Designing, Visualizing and Understanding Deep Neural Networks

Lecture 5: Convolutional Networks I

CS 182/282A Spring 2019

John Canny

Slides originated from Efros, Karpathy, Ransato, Seitz, and Palmer

Last Time - Gradient Descent

To reach a minimum of loss, we should follow the negative gradient.

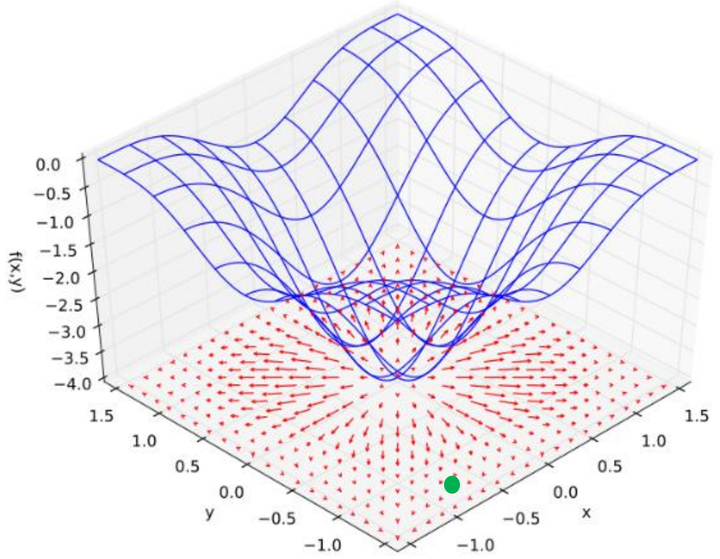
i.e. we should take small steps in direction

$$-\nabla_W L(W)$$

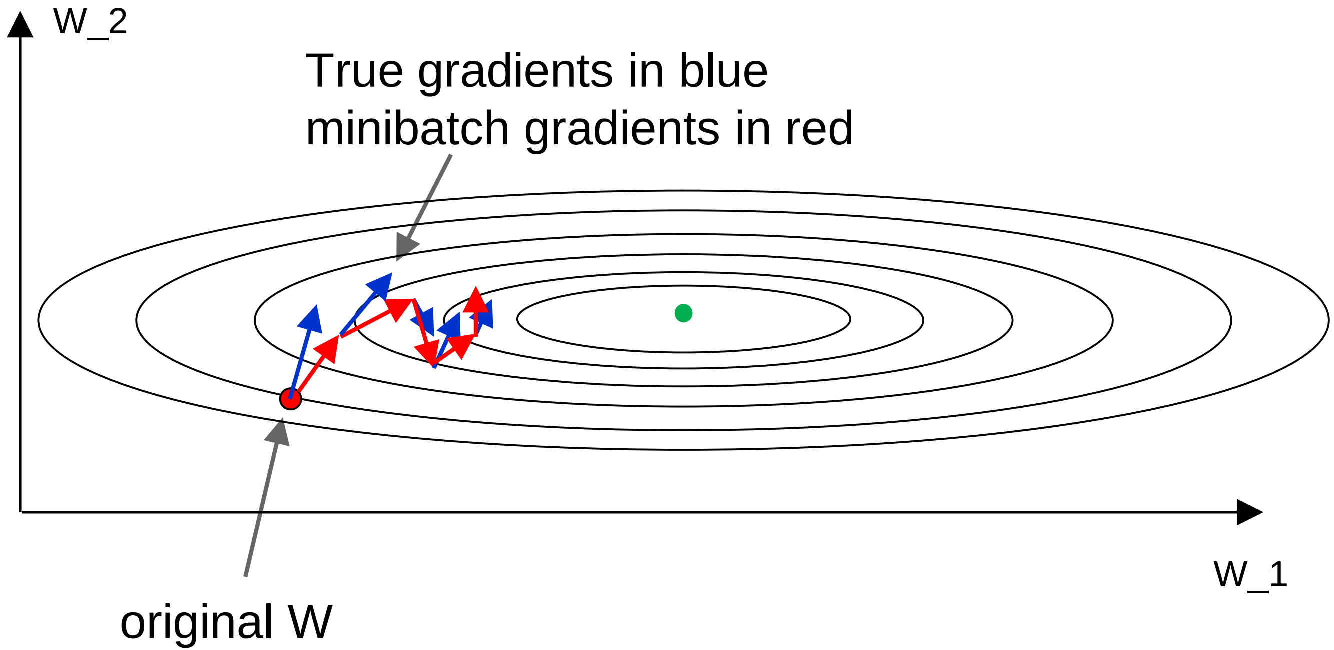
Let W^t denote the weights at step t of gradient descent. Then

$$W^{t+1} = W^t - \alpha \ \nabla_W L(W)$$

Where α is called the ***learning rate***.

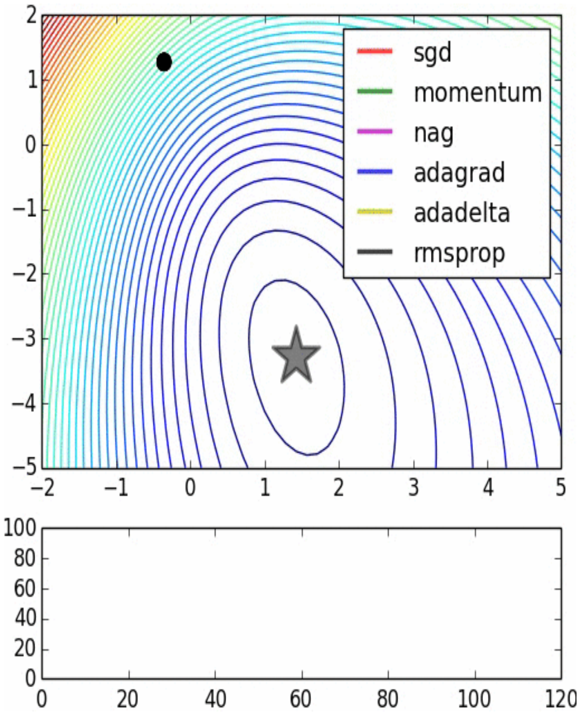


Last time: Minibatches and SGD



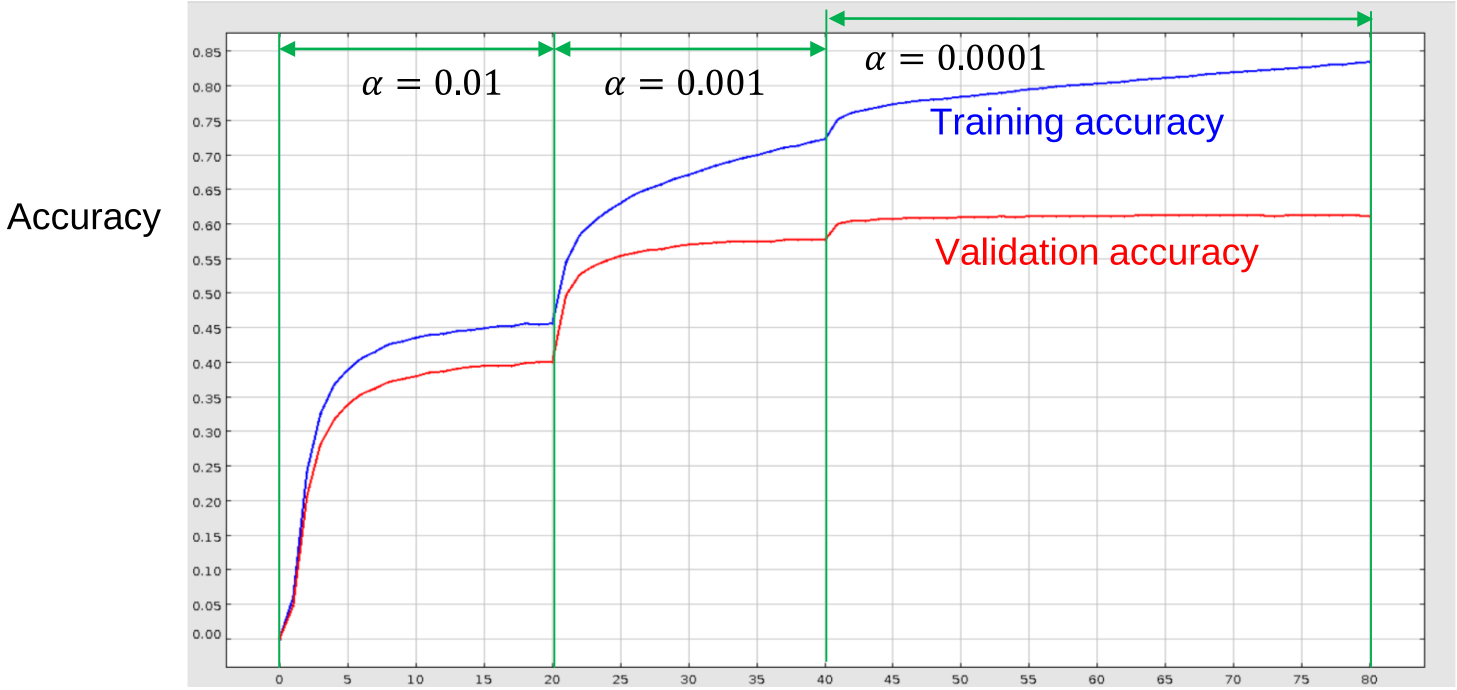
Gradients are noisy but still make good progress on average

Last Time: SGD refinements: Momentum, Nesterov Momentum, RMSprop, ADAGRAD



Last Time: Learning rate schedules

Alexnet trained on ImageNet data. α = learning rate.



Epoch: number of passes over the dataset

Updates

- Project Proposals are due in 2 ½ weeks, form your team asap, 3-4 people.
- Note: Project teams should all be registered for 182, or all for 282A as the project requirements are different.
- If you need help putting a team together, we suggest you post to Piazza in the "project_teams" folder.
- You can either post your own ideas on project topics, or respond to other groups looking to complete their team.

- Assignment 1 due on Feb 19, make sure at least that you've started by now…

This Time: Backpropagation

So far we have been using gradient methods to minimize a loss over some parameters.

Our loss is of the form $L(f(x, W), y)$.

where x is an input, y is a target, and W are the parameters.

To compute the gradient of L wrt W, we need the ***chain rule***. If f is single-valued, W a single parameter the chain rule is just:

$$\frac{dL}{dW} = \frac{dL}{df}\frac{df}{dW}$$

If W is a vector of parameters, then we have:

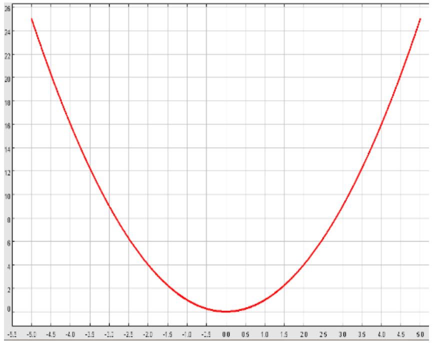
$$\nabla_W L = \frac{dL}{df}\nabla_W f$$

Which is really just the first rule applied to all the partial derivatives wrt elements of W.

Losses we have seen so far

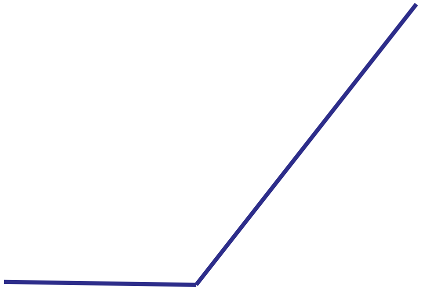
Squared Loss

$L = \left(y_i - f(x_i)\right)^2$



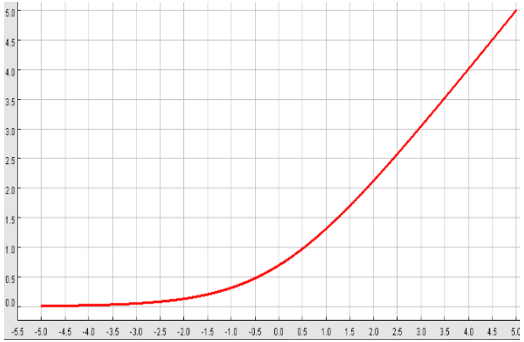
Hinge Loss, $y_i \in \{-1,1\}$

$L = \max(0, 1 - y_i f(x_i))$



Cross-entropy loss on logistic function, $y_i \in \{-1,1\}$

$L = \log(1 + \exp(-y_i f(x_i)))$



All three have "well behaved" derivatives. $f(x) = w^T x$ is a linear function of the weights W, so we can differentiate loss with respect to weights.

The Chain Rule

If f is also vector-valued with k values and W is a vector of m parameters, then we can apply the chain rule parameter-wise and then sum over the contributions:

$$\frac{\partial L}{\partial W_j} = \sum_{i=1}^{k} \frac{\partial L}{\partial f_i} \frac{\partial f_i}{\partial W_j}$$

For $j = 1, \ldots, m$. This can be written as a matrix multiply

$$J_L(W) = J_L(f)\ J_f(W)$$

Where $J_f(W)$ is a Jacobian matrix.

$$J_f(W)_{ij} = \frac{\partial f_i}{\partial W_j}$$

Jacobians

The Jacobian generalizes the gradient of a scalar-valued function f to a k-valued function. Here we think of the function as a neural layer with m inputs and k outputs.

$$J_f(W) = \begin{bmatrix} \frac{\partial f_1}{\partial W_1} & \frac{\partial f_1}{\partial W_2} & \cdots & \frac{\partial f_1}{\partial W_m} \\ \frac{\partial f_2}{\partial W_1} & \frac{\partial f_2}{\partial W_2} & \cdots & \frac{\partial f_2}{\partial W_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial W_1} & \frac{\partial f_k}{\partial W_2} & \cdots & \frac{\partial f_k}{\partial W_m} \end{bmatrix}$$

The Jacobian has dimensions $k \times m$ which is $n_{outputs} \times n_{inputs}$.

N-step Chain Rule

Now suppose we have several vector-valued functions $A(.)$, $B(.)$, $C(.)$, ... composed in a chain (e.g. a deep network):

$$A \rightarrow B \rightarrow C \rightarrow \cdots K \rightarrow \ L$$

Algebraically, that looks like:

$$L(A) = L(K(\cdots C\big(B(A)\big) \cdots))$$

Then we just multiply Jacobians (matrix multiply $*$) to get the gradient:

$$J_L(A) = J_L(K) * \cdots J_C(B) * J_B(A)$$

And $J_L(A) = (\nabla_A L)^T$ which is the gradient we need to minimize loss over A.

Backpropagation

Now all the Jacobians are matrices, and matrix multiply is associative.

$$J_L(A) = J_L(K) * \cdots J_C(B) * J_B(A)$$

So we could actually evaluate the product of Jacobians in any order, including left-to-right and right-to-left.

Backpropagation: evaluate the Jacobian product (loss gradient wrt params) left-to-right, i.e. from the output of the neural network toward its input.

Why not some other order?

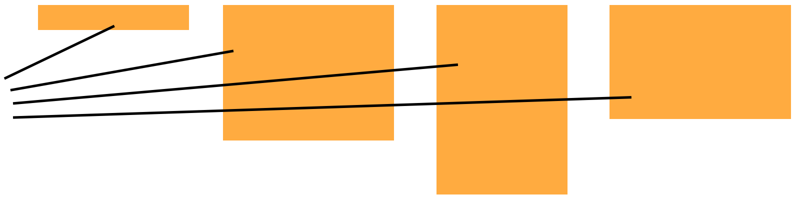
Backpropagation

Reason 1: Efficiency

Output Jacobian is always a row vector (because loss is a scalar). Matrix-vector multiply is much less expensive than matrix-matrix multiply.

$$J_L(A) = \overbrace{J_L(D) \;*\; J_D(C)}^{\text{Do first (output layer)}} \;*\; J_C(B) \;*\; J_B(A)$$

Jacobians (matrix shape)



Back (output) of network

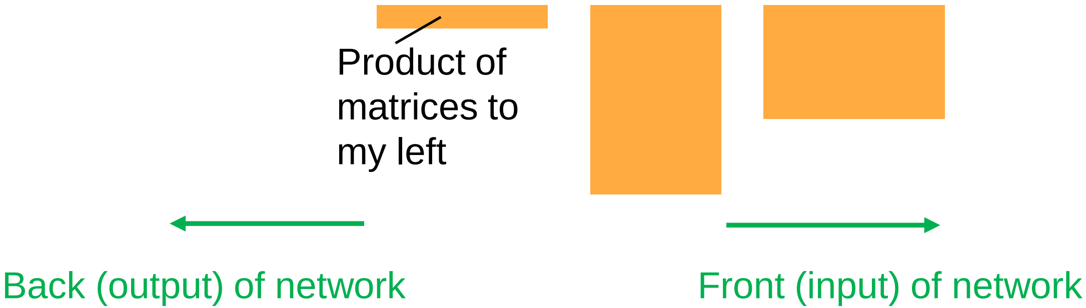
Front (input) of network

Backpropagation

Reason 1: Efficiency

Output Jacobian is always a row vector. Matrix-vector multiply is much less expensive than matrix-matrix multiply.

$$J_L(A) = \overbrace{J_L(C) \ast J_C(B)}^{\text{Do next}} \ast J_B(A)$$



Backpropagation

Reason 1: Efficiency

Output Jacobian is always a row vector. Matrix-vector multiply is much less expensive than matrix-matrix multiply.

$$J_L(A) = \overbrace{J_L(B) * J_B(A)}^{\text{Do next}}$$

Product of matrices to my left

Back (output) of network

Front (input) of network

Backpropagation

Reason 1: Efficiency

Output Jacobian is always a row vector. Matrix-vector multiply is much less expensive than matrix-matrix multiply.

Final result, without any matrix-matrix multiplies

$$J_L(A) = \qquad J_L(A)$$

Product of all matrices

Back (output) of network

Front (input) of network

For-propagation

Reason 1: Efficiency

By comparison, we could invent “for-propagation”:

$$J_L(A) = J_L(D) \; * \; J_D(C) \; * \; \overbrace{J_C(B) \; * \; J_B(A)}^{\text{Do first}}$$

Back (output) of network

Front (input) of network

For-propagation

Reason 1: Efficiency

We could invent "for-propagation": Oops, cost $O(n^3)$ instead of $O(n^2)$ for the first multiply and we still have another matrix-matrix multiply to do.

