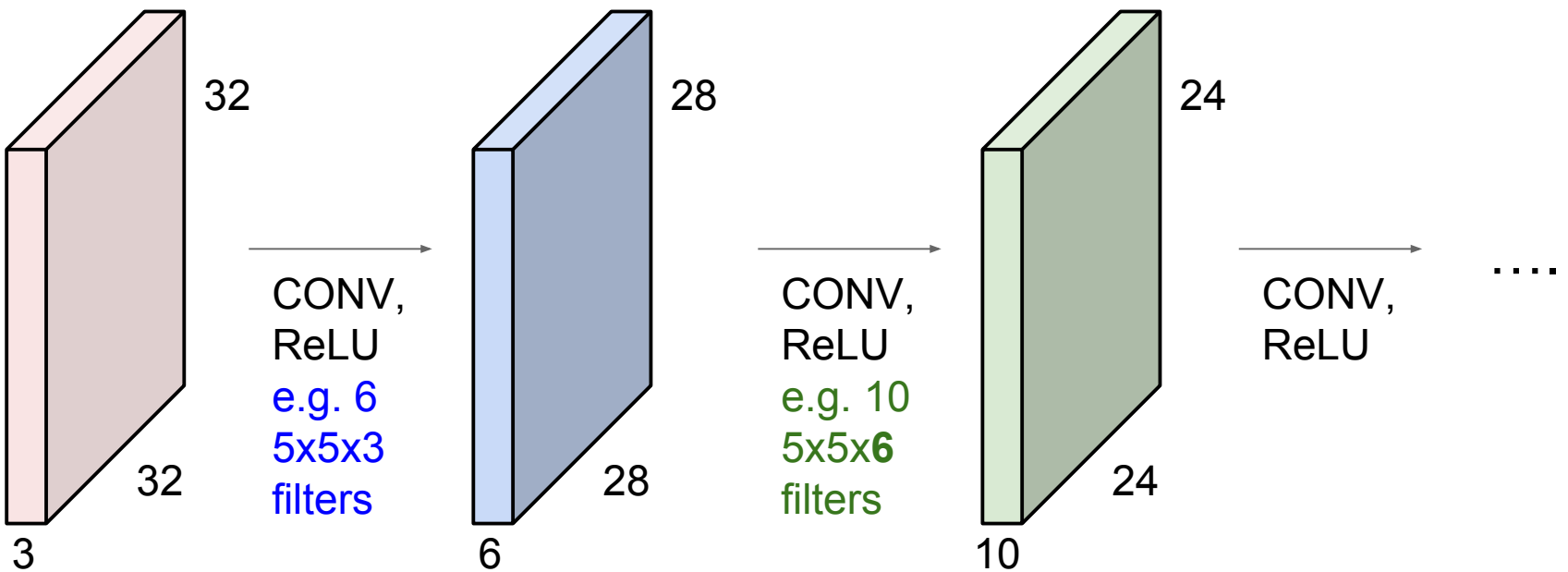
How to Stack Convolutions: Recap

1. Replace large convolutions (5×5, 7×7) with stacks of 3×3 convolutions.
2. 1×1 "bottleneck" convolutions are very efficient.
3. Can factor $N \times N$ convolutions into $1 \times N$ and $N \times 1$.
4. All of the above give fewer parameters and less compute.

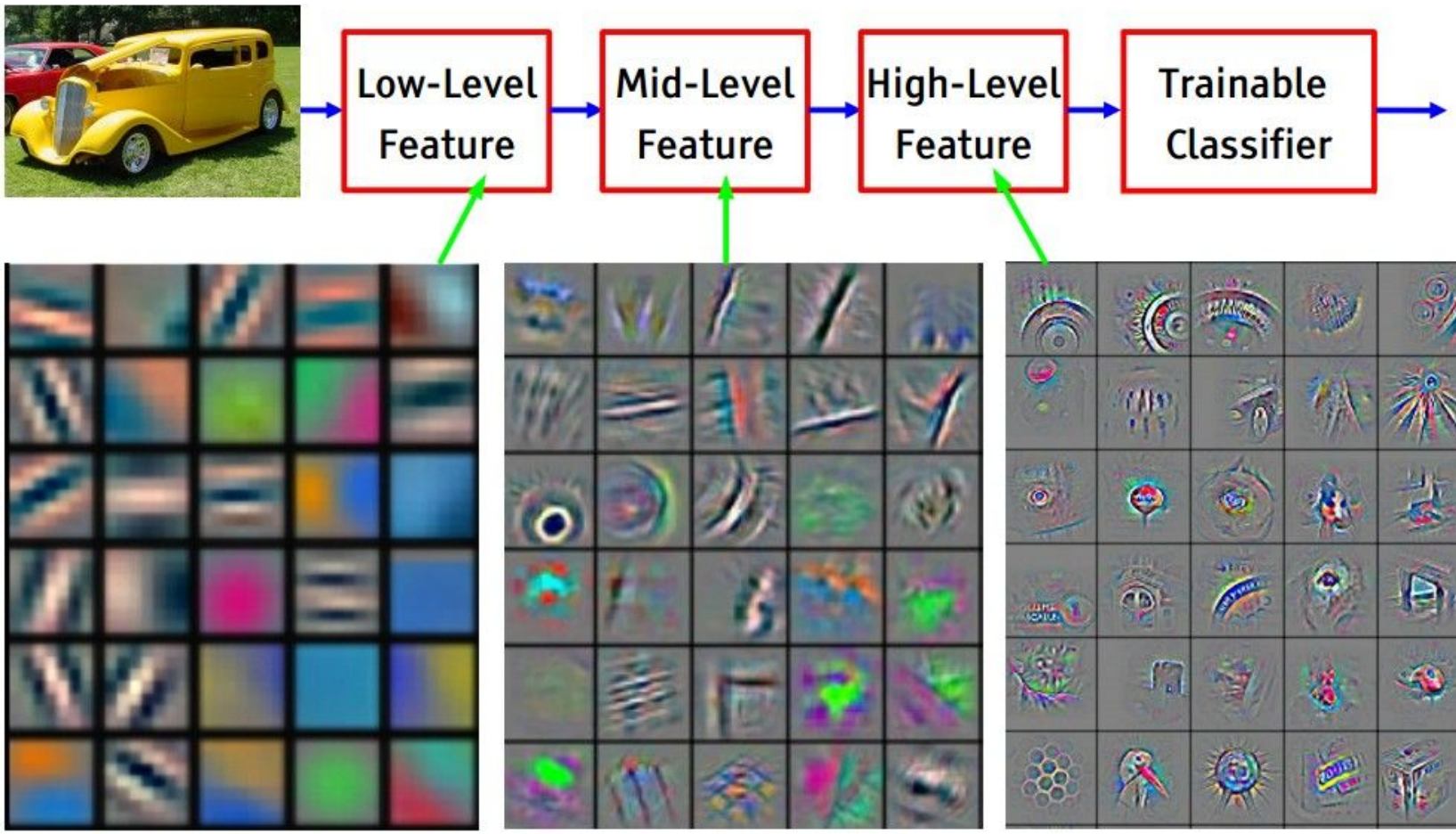
Convolutional Neural Networks (ConvNet)

