Machine Learning for Image Analysis

Lecture 3: Convolutional Neural Networks

Dagmar Kainmueller, 30.04.2024

Course Outline

- Introduction to Image Analysis
- Machine Learning Basics: Linear Regression, Basic Classifiers
- Neural Networks
- **Convolutional Neural Networks**
- Recurrent Neural Networks, Transformers, Diffusion
- Model Interpretability
- Probabilistic Machine Learning
- Generative Models

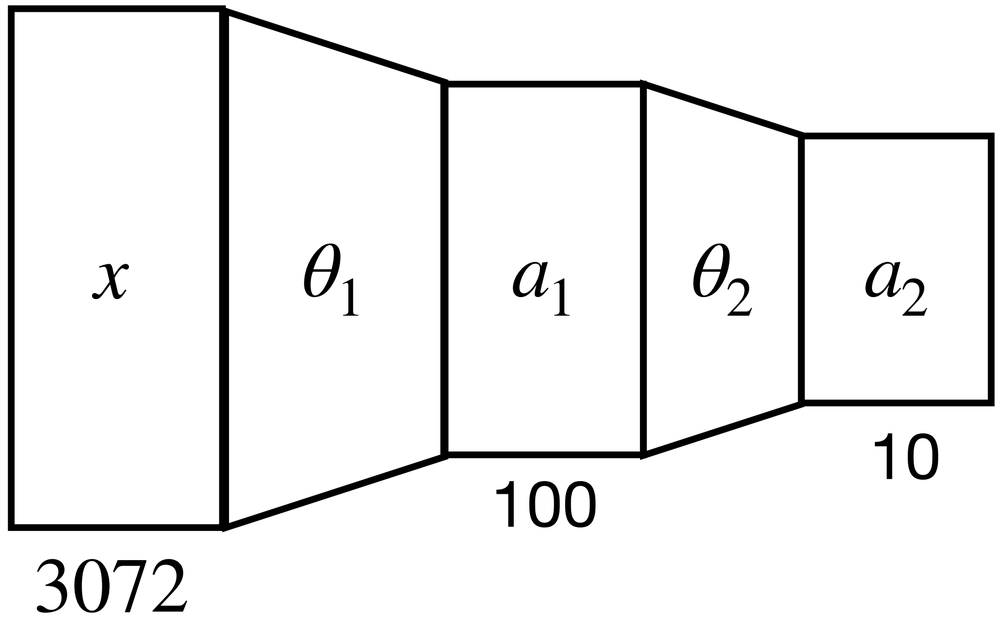
Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization; Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Recap: Neural networks for image classification

Logistic regression: $h = a(\theta x)$

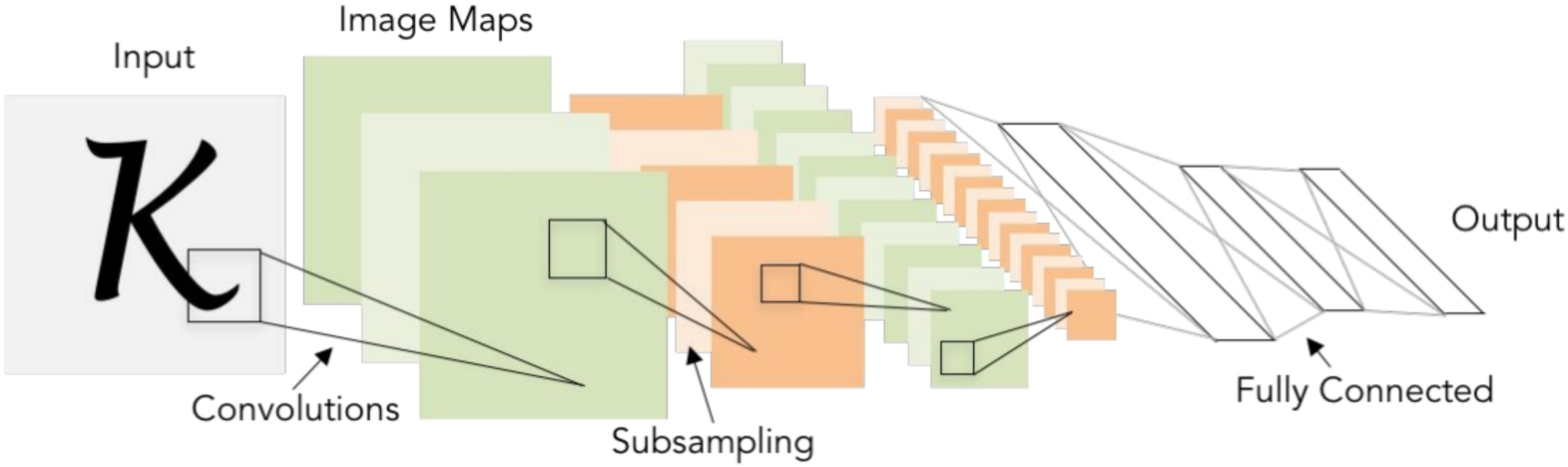
2-layer neural network: $h = a_2(\theta_2 a_1(\theta_1 x))$



Credits to Fei-Fei Li, Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

A bit of CNN history

1998: Character recognition (classification) with convolutional neural networks



LeCun, Bottou, Benigo, Haffner, 1998

A bit of CNN history

2012: Imagenet Classification with deep convolutional neural networks

Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, 2012

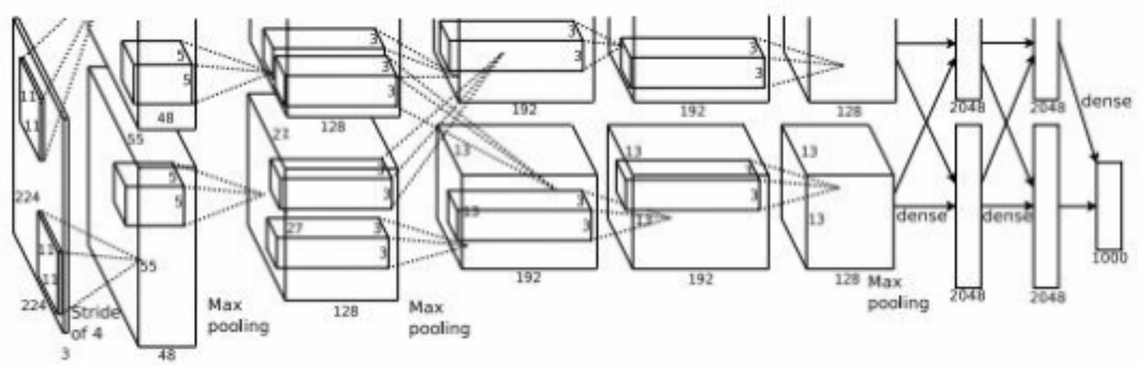
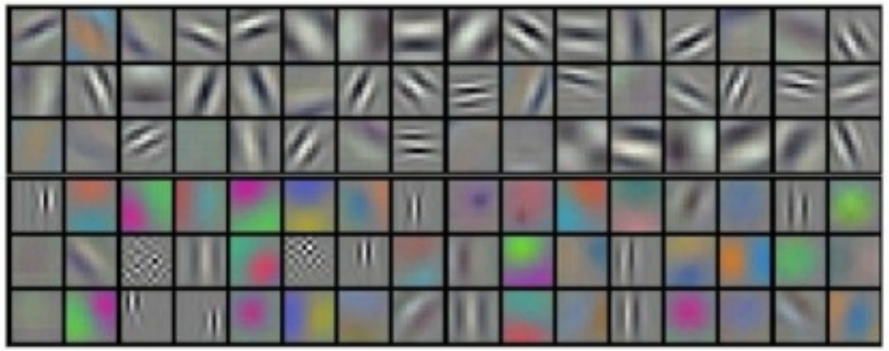
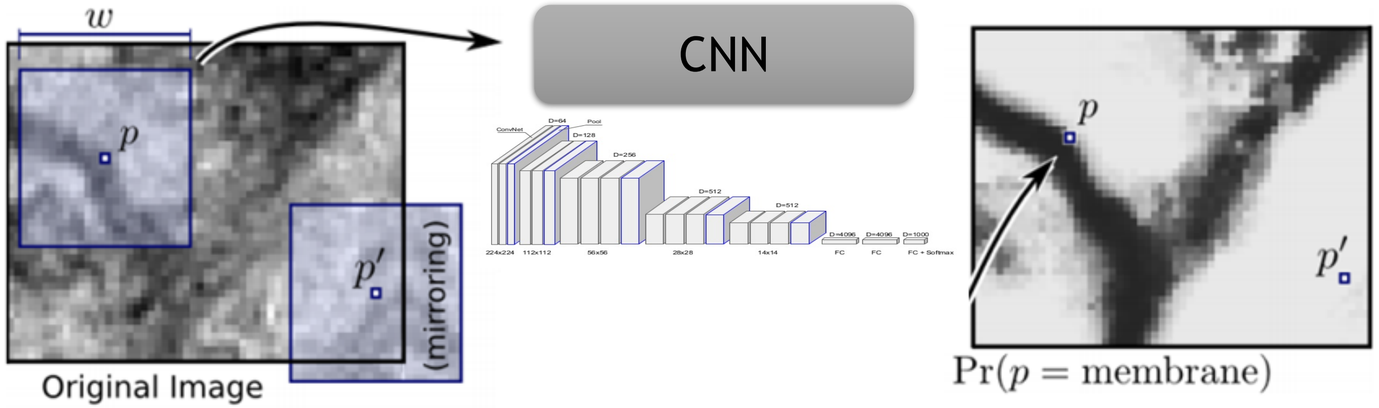


Figure copyright Alex Krizhevsky,Ilya Sutskever and Geoffrey Hinton, 2012



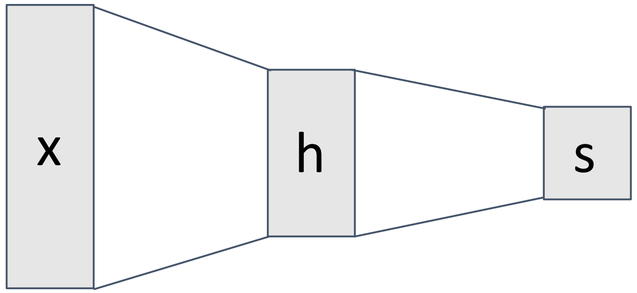
ISBI 2012 EM Segmentation Challenge



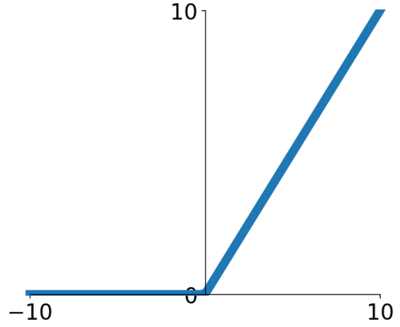
Ciresan, D.C., Gambardella, L.M., Giusti, A., Schmidhuber, J.: Deep neural networks segment neuronal membranes in electron microscopy images.

Components of Convolutional Networks

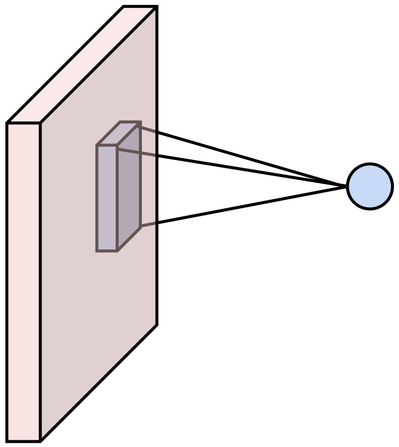
Fully-Connected Layers



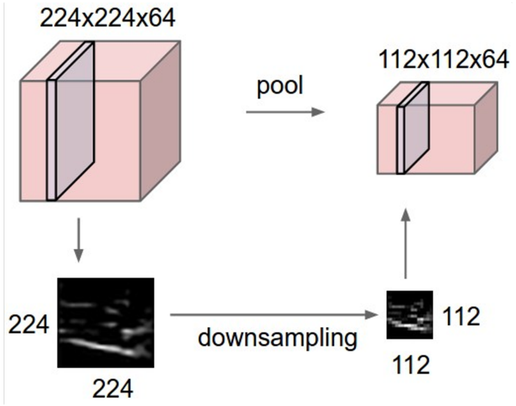
Activation Function



Convolution Layers



Pooling Layers



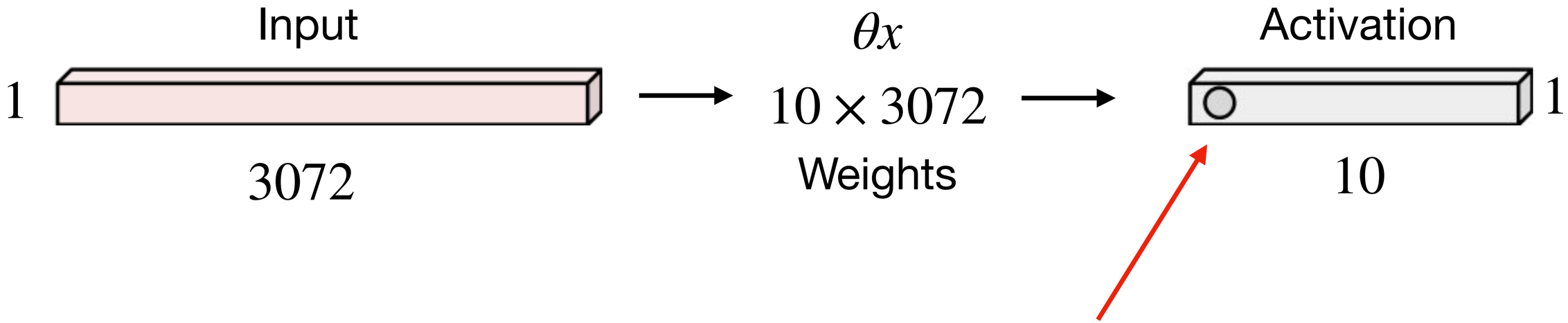
Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Credit: Justin Johnson, U Michigan, Deep Learning for Computer Vision 2019

Fully Connected Layer

$32 \times 32 \times 3$ image $\longrightarrow$ Stretch to 3072×1

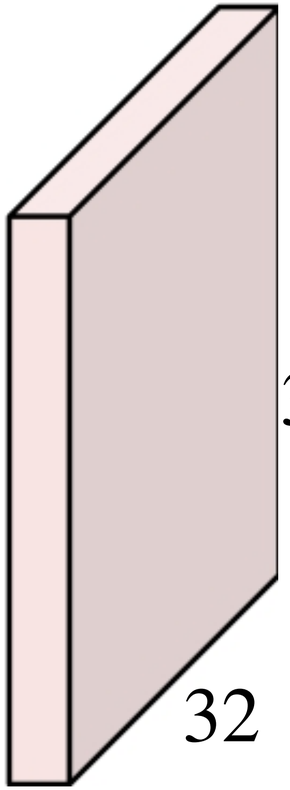


The result of taking a dot product between a row of θ and the input (a 3072-dimensional dot product)

Convolution Layer

Filters always extend the full depth of the input volume

$32 \times 32 \times 3$ image

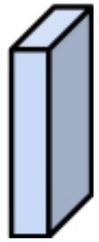


32 height

32 width

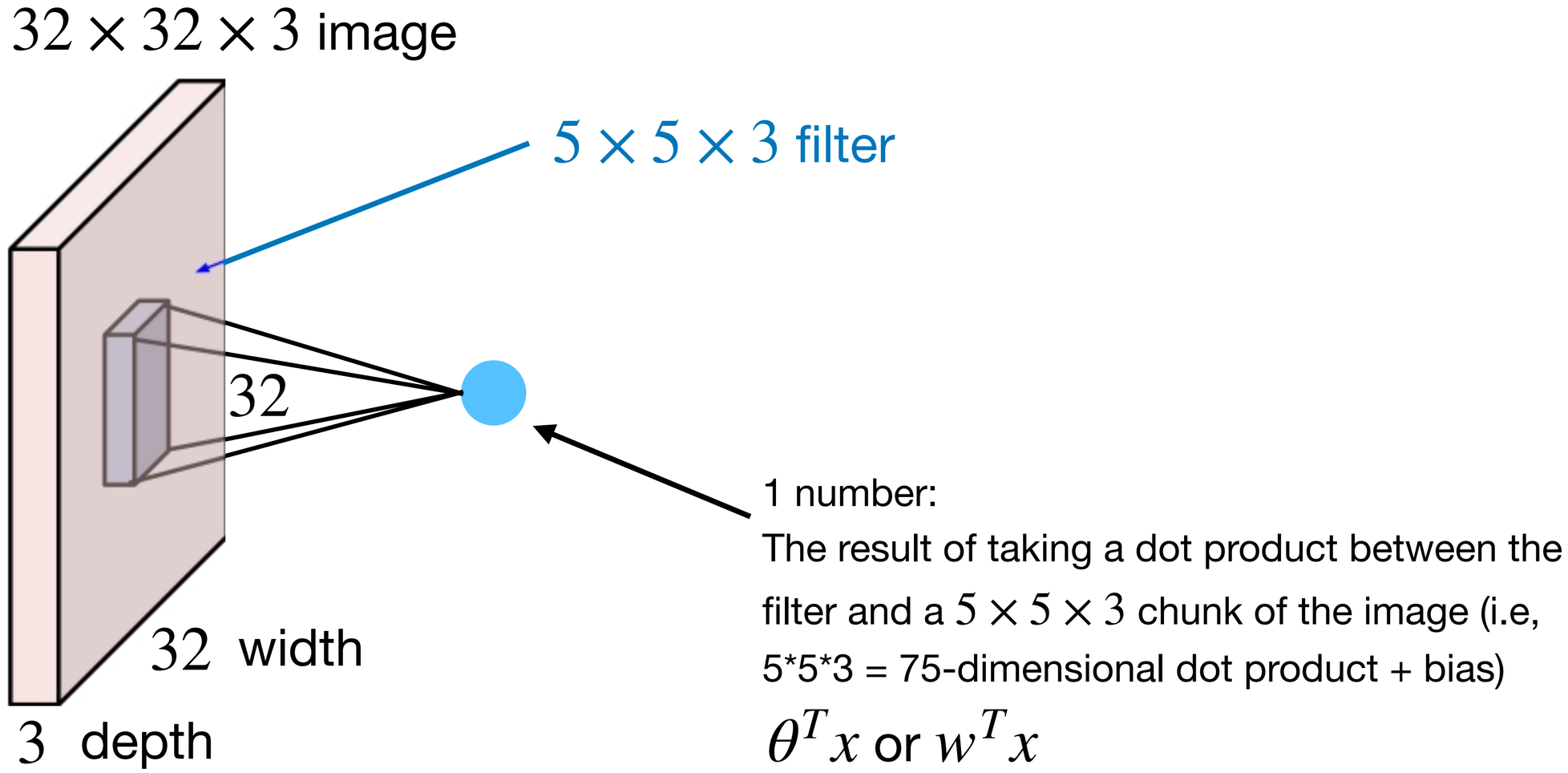
3 depth

$5 \times 5 \times 3$ parameters (θ or w), aka "filter"

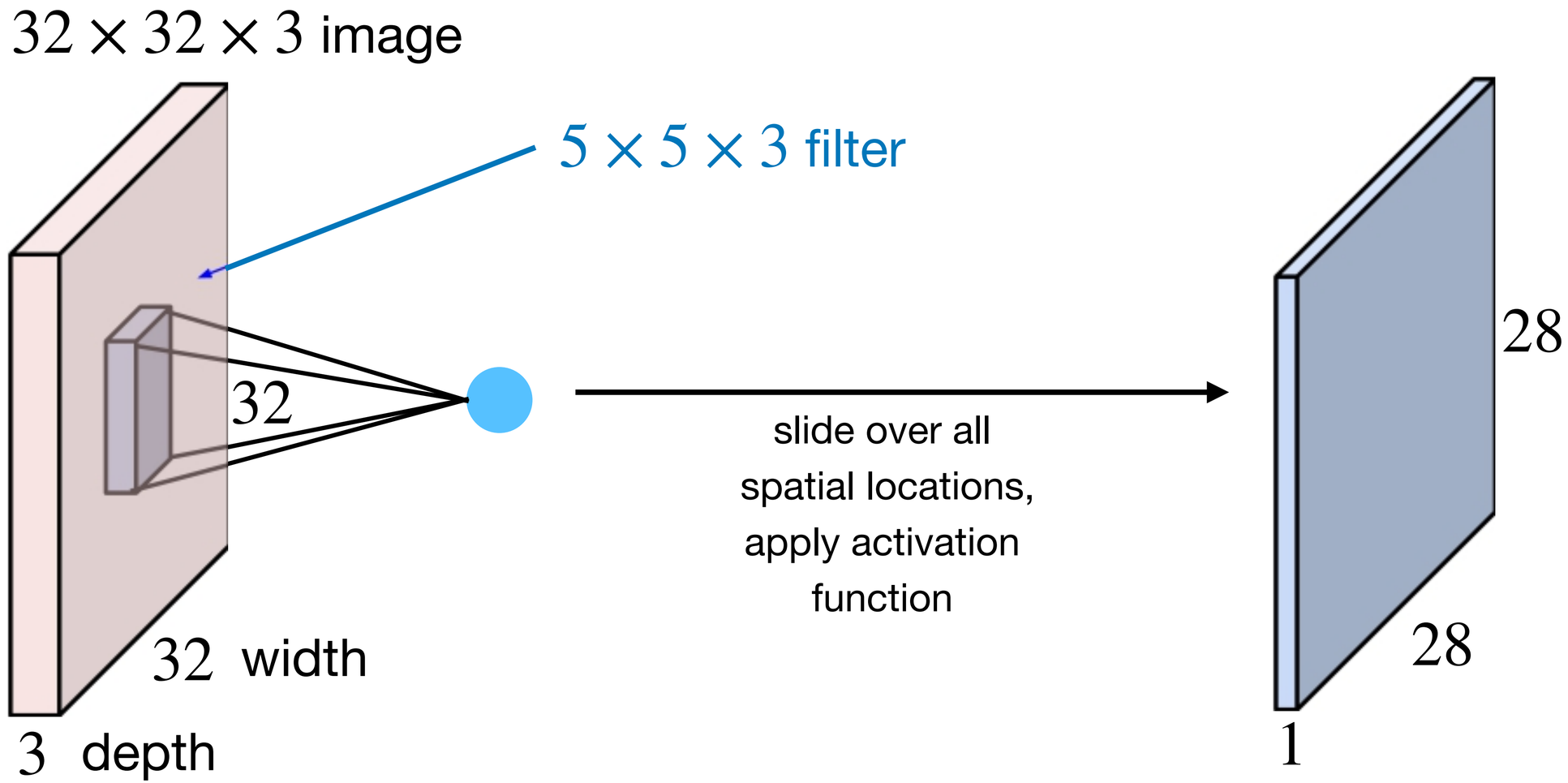


"Convolve" the filter with the image, i.e., slide the filter over the image spatially computing dot products