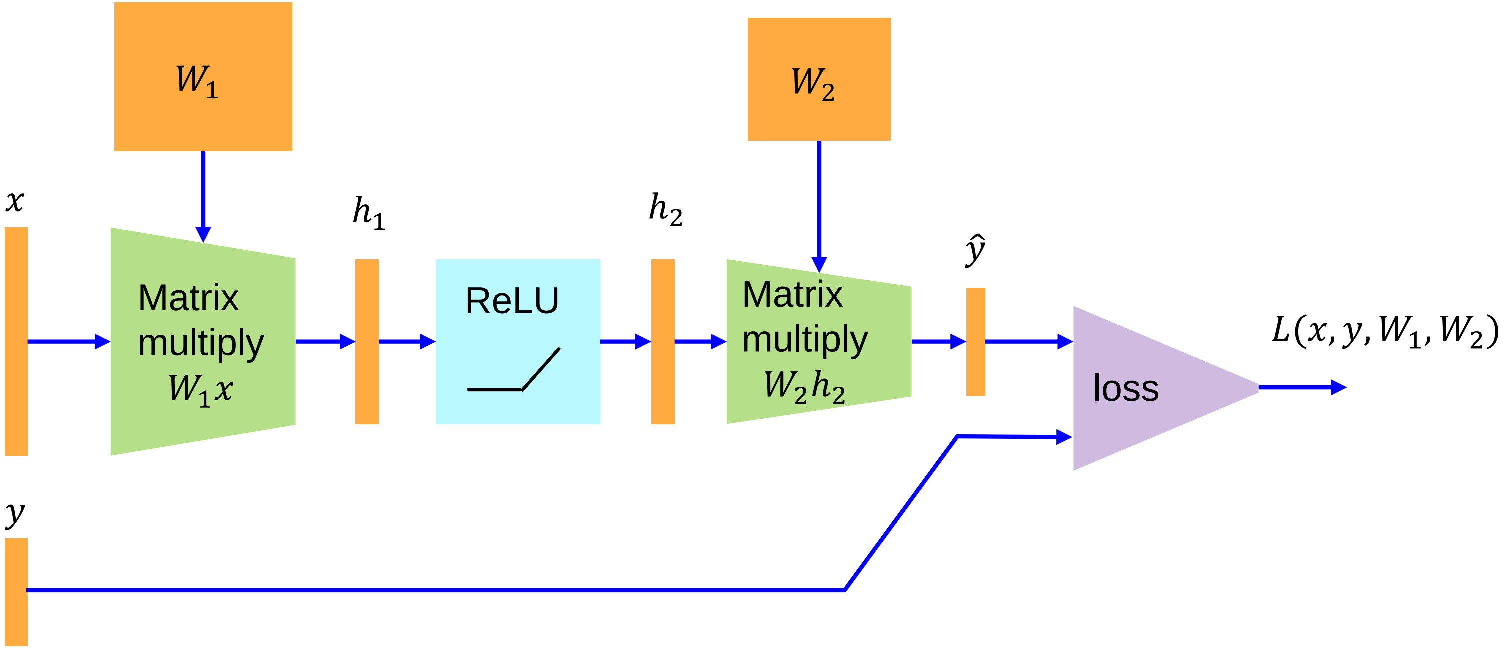
A good invention for the wastebasket…

$$J_L(A) = J_L(D) \;*\; J_D(C) \;*\; \overbrace{J_C(B) \;*\; J_B(A)}^{\text{Do first}}$$

Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

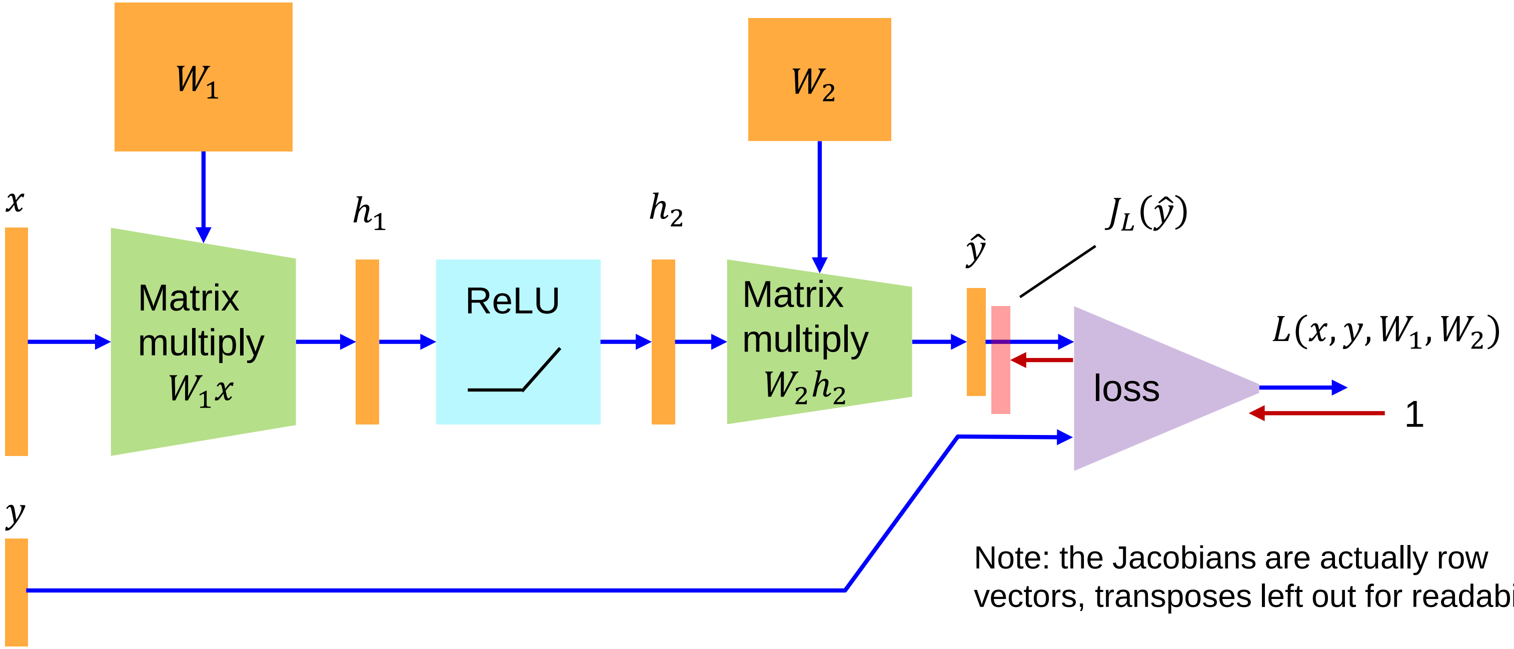
Let's build a real neural network (Tensorflow style):



Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

Let's build a real neural network (Tensorflow style):

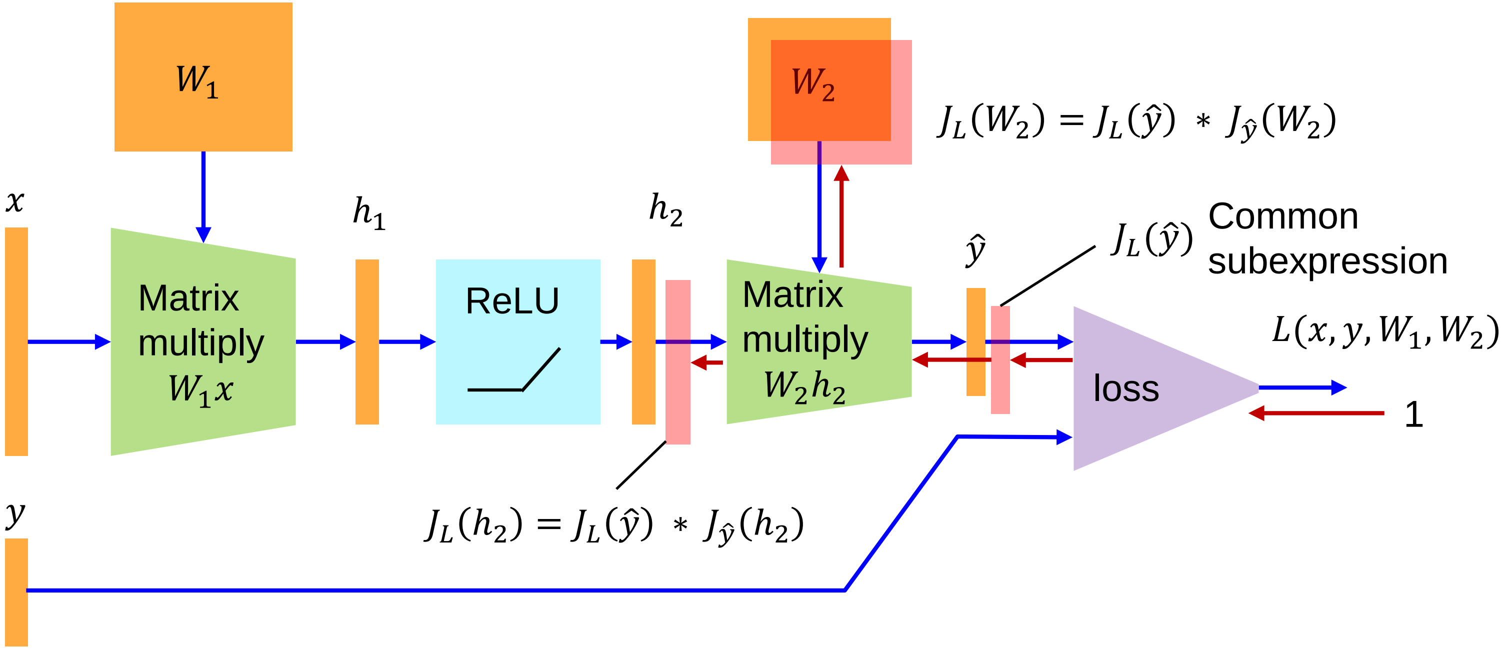


Note: the Jacobians are actually row vectors, transposes left out for readability.

Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

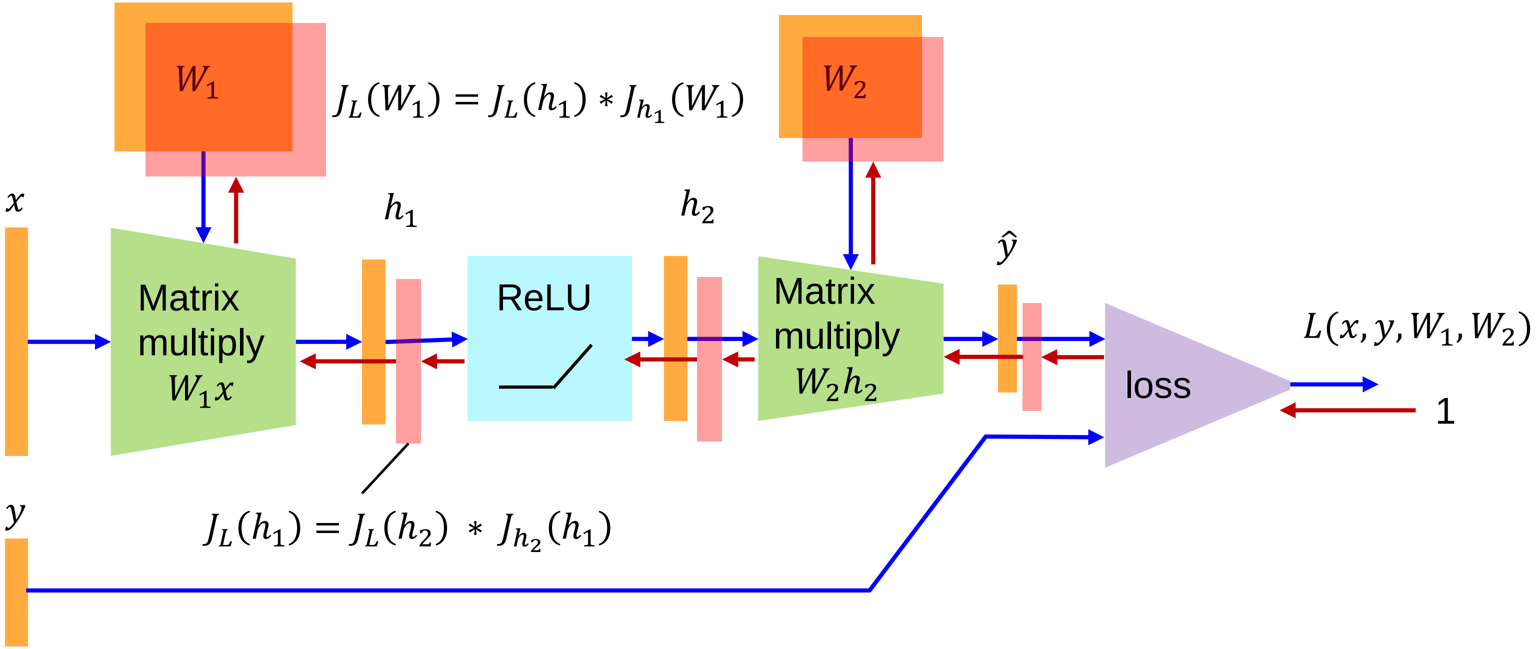
Let's build a real neural network (Tensorflow style):



Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

Let's build a real neural network (Tensorflow style):



Backpropagation Recipes: Cross Entropy Loss

- This is just calculus: always check yourself:
- Cross-entropy loss (correct class label $y \in \{1, \ldots, n\}$ known exactly, input vector x):

$$L = -\log x_y$$

So

$$J_L(x)_i = \nabla_x L_i^T = \begin{cases} -\frac{1}{x_i} & \text{for } i = y \\ 0 & \text{otherwise} \end{cases}$$

In general if y is a distribution, $L = -\sum_{i=1}^{n} y_i \log x_i$, then

$$J_L(x)_i = \nabla_x L_i^T = -\frac{y_i}{x_i}$$

Backprop Recipes: Multiclass SVM Loss

- Assume label $y \in \{1, \ldots, n\}$ known exactly, input vector x:

$$L = \sum_{i=1}^{n} \max(0, 1 - x_y + x_i)$$

So

$$J_L(x)_i = \nabla_x L_i^T = \begin{cases} 1 & \text{if } x_y - x_i < 1 \\ 0 & \text{otherwise} \end{cases}$$

Backprop Recipes: Multiclass Logistic Fn

Assume input vector x, output y of same dimension n.

$$y_i = \frac{\exp x_i}{\exp x_1 + \cdots + \exp x_n} = \frac{f}{g}$$

Then

$$\frac{\partial y_i}{\partial x_j} = \frac{f' g - g' f}{g^2} = \frac{\exp x_i \, g \delta_{ij} - \exp x_j \exp x_i}{g^2}$$

Simplifying:

$$J_y(x) = \frac{\partial y_i}{\partial x_j} = y_i \delta_{ij} - y_i y_j$$

or

$$J_L(x) = J_L(y) \circ y^T - J_L(y) y y^T$$

Where $\circ$ is element-wise product and:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Backprop Recipes: ReLU

Assume input vector x, output y of same dimension n.

$$y_i = \max(0, x_i)$$

Then

$$J_y(x) = \frac{\partial y_i}{\partial x_j} = \begin{cases} 1 \text{ if } i = j \text{ and } x_i > 0 \\ 0 \text{ otherwise} \end{cases}$$

Or more directly

$$J_L(x) = (y > 0)^T \circ J_L(y)$$

Where $\circ$ is element-wise product.

In-place ReLU: Since backpropagation can be done with only y (and not x), we can use the same storage for x and y, i.e. we overwrite x with y during the forward pass.

“Tensors” (not really)

Deep networks are often applied to structured data like images or time series or texts.

Activations are also grouped into minibatches of some dimension N.

Its useful to organize layer activations and model weights to represent this structure. They are therefore most naturally represented as ***multi-dimensional arrays*** (incorrectly called “tensors”).

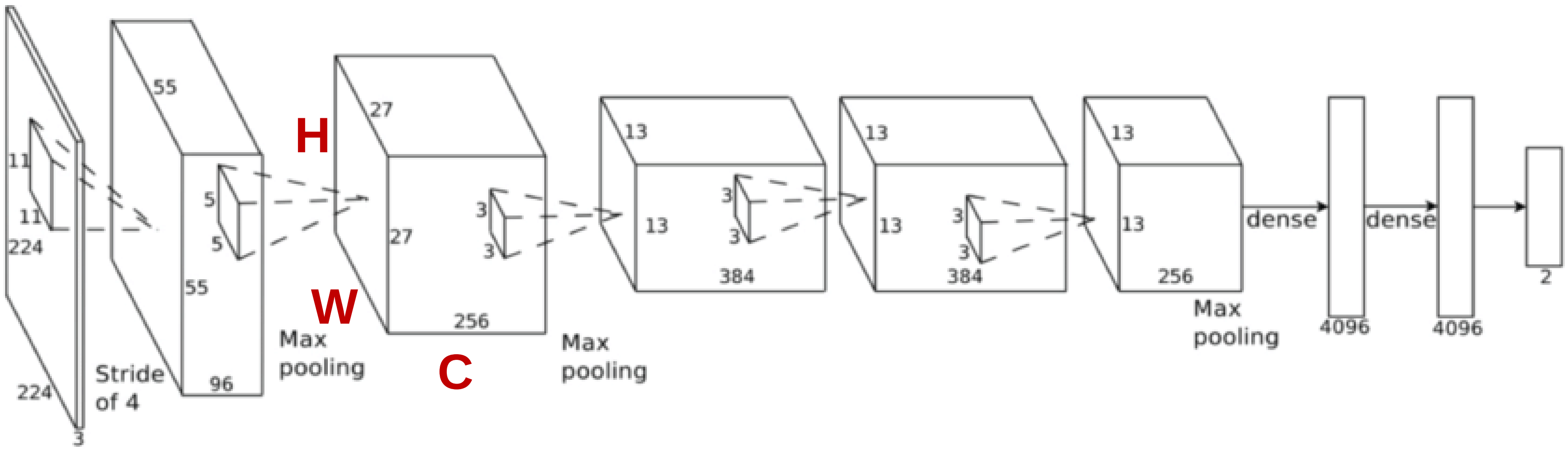
e.g. the activations in layers of most 2D image processing networks have dimensions:

N x C x H x W   or   N x H x W x C

N = minibatch dimension
C = number of channels or filters
H = image height
W = image width

“Tensors” (not really)

Alexnet (simplified): The 3D blocks show activations with C, H, W dimensions.
N, the minibatch dimension, is not shown.