Convolutional Neural Networks

ConvNet is a sequence of Convolutional Layers, interspersed with activation functions

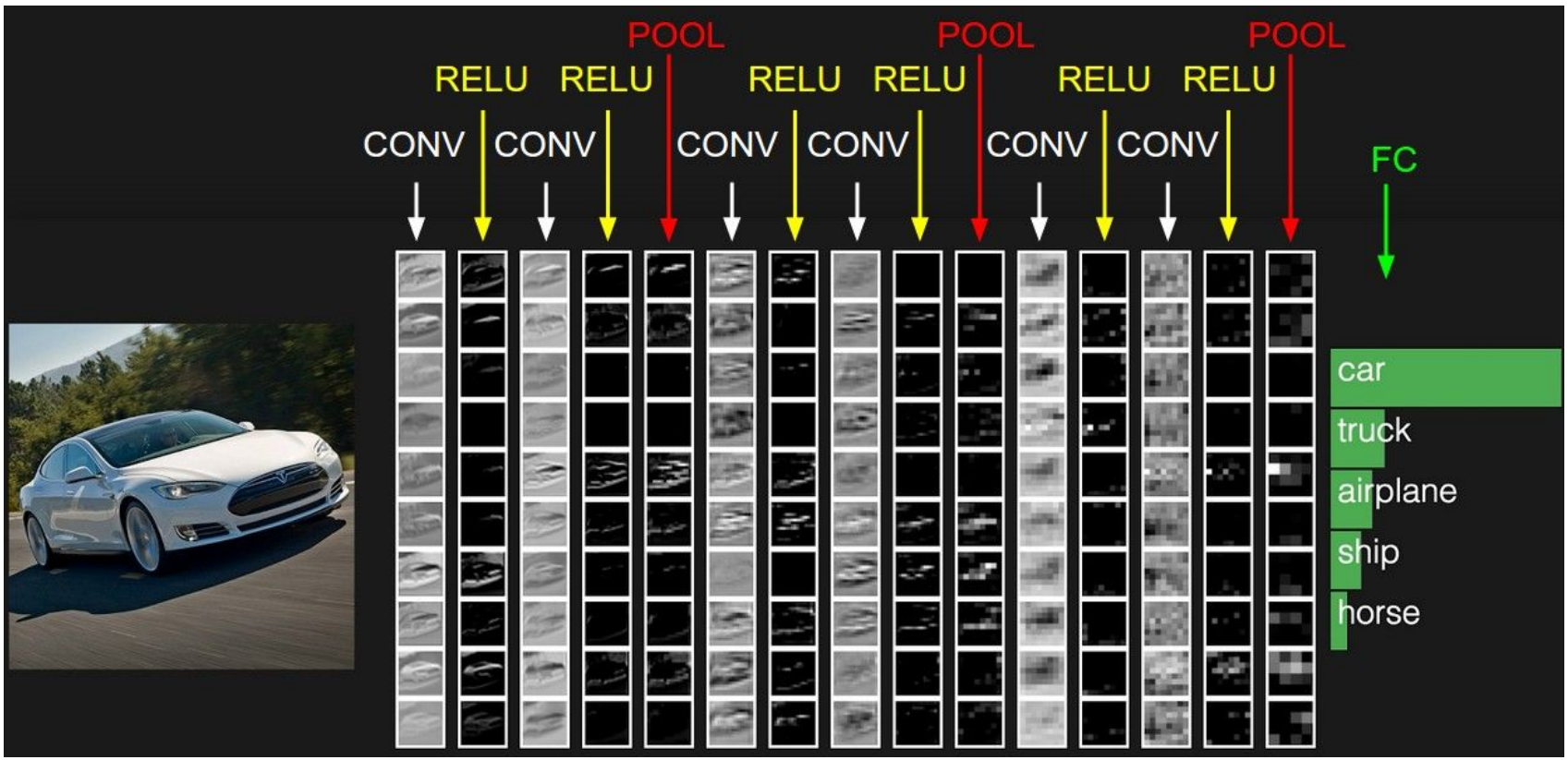


Features from ConvNet



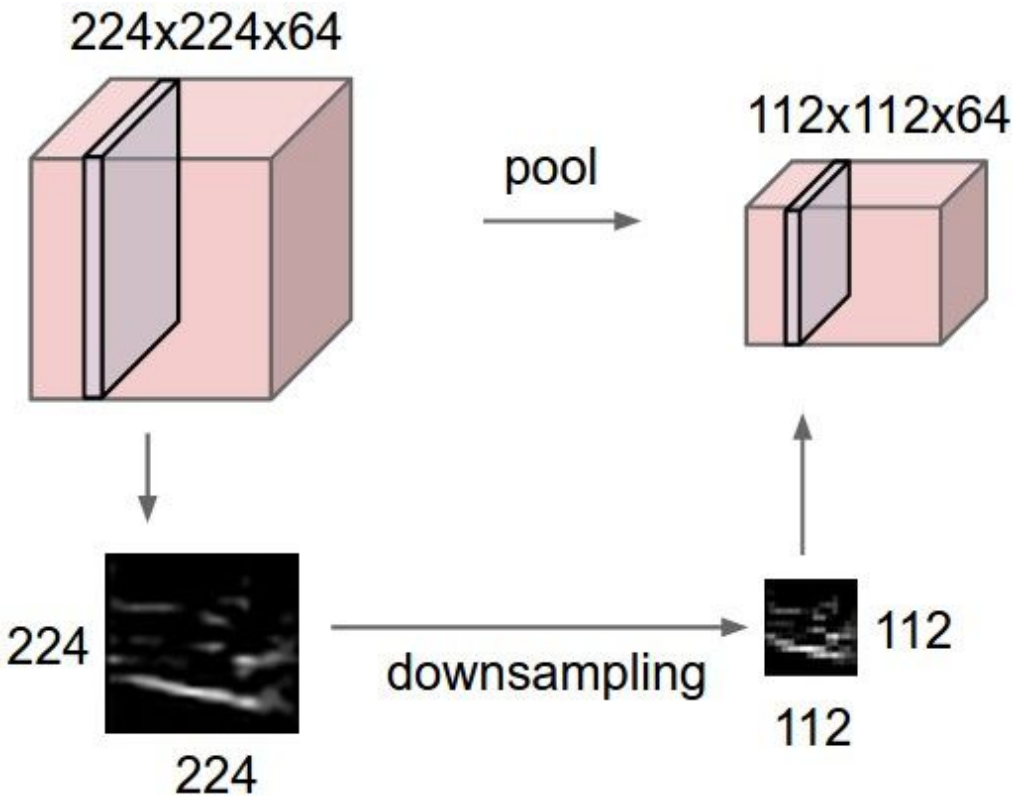
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Two More Layers to Go: POOL/FC