Convolution Layer

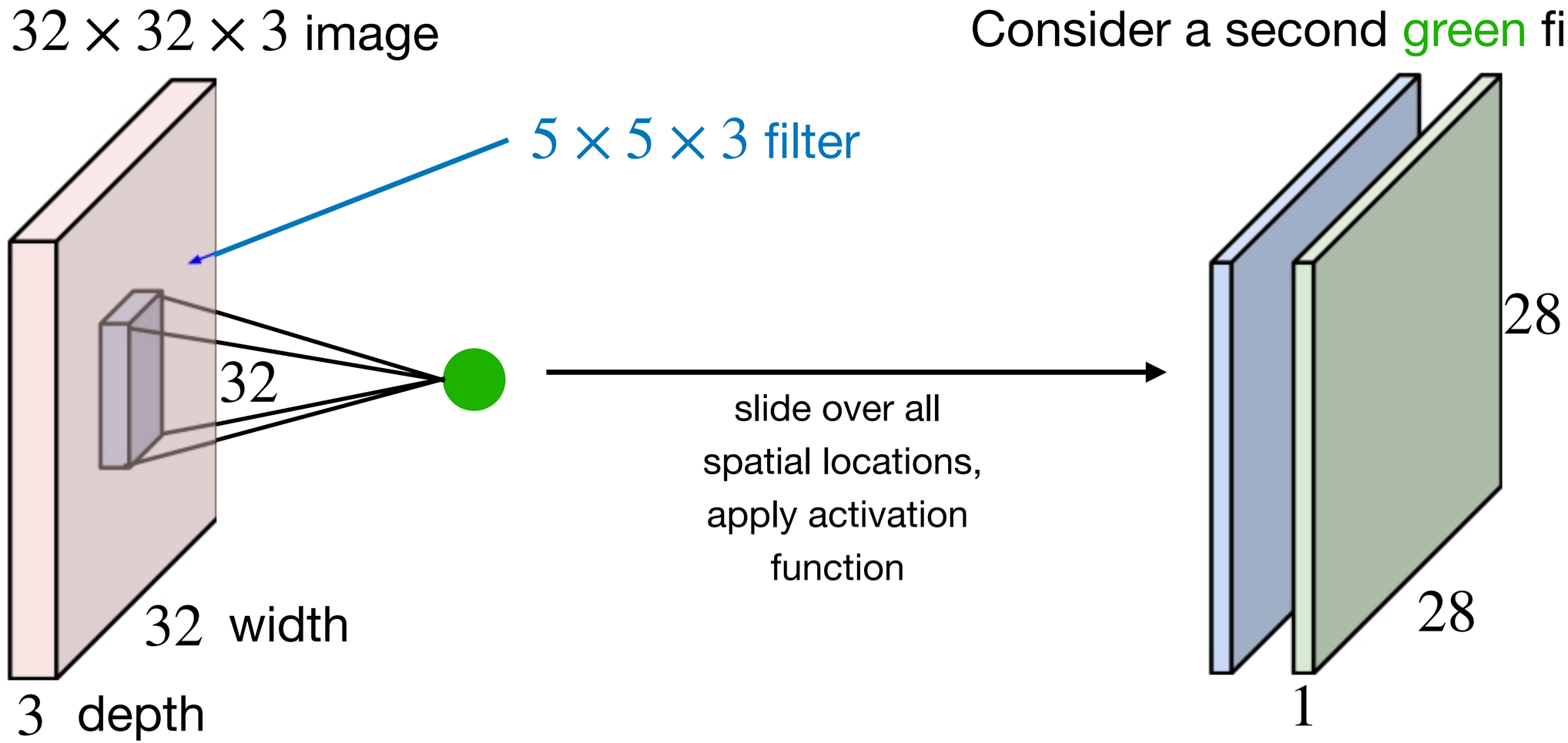


Convolution Layer



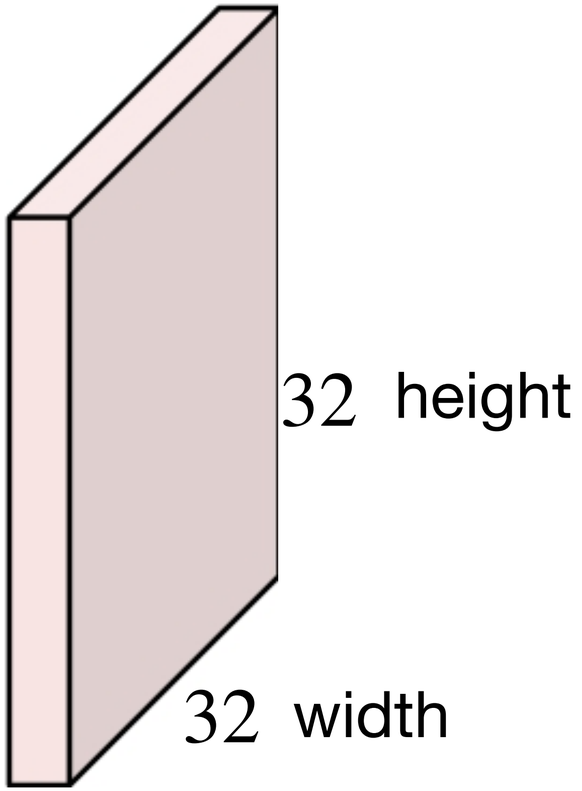
Credits to Fei-Fei Li, Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Convolution Layer



Convolution Layer

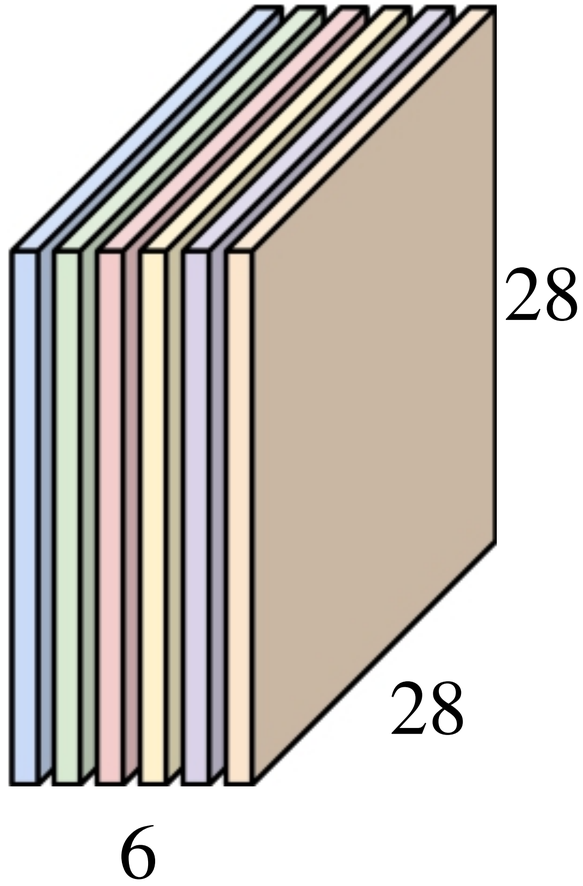
$32 \times 32 \times 3$ image



convolution layer

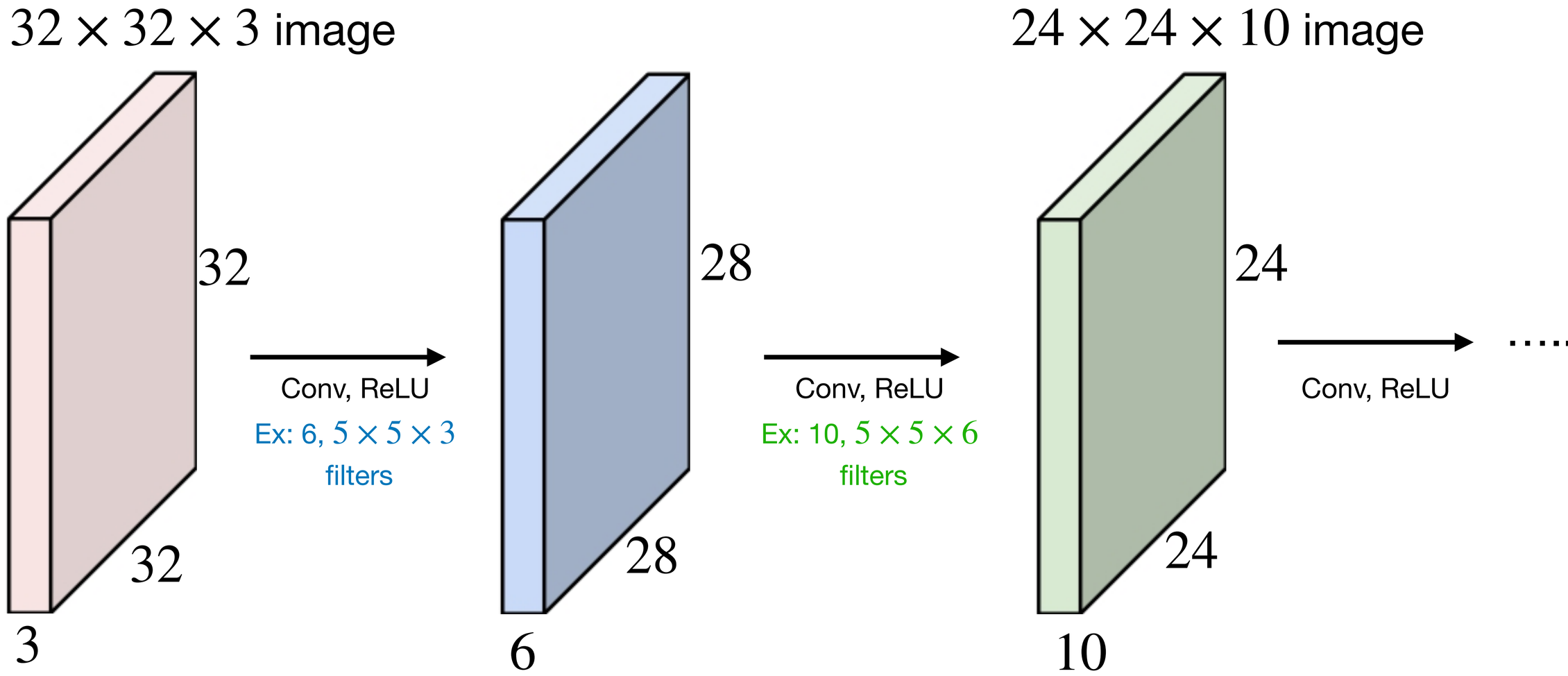
E.g., 6 different 5x5 filters followed by activations yield 6 activation maps; Stacked up, these form a “new image” of size 28x28x6