Backprop recipes: Matrix Multiply, Affine or Fully-Connected (FC) Layer

Assume input vector x of dimension n, output y of dimension m, layer $y = Wx$

$$y_i = \sum_{j=1}^{n} W_{ij} x_j$$

Then

$$J_y(x) = \frac{\partial y_i}{\partial x_j} = W_{ij}$$

and so

$$J_L(x) = J_L(y) W$$

Also we want to backprop to the weight matrix and compute:

$$\frac{\partial y_i}{\partial W_{jk}}$$

Which is a tensor (ouch), but notice that its zero unless $i = j$

Backprop recipes: Matrix Multiply, Affine or Fully-Connected (FC) Layer

We want: $\frac{\partial y_i}{\partial W_{jk}}$

Which is a tensor, but notice that its zero unless $i = j$

So it simplifies to $\frac{\partial y_i}{\partial W_{ik}} = x_k$

And if we already know the output loss Jacobian (from backprop): $J_L(y)$, then

$$\frac{\partial L}{\partial W_{ik}} = \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial W_{ik}} = J_L(y)_i x_k$$

Or more compactly $\boxed{J_L(W) = \frac{\partial L}{\partial W_{ik}} = \underbrace{J_L(y)^T}_{\text{Col vector}}\underbrace{x^T}_{\text{Row vector}}}$

Data and Model Derivatives

So far we've seen two distinct types of jacobians:

- Derivatives of loss wrt an input vector (called the “data” path)
- Derivatives of loss wrt to model parameters (called the “model” path)

These are treated differently across samples in a minibatch.

- Along the data path, samples are treated separately
- Along the model path, sample contributions are added to compute the gradient.

This just “works” if we treat the jacobians $J_L(X)$ as BxK matrices, where B is the batch size and K is the dimension of X.

Matrix Multiply, Affine or Fully-Connected (FC) Layer

Assume input vector x of dimension n, output y of dimension m, layer $y = Wx$

$$y_i = \sum_{j=1}^{n} W_{ij} x_j$$

The minibatch adds rows to these jacobians

Then the data derivative is

$$J_y(x) = \frac{\partial y_i}{\partial x_j} = W_{ij}$$ and so $$J_L(x) = J_L(y)W$$

For the model derivative we have:

$$J_L(W) = J_L(y)^T x^T$$

The minibatch adds columns+ rows inside this product, but the model jacobian size is fixed

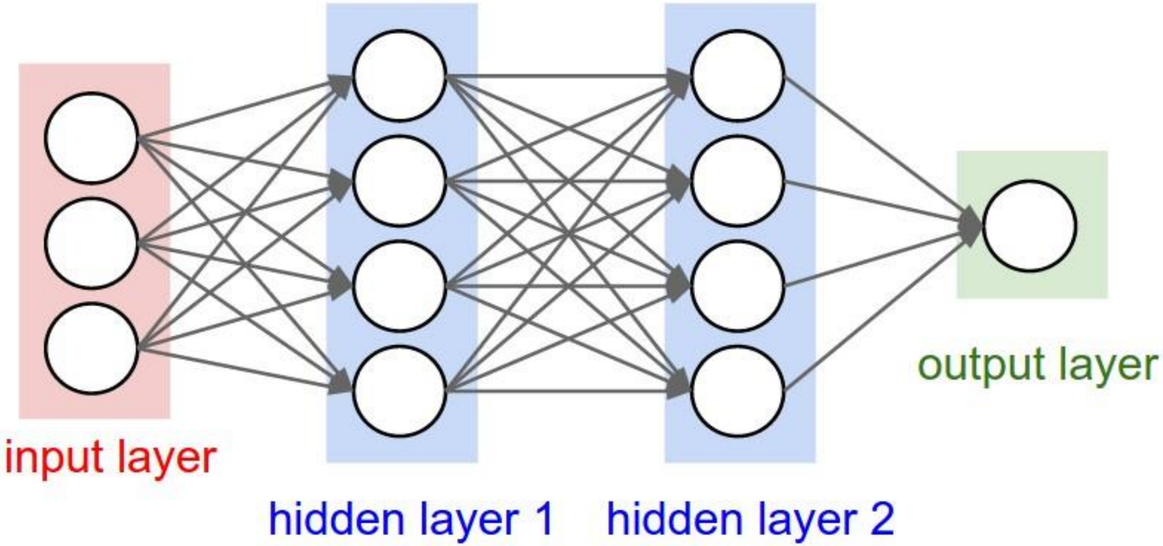
Across a minibatch of size B, we change the rows of any loss jacobian $J_L(z)$ to B.

We also change the number of columns of data blocks x to be B.

Backpropagation

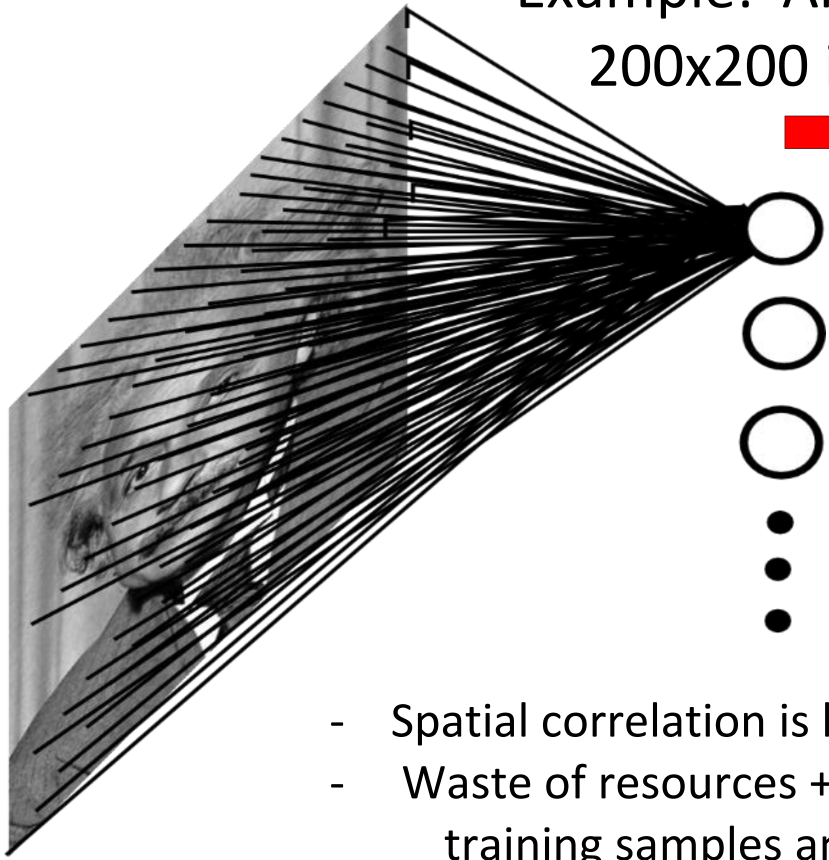
- Compute function values (activations) from the first layer to the last.
- Compute derivatives of the loss wrt other layers from the last layer to the first (backpropagation).
- This only requires matrix-vector multiplies.
- Paths from the loss layer to inner layers are re-used.

This time: Neural Networks for Visual Data



Fully Connected Layer for Visual Data

Example: An affine (FC) layer from 200x200 images to 40K hidden units

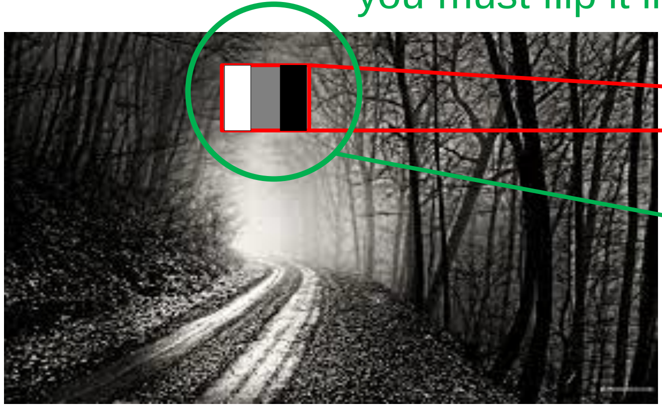


~2B parameters!!!

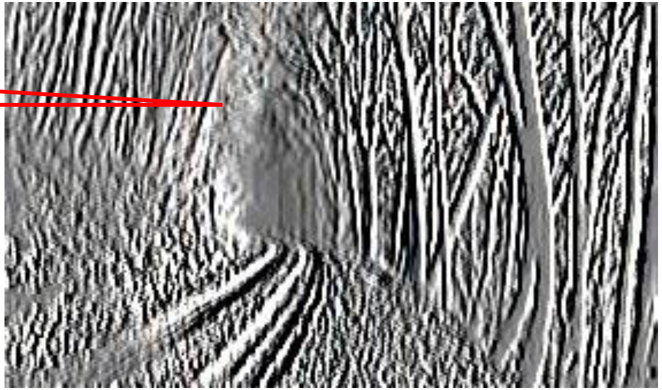
- Spatial correlation is local
- Waste of resources + we have not enough training samples anyway..

Convolutional of Two Signals

Note: to implement convolution with a sliding mask,
you must flip it in x and y because of the negated i,j indices



$$* \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} =$$



$$g(x,y) = (h * f)(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(i,j) f(x-i, y-j)$$

- elementwise multiplication and sum of a filter and the signal (image)

Convolutional of Two Signals

The convolution formula is:

$$(h * f)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(i, j) f(x - i, y - j)$$

It follows (by redefining indices) that convolution is commutative:

$$h * f = f * h$$

Finite filters: Typically one function, the "filter" (here its h) is defined over a finite support $[-w/2, w/2]^2$, then

$$(h * f)(x, y) = \sum_{i=-w/2}^{w/2} \sum_{j=-w/2}^{w/2} h(i, j) f(x - i, y - j)$$

Correlation of Two Signals

The ***correlation*** formula is:

$$(h * f)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(i, j) f(x + i, y + j)$$

NOTE: Convolution and correlation are widely confused in deep learning toolkits and descriptions on web sites.

Many online descriptions of convolution forget to invert the i,j indices and actually represent correlation operations. Also many toolkits actually implement correlation instead of convolution. E.g. the convolution layers in assignment 1 are actually correlations.

This is OK for convolutional networks which learn the filter weights, because forward and backward passes will be defined consistently. But it can cause trouble when transferring models between toolkits.

Image filtering

$g = h * f$

$h[\cdot\,,\cdot\,]\ \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$

$f[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g[.,.]$

Credit: S. Seitz

Image filtering

$$g = h * f$$

$$h[\cdot\,,\cdot\,]\ \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

$f\,[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g[.,.]$

	0	10							

Credit: S. Seitz

Image filtering

$$g = h * f$$

$$h[\cdot,\cdot]\ \frac{1}{9}$$

1	1	1
1	1	1
1	1	1

$f[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g[.,.]$

	0	10	20						

Credit: S. Seitz

Image filtering

$$g = h * f$$

$h[\cdot\,,\cdot\,]\ \frac{1}{9}$

1	1	1
1	1	1
1	1	1

$f\,[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g\,[.,.]$

	0	10	20	30					

Credit: S. Seitz

Image filtering

$$g = h * f$$

$h[\cdot\ ,\cdot\ ]\ \frac{1}{9}$

1	1	1
1	1	1
1	1	1

$f[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g[.,.]$

	0	10	20	30	30				

Credit: S. Seitz

Image filtering

$$g = h * f$$

$$h[\cdot\,,\cdot\,]\ \frac{1}{9}$$

1	1	1
1	1	1
1	1	1

$f\,[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g[.,.]$

	0	10	20	30	30				
				?					

Credit: S. Seitz

Image filtering

$$g = h * f$$

$$h[\cdot\,,\cdot\,]\ \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$f[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g[.,.]$

	0	10	20	30	30				
						?			
				50					

Credit: S. Seitz

Image filtering

$g = h * f \quad h[\cdot\ ,\cdot\ ] \quad \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$

$f[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$g[.,.]$

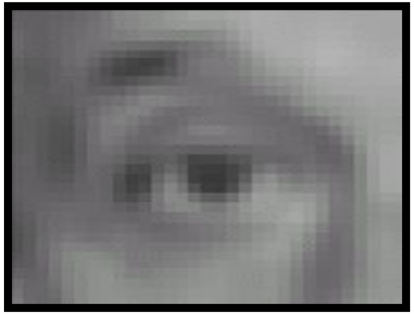
	0	10	20	30	30	30	20	10	
	0	20	40	60	60	60	40	20	
	0	30	60	90	90	90	60	30	
	0	30	50	80	80	90	60	30	
	0	30	50	80	80	90	60	30	
	0	20	30	50	50	60	40	20	
	10	20	30	30	30	30	20	10	
	10	10	10	0	0	0	0	0	

Credit: S. Seitz

Linear filters: examples

Original

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} =$$



Blur (with a mean filter)

Source: D. Lowe

Practice with linear filters

Original

0	0	0
0	1	0
0	0	0

?

Source: D. Lowe

Practice with linear filters

Original

0	0	0
0	1	0
0	0	0

Filtered
(no change)

Source: D. Lowe

Practice with linear filters

Original

0	0	0
0	0	1
0	0	0

?

Source: D. Lowe

Practice with linear filters

0	0	0
0	0	1
0	0	0

Convolutional Filter Weights

Original

0	0	0
1	0	0
0	0	0

Sliding Mask (Correlation) Weights

Shifted right
By 1 pixel

Source: D. Lowe

Impulse response

Image

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

$*$

1	2	3
4	5	6
7	8	9

Convolutional Filter

$=$

0	0	0	0	0
0	1	2	3	0
0	4	5	6	0
0	7	8	9	0
0	0	0	0	0

Output

The impulse response (response to a single “1” pixel at the center of an image) should be a copy of the filter weights. True for convolution, not for correlation

Impulse response

Image

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

$*$

1	2	3
4	5	6
7	8	9

Convolutional Filter

9	8	7
6	5	4
3	2	1

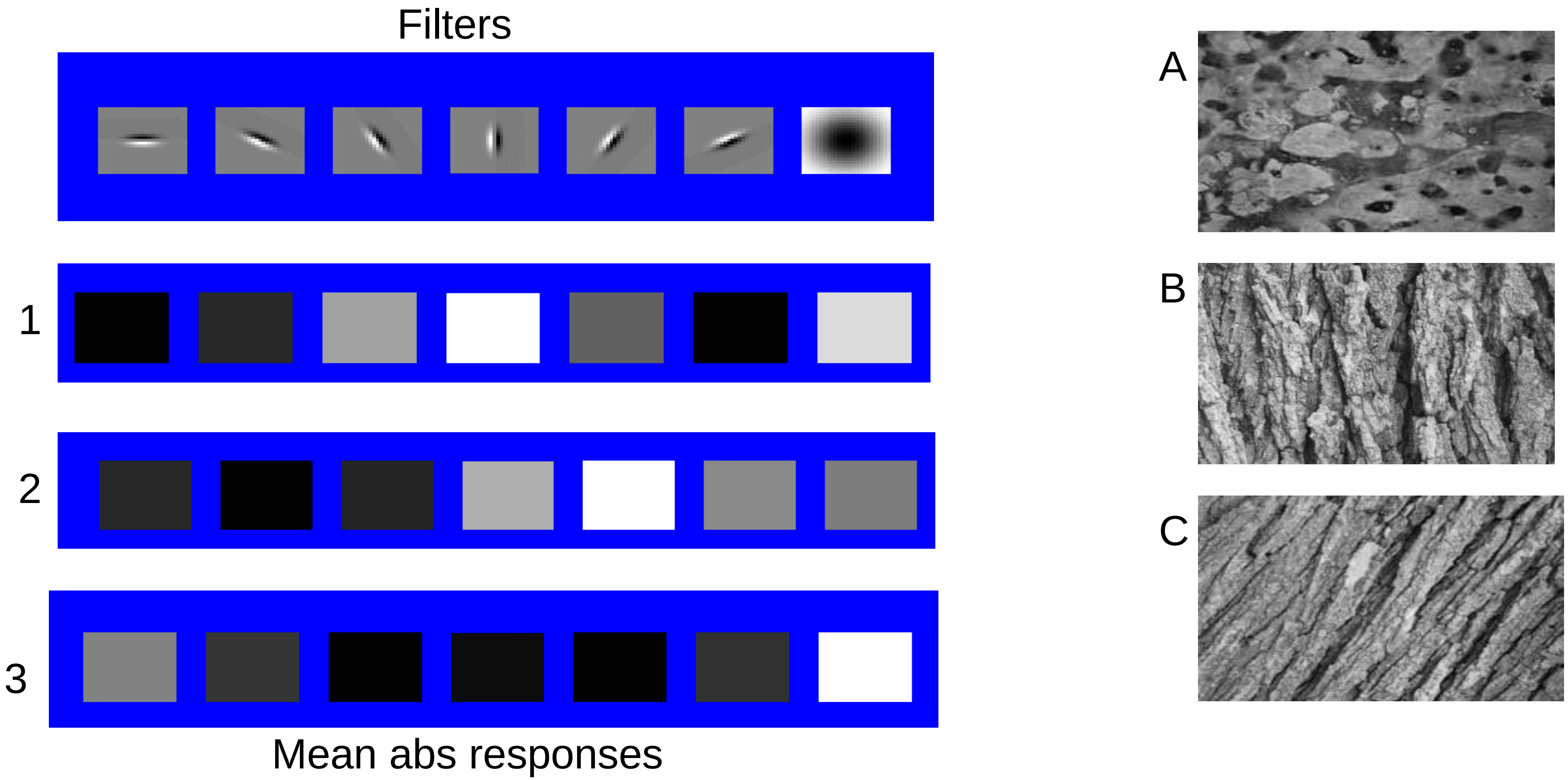
Sliding (Correlation) Mask

$=$

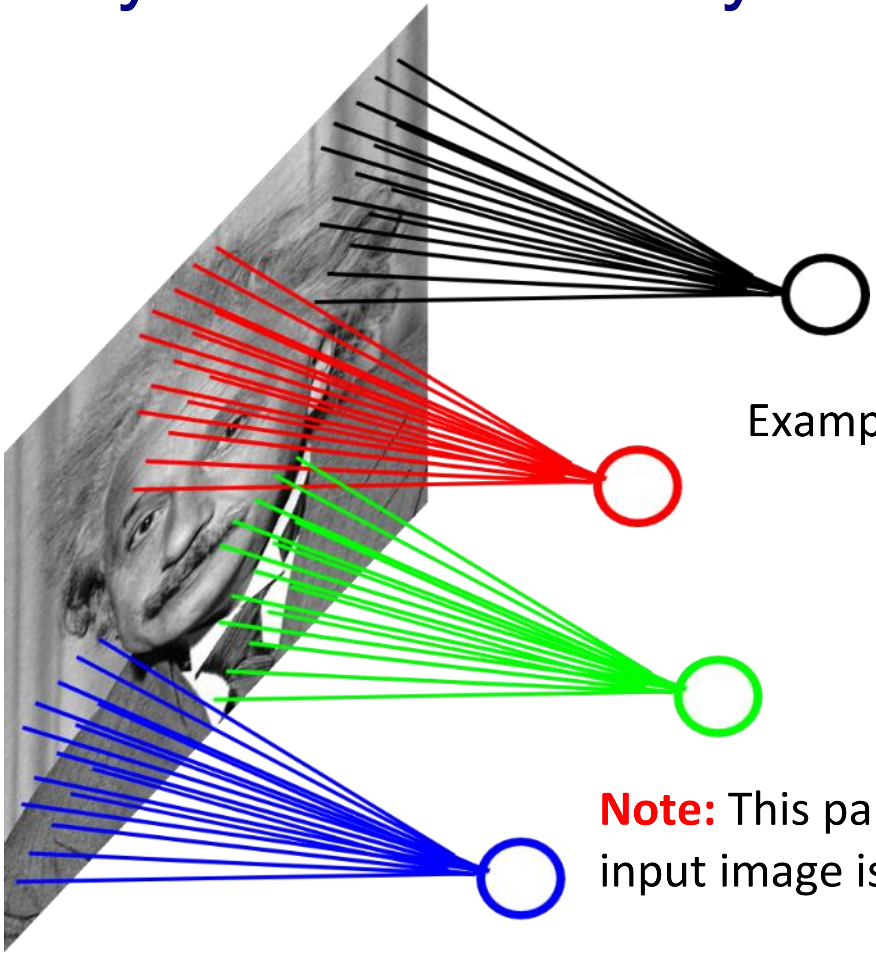
0	0	0	0	0
0	1	2	3	0
0	4	5	6	0
0	7	8	9	0
0	0	0	0	0

Output

Can you match the texture to the response?