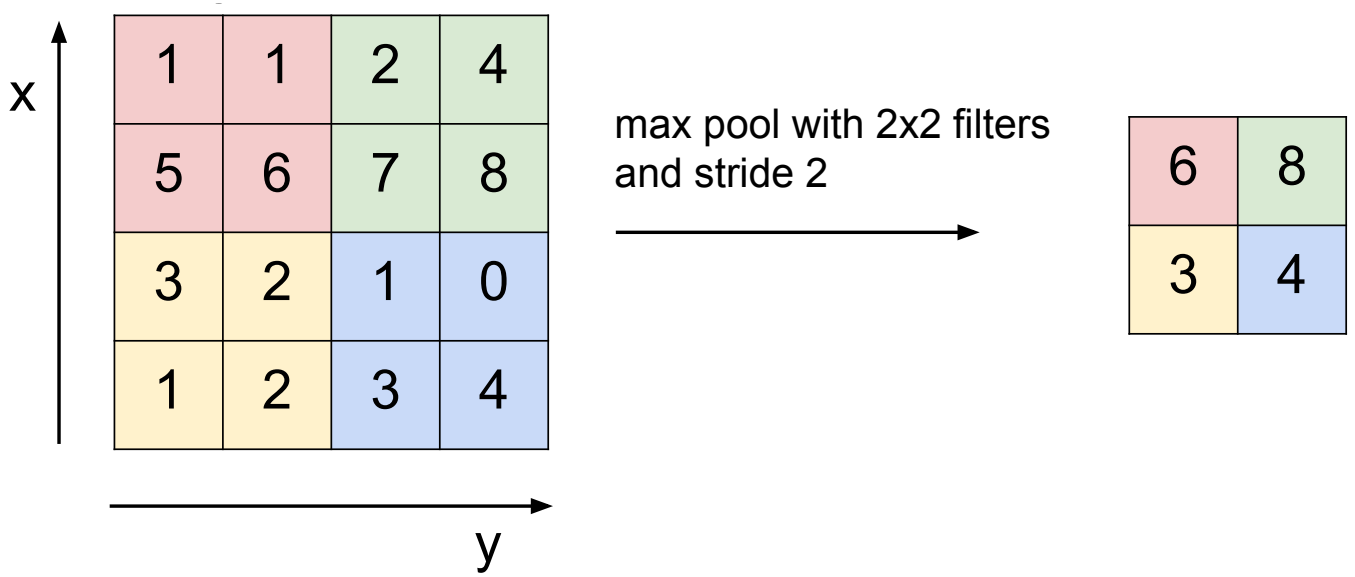


Pooling Layer

1. Makes the representations smaller and more manageable.
2. Operates over each activation map independently.



Max Pooling



Common settings:

- Filter size: 2×2 or 3×3
- Stride: 2

Max pooling is a non-linear operator.