activation maps



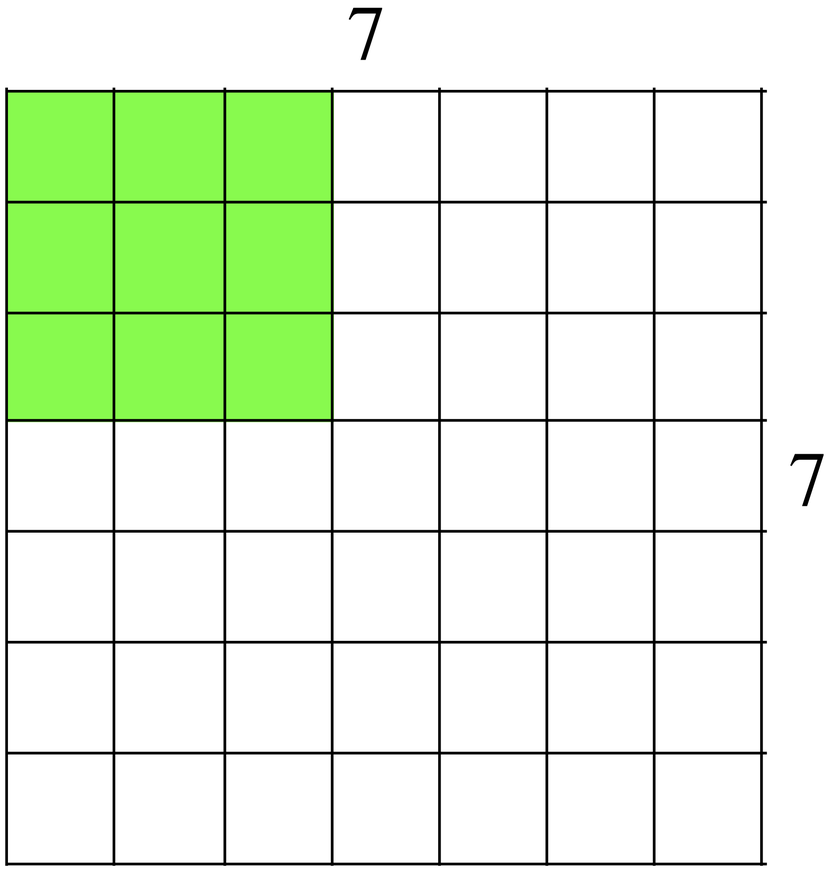
Convolution Layer

A CNN contains sequences of convolution layers



Convolution Layer: Stride

Stride 1

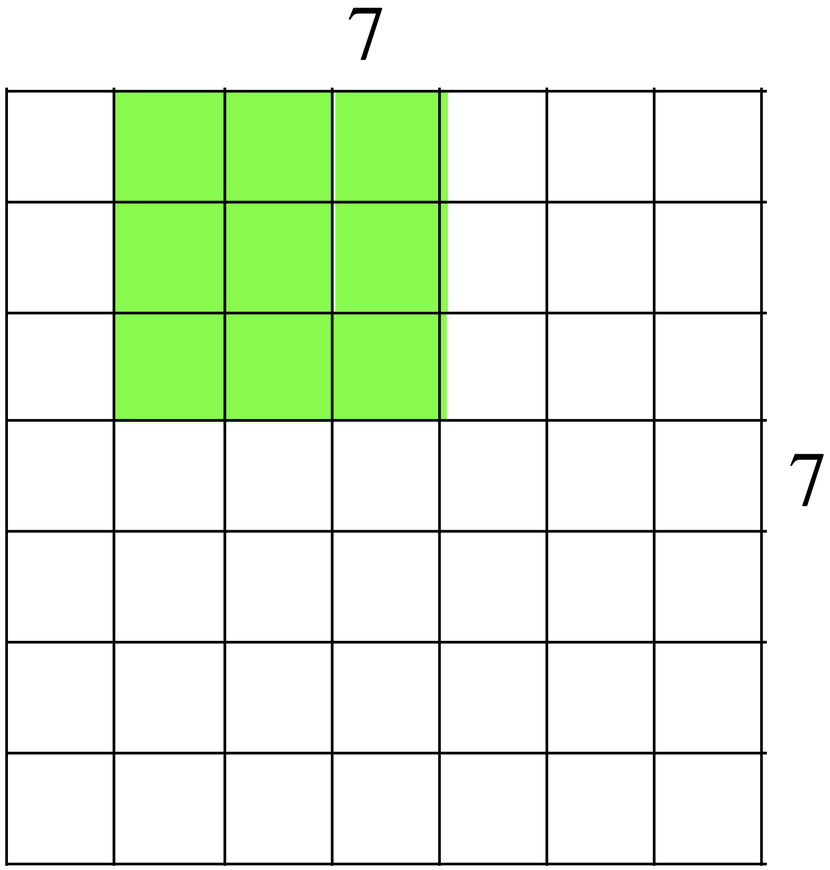


A 7×7 image (input),

Let's assume a 3×3filter

Convolution Layer: Stride

Stride 1

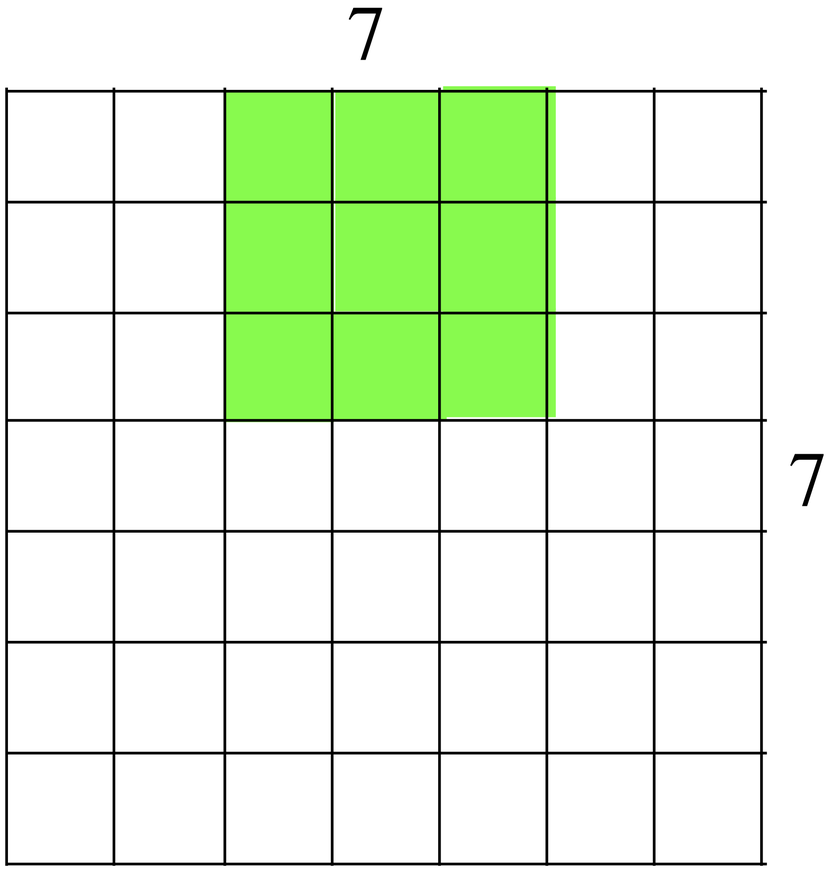


A 7×7 image (input),

Let's assume a 3×3filter

Convolution Layer: Stride

Stride 1

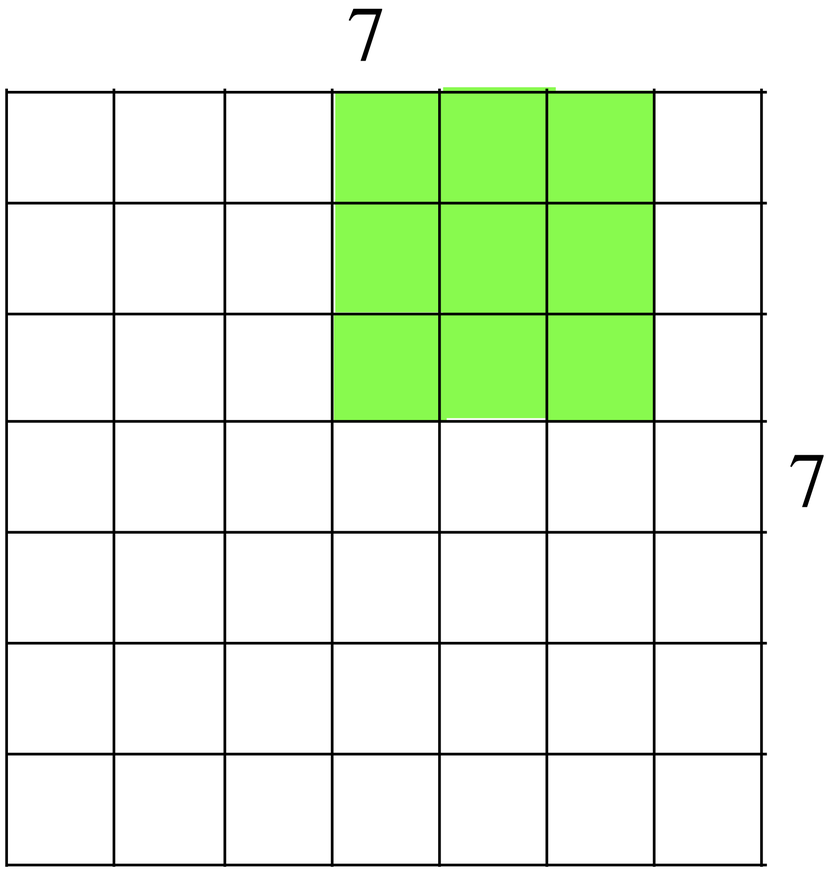


A 7×7 image (input),

Let's assume a 3×3filter

Convolution Layer: Stride

Stride 1

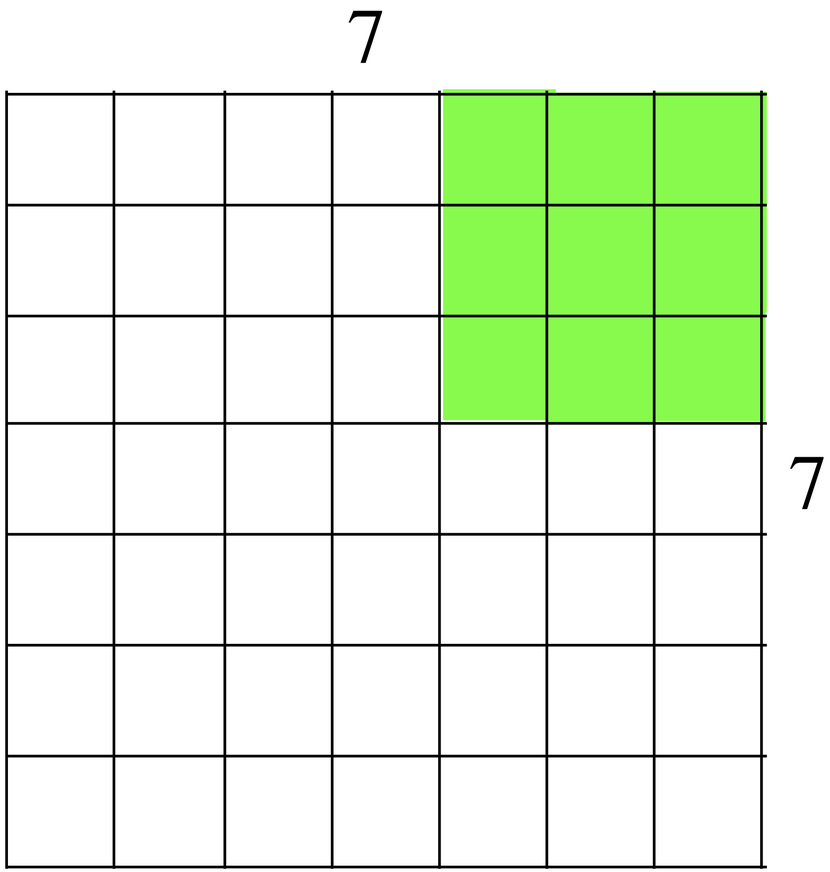


A 7×7 image (input),

Let's assume a 3×3filter

Convolution Layer: Stride

Stride 1



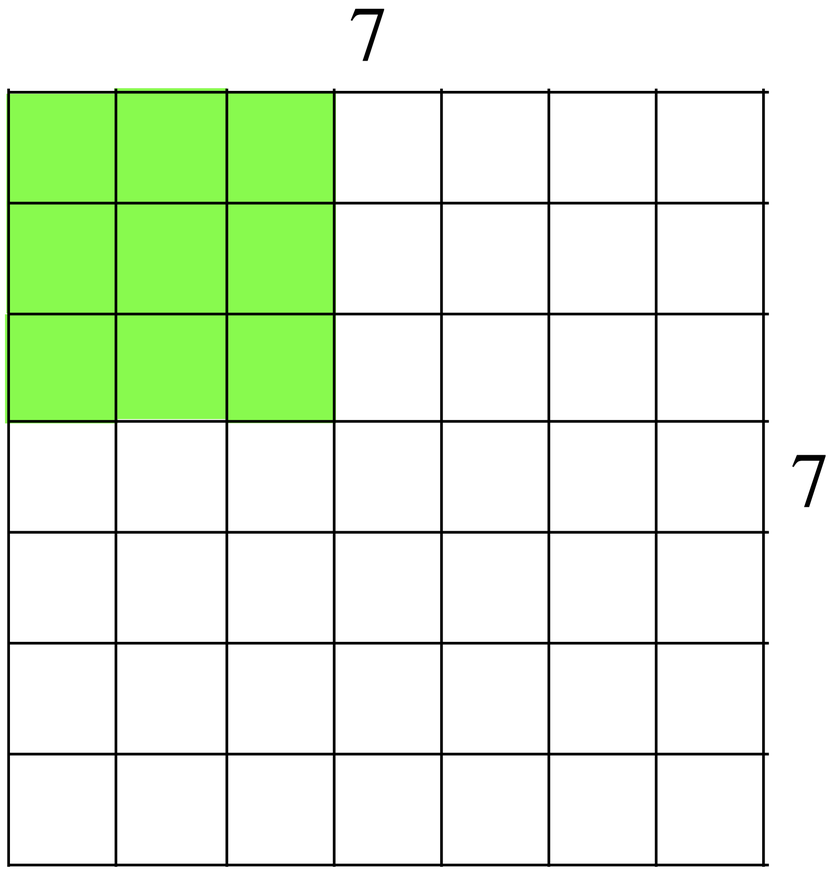
A 7×7 image (input),

Let's assume a 3×3filter

5×5 output

Convolution Layer: Stride

Stride 2

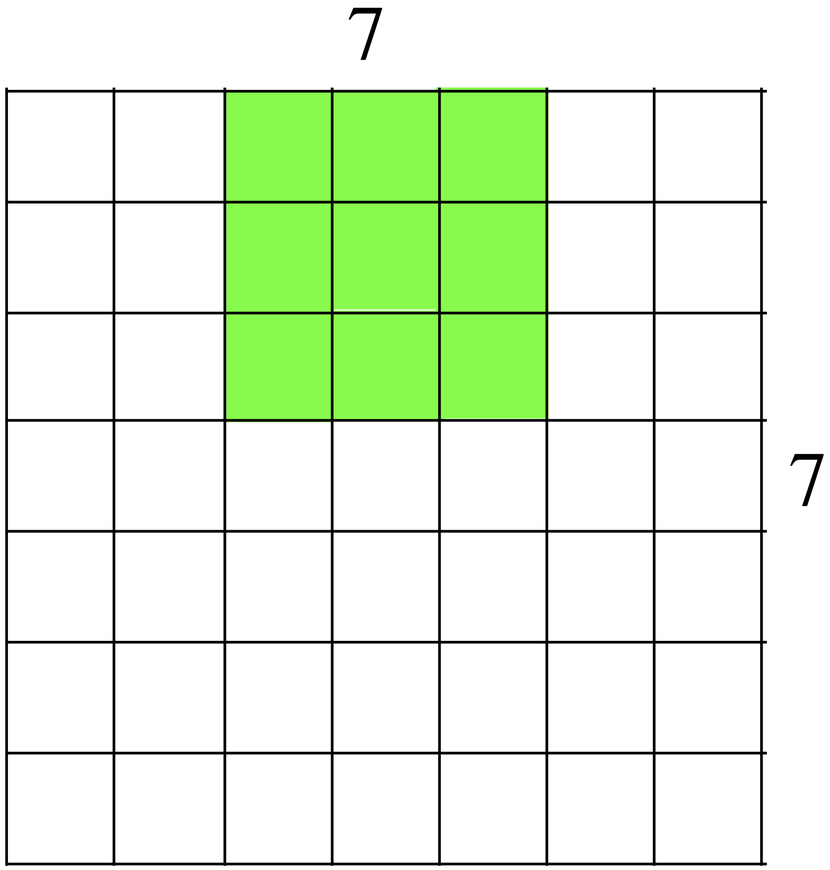


A 7×7 image (input),

Let's assume a 3×3filter

Convolution Layer: Stride

Stride 2

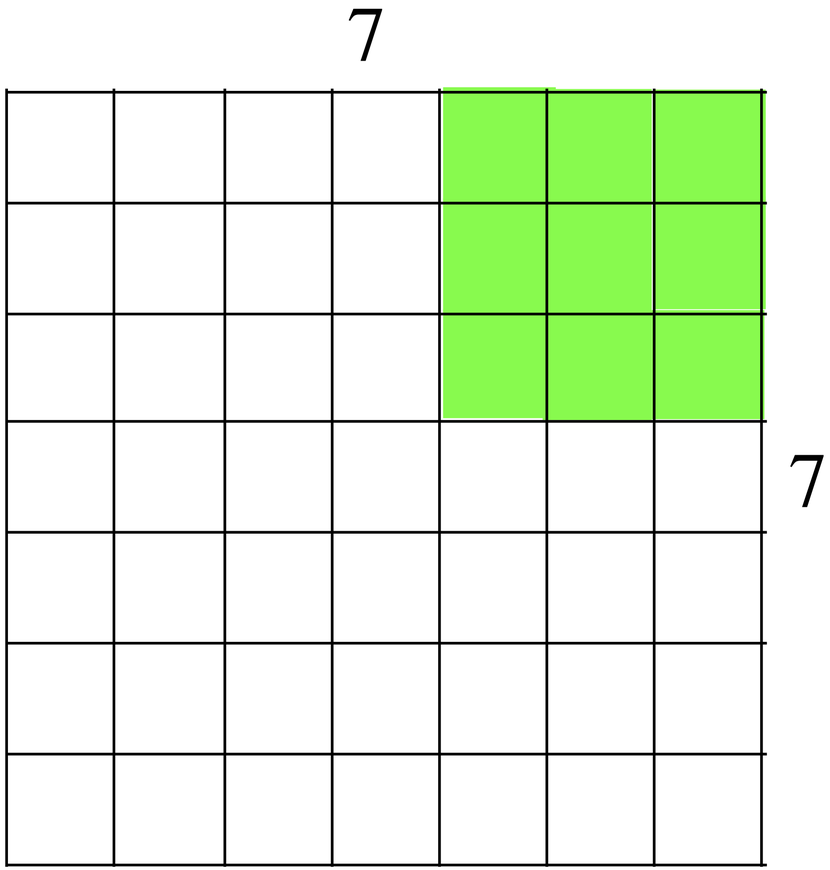


A 7×7 image (input),

Let's assume a 3×3filter

Convolution Layer: Stride

Stride 2



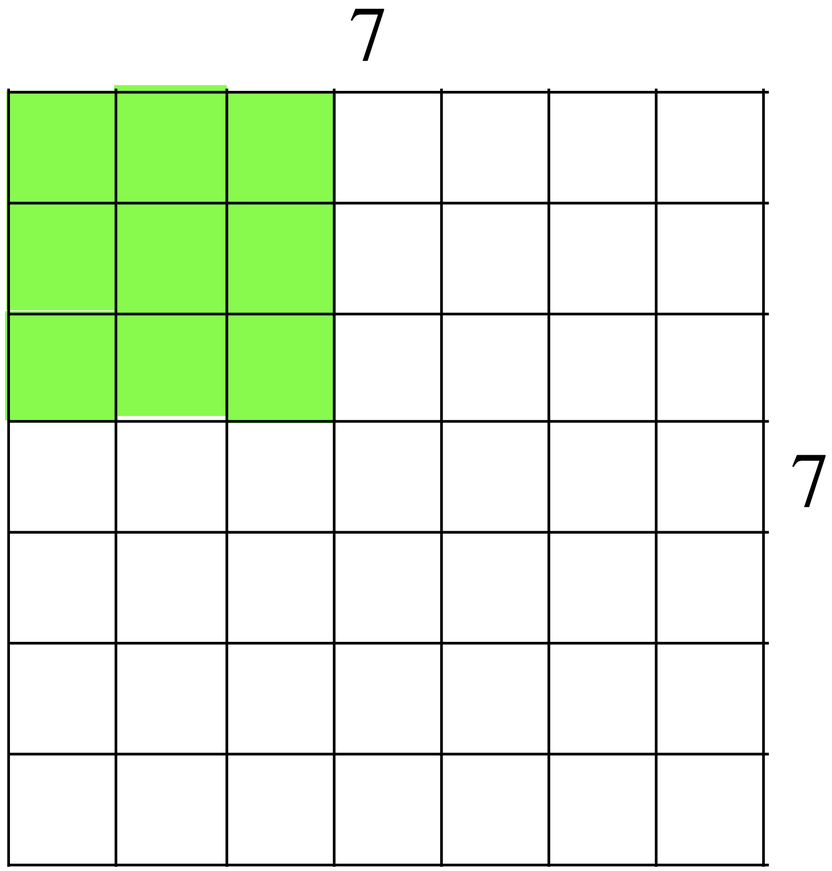
A 7×7 image (input),

Lets assume a 3×3filter

3×3 output

Convolution Layer: Stride

Stride 3



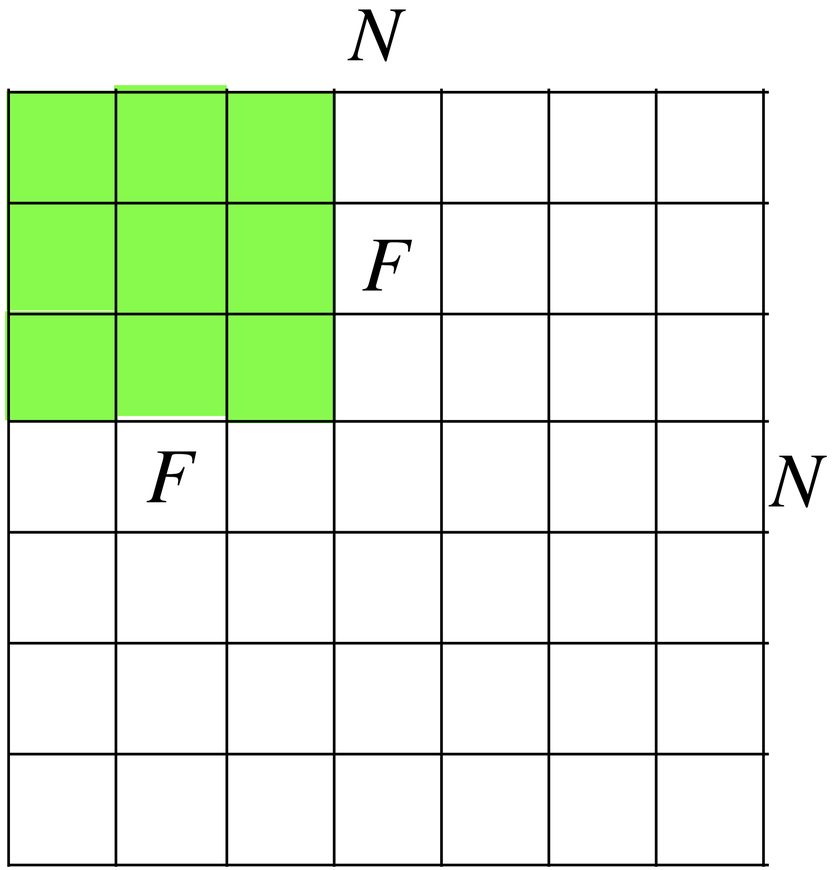
A 7×7 image (input),

Lets assume a 3×3filter

Does not fit!

We cannot apply a 3×3 filter on a 7×7 input with stride 3

Convolution Layer: Output Size



Output size:

$(N - F)/stride + 1$

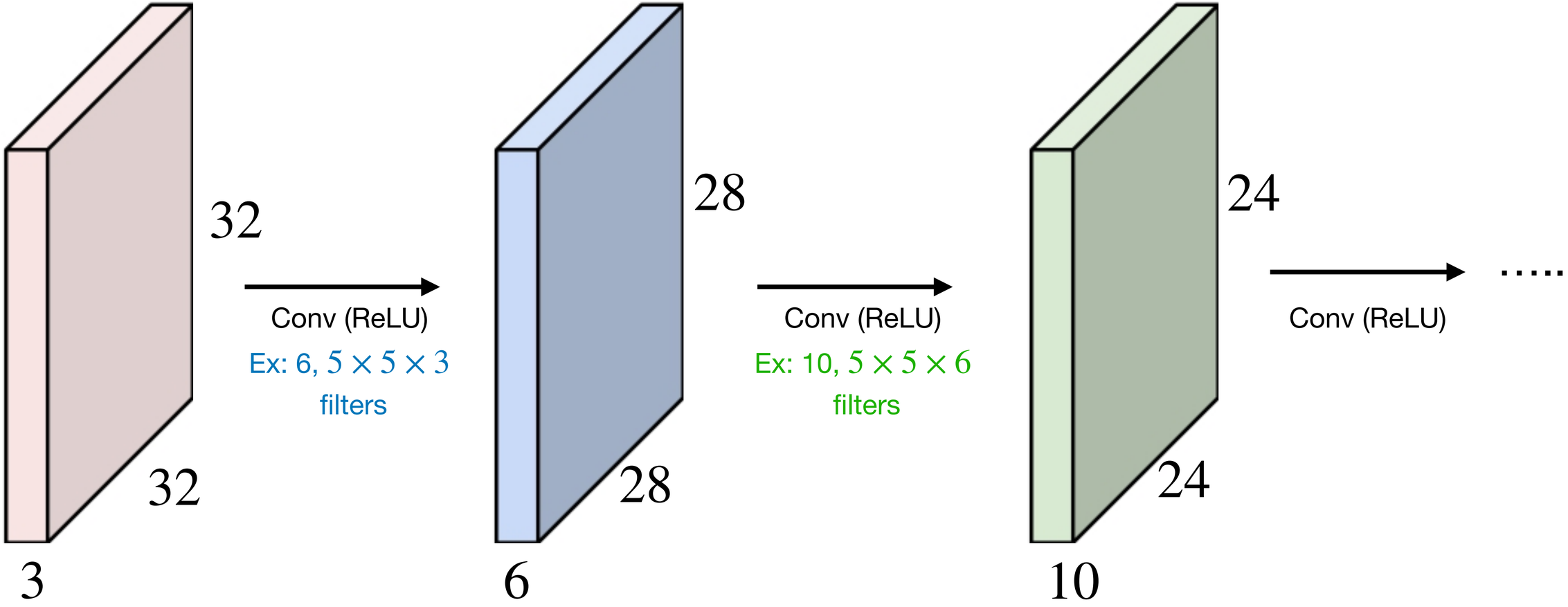
Example: $N = 7, F = 3$:

Stride 1 : $(7 - 3)/1 + 1 = 5$

Stride 2 : $(7 - 3)/2 + 1 = 3$

Stride 3 : $(7 - 3)/3 + 1 = 2.33$ **X**

Convolution Layer: Output Size

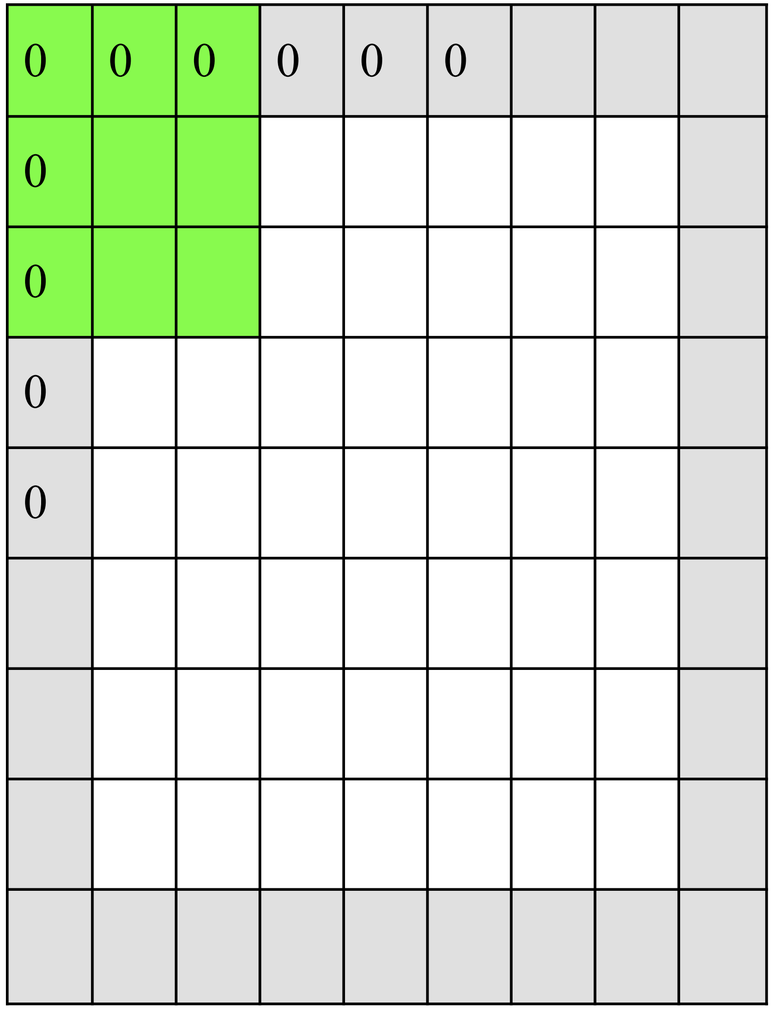


32×32 input convolved repeatedly with 5×5 filters shrinks volume spatially;

Too much shrinking can be problematic

Convolution Layer: Padding

Common to zero-pad the border



Ex: input 7×7

3×3 filter, applied with stride 1

Pad with 1 pixel border — what would the output be?

Remember the output size formula

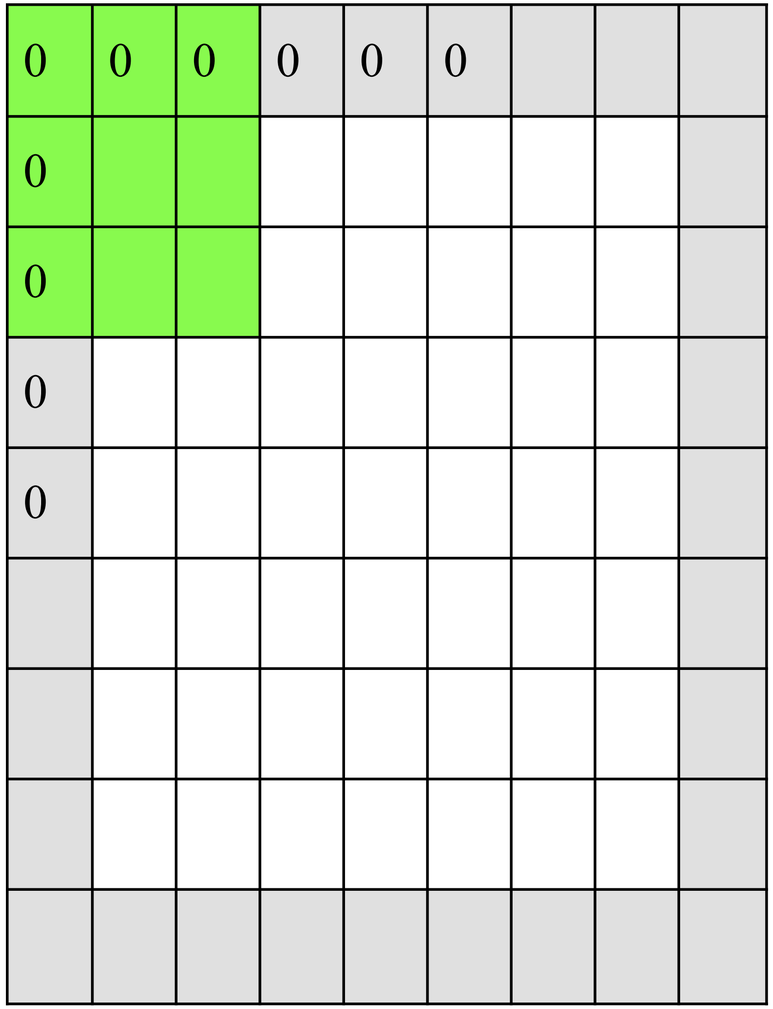
Output size:

$(N - F)/stride + 1$

The output would be 7×7

Convolution Layer: Padding

Common to zero-pad the border



Ex: input 7×7

3×3 filter, applied with stride 1

Pad with 1 pixel border — what would the output be?

7×7

It is very common to see convolution layers with stride 1, filters of size FxF and zero-padding with $(F - 1)/2$.

Ex: F = 3 — zero padding with 1

F = 5 —zero padding with 2

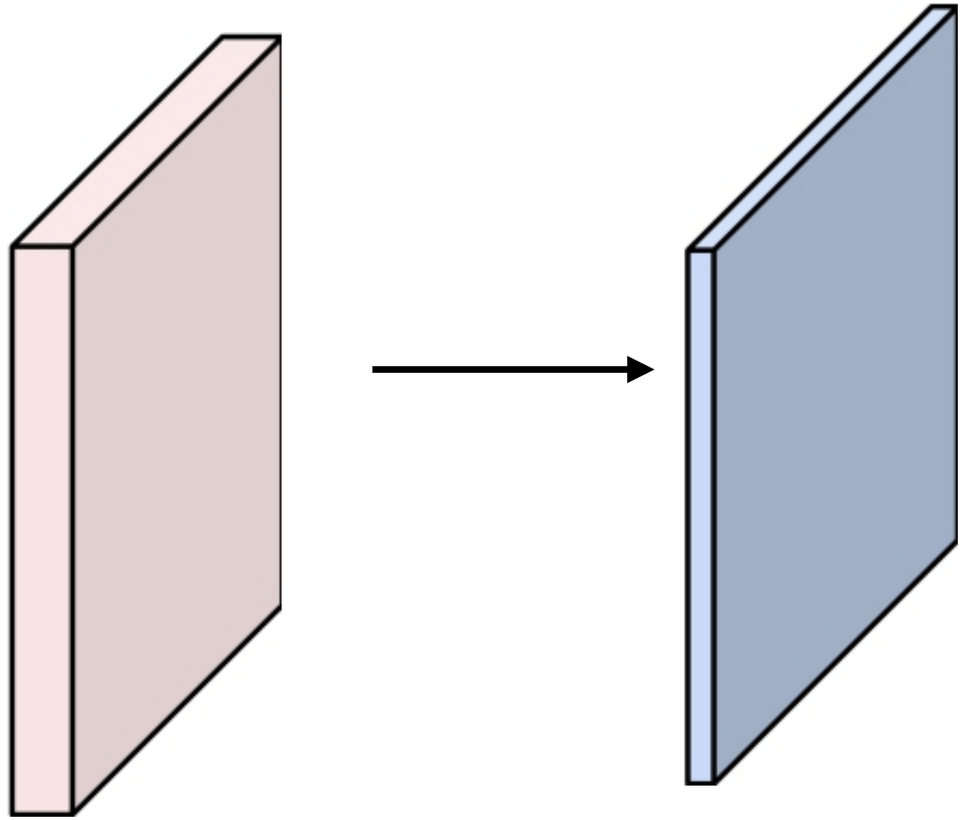
F = 7, zero padding with 3

Convolution Layer: Quiztime

Input Volume: $32 \times 32 \times 3$

10 5×5 filters with stride 1 and padding 2

What is the output size?



N = 32, F = 5, Stride = 1, padding = 2

$(32 + (2 \times 2) - 5)/1 + 1 = 32$

10 filters

Output size is $32 \times 32 \times 10$

Convolution Layer: Quiztime

Input Volume: $32 \times 32 \times 3$

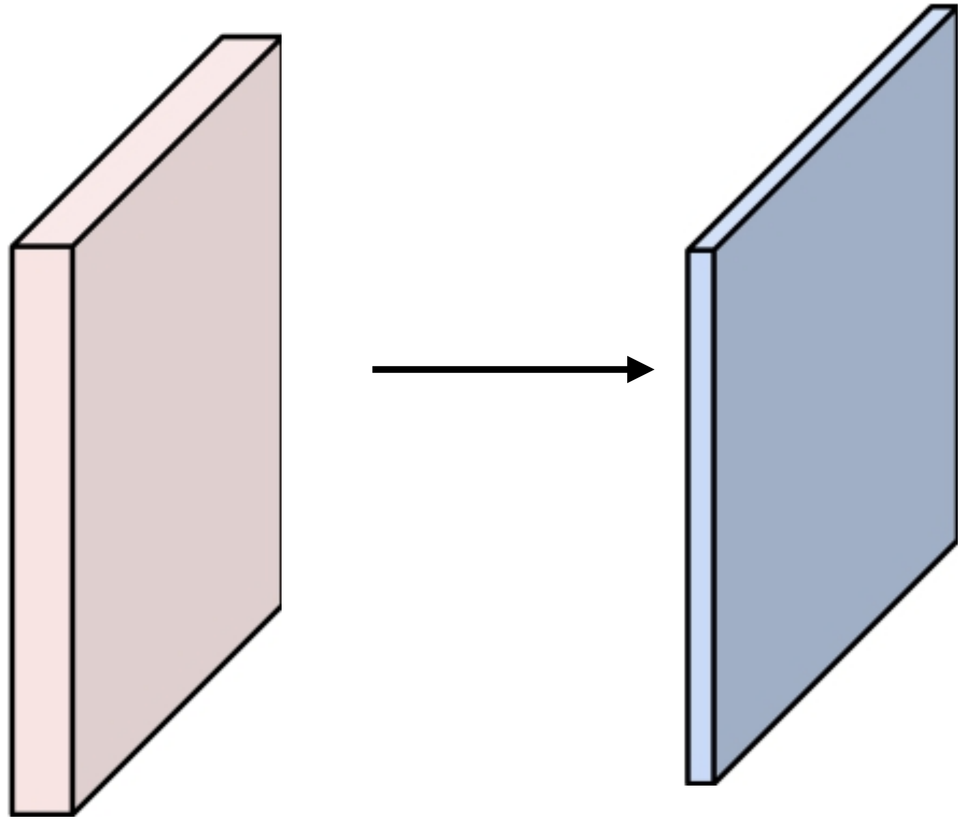
10 5×5 filters with stride 1 and padding 2

How many parameters are there in this layer?