Locally Connected Layers

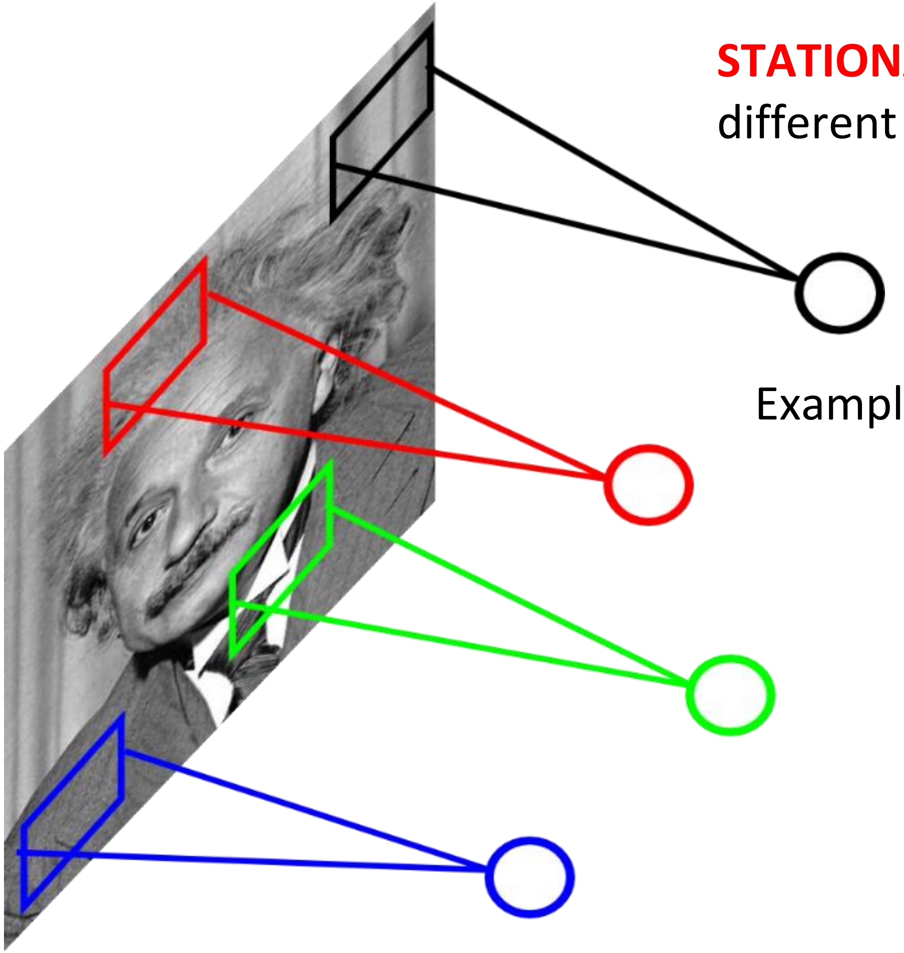


Example: 200x200 image
40K hidden units
Filter size: 10x10
→ 4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

Ranzato

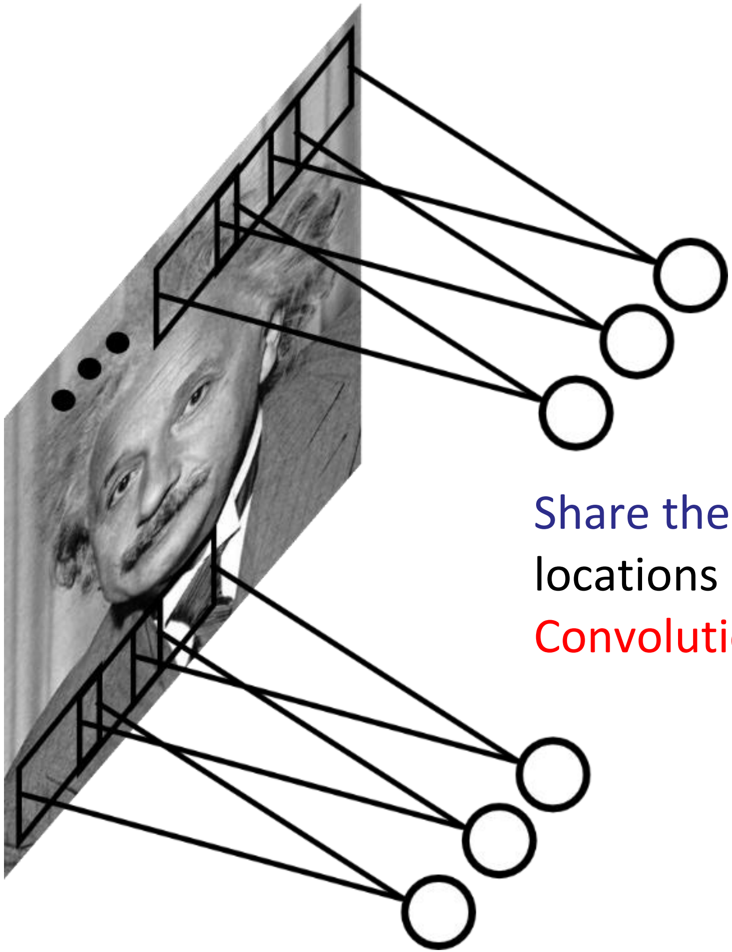
Locally Connected Layers



STATIONARITY? Statistics is similar at different locations

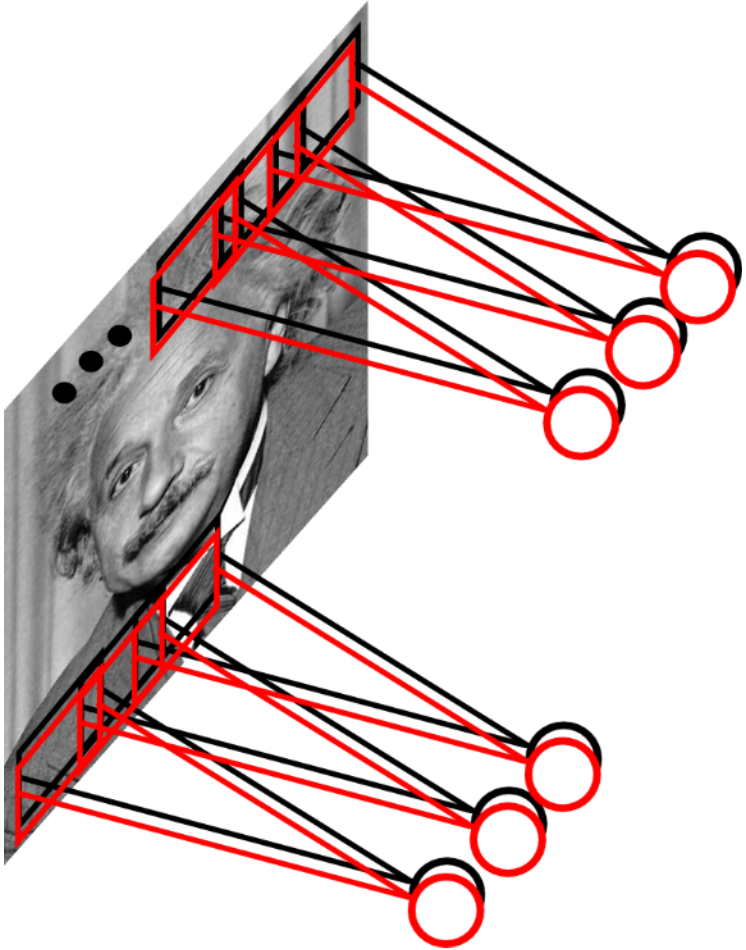
Example: 200x200 image
40K hidden units
Filter size: 10x10
→ 4M parameters

Convolutional Layer



Share the same parameters across different locations (assuming input is stationary): Convolutions with learned kernels

Convolutional Layer



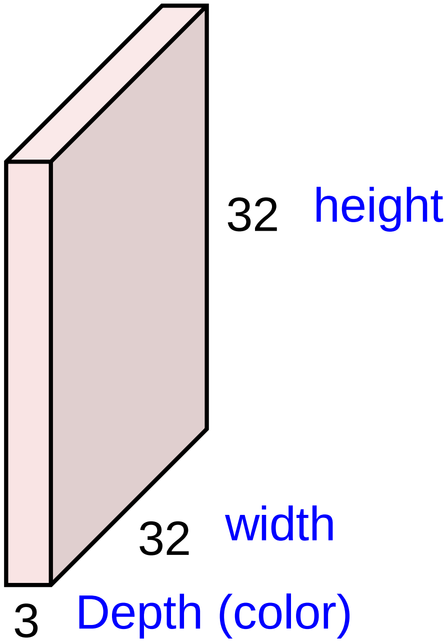
Learn multiple filters.

E.g.: 200x200 image
100 Filters
Filter size: 10x10
10K parameters

Ranzato

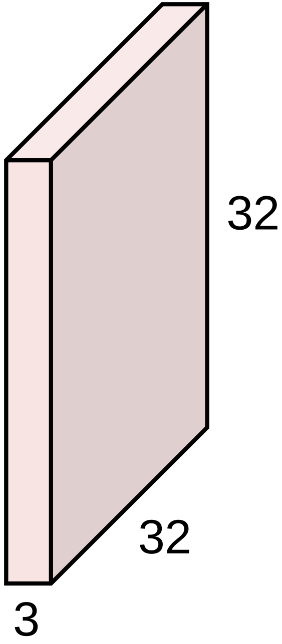
Convolution Layer

32x32x3 image

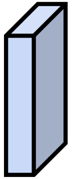


Convolution Layer

32x32x3 image



5x5x3 filter

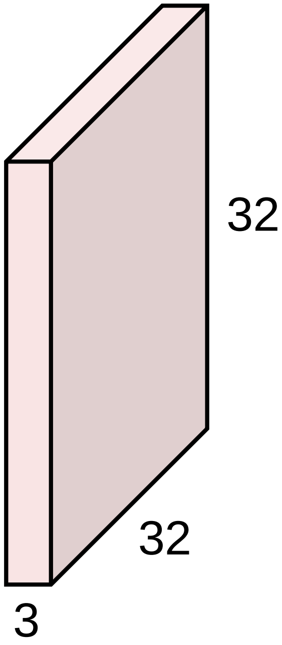


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

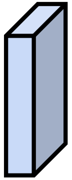
Convolution Layer

Filters always extend to the full depth of the input volume

32x32x3 image

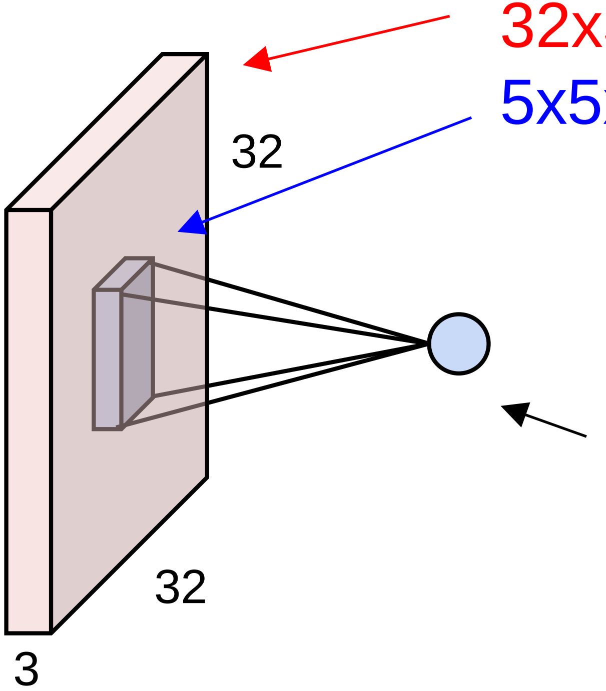


5x5x3 filter



Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

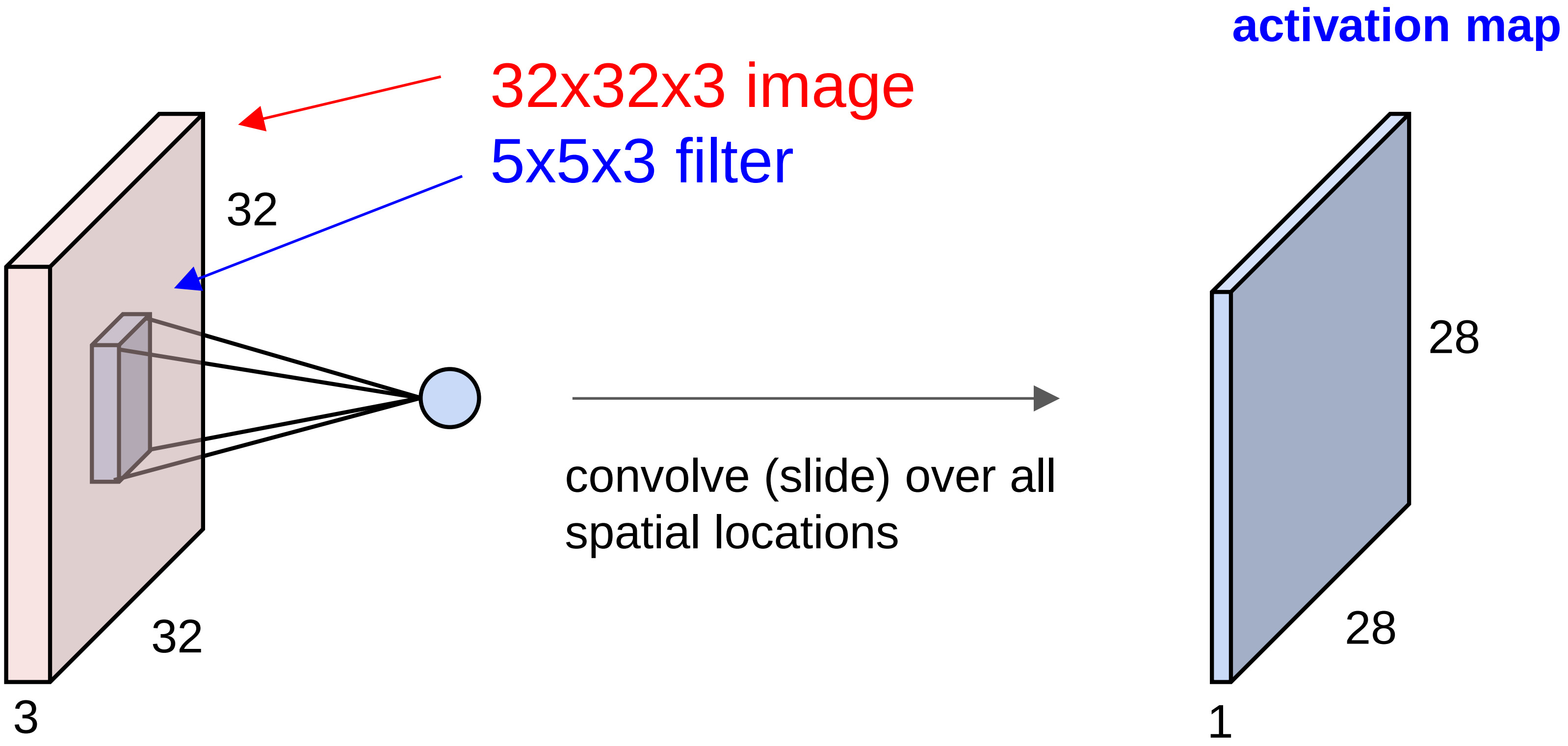
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