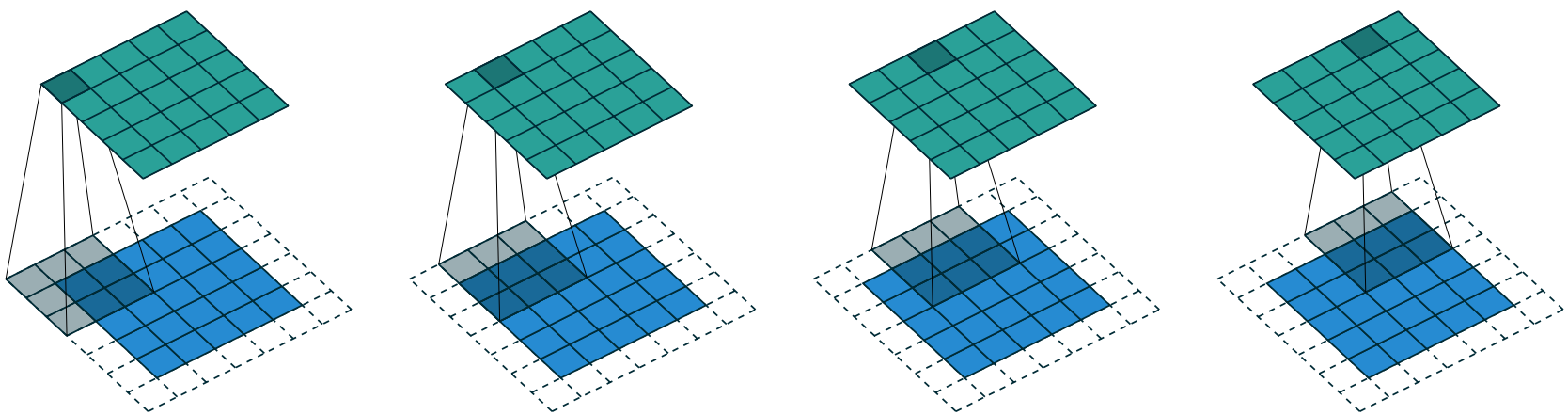
Max Pooling

Q: What is the gradient of a max pooling operation?

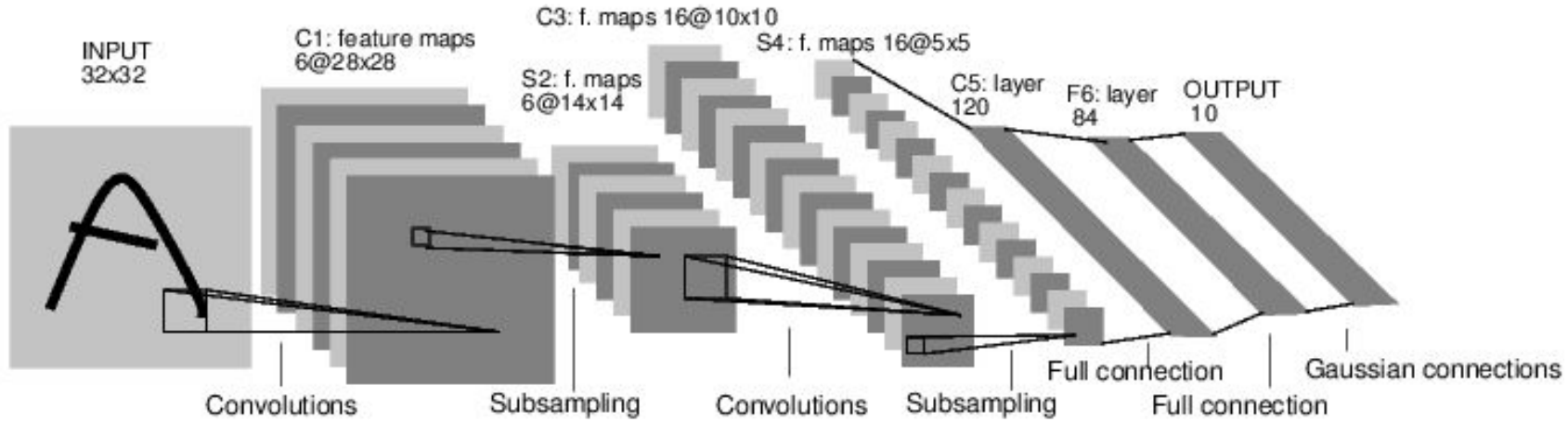
Case Study: CNN for MNIST

Padding in Tensorflow

- Input width = 13; Filter width = 6; Stride = 5.
- "VALID" only ever drops the right-most columns (or bottom-most rows).
- "SAME" tries to pad evenly left and right, but if the amount of columns to be added is odd, it will add the extra column to the right.



Case Study: LeNet-5 [LeCun *et al.*, 1998]



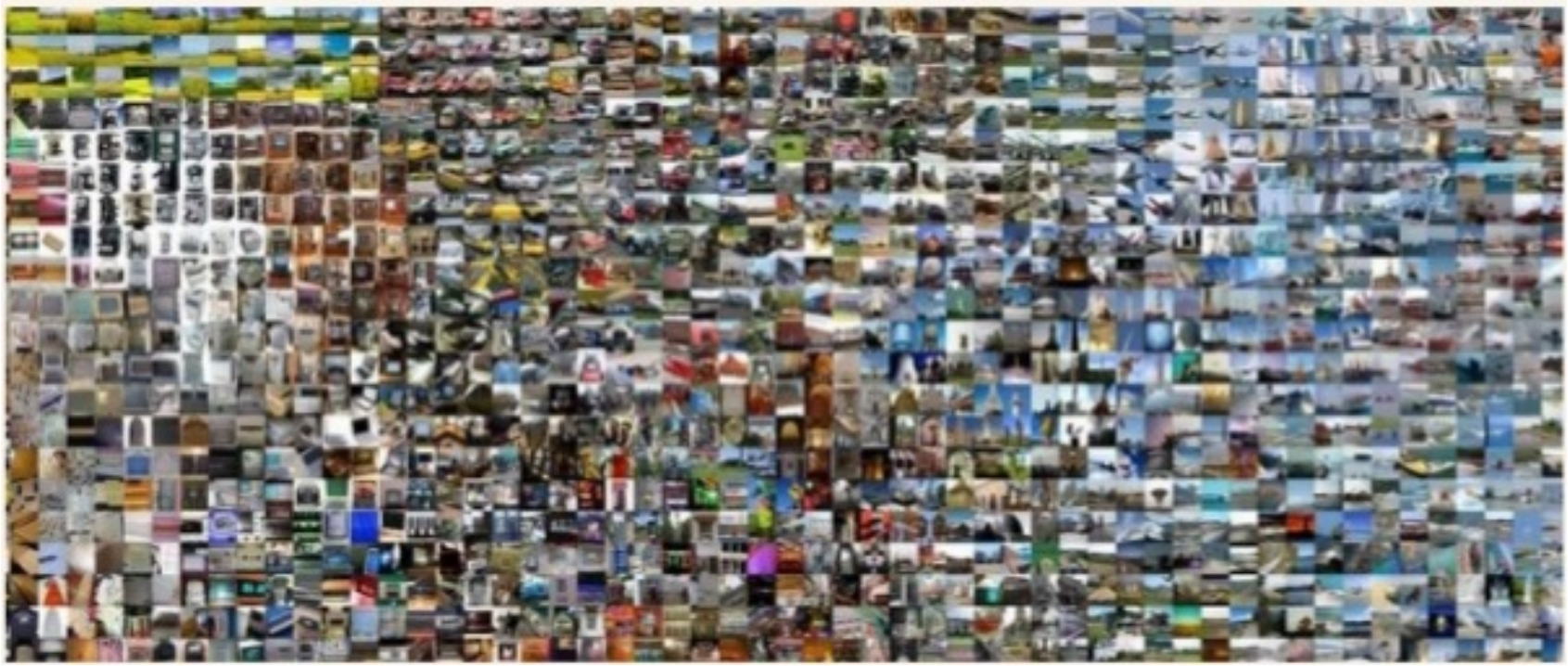
Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