Each filter has $5 \times 5 \times 3 + 1 = 76$ parameters

10 filters

Total number of parameters : $76 \times 10 = 760$



Convolution Layer: Summary

➡ Accepts an input of dimension $width_{input} \times height_{input} \times depth_{input}$

➡ Four hyper-parameters:

- Number of filters K
- Filter spatial extent F
- Stride S
- Padding P

➡ Produces of output of dimension $width_{output} \times height_{output} \times depth_{output}$ where

- $width_{output} = (width_{input} - F + 2P)/S + 1$
- $height_{output} = (height_{input} - F + 2P)/S + 1$
- $depth_{output} = K$

Convolution Layer: Summary

➡ $F \times F \times depth_{input}$ weights and 1 bias per filter

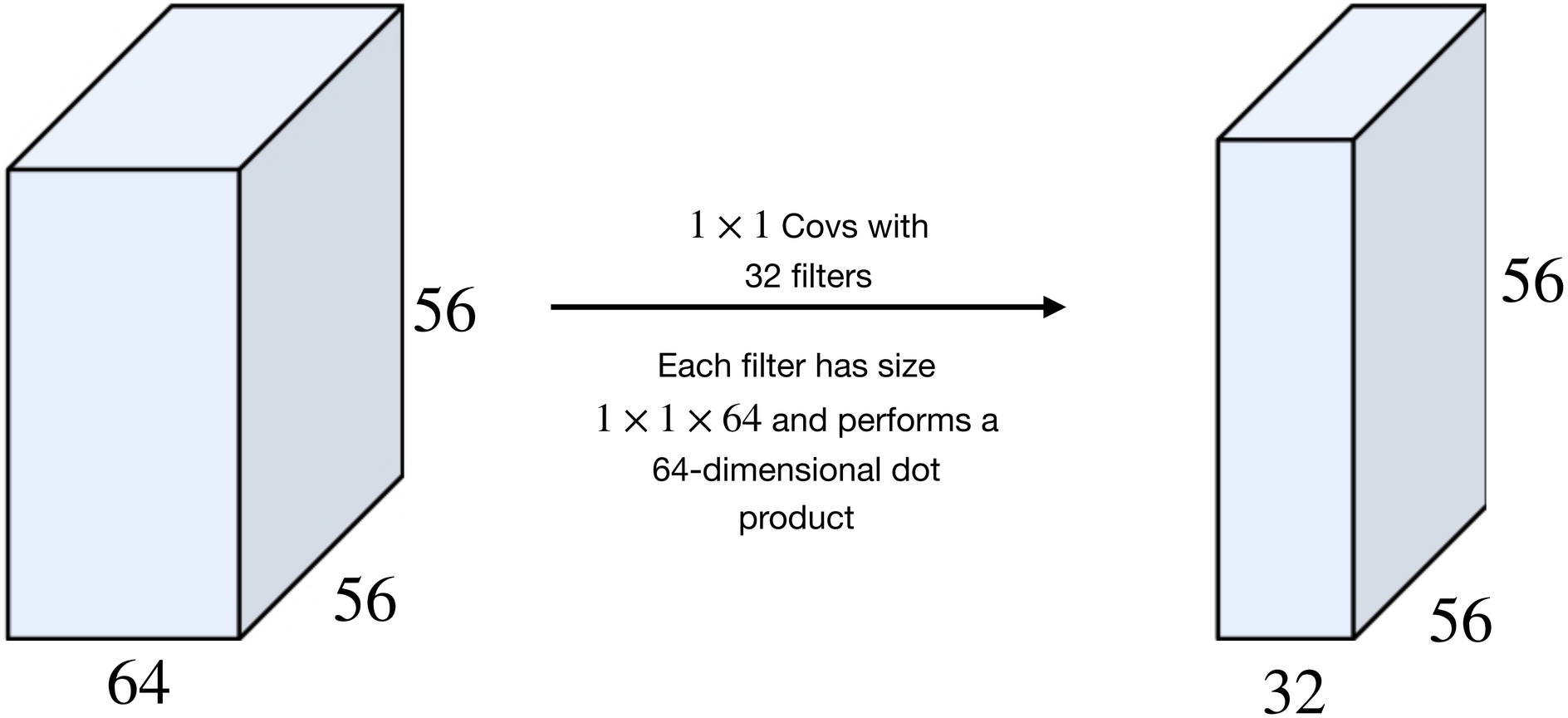
➡ Total of $(F \times F \times depth_{input}) \times K$ weights and K biases

➡ In the output, the nth slice (of size $width_{output} \times height_{output}$) results from convolution of the input with nth filter with stride S and then offset by the nth bias.

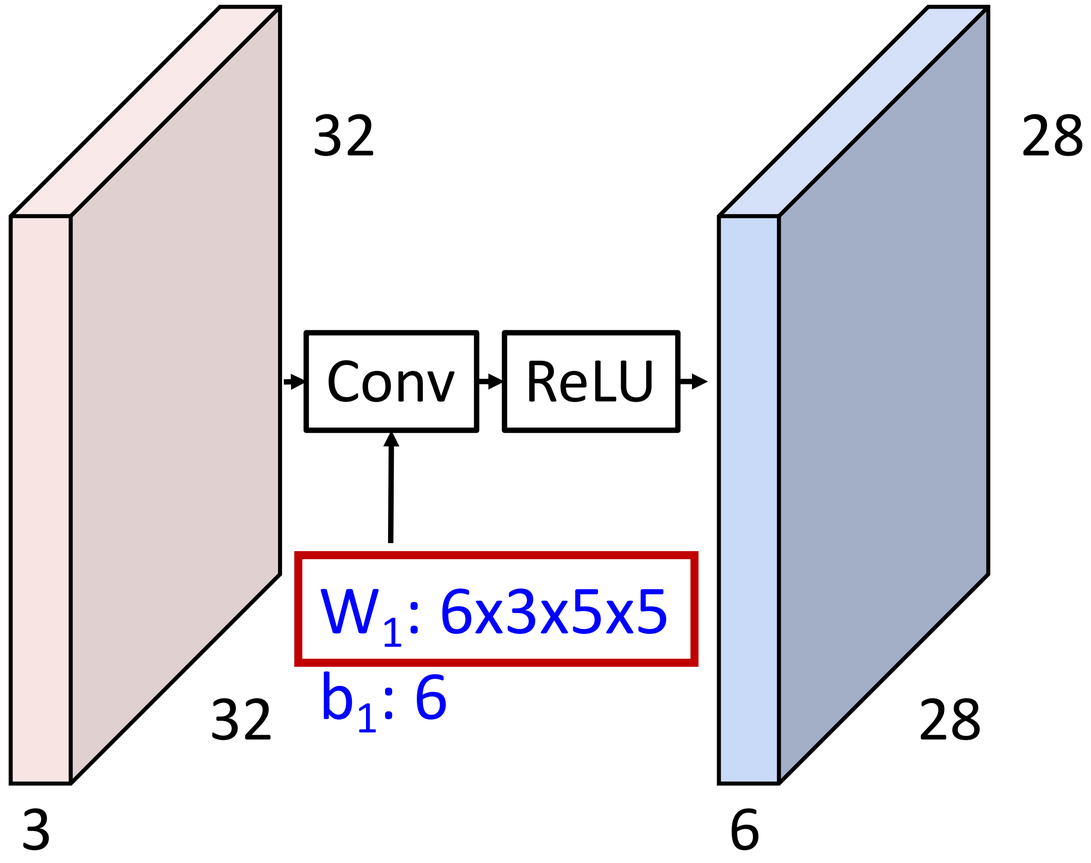
➡ Note: usually K = (powers of 2, e.g: 32,64,128,256) “just to be nice”

- 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

Note: 1×1 convolution also makes sense



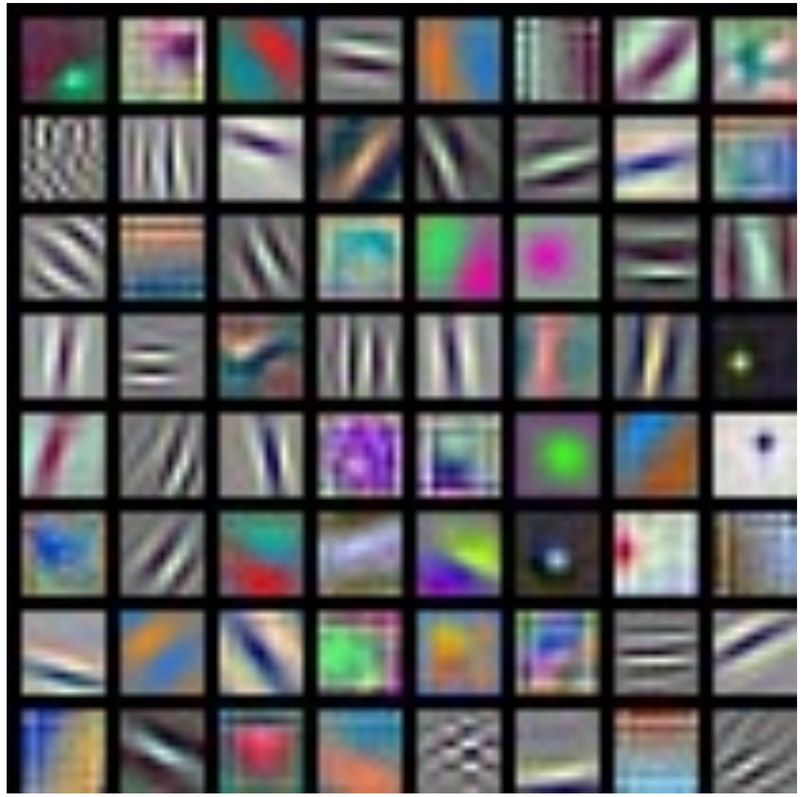
What do convolutional filters learn?



Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

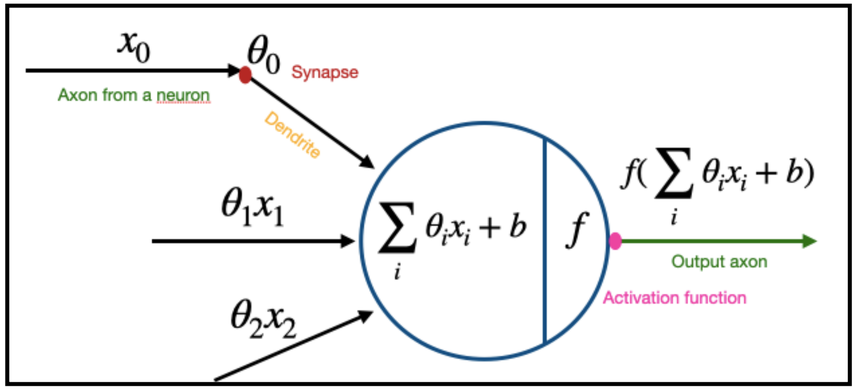
First-layer conv filters: local image templates
(Often learns oriented edges, opposing colors)

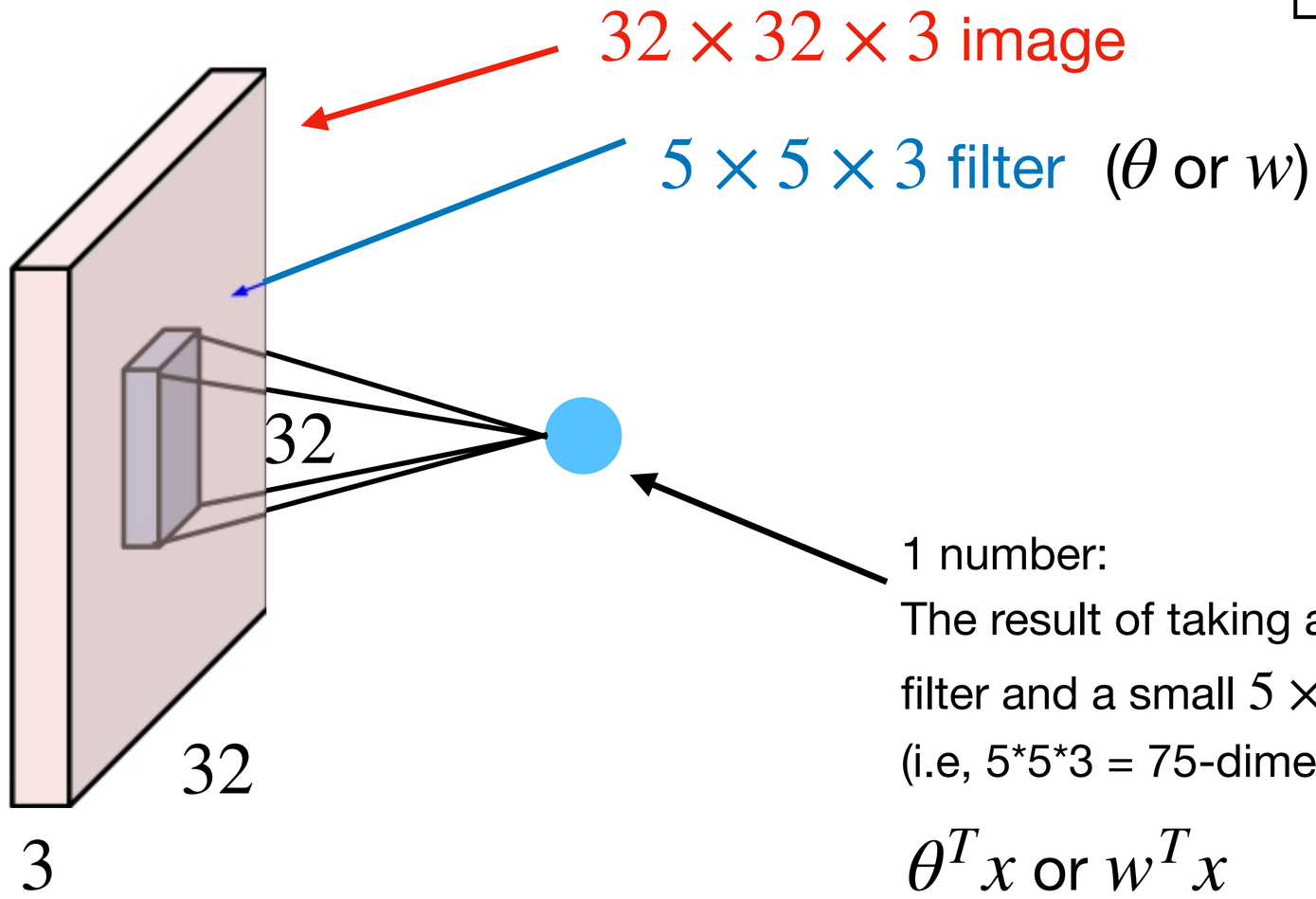


AlexNet: 64 filters, each 3x11x11

Convolution layer

In terms of neurons





1 number:

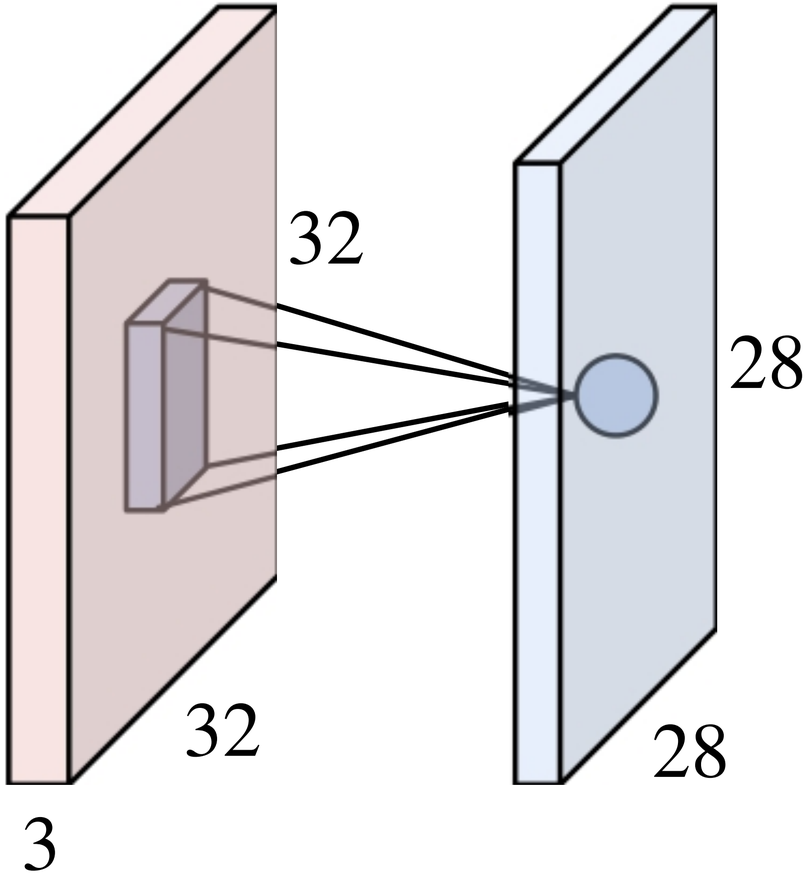
The result of taking a dot product between the filter and a small $5 \times 5 \times 3$ chunk of the image

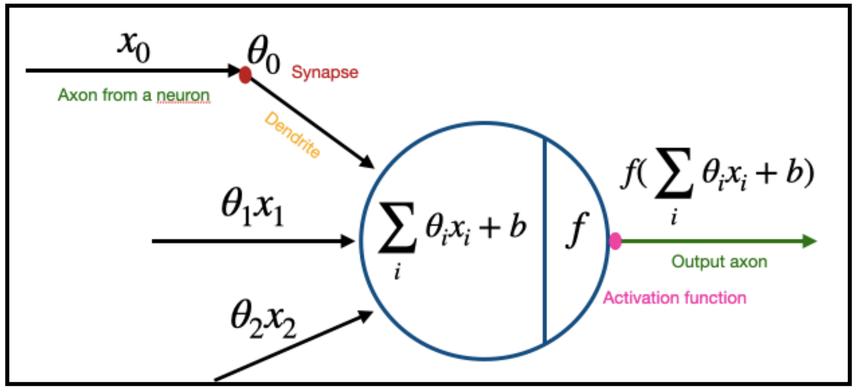
(i.e, 5*5*3 = 75-dimensional dot product + bias)

$\theta^T x$ or $w^T x$

Convolution layer

In terms of neurons





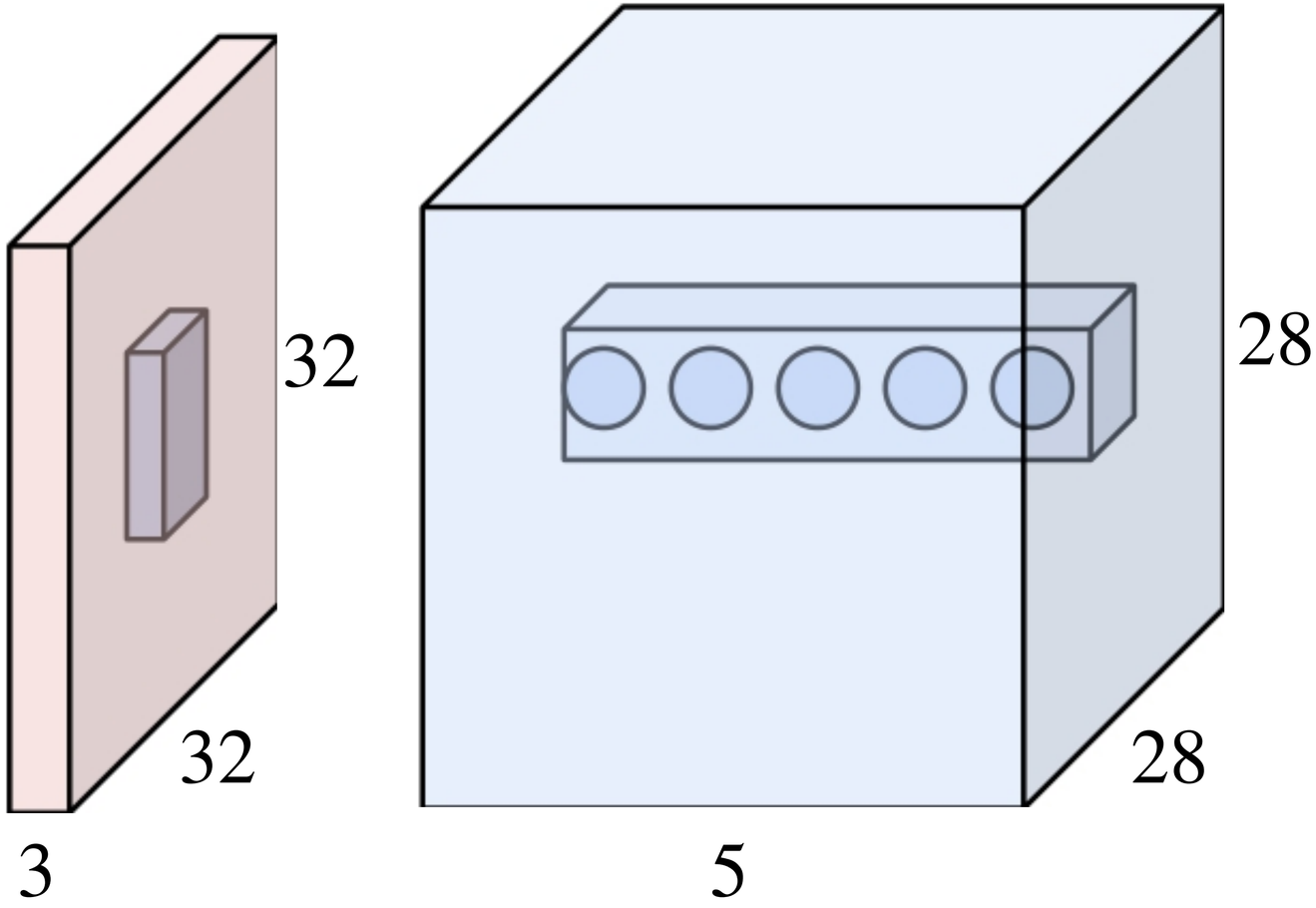
An activation map is a 28×28 sheet of neuron outputs

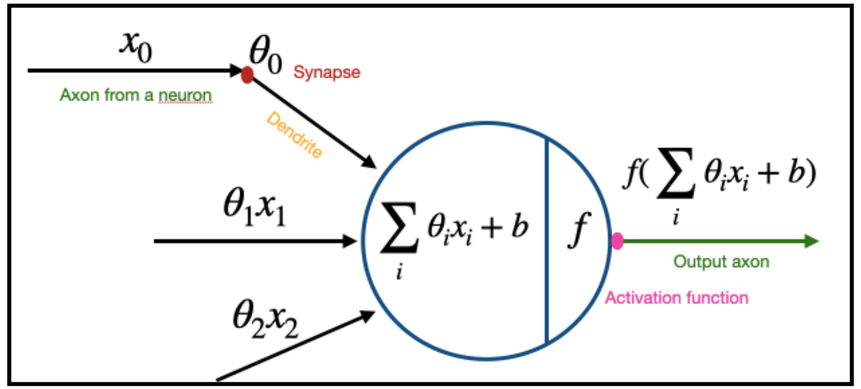
✦ Each neuron is connected to a small region in the input

✦All of them share their parameters

Convolution layer

In terms of neurons



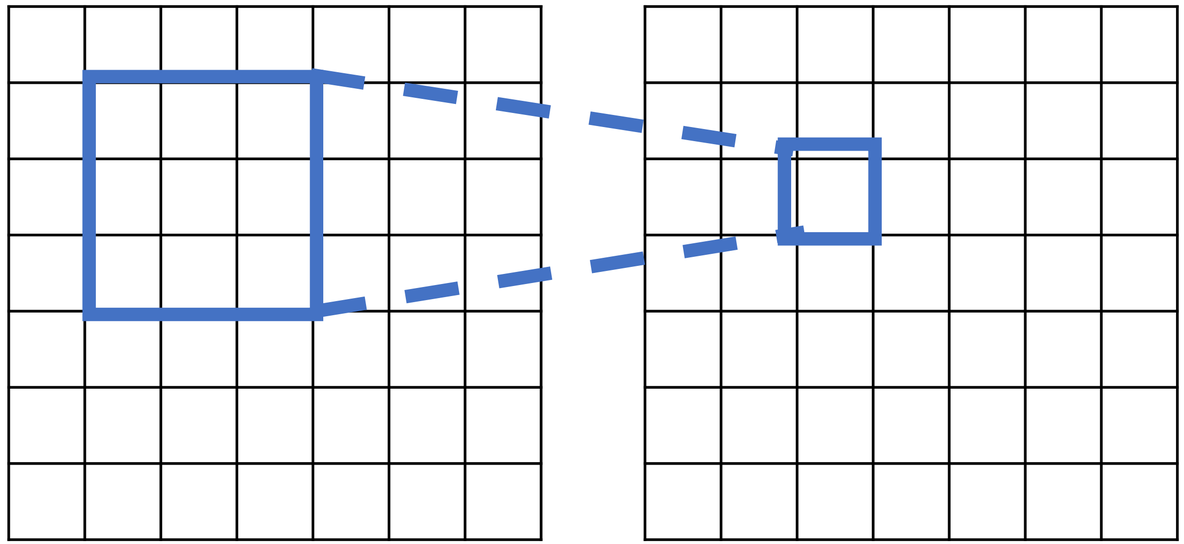


Ex: with 5 filters, convolution layer consists of neurons arranged in a 3D grid $(28 \times 28 \times 5)$

There will be 5 different neurons all looking at the same region in the input volume.