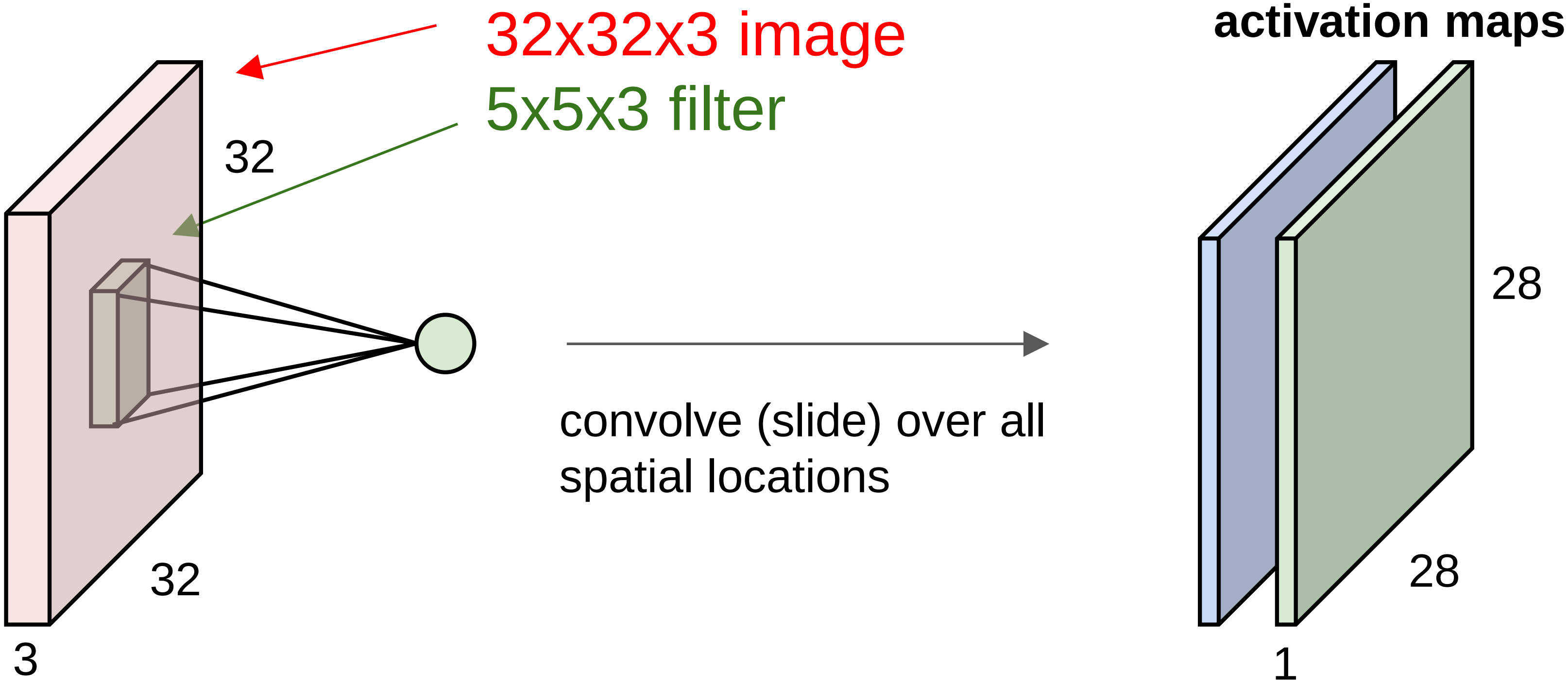
$$w^T x + b$$

Convolution Layer

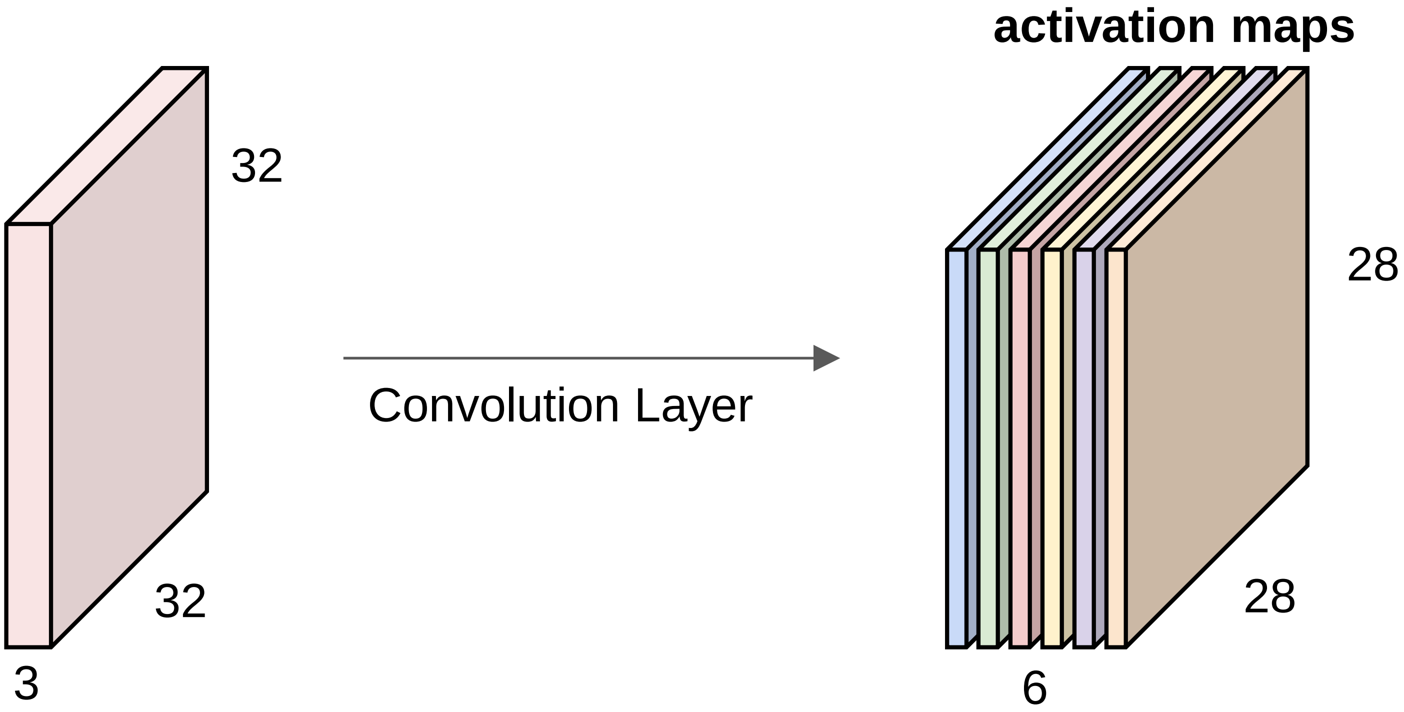


Convolution Layer

consider a second, green filter



For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



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

“Tensors” again

Because there are multiple channels in each data block, convolutional filters are normally specified by 4D arrays. Common dimensions are:

$$C_{out} \times C_{in} \times F_H \times F_W$$

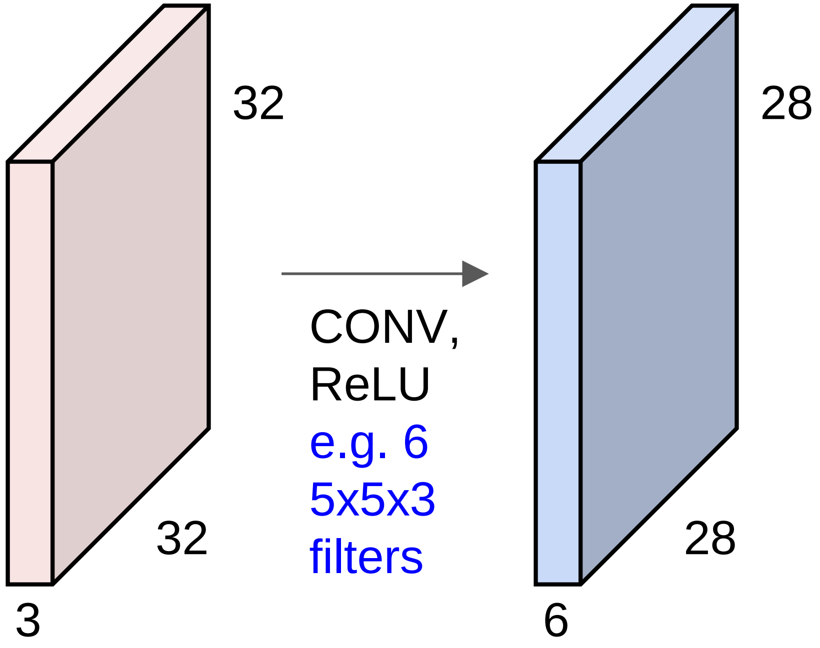
C_{out} = number of output channels

C_{in} = number of input channels

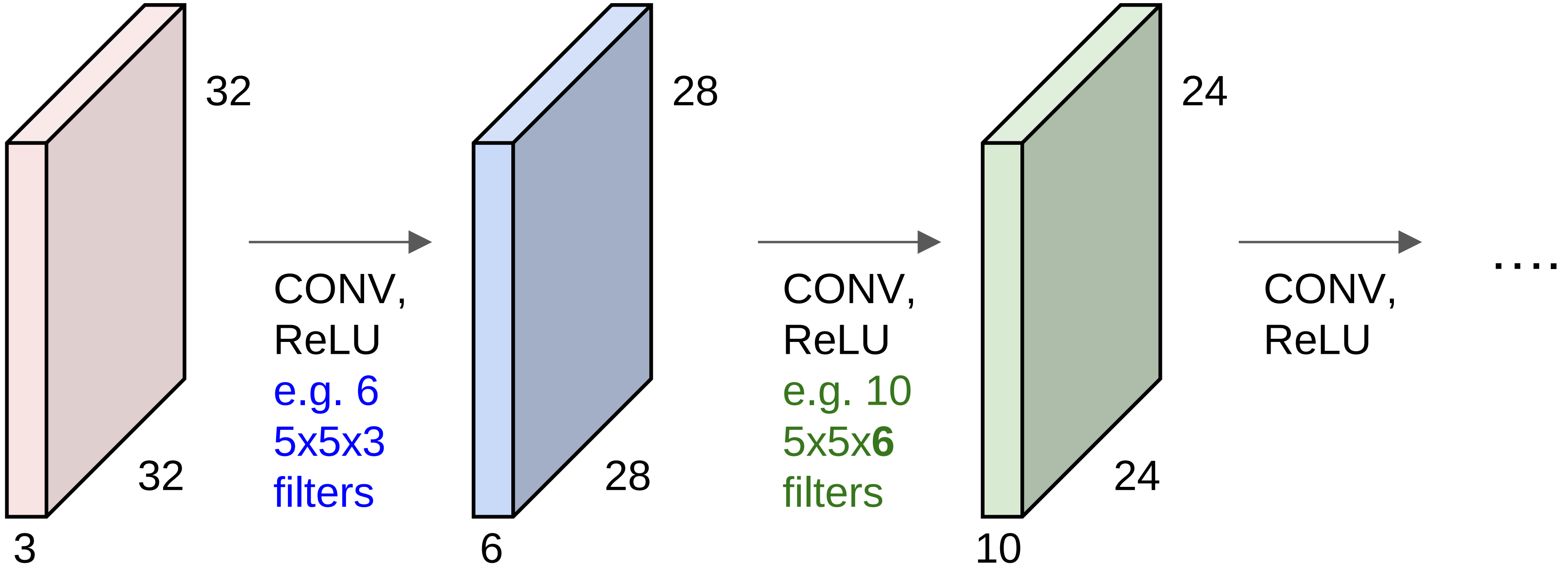
F_H = filter height

F_W = filter width

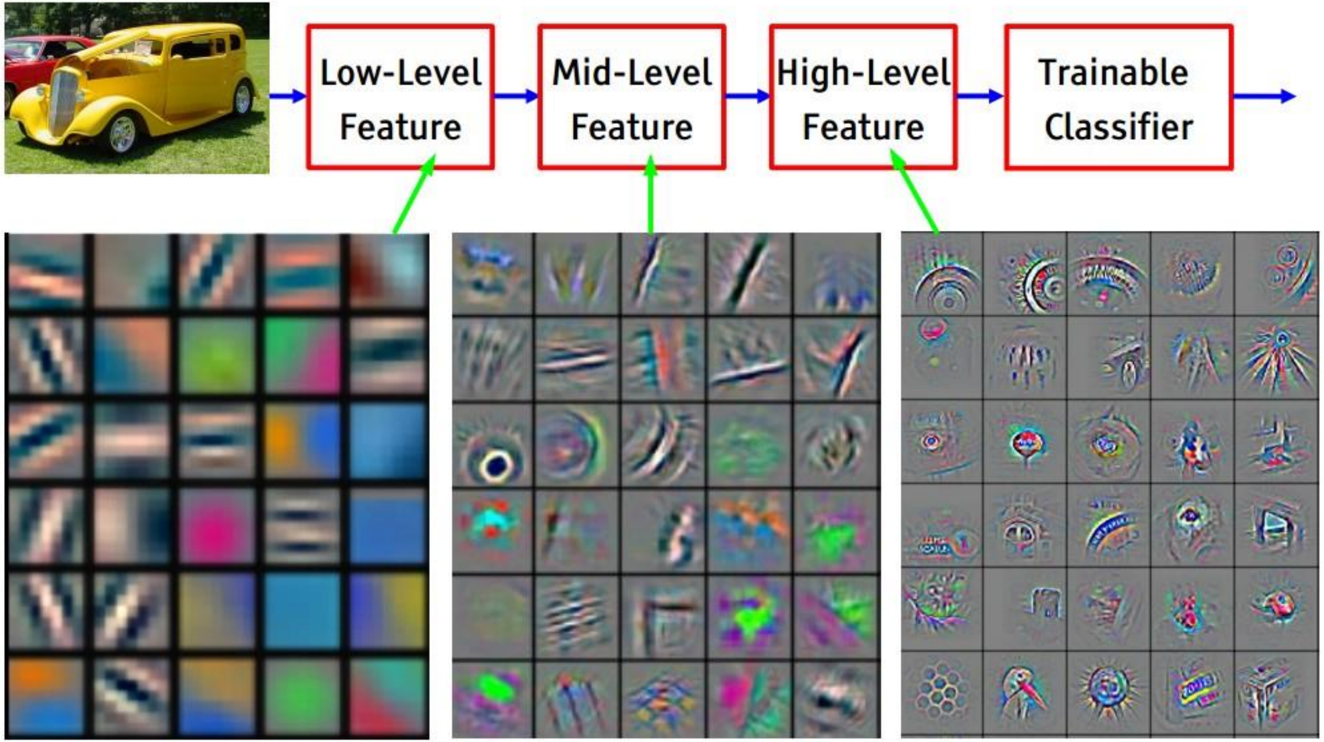
Preview: ConvNet is a sequence of Convolution Layers, interspersed with non-linear activation functions



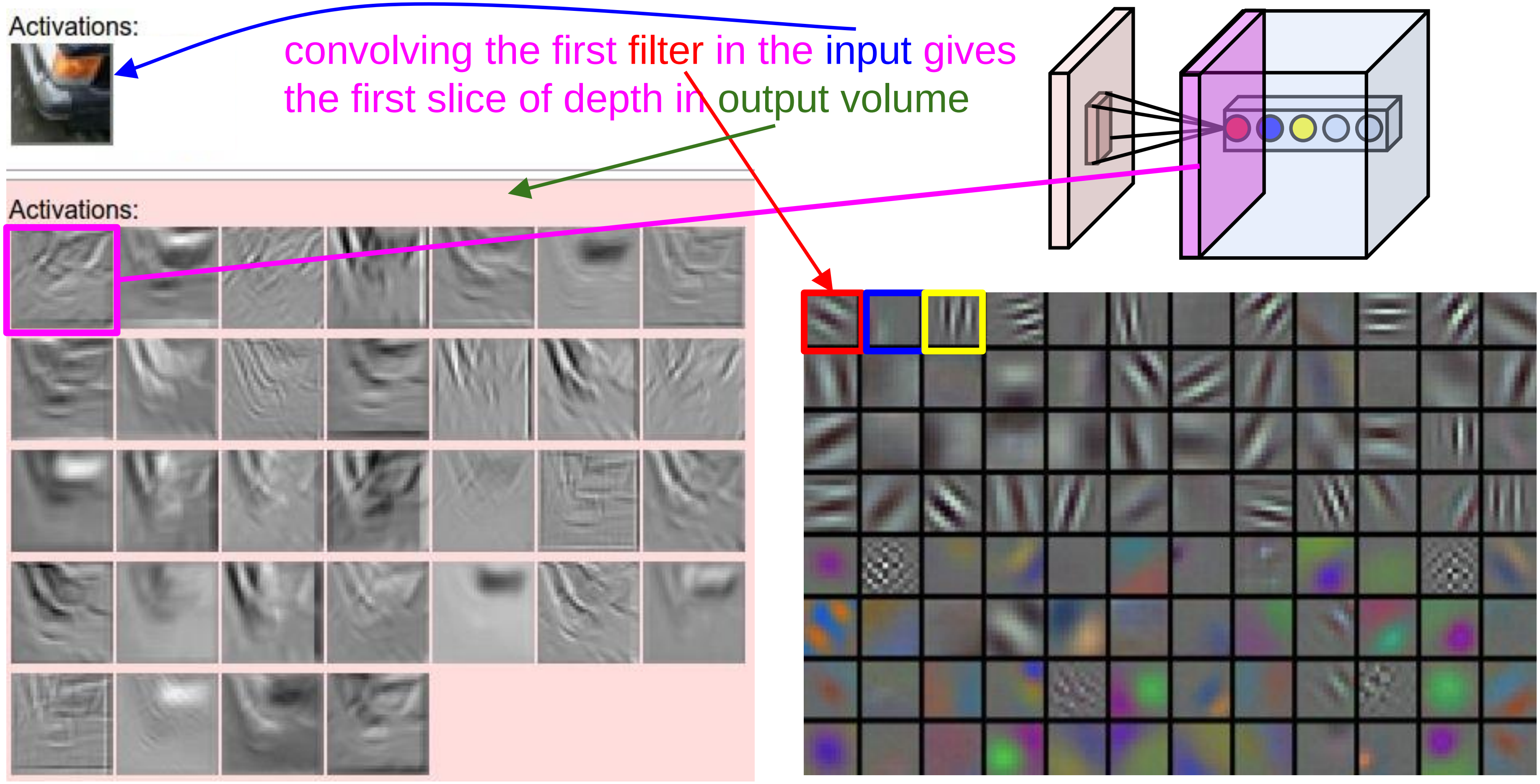
Preview: ConvNet is a sequence of Convolution Layers, interspersed with non-linear activation functions