Implementation

https://github.com/sujaybabruwad/LeNet-in-Tensorflow/blob/master/LeNet-Lab.ipynb

ImageNet

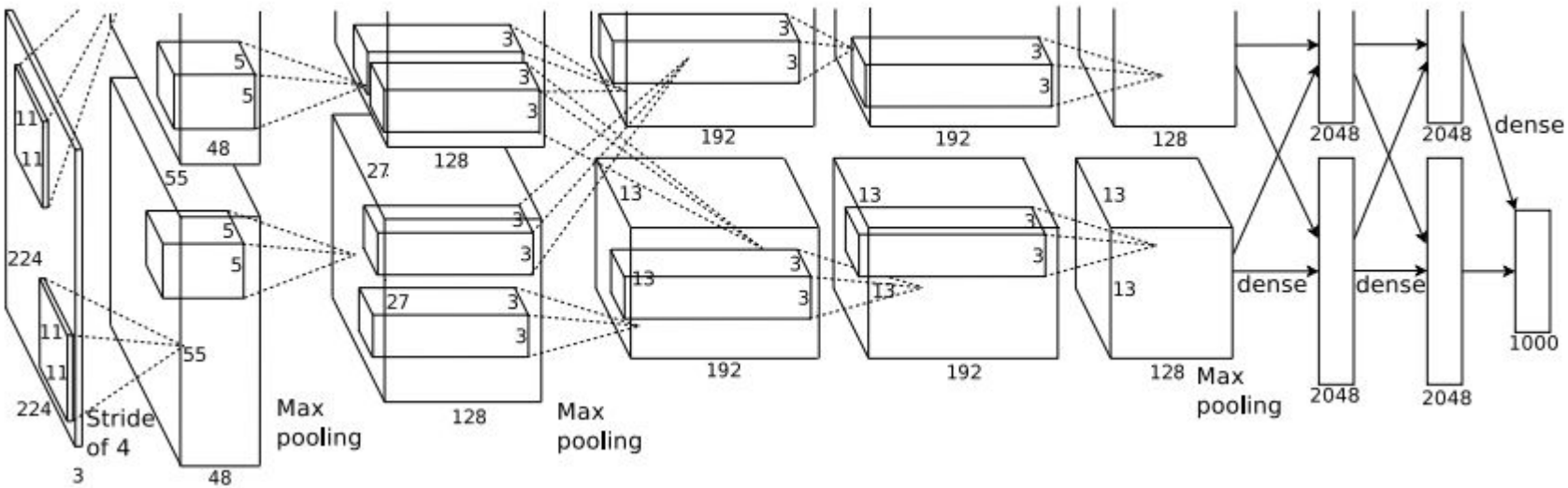
- About 14M images, 1000 classes.



Reference:
http://www.image-net.org/

Case Study: AlexNet [Krizhevsky *et al.*, 2012]

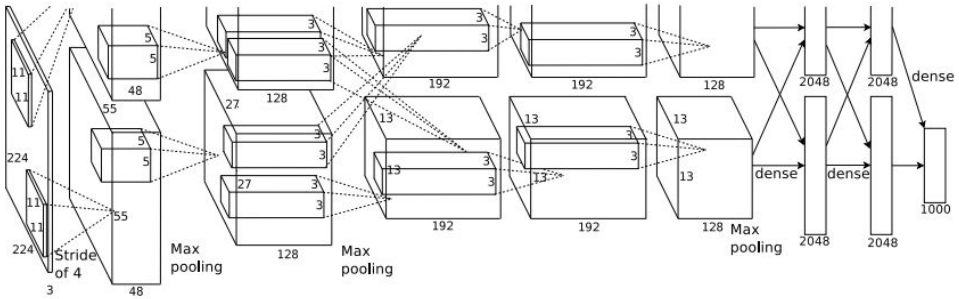
- Input: $227 \times 227 \times 3$ images.
- First layer (CONV1): 96 11×11 filters applied at stride 4.



- Q: What are the output volume and number of parameters in each layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

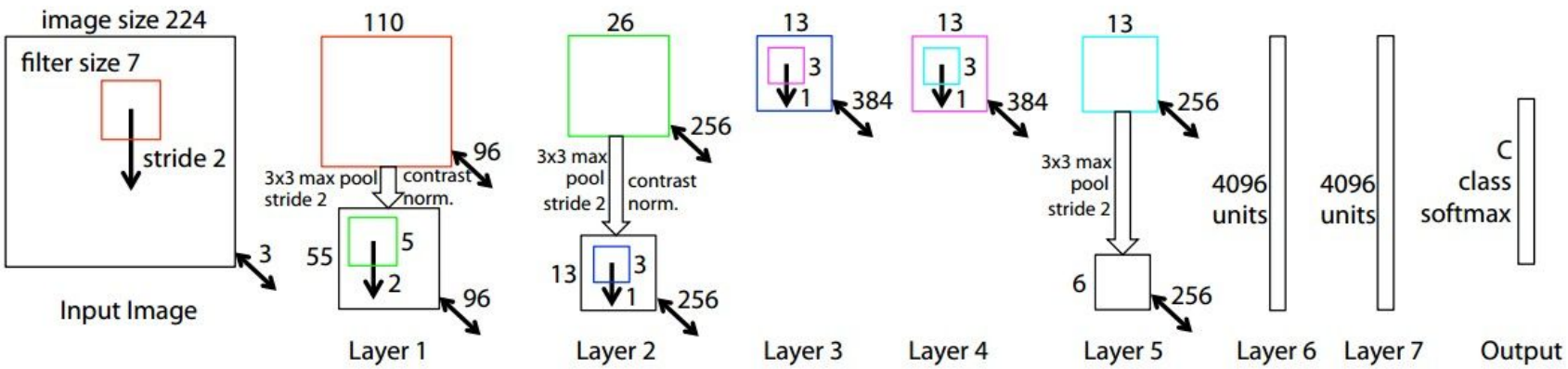
Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/AlexNet.py