Receptive Field

For convolution with kernel size K, each element in the output depends on a K x K **receptive field** in the input

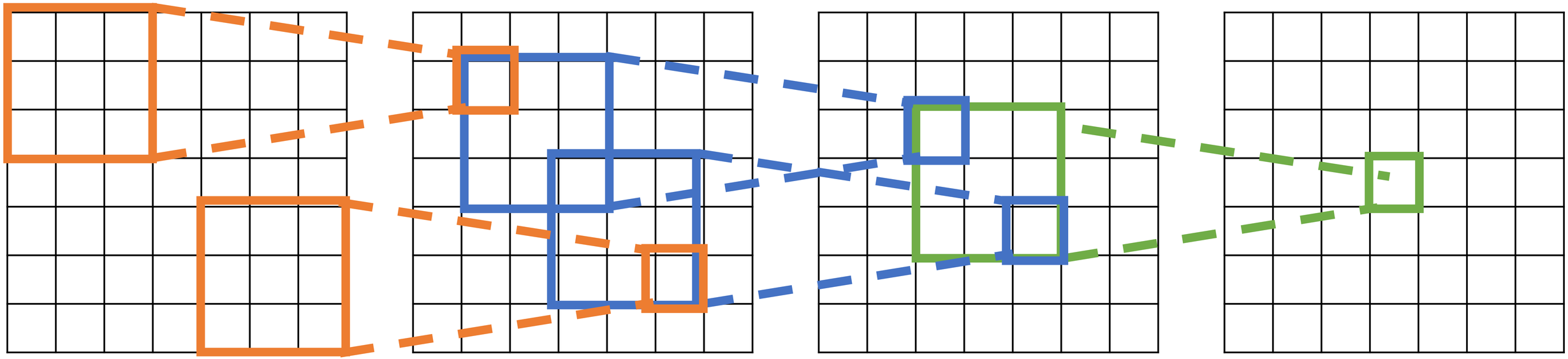


Input Output

Receptive Field

Each successive convolution adds K – 1 to the receptive field size

With L layers the receptive field size is 1 + L * (K – 1)



Input

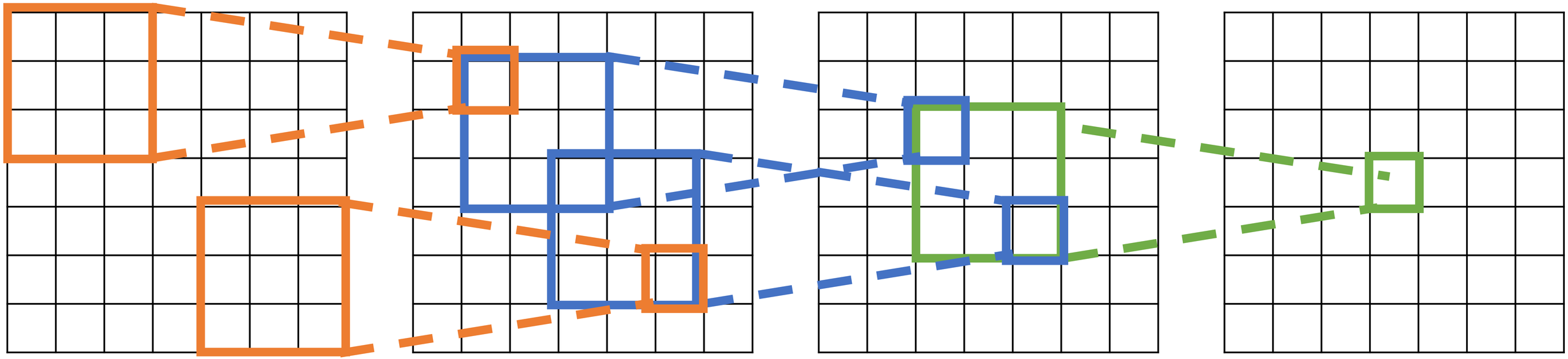
Output

Be careful – ”receptive field in the input” vs “receptive field in the previous layer”

Receptive Field

Each successive convolution adds K – 1 to the receptive field size

With L layers the receptive field size is 1 + L * (K – 1)



Input

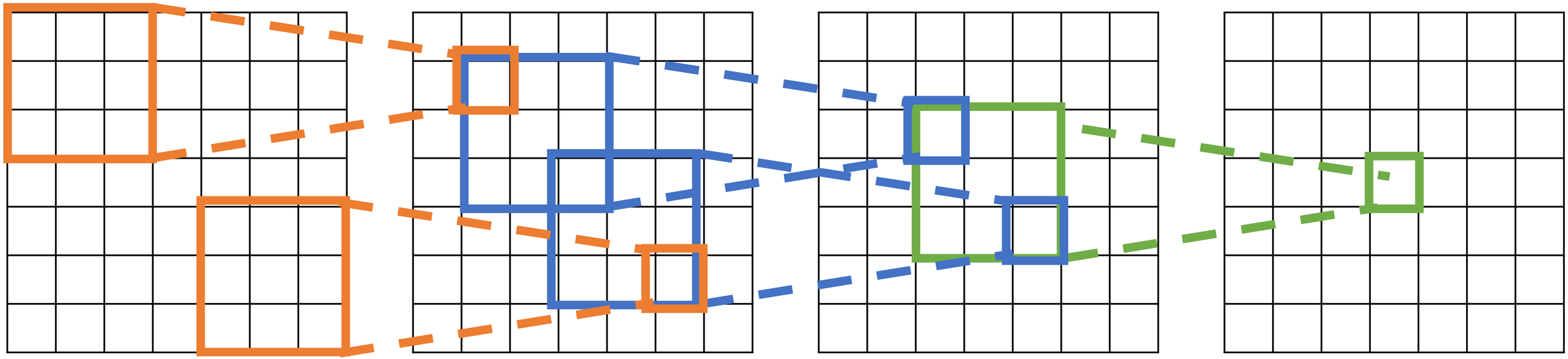
Output

Problem: For large images we need many layers for each output to “see” the whole image image

Receptive Field

Each successive convolution adds K – 1 to the receptive field size

With L layers the receptive field size is 1 + L * (K – 1)



Input

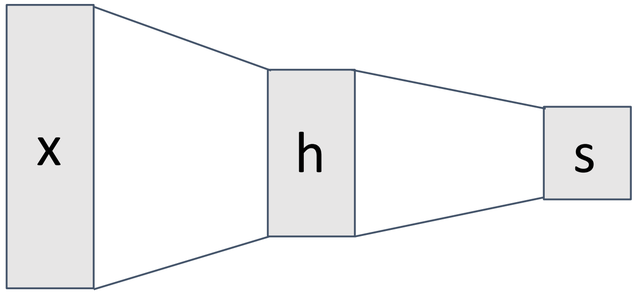
Output

Problem: For large images we need many layers for each output to “see” the whole image image

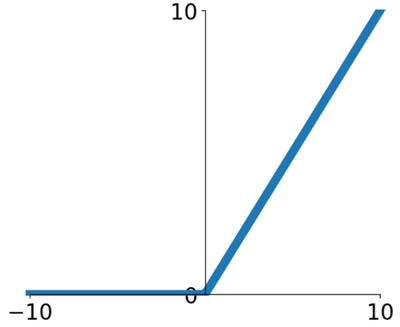
Solution: Downsample inside the network

Components of Convolutional Networks

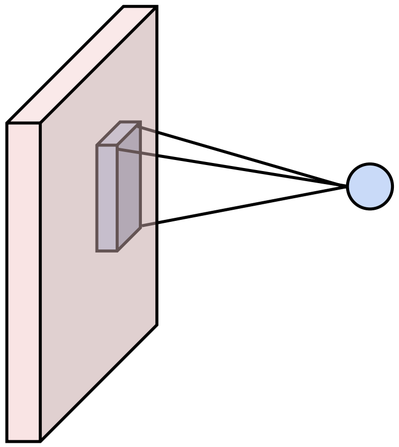
Fully-Connected Layers



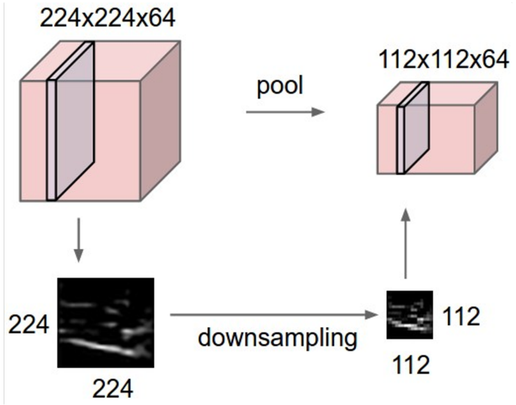
Activation Function



Convolution Layers



Pooling Layers

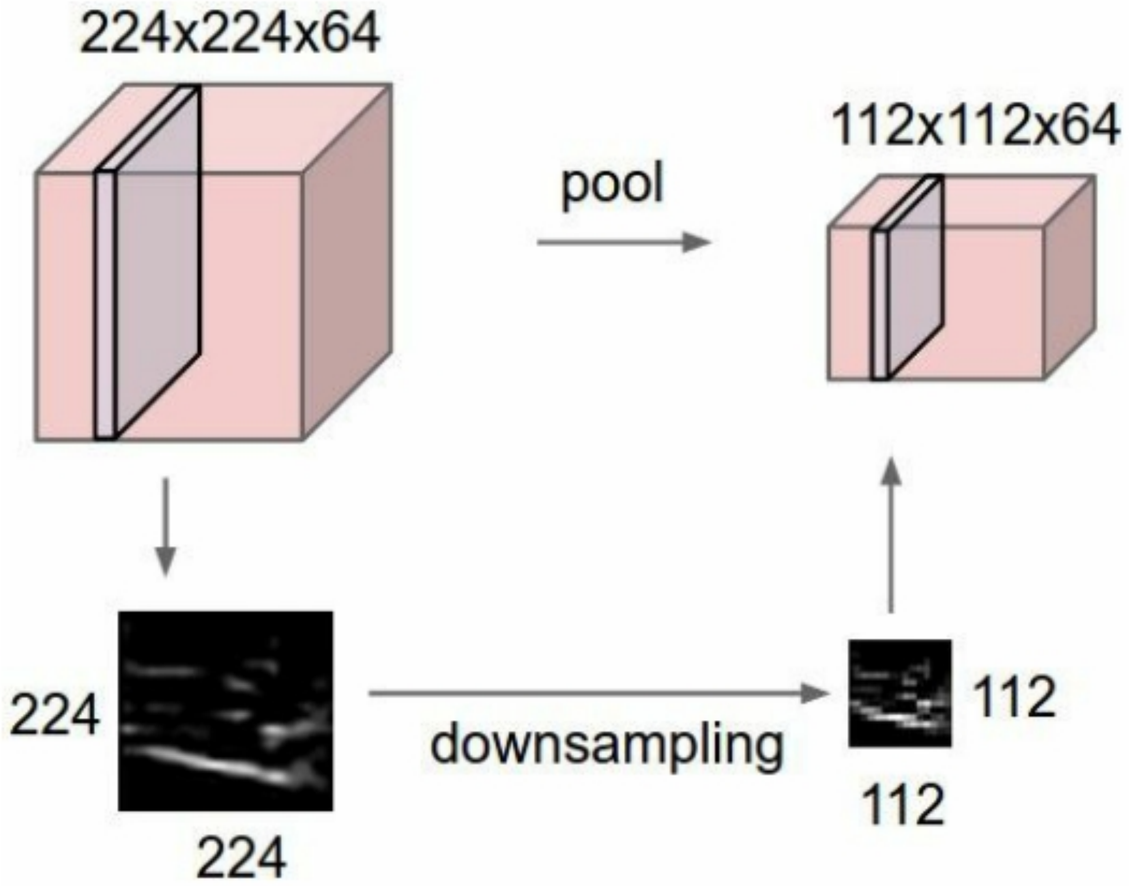


Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Pooling layer

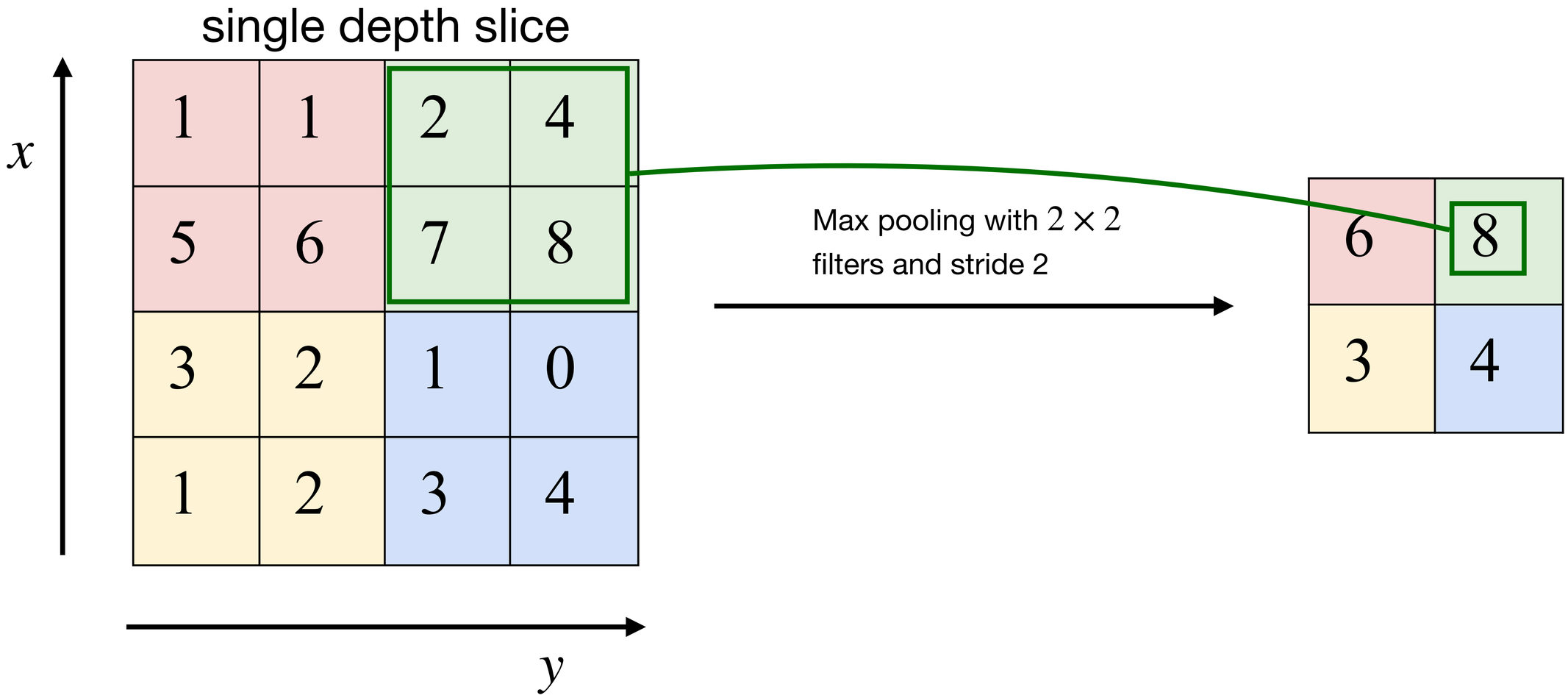
- Makes the representations smaller and more manageable
- Operates over each activation map independently



Credits to Fei-Fei Li, Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

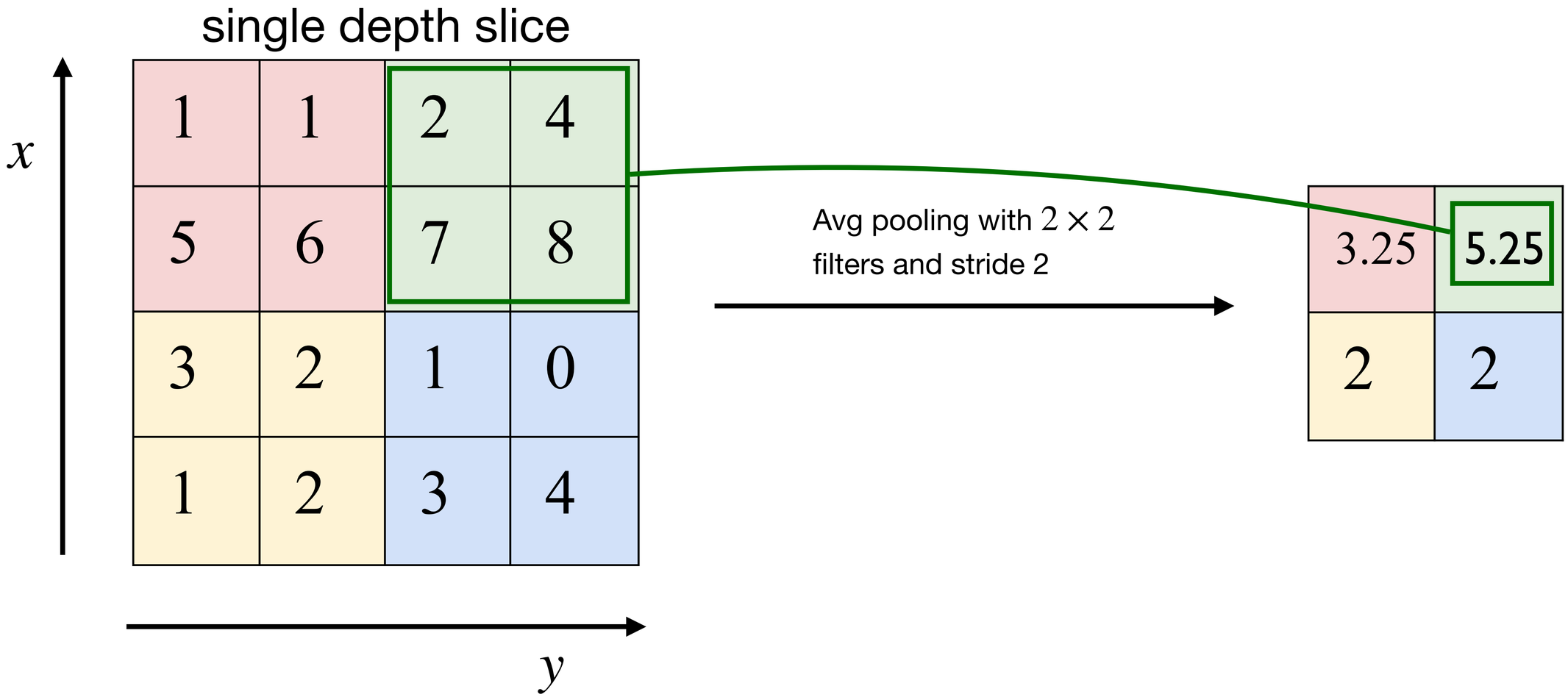
Max pooling

Type of pooling layer



Average pooling

Type of pooling layer



Pooling layer

A summary

➡ Accepts an input of dimension $width_{input} \times height_{input} \times depth_{input}$

➡ Hyper-parameters:

- Filter dimension F
- Stride S

➡ Produces of output of dimension $width_{output} \times height_{output} \times depth_{output}$ where,

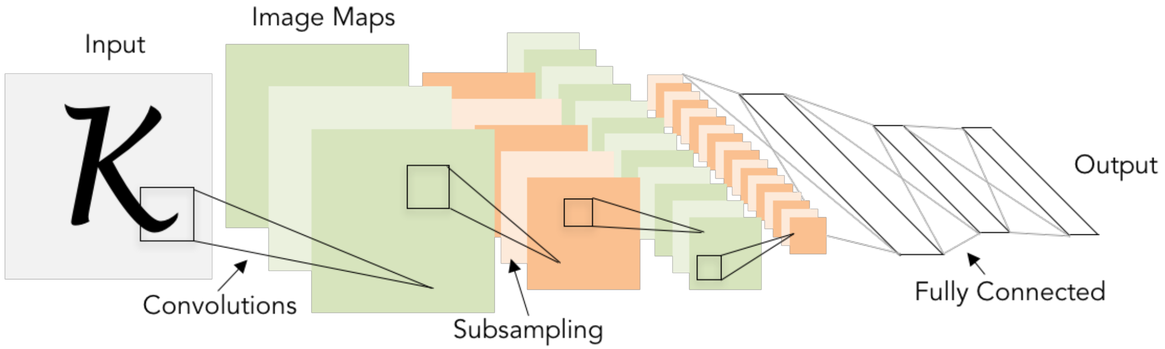
- $width_{output} = (width_{input} - F)/S + 1$
- $height_{output} = (height_{input} - F)/S + 1$
- $depth_{output} = depth_{input}$

Introduces zero parameters since it computes a fixed function of the input

Common settings: F=2, S=2; F=3, S=3

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10



As we go through the network:

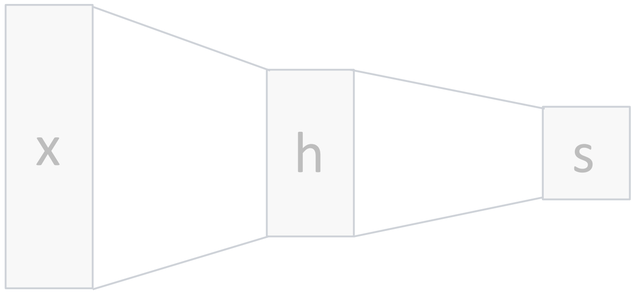
Spatial size **decreases**
(using pooling or strided conv)

Number of channels **increases**
(total "volume" is preserved!)