Recall

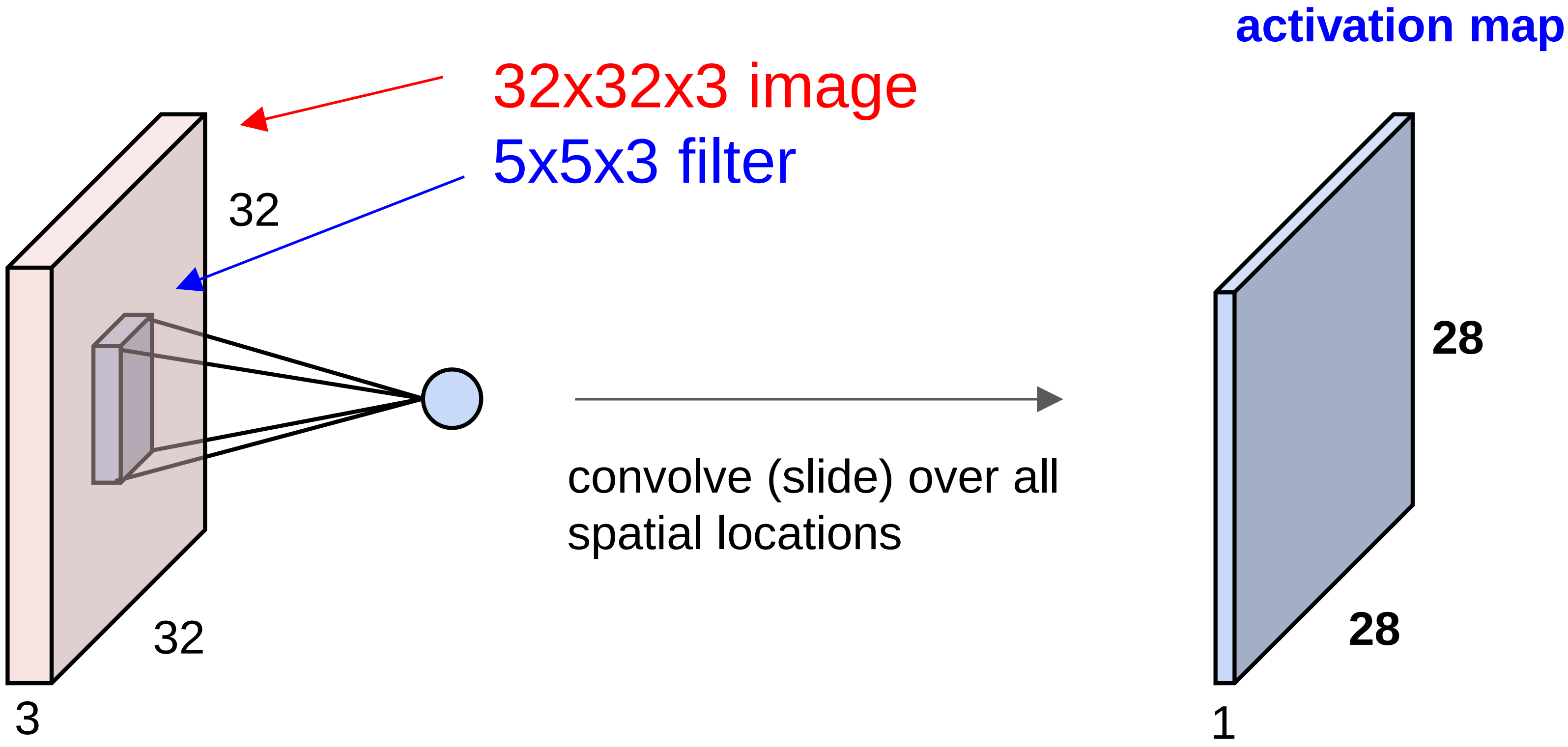


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

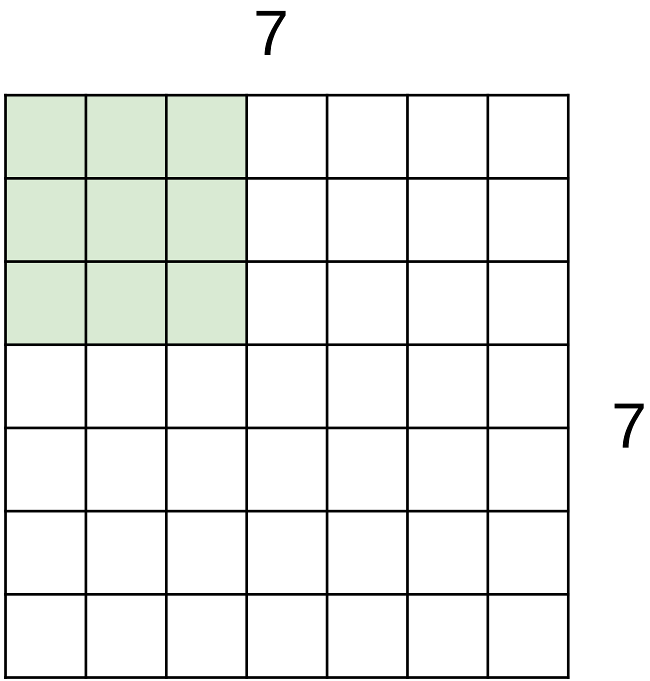


Slide based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

A closer look at spatial dimensions:

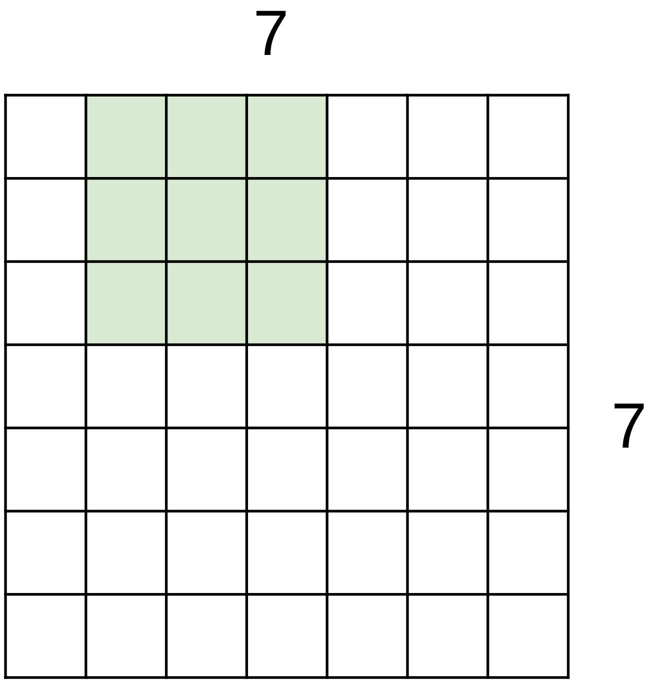


A closer look at spatial dimensions:



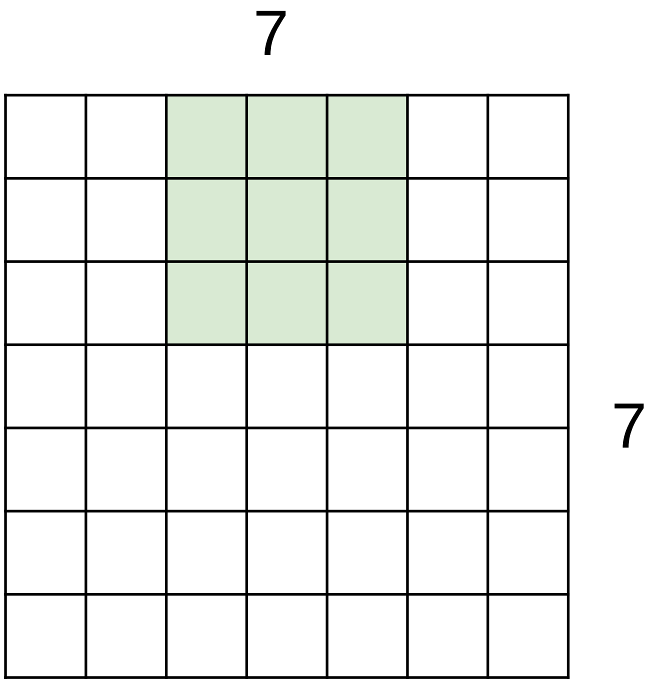
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



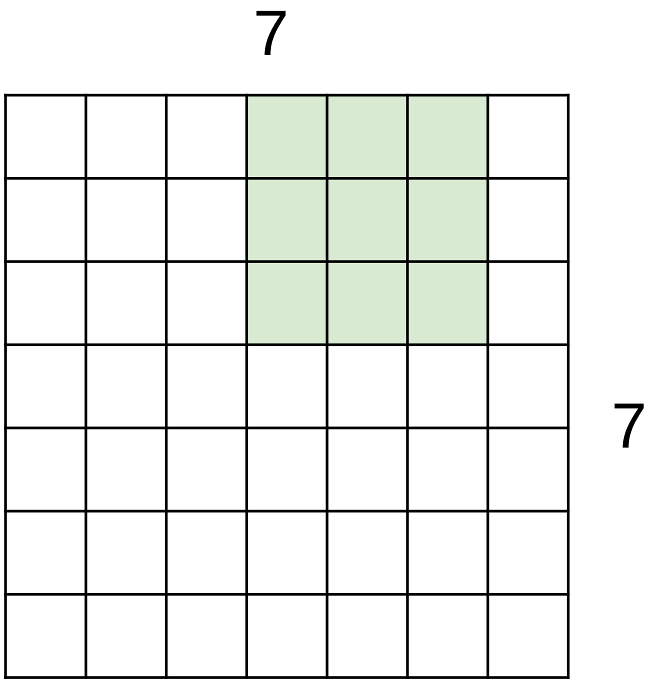
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



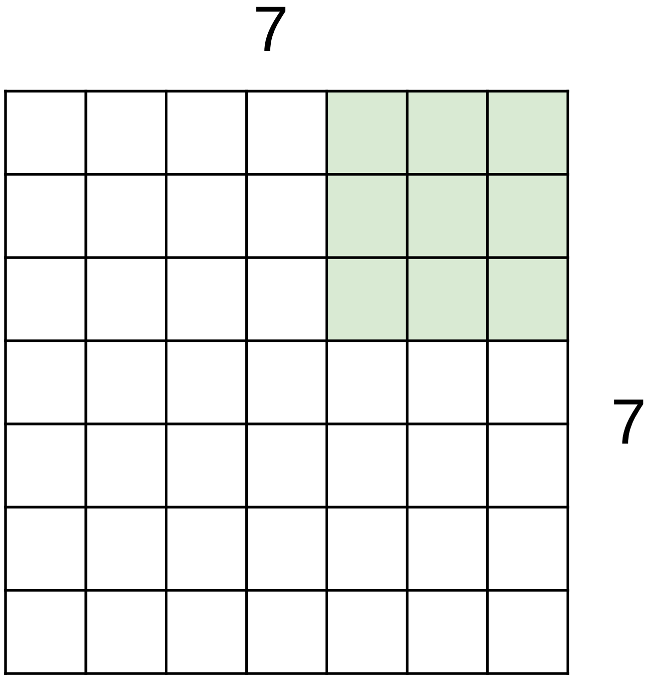
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

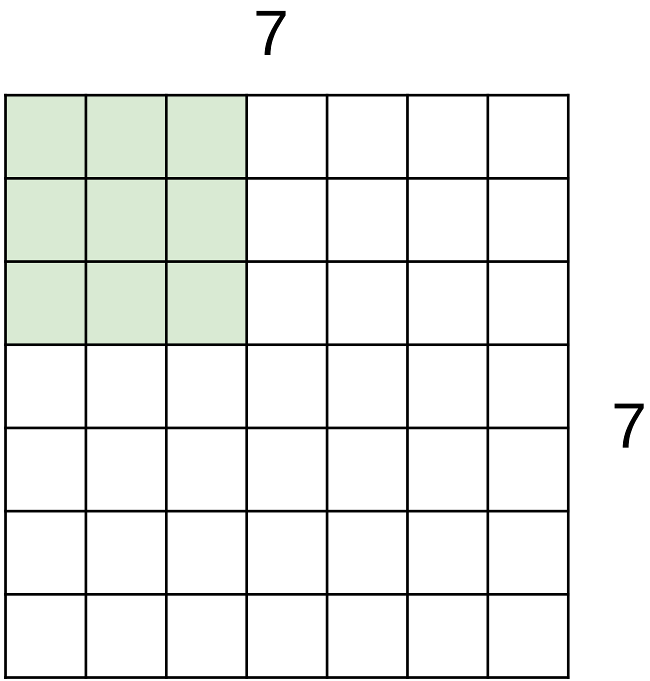
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

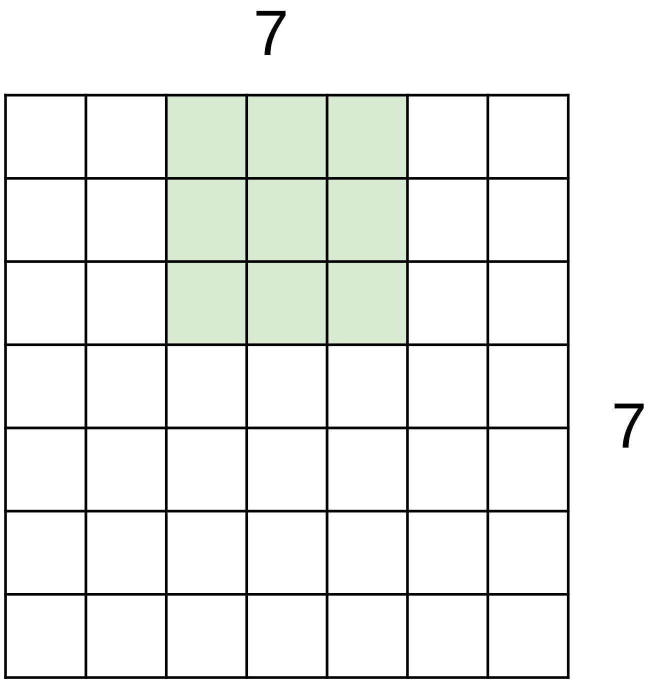
=> 5x5 output

A closer look at spatial dimensions:



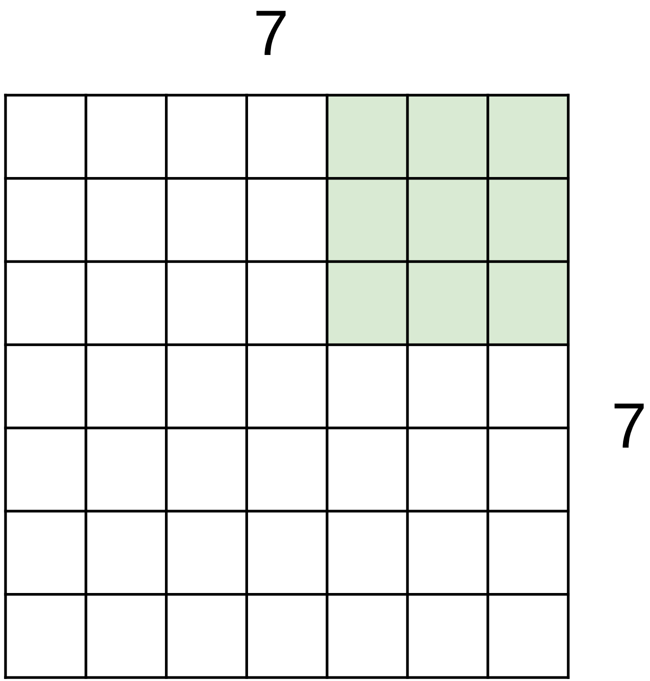
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



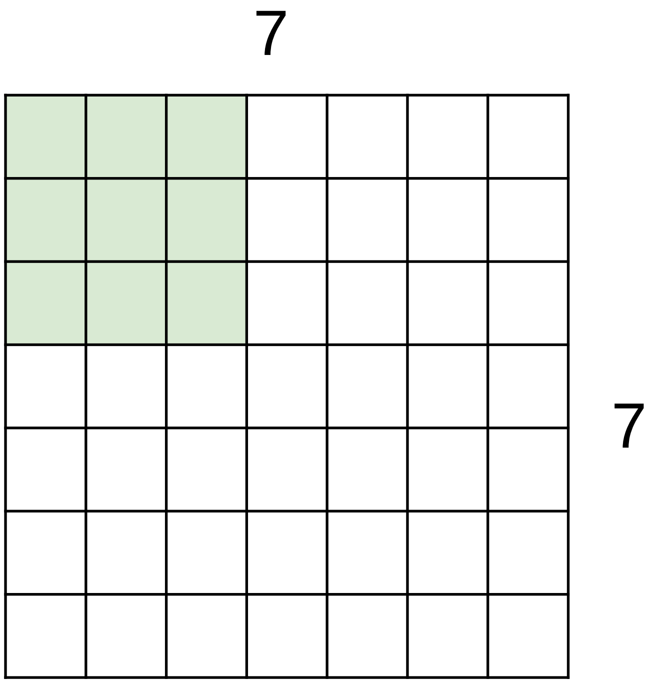
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



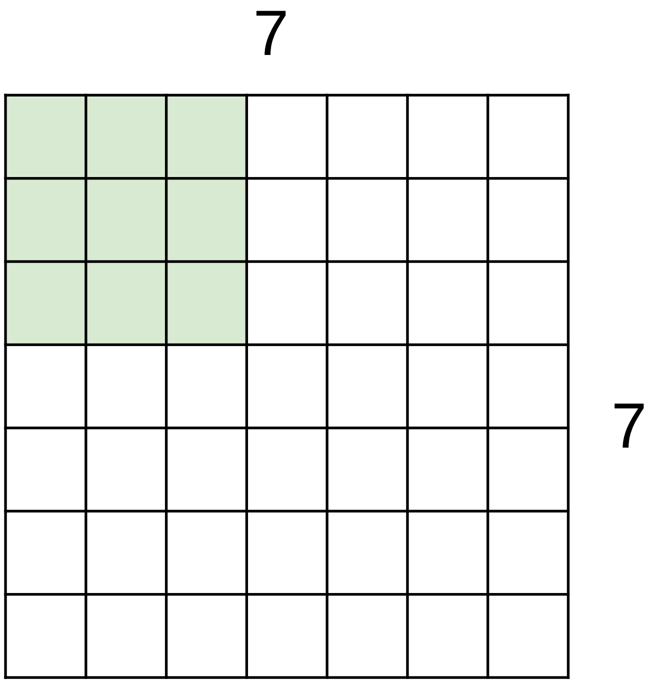
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