Case Study: ZFNet [Zeiler and Fergus, 2013]

- AlexNet but:
  - CONV1: change from (11×11 stride 4) to (7×7 stride 2)
  - CONV3, 4, 5: instead of 384, 384, 256 filters use 512, 1024, 512
- ImageNet top 5 error: 15.4% -¿ 14.8%.



Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/ZFNet.py

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

best model

11.2% top 5 error in ILSVRC 2013
->
7.3% top 5 error

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 **conv1-256**	conv3-256 **conv3-256**	conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

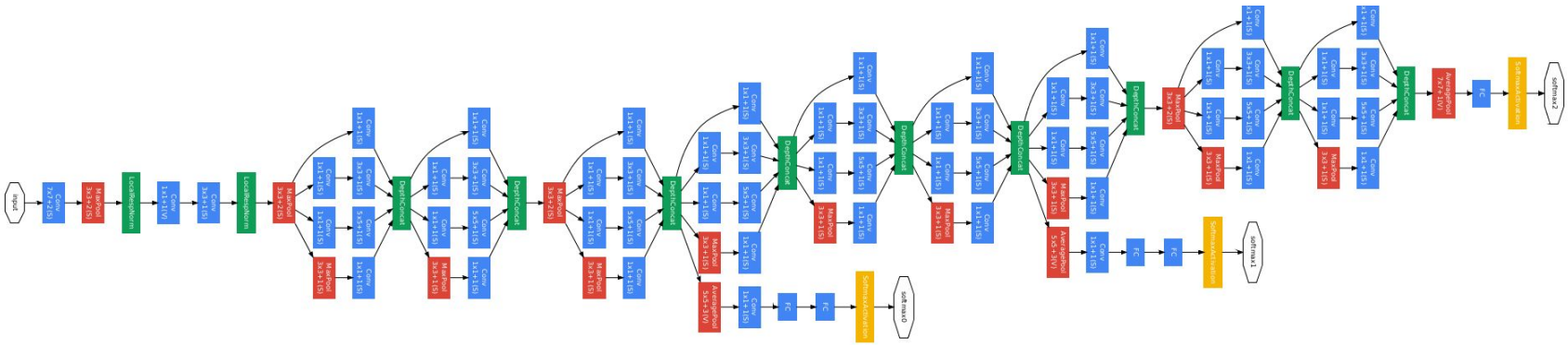
Table 2: **Number of parameters** (in millions).

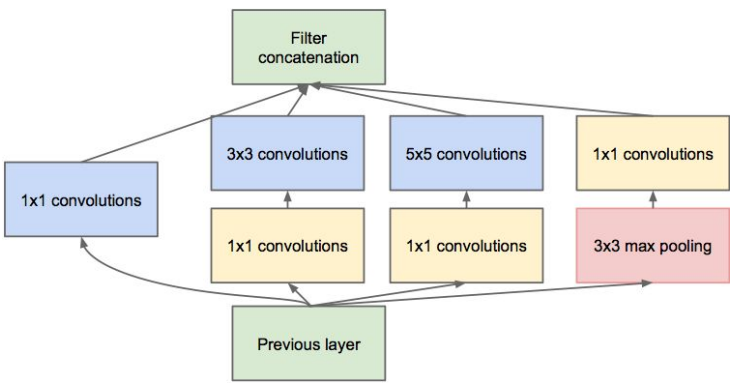
Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/VGGNet.py

Case Study: GoogLeNet [Szegedy *et al.*, 2014]





Inception module

ILSVRC 2014 winner (6.7% top 5 error)

Case Study: GoogLeNet [Szegedy *et al.*, 2014]

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Fun features:

- Only 5 million params! (Removes FC layers completely)