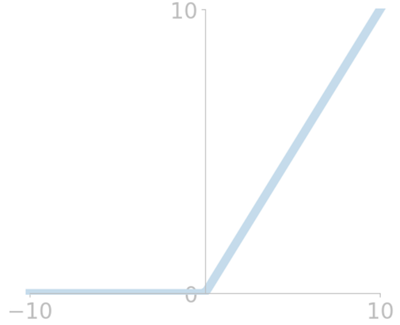
Lecun et al, "Gradient-based learning applied to document recognition", 1998

Components of Convolutional Networks

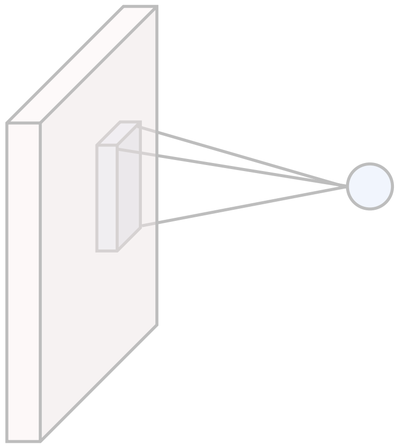
Fully-Connected Layers



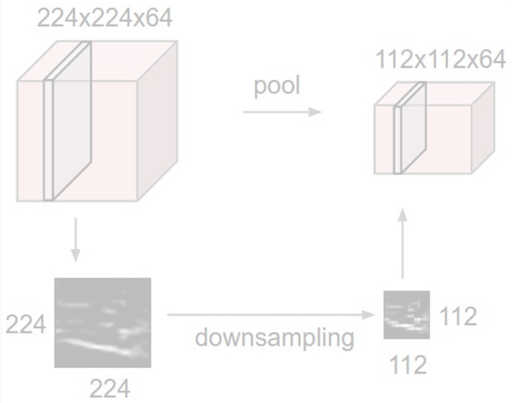
Activation Function



Convolution Layers



Pooling Layers



Normalization

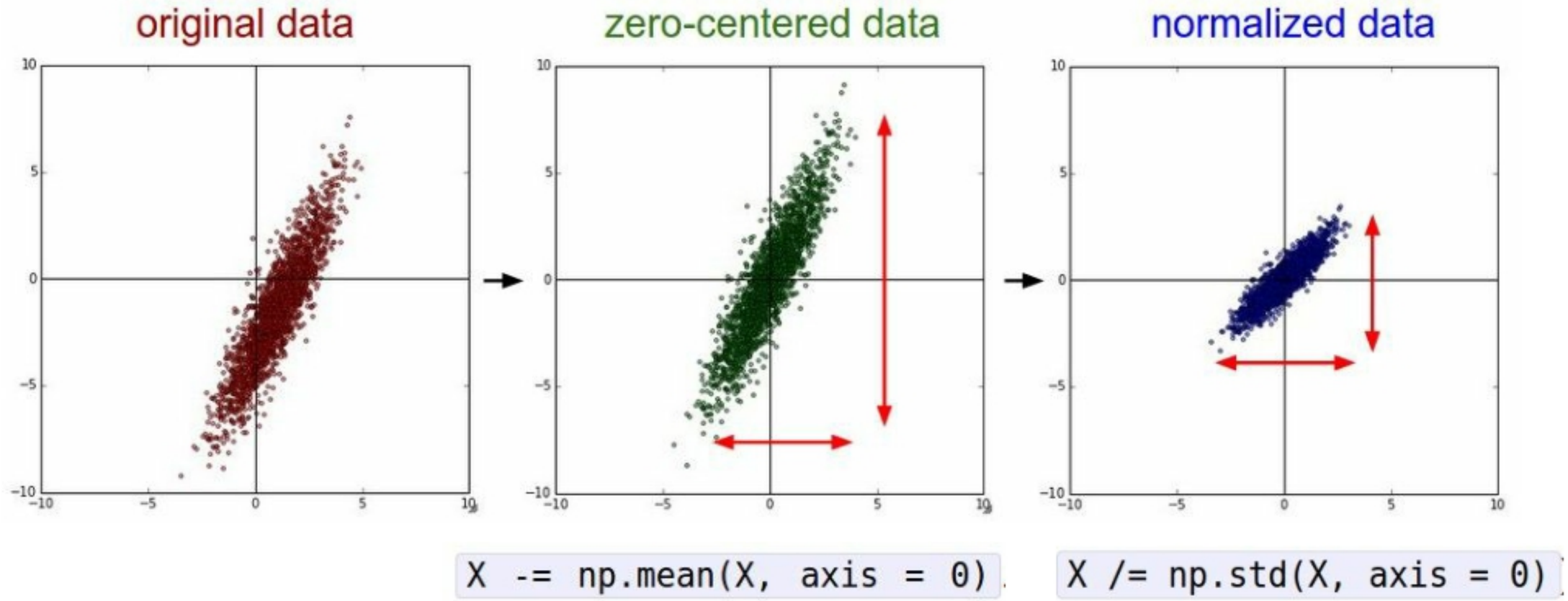
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Batch Normalization

Idea: “Normalize” the outputs of a layer so they have zero mean and unit variance (Recap: We normalize the inputs!)

Intuition: Weight search at same order of magnitude across input dimensions => learning rate good (or bad) for all; Zero-centered inputs facilitate convergence

Recap: Input Normalization



$X \in \mathbb{R}^{N \times D}$ is the data matrix, N=number of samples, D=dimensionality of each sample

Recap: Input Normalization

For images, e.g. CIFAR-10 with size [32, 32, 3] images:

- Subtract the mean image (e.g. AlexNet)

  (mean image = [32, 32, 3] array)

- Subtract per-channel mean value (e.g. VGGNet)

  (mean along each channel = 3 numbers)

Batch Normalization

Idea: “Normalize” the outputs of a layer so they have zero mean and unit variance (Recap: We normalize the inputs!)

Why? Benefits of input normalization across all layers and iterations

We can normalize a batch of activations like this:

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!

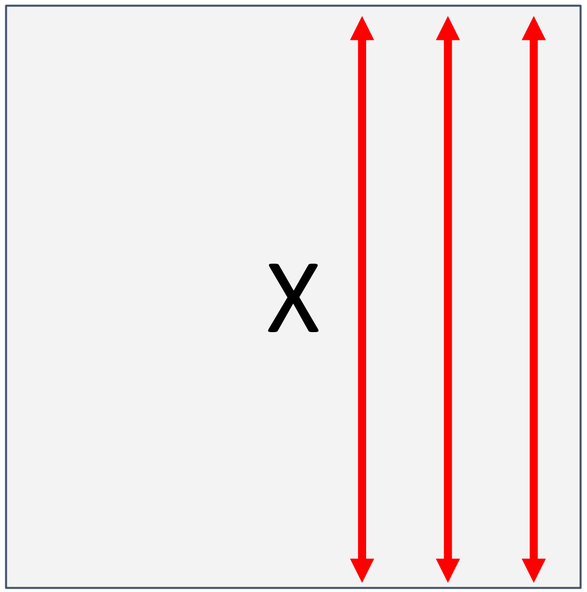
Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D



$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} \left(x_{i,j} - \mu_j\right)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

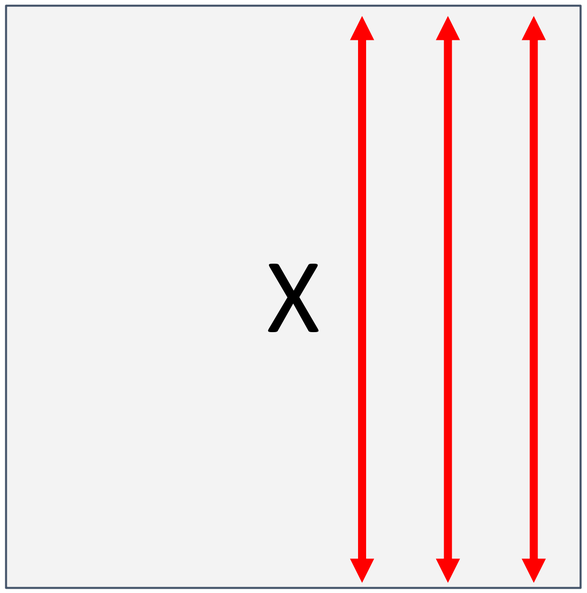
Normalized x, Shape is N x D

N=number of samples, D=dimensionality of each sample

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}\left(x_{i,j} - \mu_j\right)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^{D}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} \left(x_{i,j} - \mu_j\right)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization

Problem: Estimates depend on minibatch; can't do this at test-time!

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^{D}$$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} \left(x_{i,j} - \mu_j\right)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test Time

Input: $x \in \mathbb{R}^{N \times D}$

$\mu_j =$ (Running) average of values seen during training

Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

$\sigma_j^2 =$ (Running) average of values seen during training

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test Time

Input: $x \in \mathbb{R}^{N \times D}$

$\mu_j =$ (Running) average of values seen during training

Per-channel mean, shape is D

Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^D$$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j^{test} = 0$$

For each training iteration:

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

$$\mu_j^{test} = 0.99\, \mu_j^{test} + 0.01\, \mu_j$$

(Similar for σ)

Batch Normalization: Test Time

Input: $x \in \mathbb{R}^{N \times D}$

$\mu_j =$ (Running) average of values seen during training

Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

$\sigma_j^2 =$ (Running) average of values seen during training

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test Time

Input: $x \in \mathbb{R}^{N \times D}$

$\mu_j =$ (Running) average of values seen during training

Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

$\sigma_j^2 =$ (Running) average of values seen during training

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization for CNNs

Batch Normalization for
fully-connected networks

$$x : N \times D$$

Normalize ↓

$$\mu, \sigma : 1 \times D$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$$x : N \times C \times H \times W$$

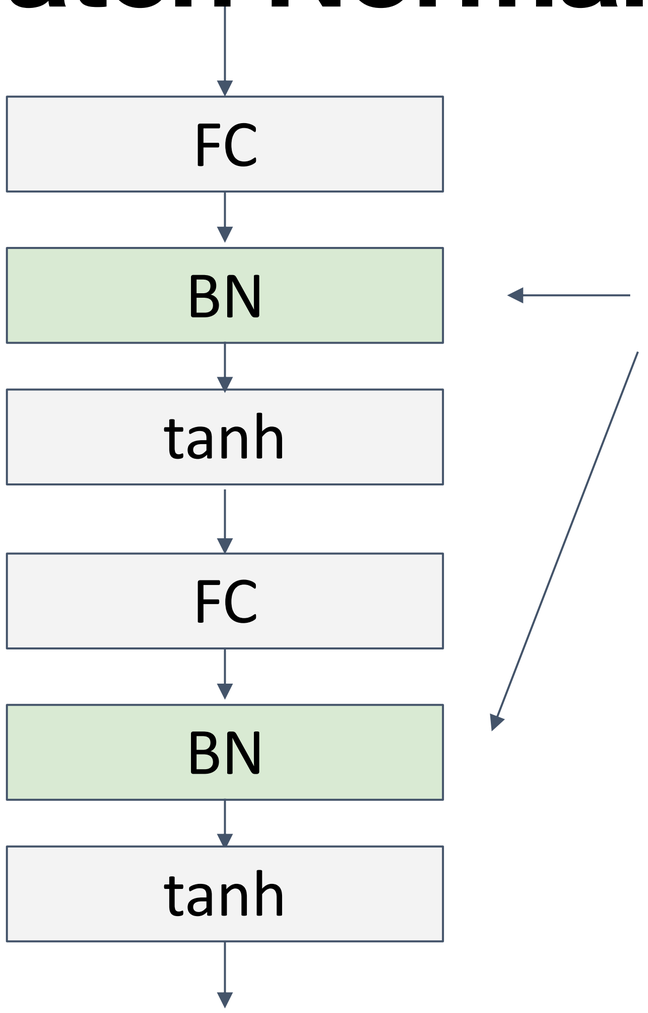
Normalize ↓ ↓ ↓

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Batch Normalization

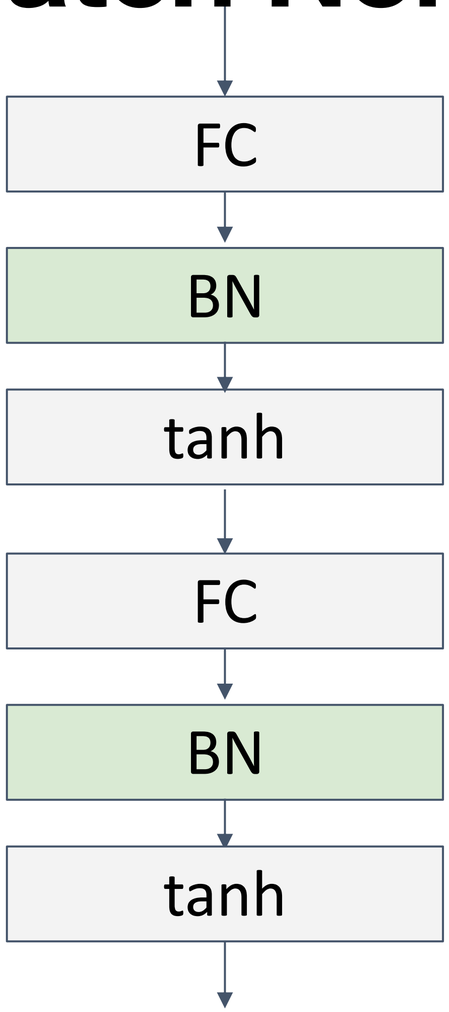


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

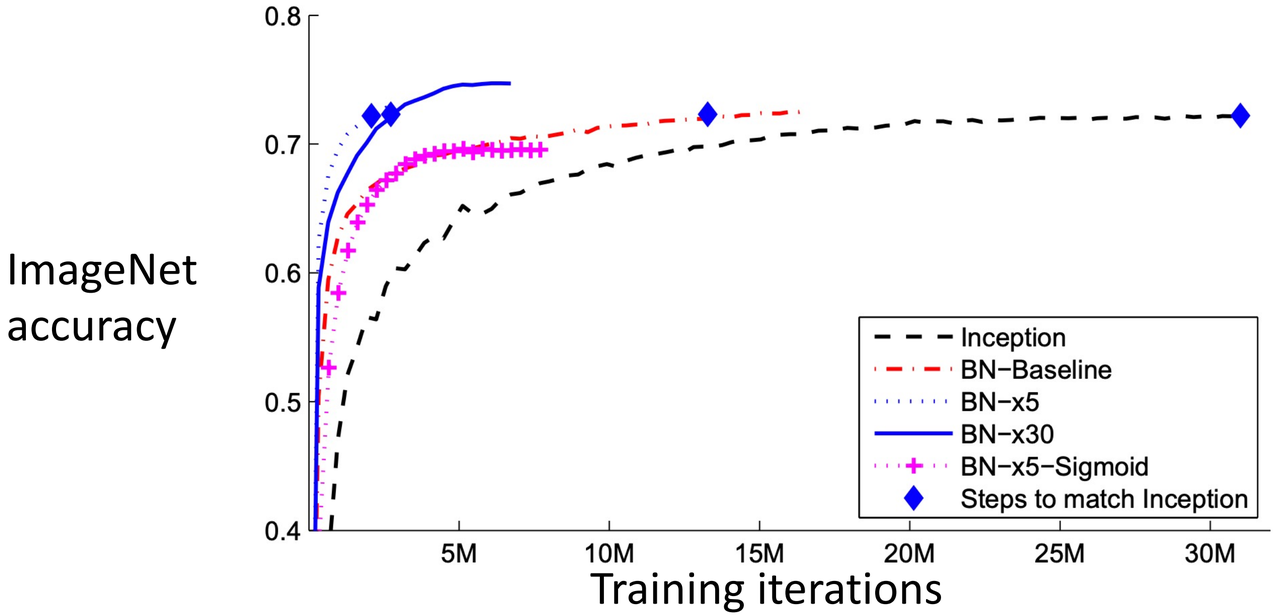
$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

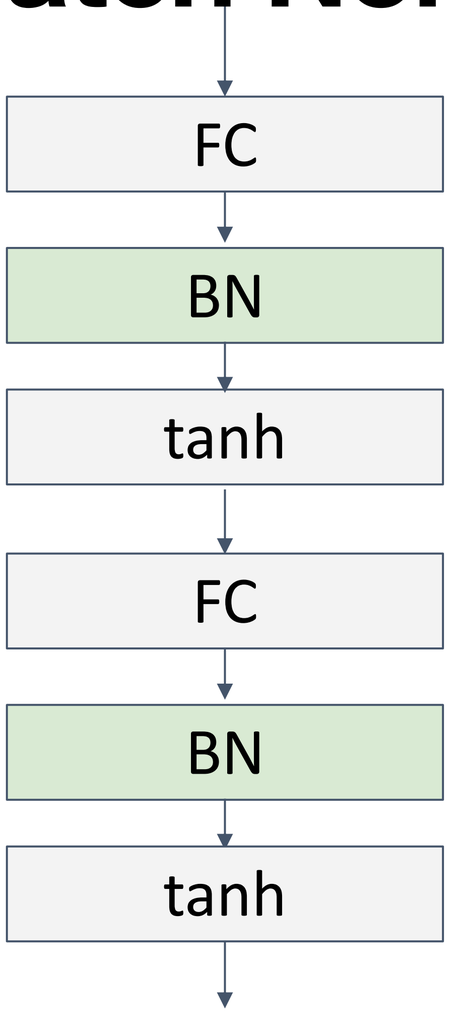


- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!



Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization



- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is a very common source of bugs!

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Layer Normalization

Batch Normalization for
fully-connected networks

$$x : N \times D$$

Normalize ↓

$$\mu, \sigma : 1 \times D$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Layer Normalization for fully-connected networks
Same behavior at train and test!
Used in RNNs, Transformers

$$x : N \times D$$

Normalize ↓

$$\mu, \sigma : N \times 1$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Instance Normalization

Batch Normalization for convolutional networks

$$x : N \times C \times H \times W$$

Normalize

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Instance Normalization for convolutional networks

$$x : N \times C \times H \times W$$

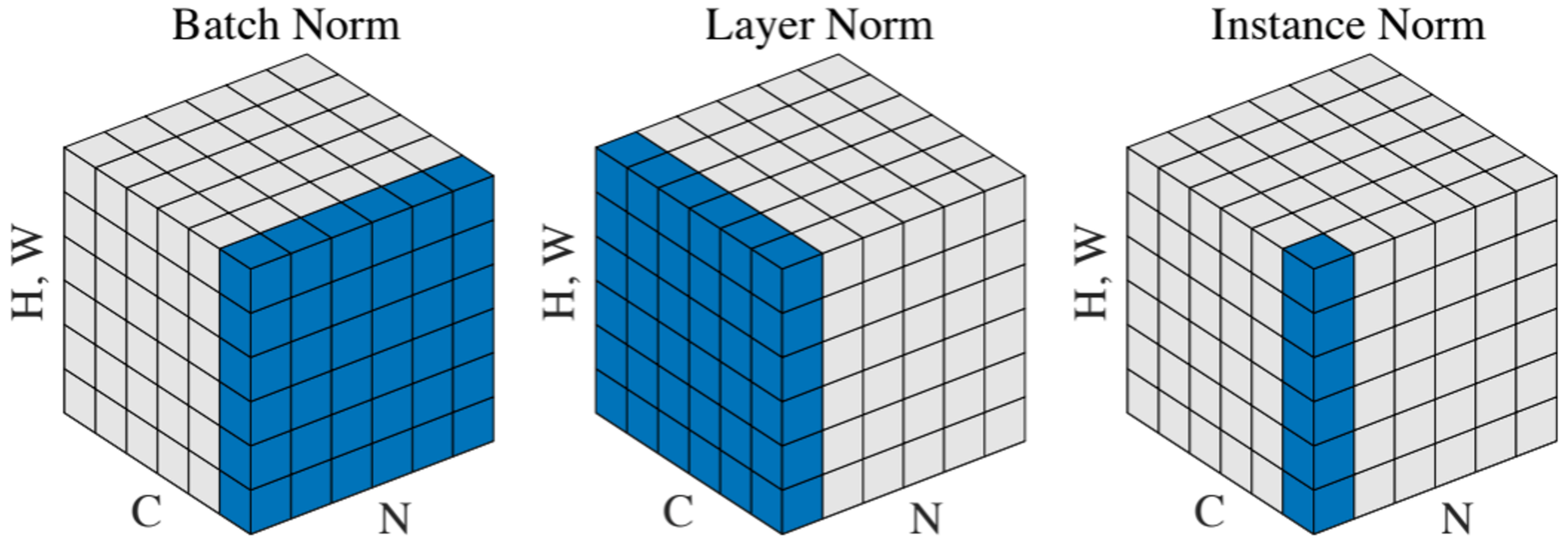
Normalize

$$\mu, \sigma : N \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Comparison of Normalization Layers



What does Layer- and Instance Norm do to the receptive field?

Wu and He, “Group Normalization”, ECCV 2018

Convolutional Neural Networks: Summary

- CNNs stack conv-, pooling-, fully connected layers, and normalizaion
- What is the right way to combine these components?
- Typical classification architectures look like:

  [(conv,ReLU)*N, Pool]*M, (FC,ReLU)*K, Softmax

  Typical $M \in \{5, \ldots, 152\}$, $N \in \{2, 3\}$, and $K \in \{0, 1, 2\}$

- Batch-, Layer- or InstanceNorm after conv or FC for faster convergence

Advanced Optimization

Stochastic Gradient Descent

Momentum

Adam

Recap

Linear Regression with Gradient Descent

Hypothesis = $h_\theta = \sum_{j=0}^{n} \theta_j x_j$

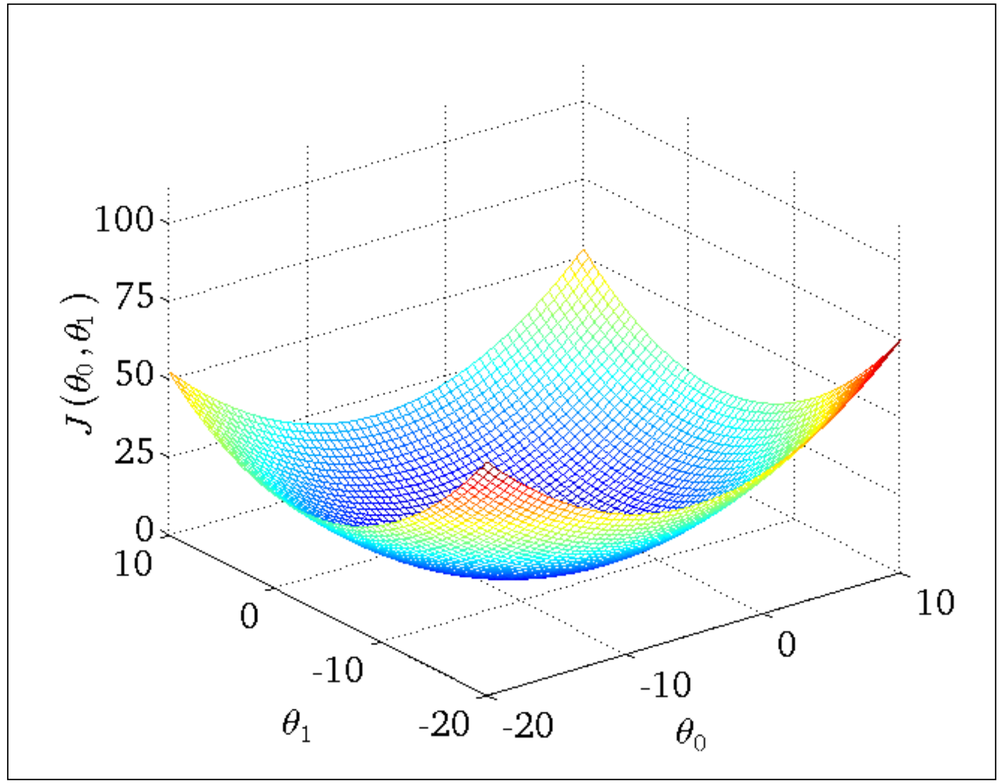
Cost function = $J(\theta) = \frac{1}{2m} \sum_{i=q}^{m} (h_\theta(x^{(i)} - y^{(i)})^2$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

for every $j = 0,1,....,n$

}



Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Recap

Linear Regression with Gradient Descent

Hypothesis = $h_\theta = \sum_{j=0}^{n} \theta_j x_j$

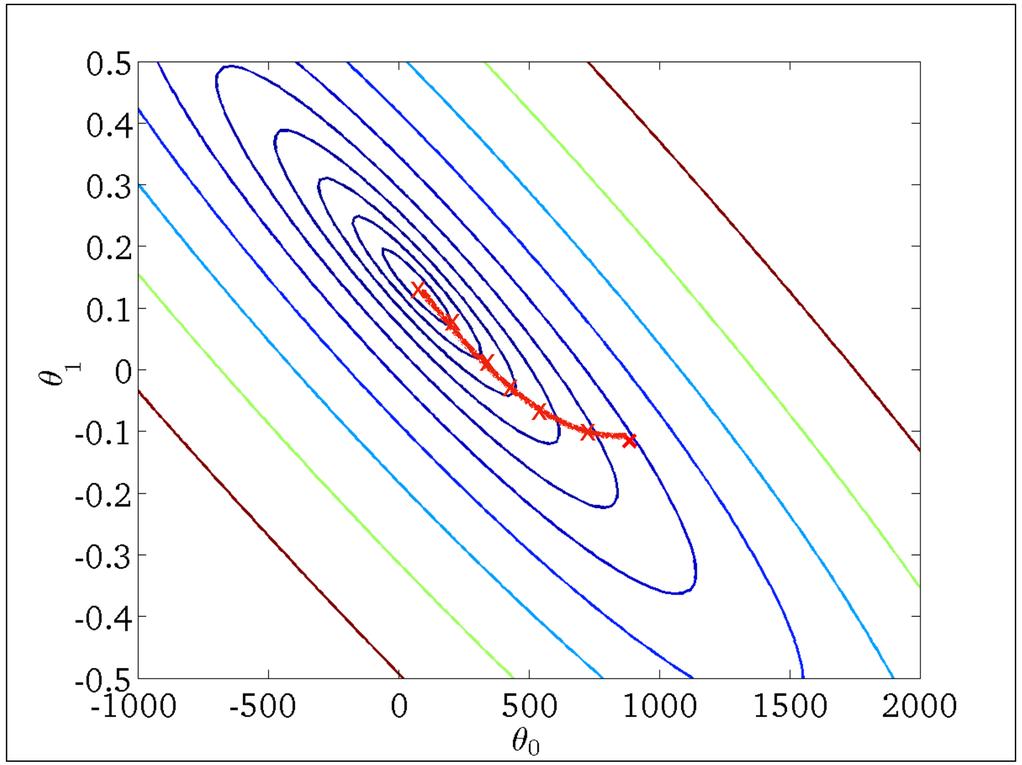
Cost function = $J(\theta) = \frac{1}{2m} \sum_{i=q}^{m} (h_\theta(x^{(i)} - y^{(i)})^2$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

for every $j = 0,1,....,n$

}



Batch gradient descent:
For large datasets, computation is expensive

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Batch- vs. stochastic gradient descent