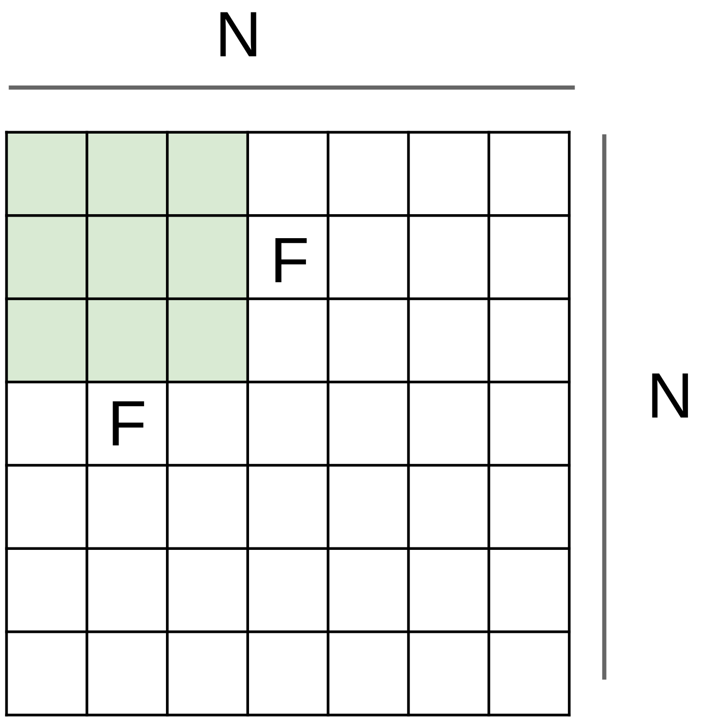
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



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

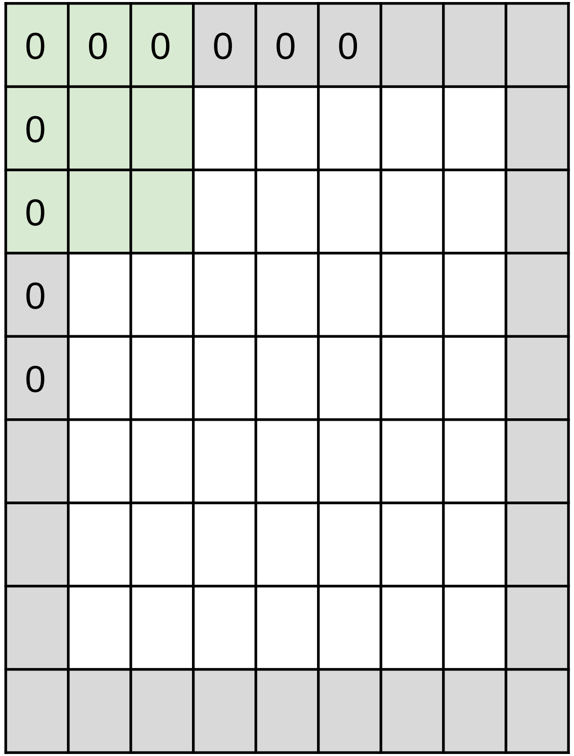
doesn't fit!
Wastes some input pixels.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

In practice: Common to zero pad the border

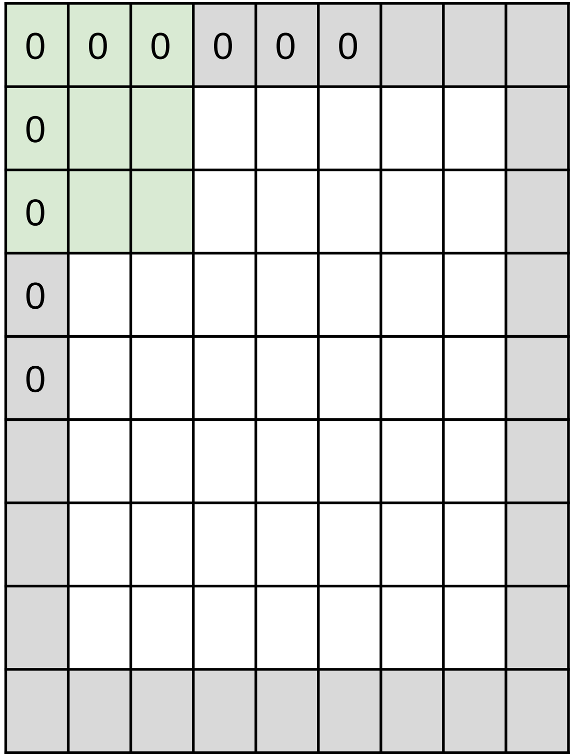


Aside: remove image mean first!

e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

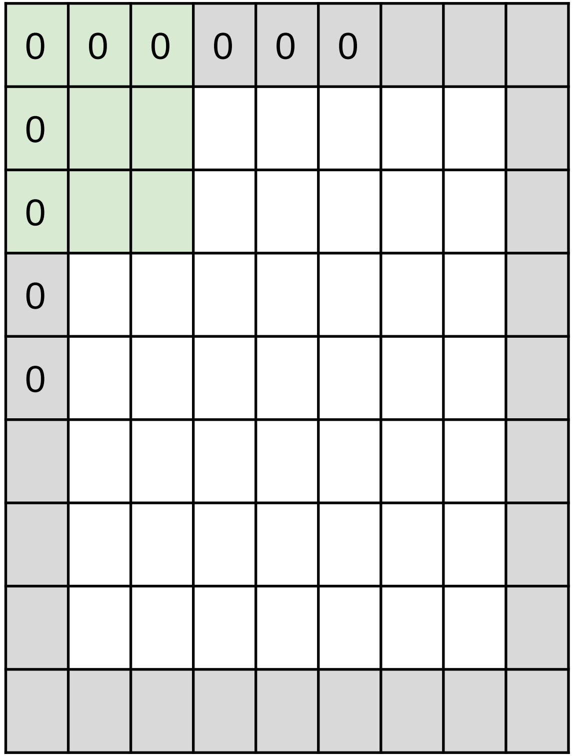
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

In practice: Common to zero pad the border

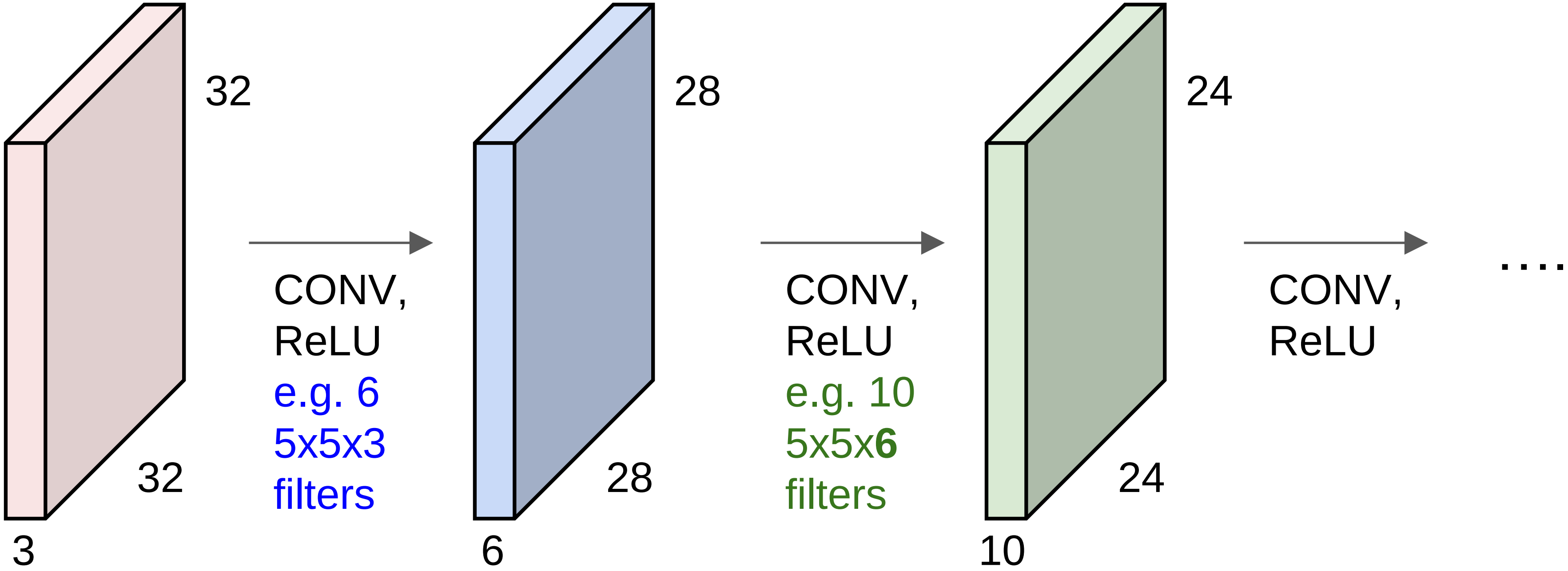


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



When the Filter doesn't fit:

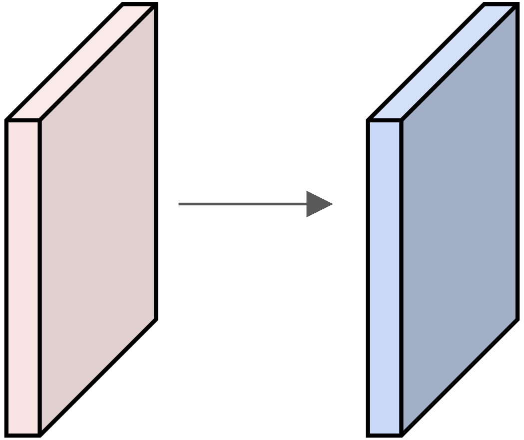
Our new formula with padding on both sides is:

$$D_{out} = (N + 2P - F)/\mathrm{stride} + 1$$

(the effective image size is now $N + 2P$).
This means that the number of strided steps we can take may not be an integer for $S > 1$.
But we can still take the next-smallest integer number of steps.
In practice almost all deep learning toolkits simply round down to the nearest integer in this case (disregard the cs231n notes on this point). i.e.

$$D_{out} = \lfloor (N + 2P - F)/\mathrm{stride} \rfloor + 1$$

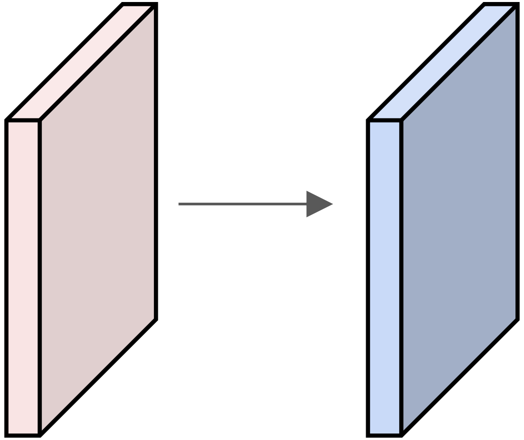
Examples time:



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

Output volume size: ?

Examples time:



Input volume: **32x32x3**

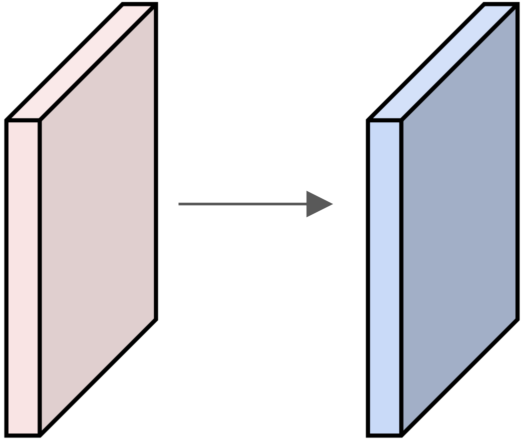
10 5x5 filters with stride 1, pad 2

Output volume size:

(32+2*2-5)/1+1 = 32 spatially, so

32x32x10

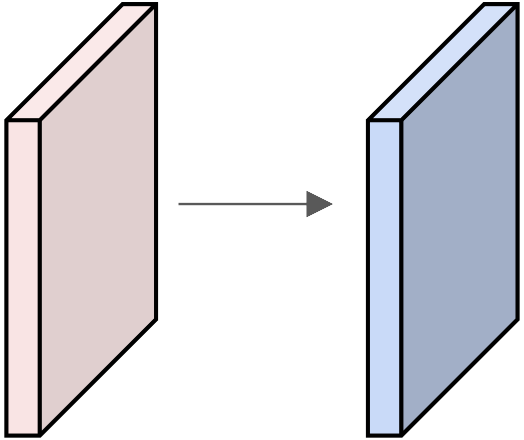
Examples time:



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

Number of parameters in this layer?

Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

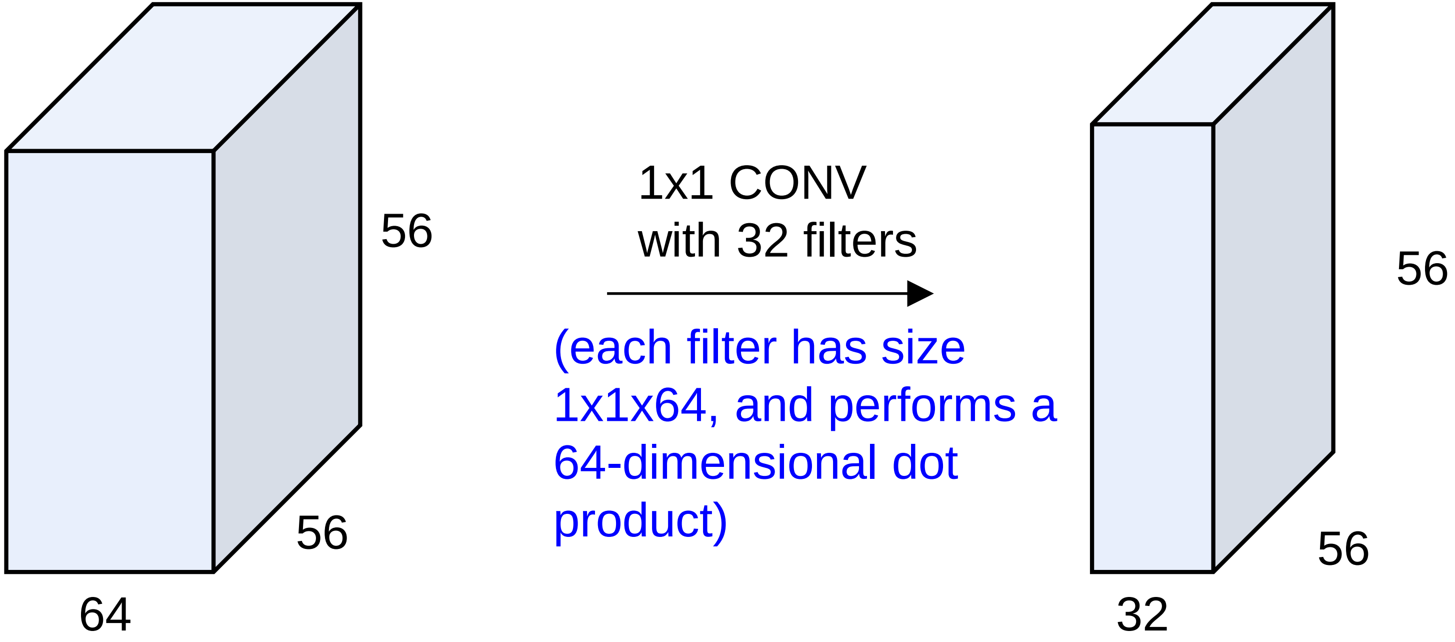
each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**

Summary. To summarize, the Conv Layer:

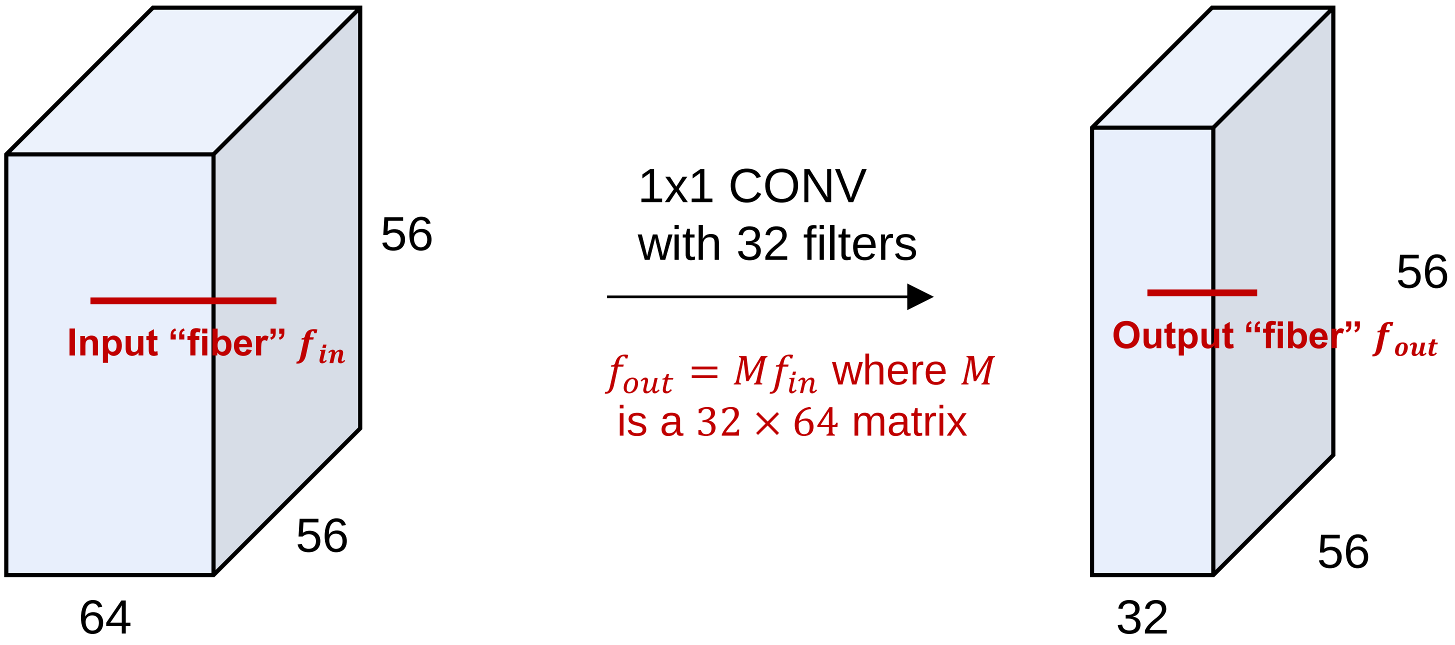
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