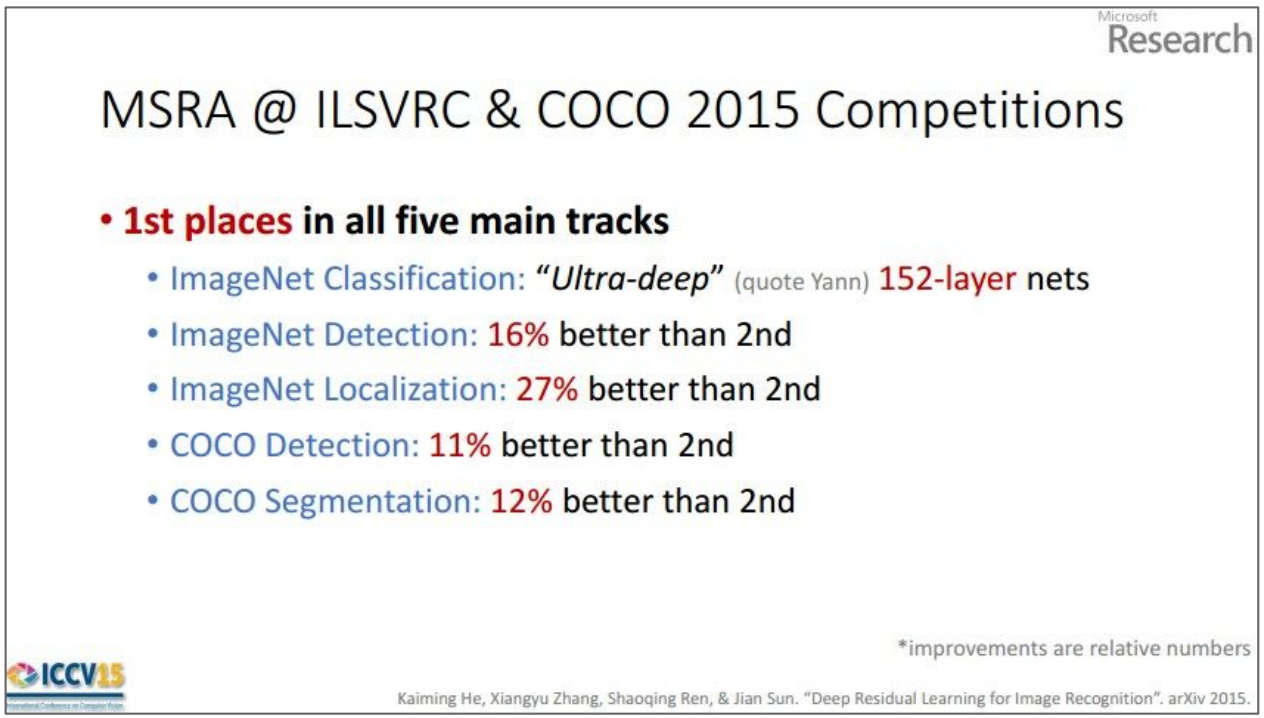
Compared to AlexNet:
- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)

Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/GoogLeNet.py

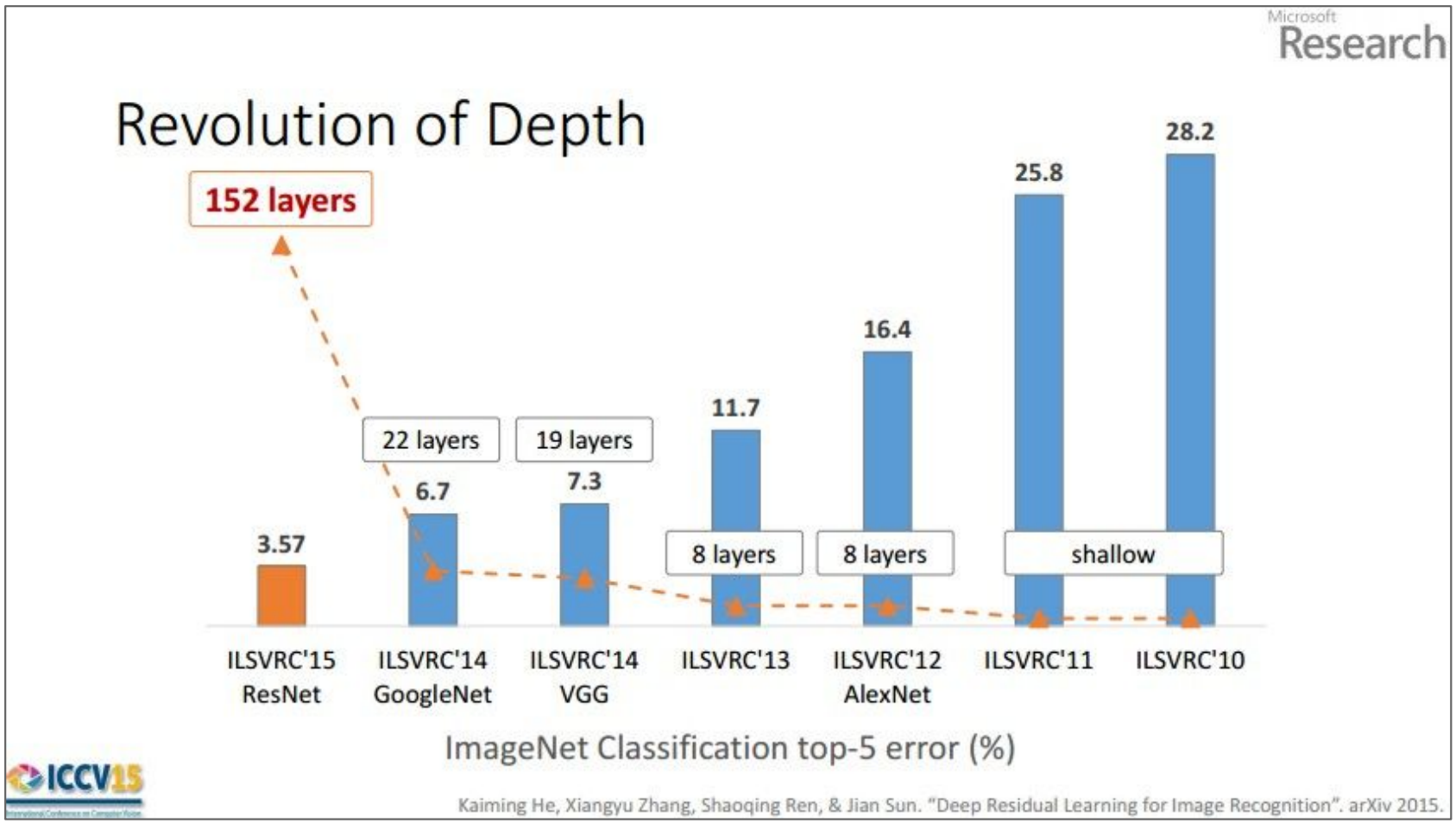
Case Study: ResNet [He *et al.*, 2015]

ILSVRC 2015 winner (3.6% top 5 error)