Batch gradient descent

$$J(\theta) = \frac{1}{2m}\sum_{i=q}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$$

Repeat {

$$\theta_j := \theta_j - \alpha\boxed{\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}}$$

$$j = 0,1,.....,n$$

$$= \frac{\partial}{\partial\theta_j}J(\theta)$$

Stochastic gradient descent

$$cost(\theta,(x^{(i)},y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}cost(\theta,x^{(i)},y^{(i)}))$$

1. Randomly shuffle dataset
2. Repeat {
   for i = 1,...,m {

$$\theta_j := \theta_j - \alpha\boxed{(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}}$$

   for j =0,…n

   } }

$$= \frac{\partial}{\partial\theta_j}cost(\theta,(x^{(i)},y^{(i)}))$$

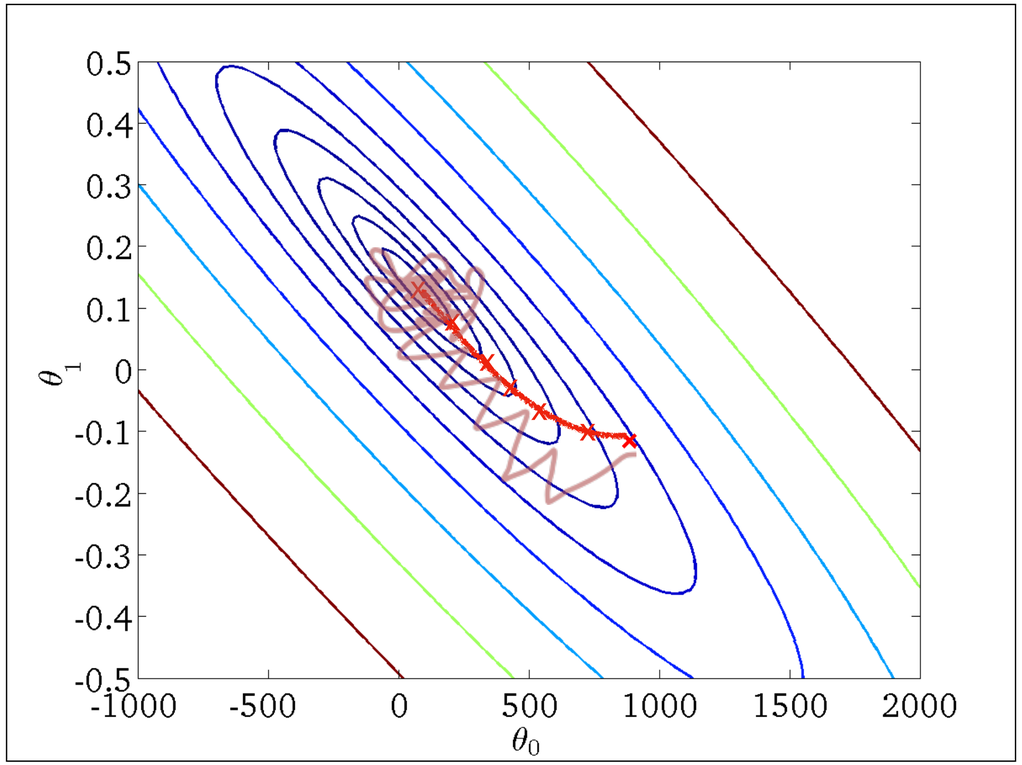
Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Stochastic gradient descent

1. Randomly shuffle dataset
2. Repeat { **= Epoch**
   for i = 1,...,m {

$$\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

   for j =0,…,n
   }
   }



Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- When the training set size m is very large, SGD can be much faster than gradient descent
- The cost function $J_{train}(\theta) = \frac{1}{2m}\sum_{i=q}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$ should go down with every iteration of batch gradient descent (assuming appropriate α) but not necessarily with SGD
- SGD is applicable only to linear regression but not to other models (like logistic regression or neural networks)
- Before beginning the main loop of SGD, it is a good idea to “shuffle” your training data into a random order.

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Mini-batch Gradient Descent

Batch gradient descent: Use all *m* examples in each iteration
Stochastic gradient descent: Use 1 example in each iteration
Mini-batch gradient descent: Use b examples in each iteration

b = mini-batch size e.g., b= 10

$$\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{i=k \cdot b}^{(k+1) \cdot b - 1} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

vectorized computation, parallelize

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

- Suppose you use mini-batch gradient descent on a training set of size m, and you use a mini-batch size of B. The algorithm becomes the same as batch gradient descent if:

  - B = 1
  - B = m/2
  - B = m 
  - NOTA

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Stochastic gradient descent convergence

How to set the learning rate?

Batch gradient descent:

Plot loss as a function of the number of gradient steps

Stochastic gradient descent:

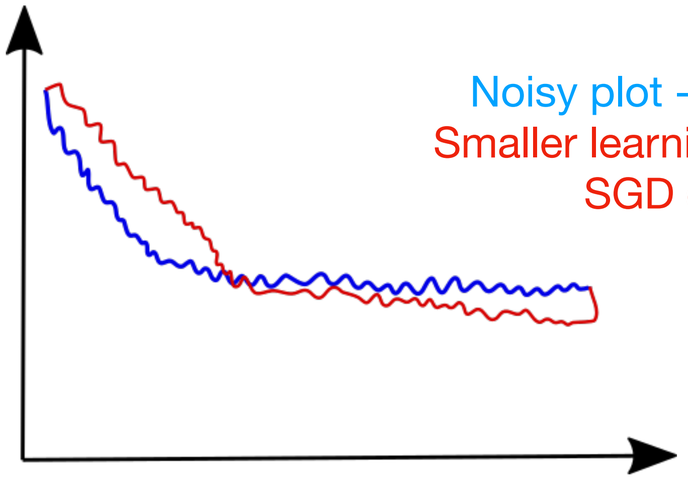
Compute loss contribution of current sample in each gradient step

Every 100 iterations, plot losses averaged over the last 100 samples

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

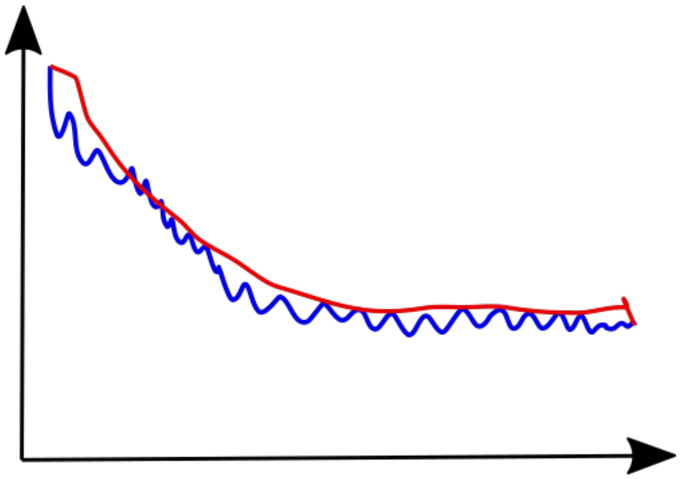
Checking for convergence

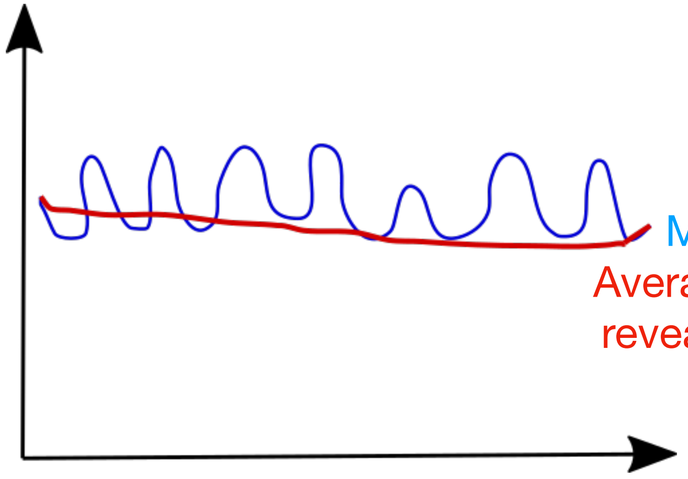
Plot loss averaged over last (e.g.) 1000 samples



Noisy plot - cost is going down
Smaller learning rate -learns slowly
SGD oscillates less

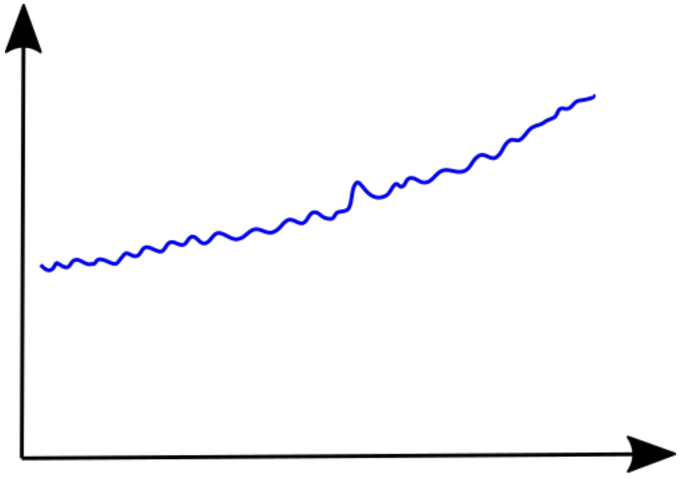
Smoother curve by averaging
over more samples





Model is not learning?
Averaging over more samples
reveals that cost going down

Diverging model
Decrease learning rate



Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- Picking a learning rate α that is very small has no disadvantage and can only speed up learning
- If we reduce the learning rate α (and run SGD), its possible that we may find a set of better parameters than with larger α 
- If we want SGD to converge to a (local) minimum rather than wander or "oscillate" around it, we should slowly increase α over time.
- If we plot $cost(\theta, (x^{(i)}, y^{(i)}))$ (averaged over the last 1000 examples) and SGD does not seem to be reducing the cost, one possible problem may be that the learning rate α is poorly tuned 

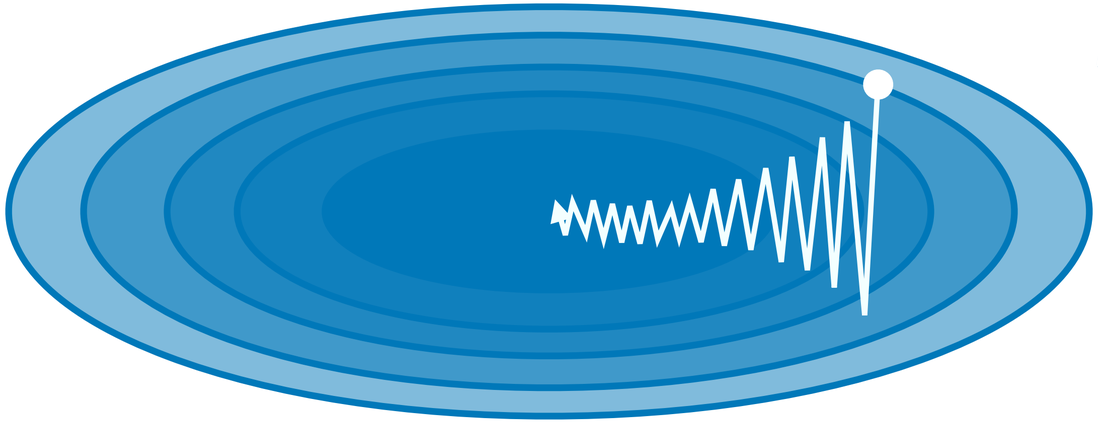
Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Problems of SGD

#1 Jitter

Large gradient in one direction, small gradient in other

⟶ *Slow progress, jitter*

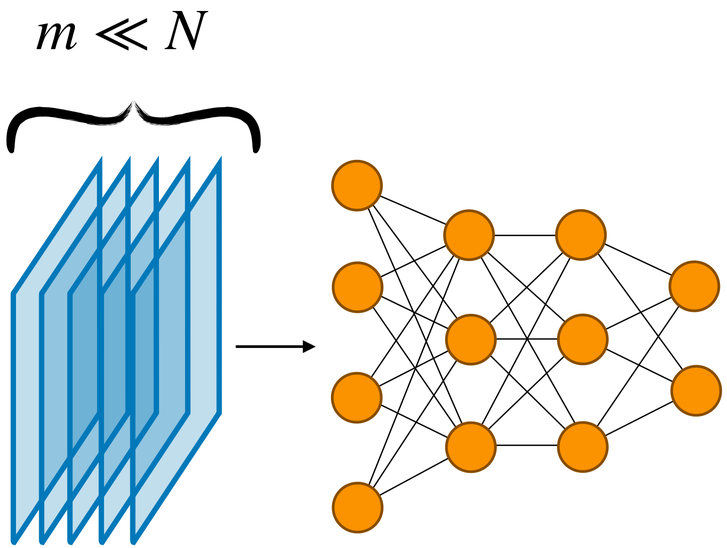


Loss function

High ratio between smallest and largest value in Hessian matrix - loss function has high condition number

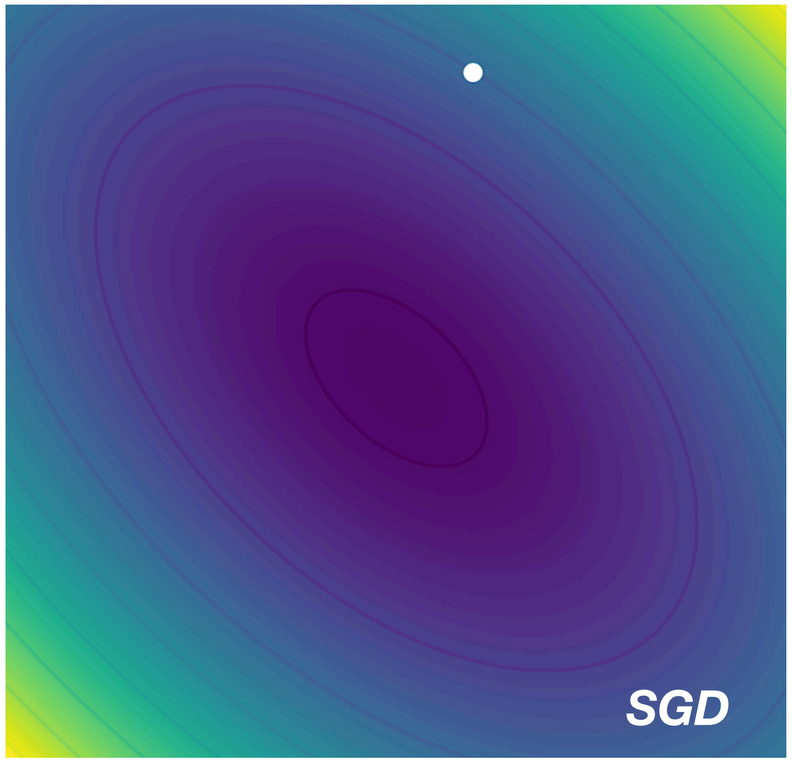
Problems of SGD

#2 noisy gradients



Gradients come from mini-batches

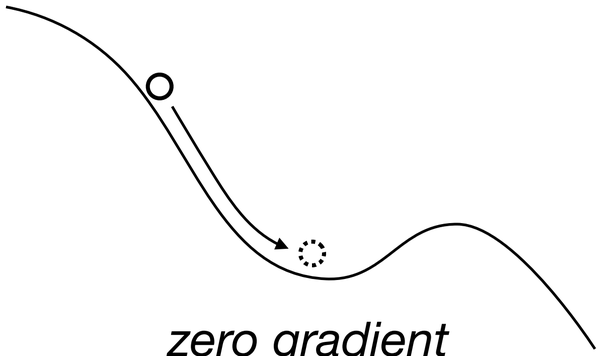
(average gradient of mini-batch)



Problems of SGD

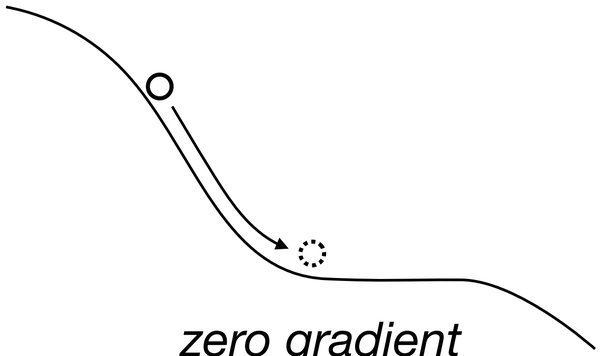
#3 local minima and saddle points

local minimum



zero gradient

saddle point → *More common in high dimensions*

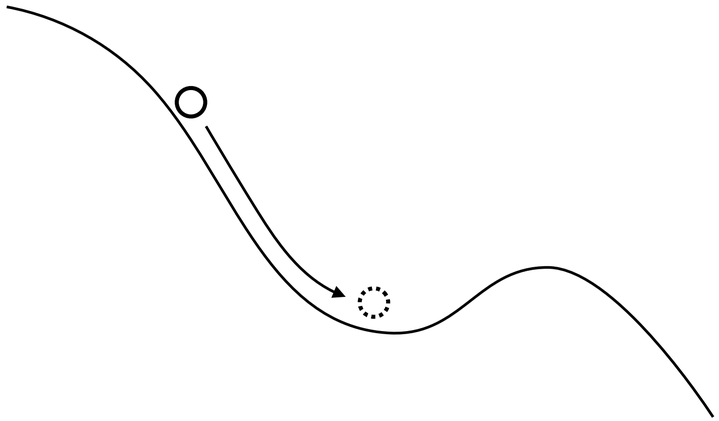


zero gradient

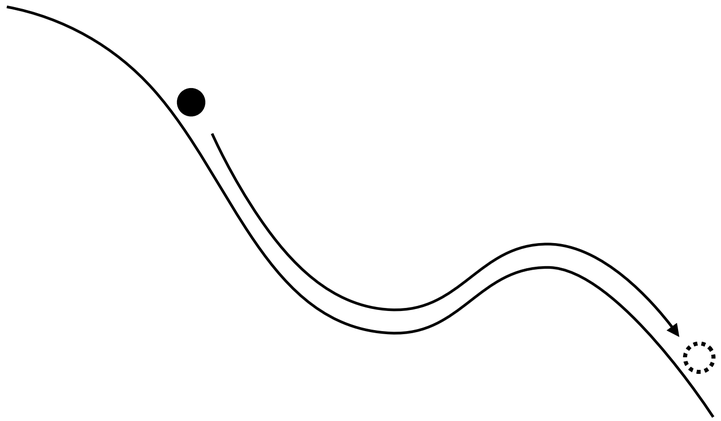
Introducing a momentum

potential picture

SGD: Weightless particle



Introduce mass (momentum)



Introducing a momentum

Algorithm

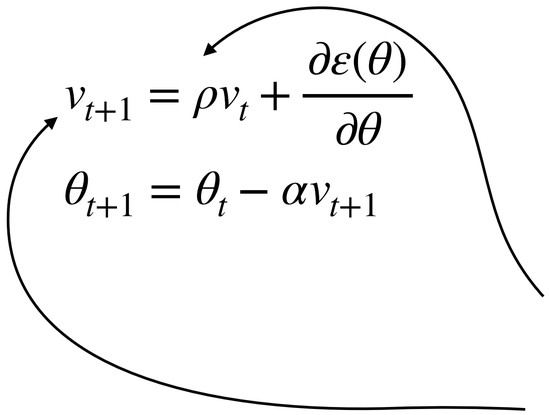
SGD

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

```
for trainingSteps:

  weights += -learning_rate*gradient
```

SGD + Momentum



$$v_{t+1} = \rho v_t + \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

```
vel = 0

for trainingSteps:

  vel      = rho*vel + gradient
  weights += -learning_rate*vel
```

Friction ρ *typically set to 0.9 or 0.99*

(negative) ***Velocity*** v *is an accumulation of gradients (exponential decaying influence)*

Introducing a momentum

Algorithm

SGD

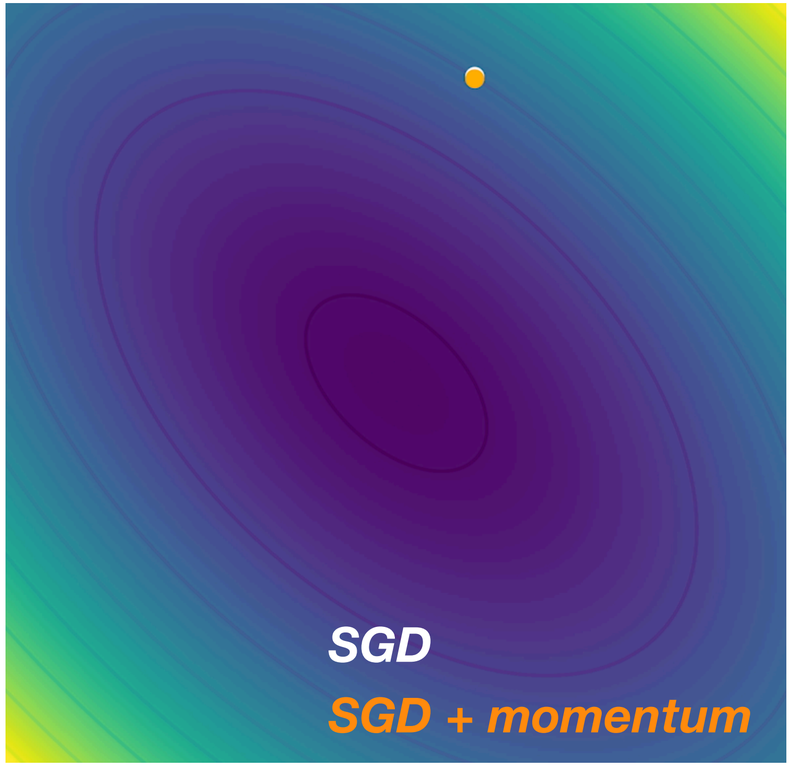
$$\theta_{t+1} = \theta_t - \alpha \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