(btw, 1x1 convolution layers make perfect sense)

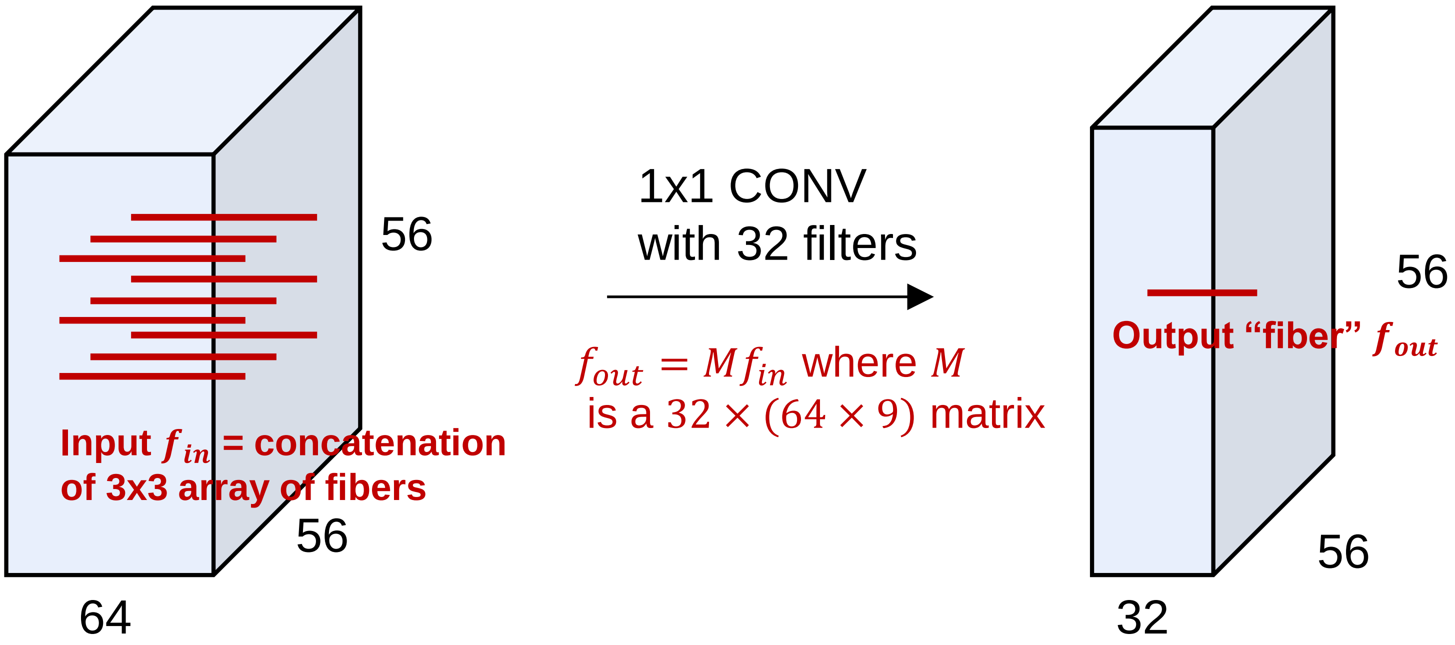


Slide based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

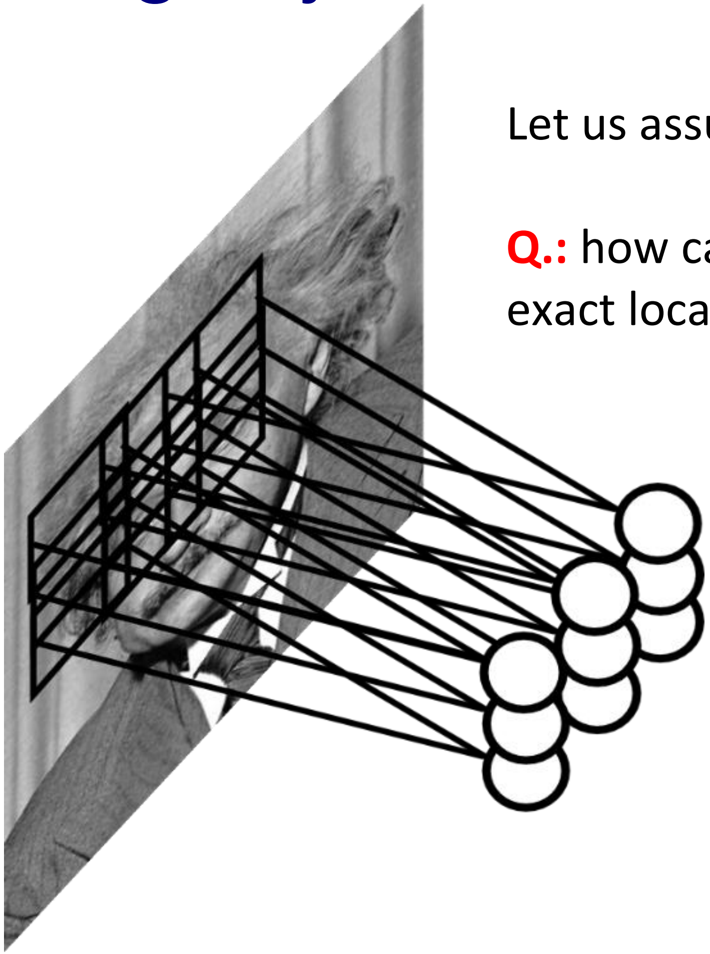
(btw, 1x1 convolution layers make perfect sense)



Aside: convolution via matrix multiply: im2col



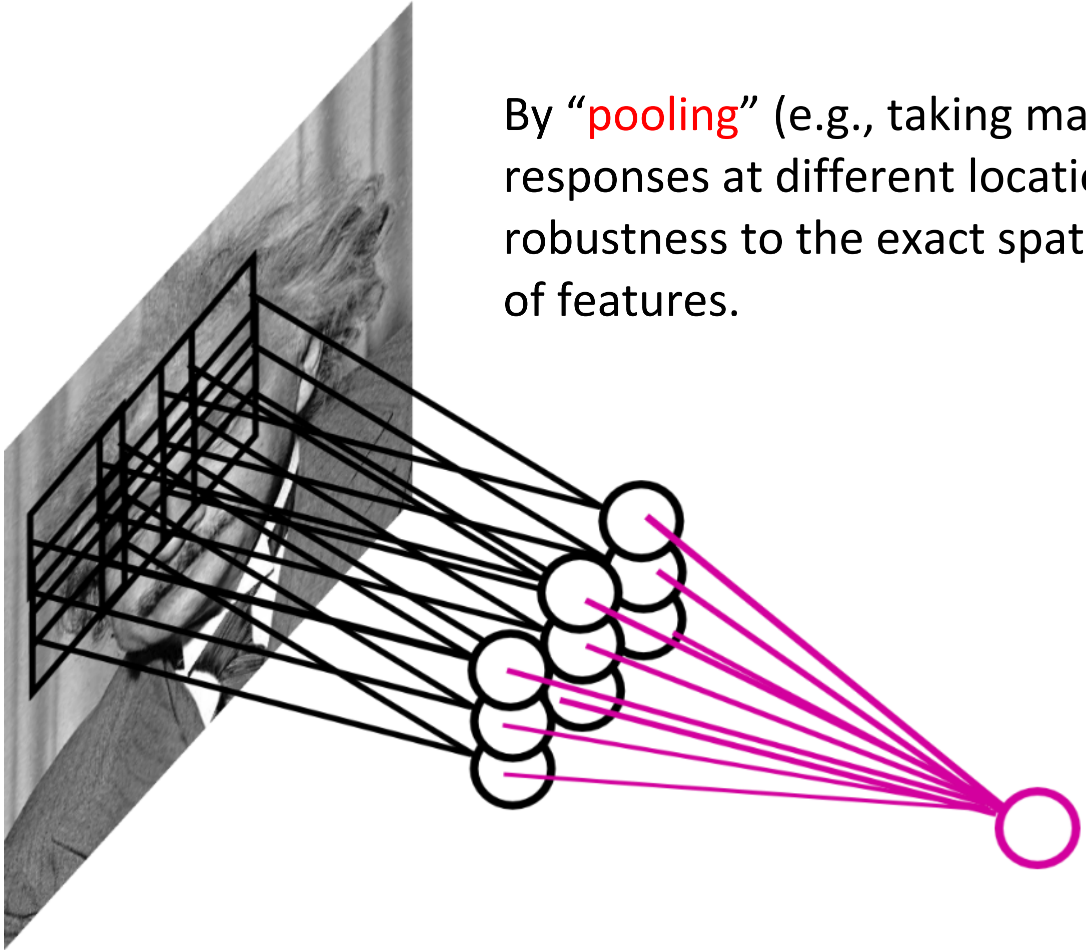
Pooling Layer



Let us assume filter is an “eye” detector.

Q.: how can we make the detection robust to the exact location of the eye?

Pooling Layer

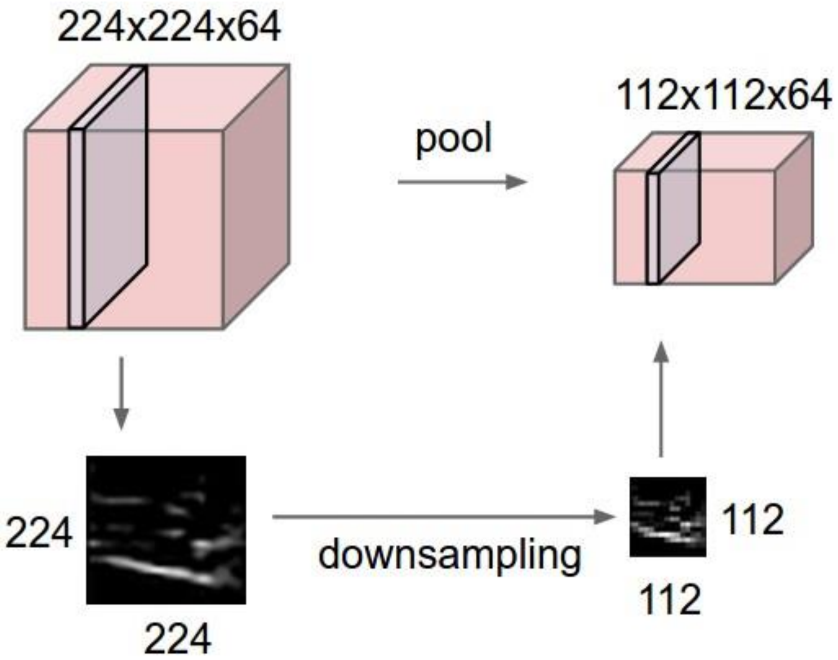


By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

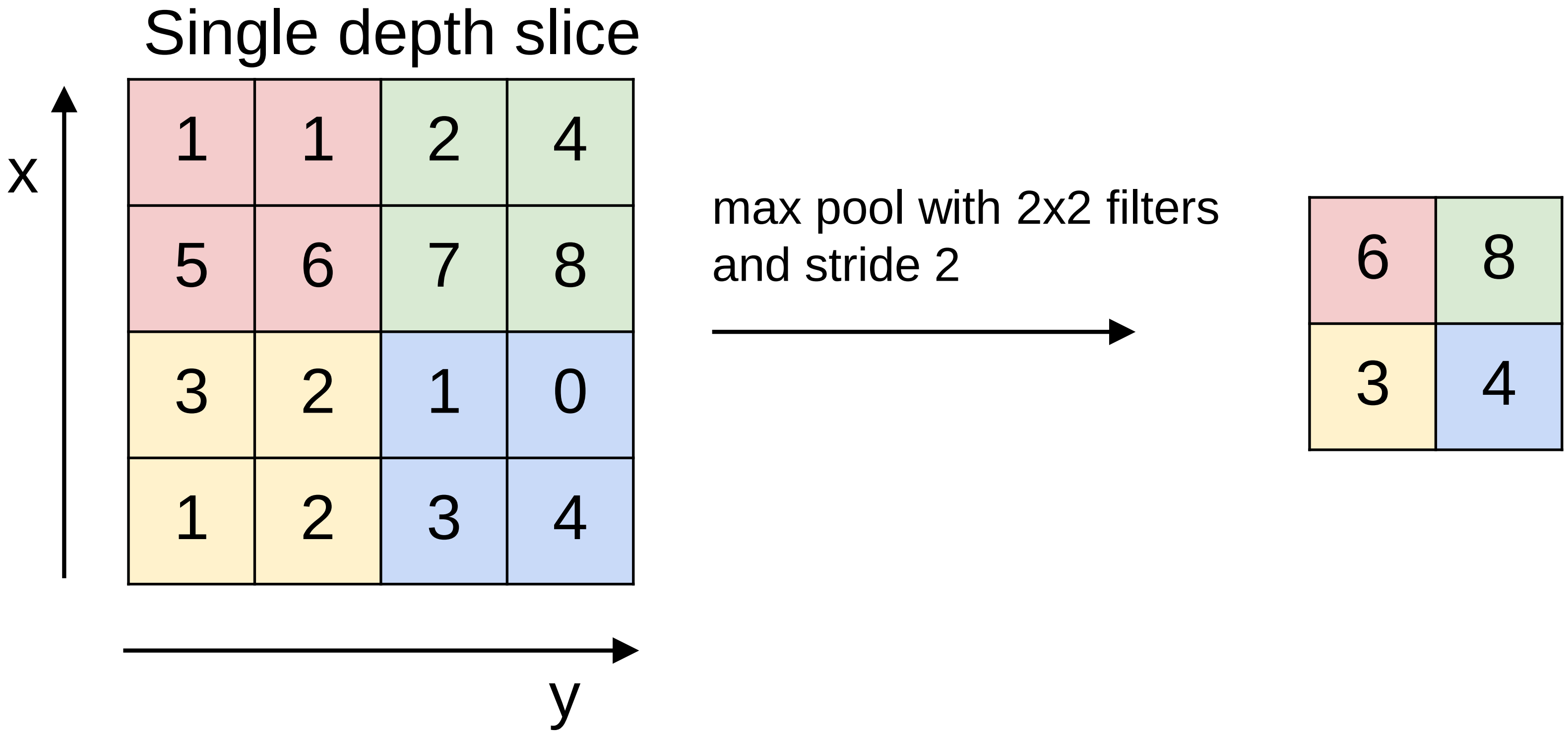
Ranzato

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



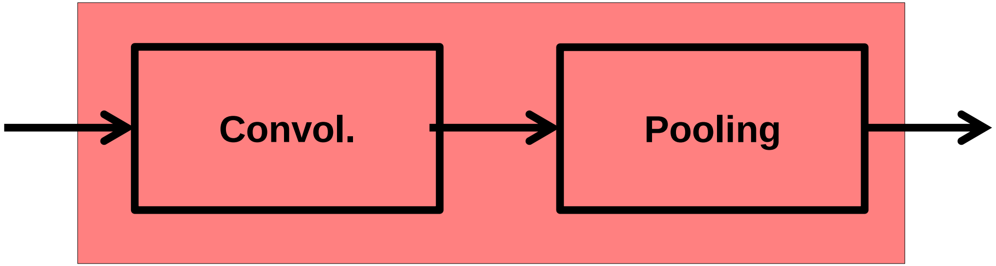
MAX POOLING

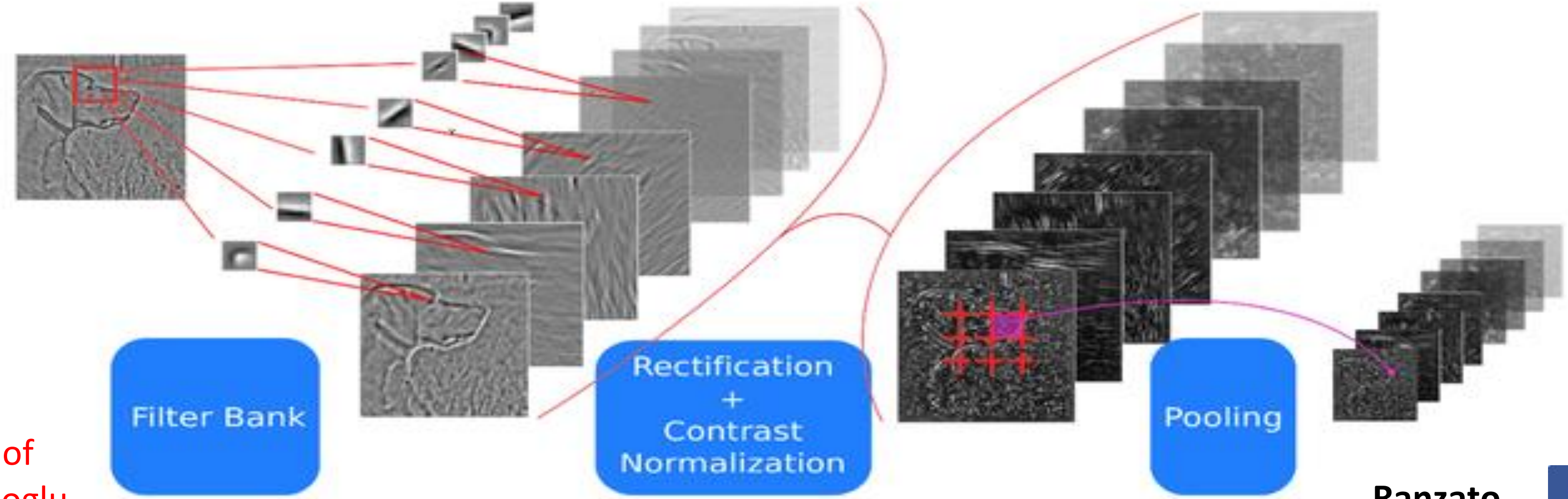


Slide based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

ConvNets: Typical Stage

One stage (zoom)





courtesy of
K. Kavukcuoglu

Convnets Summary

- Convolution layers represent local, shift-invariant operations on data blocks.
- Layer activations commonly organized into 4D blocks, NCHW or NHWC.
- Convolutional filters are also usually 4D arrays, e.g. $C_{out} C_{in} F_H F_W$
- Convnets typically follow conv layers with ReLU non-linearities.
- Spatial resolution decreases through the network, while number of channels increases.
- Pooling layers or strided convolutions reduce the spatial resolution in steps.