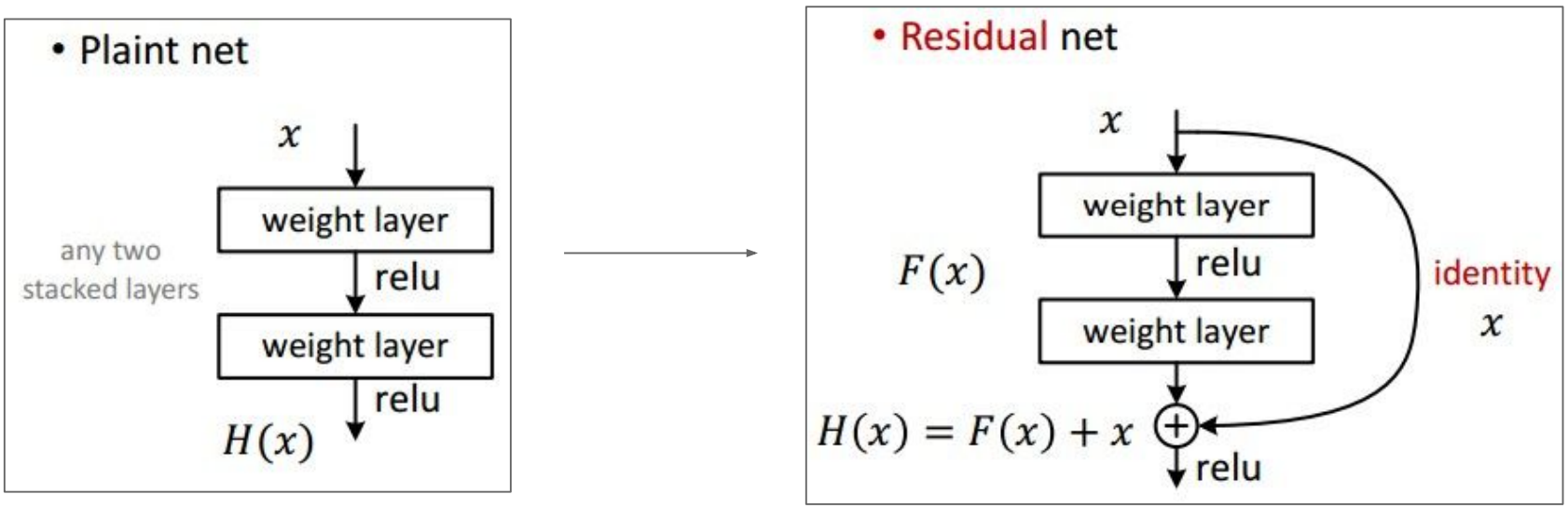


Slide from Kaiming He's recent presentation https://www.youtube.com/watch?v=1PGLj-uKT1w

Case Study: ResNet [He *et al.*, 2015]



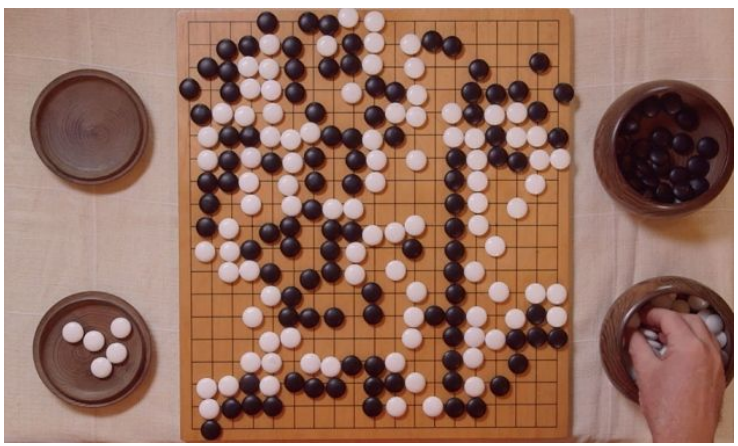
Case Study: ResNet [He *et al.*, 2015]

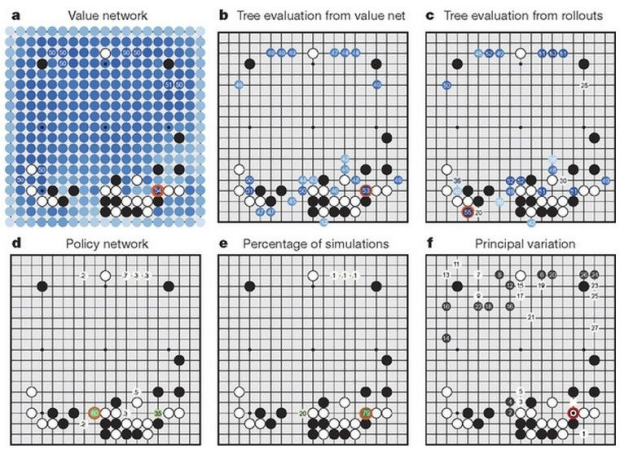


Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/ResNet.py

Case Study Bonus: DeepMind's AlphaGo





Case Study Bonus: DeepMind's AlphaGo

> The input to the policy network is a 19 × 19 × 48 image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23 × 23 image, then convolves *k* filters of kernel size 5 × 5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21 × 21 image, then convolves *k* filters of kernel size 3 × 3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1 × 1 with stride 1, with a different bias for each position, and applies a softmax function. The match version of AlphaGo used *k* = 192 filters; Fig. 2b and Extended Data Table 3 additionally show the results of training with *k* = 128, 256 and 384 filters.

policy network:
[19x19x48] Input
CONV1: 192 5x5 filters , stride 1, pad 2 => [19x19x192]
CONV2..12: 192 3x3 filters, stride 1, pad 1 => [19x19x192]
CONV: 1 1x1 filter, stride 1, pad 0 => [19x19] *(probability map of promising moves)*

Summary

1. ConvNets stack CONV,POOL,FC layers.
2. Trend towards smaller filters and deeper architectures.
3. Trend towards getting rid of POOL/FC layers (just CONV and action layers).
4. Typical architectures look like:

   [(CONV-ReLU)* N**-Pool?]*** M**-(FC-ReLU)*** K**, SOFTMAX**

   - N is usually up to ~ 5, M is large, $0 \leq K \leq 2$.
   - recent advances such as ResNet/GoogLeNet challenge this paradigm.