SGD + Momentum

$$v_{t+1} = \rho v_t + \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

Tackles previous problems (#1 jitter, #2 noisy gradients, #3 local minima / saddle points)

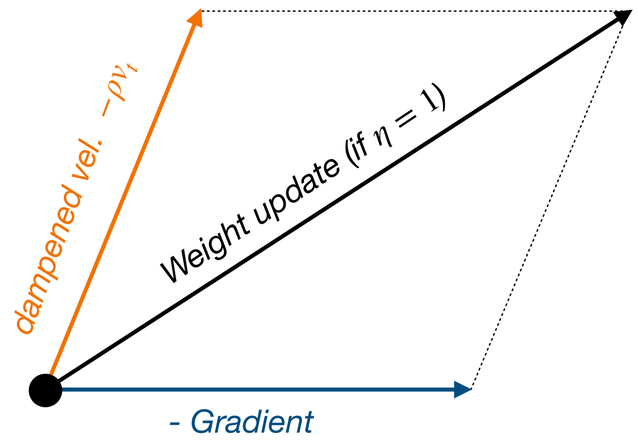


Nesterow Momentum

Momentum:

$$v_{t+1} = \rho v_t + \nabla \varepsilon(\theta)$$

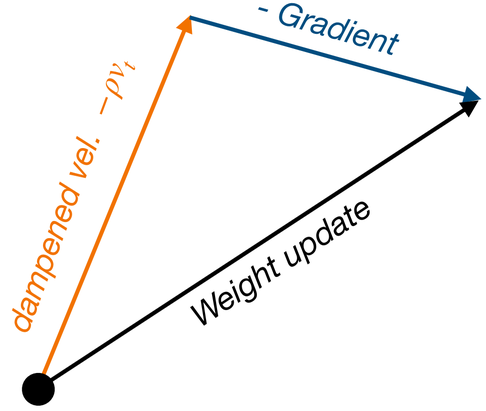
$$\theta_{t+1} = \theta_t - \eta v_{t+1}$$



Nesterov Momentum:

$$v_{t+1} = \rho v_t + \nabla \varepsilon(\theta_t - \eta \rho v_t)$$

$$\theta_{t+1} = \theta_t - \eta v_{t+1}$$



AdaGrad

SGD

```
for _ in range(training_steps):

  grad = compute_gradients(weights)
  weights += -learning_rate * grad
```

AdaGrad

```
grad_squared = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)
  grad_squared += grad * grad
  weights += -learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

Scaling gradients in each dimension by accumulation of squared gradients

RMSprop

AdaGrad

```
grad_squared = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)
  grad_squared += grad * grad
  weights += -learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

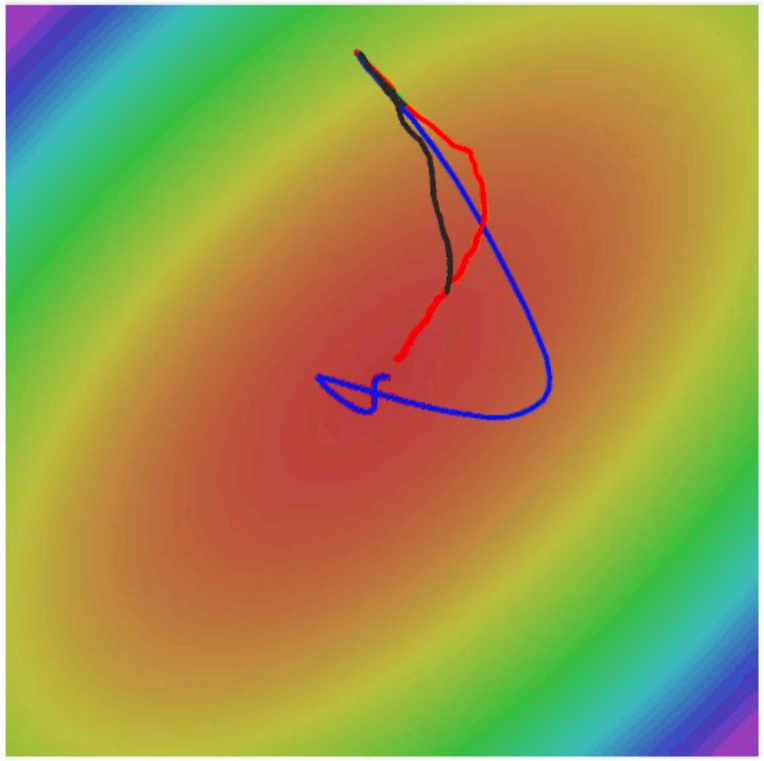
RMSProp

```
grad_squared = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)
  grad_squared += decay_rate * grad_squared + (1 - decay_rate) * grad * grad
  weights += -learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

RMSprop



Adam

Derivation

```
first_moment = 0
second_moment = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  weights -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Momentum

AdaGrad / RMSProp

Q: What happens at first timestep?

Adam

Algorithm

```
first_moment = 0
second_moment = 0

for t in range(1, training_steps + 1):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  first_unbias  = first_moment  / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)

  weights -= learning_rate * first_unbias / (np.sqrt(second_unbias)  +1e-7)
```

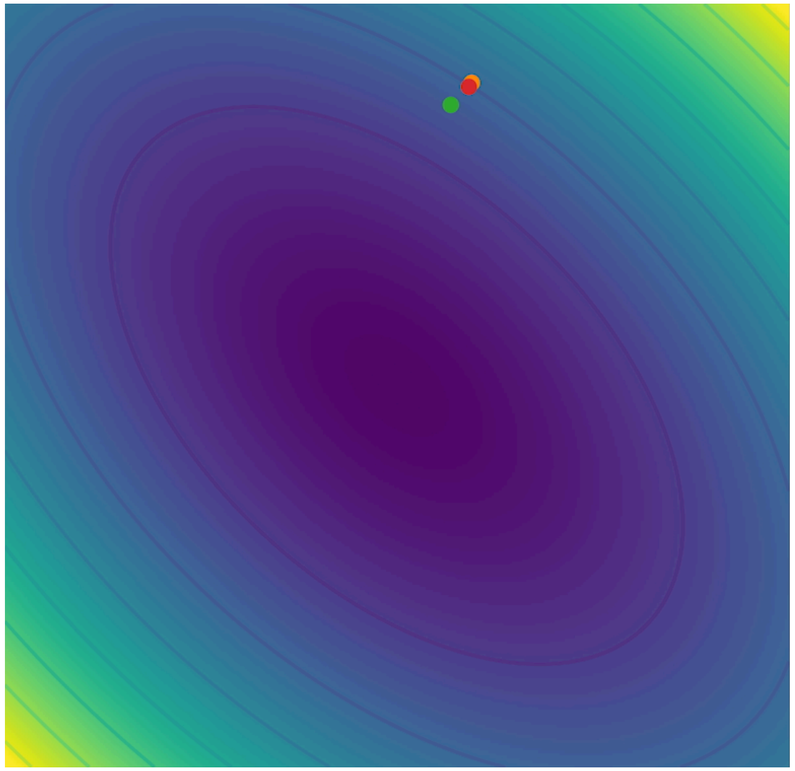
Momentum

Bias correction *accounting for first / second moment estimates starting at zero*

AdaGrad / RMSProp

Practical tip: *Adam with beta1 = 0.9 / 0.999 and learning_rate = 1e-3 / 5e-4 is good first choice for many models.*

Adam



SGD

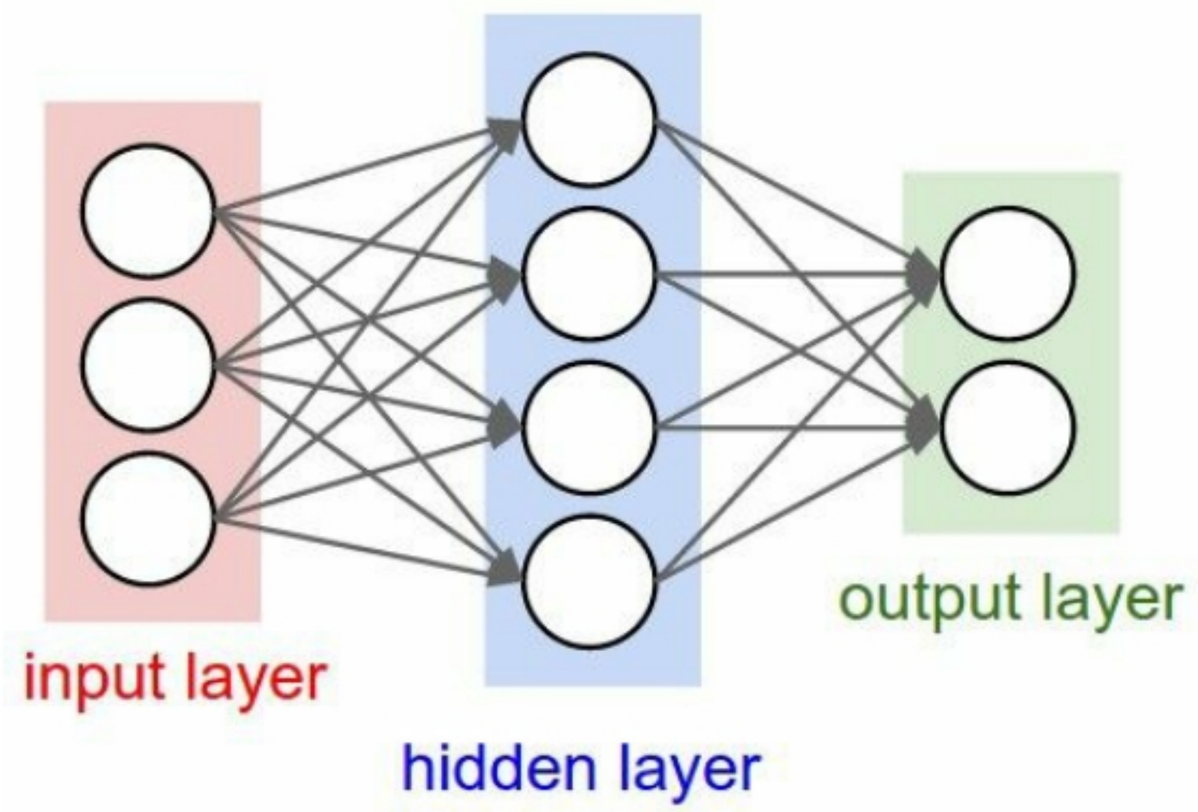
SGD + momentum

RMSProp

Adam

Weight Initialization

Q: What happens when $\theta \equiv$ const initialization is used? E.g., $\theta \equiv 0$?



Credits to Fei-Fei Li, Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

First idea: **Small random numbers**

(gaussian with zero mean and 0.01 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

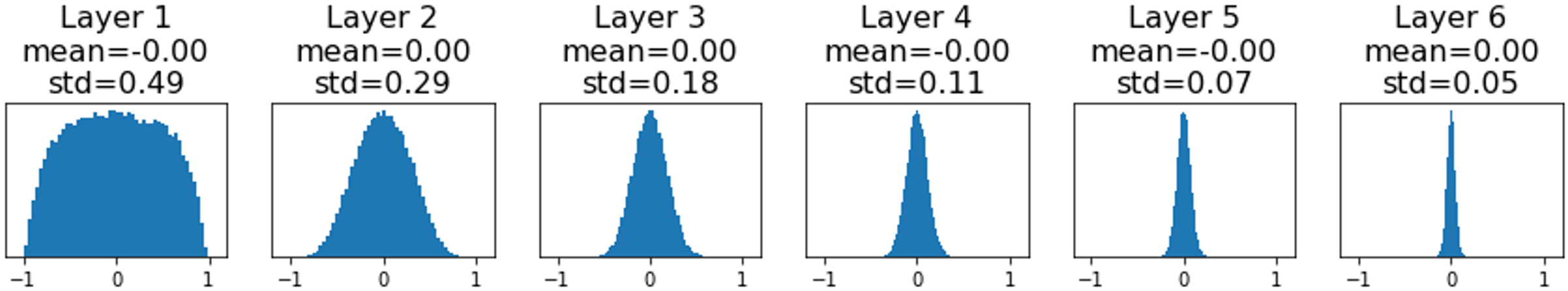
Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?



Weight Initialization: Activation Statistics

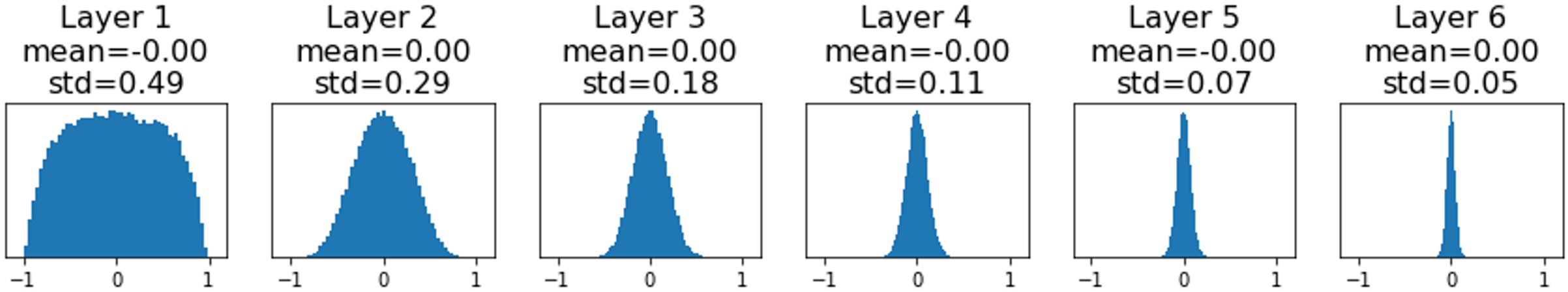
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: All zero, no learning =(



Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

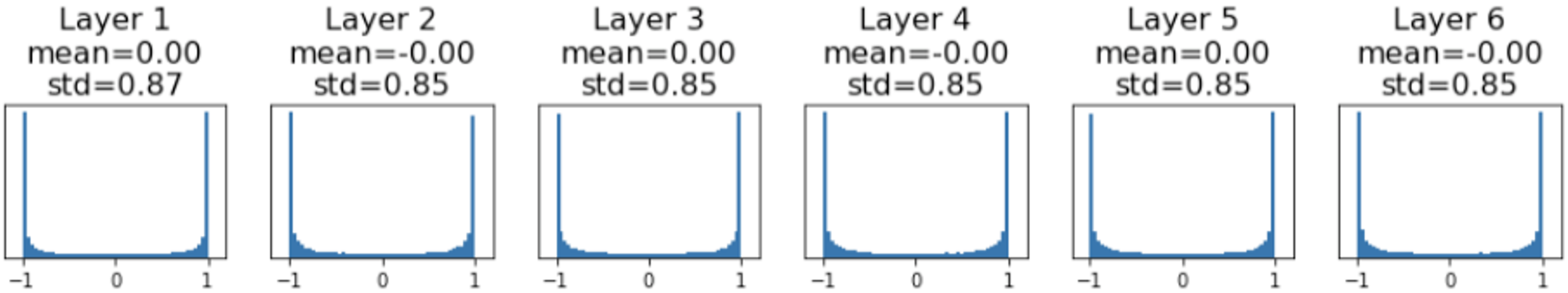
Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



Weight Initialization: Activation Statistics

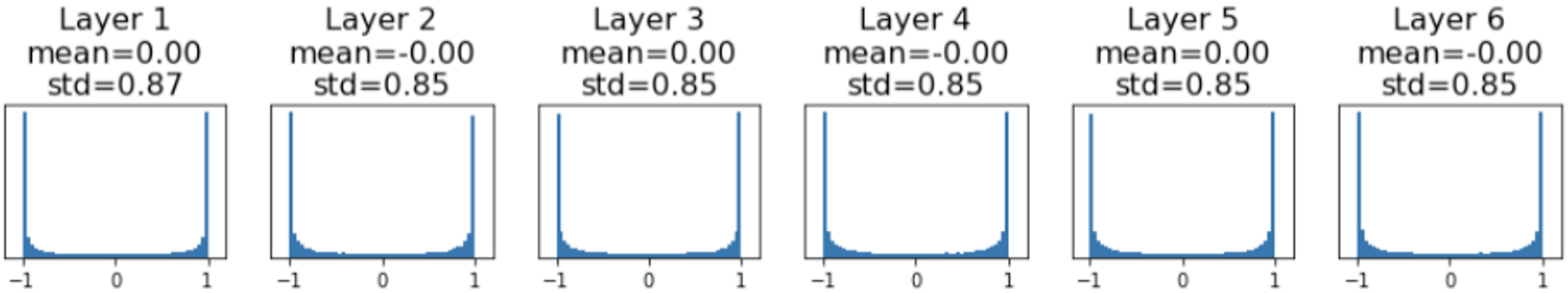
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



Credit: Justin Johnson, U Michigan, Deep Learning for Computer Vision 2019

Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

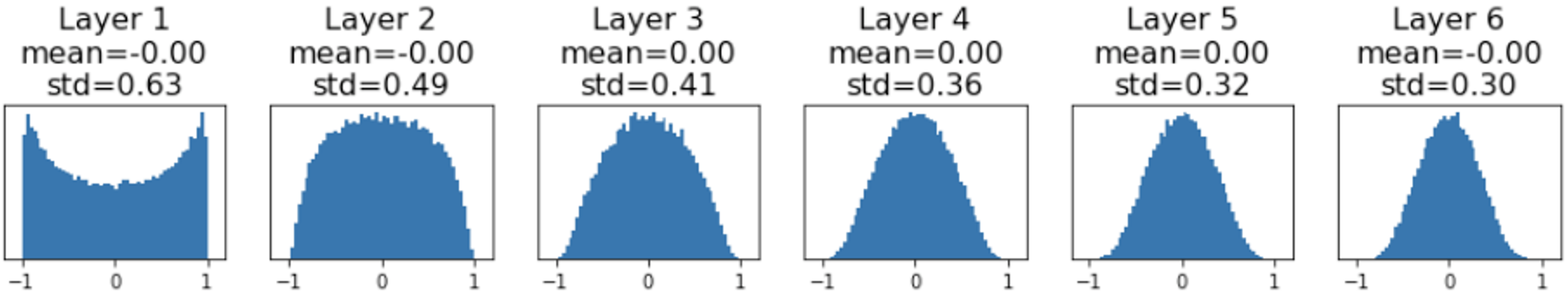
Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

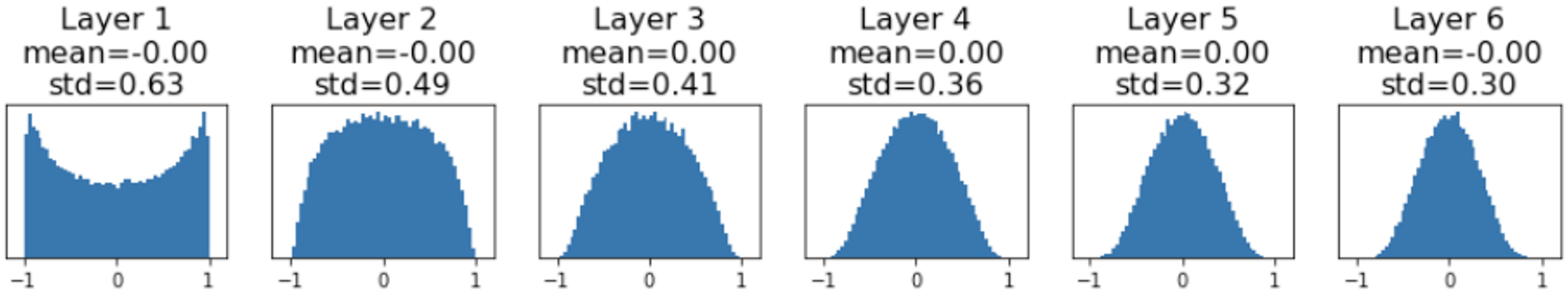
Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is kernel_size^2 * input_channels



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

y = Wx

$$y_i = \sum_{j=1}^{Din} x_j w_j$$

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$y = Wx$

$$y_i = \sum_{j=1}^{Din} x_j w_j$$

$Var(y_i) = Din * Var(x_i w_i)$

[Assume x, w are iid]

$= Din * Var(x_i) * Var(w_i)$

[Assume x, w zero-centered,
assume x^2, w^2 uncorrelated]

If $Var(w_i) = 1/Din$ then $Var(y_i) = Var(x_i)$

Weight Initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

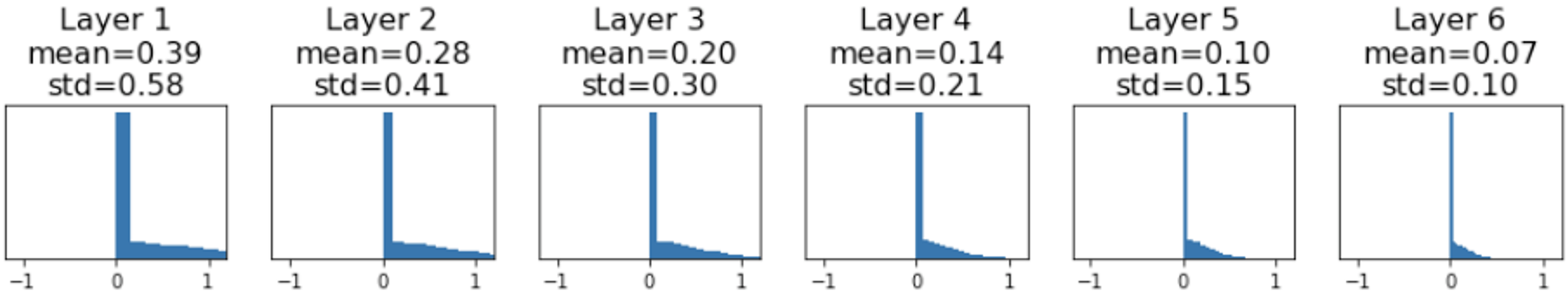
Weight Initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(

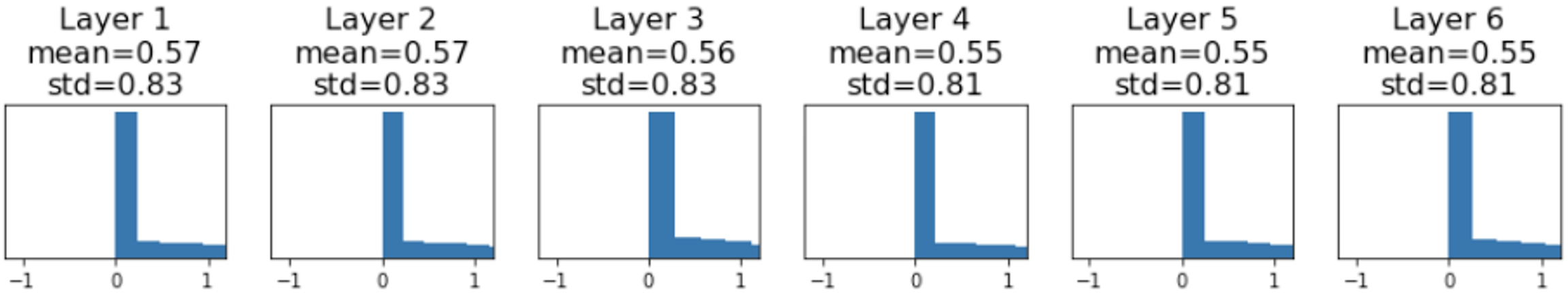


Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

”Just right” – activations nicely scaled for all layers



He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization; Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Course Outline

- Introduction to Image Analysis
- Machine Learning Basics: Linear Regression, Basic Classifiers
- Neural Networks
- **Convolutional Neural Networks**
- Recurrent Neural Networks, Transformers, Diffusion
- Model Interpretability
- Probabilistic Machine Learning
- Generative Models