Introduction to Computer Vision

Lecture 4 - Deep Learning II

Prof. He Wang

Outline

- Set up the task
- Prepare the data —> Need a labeled dataset.
- Built a model —> construct your neural network
- Decide the fitting/training objective —> Loss function
- Perform fitting —> Training by running optimization
- Testing —> Evaluating on test data

Task: Binary Classification — Is This Digit a 5?

Data

Model

Output

$x =$ 

$h($  $,\theta)$

?

$y \in \{0,1\}$

Parameters or weights

Probability of being a 5

$p(y = 1 \mid x)$

MINIST Dataset of handwritten digits

Loss: Negative Log-likelihood

- Loss: the thing you want to minimize

- Negative log-likelihood (NLL) loss

$$\mathscr{L}(\theta) = -\log p(Y|X;\theta)$$

$$= -\sum_{i=1}^{n} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

Outline

- Set up the task
- Prepare the data —> Need a labeled dataset.
- Built a model —> construct your neural network
- Decide the fitting/training objective —> Loss function
- Perform fitting —> Training by running optimization
- Testing —> Evaluating on test data

Optimization 101

$\mathscr{L}(\theta)$

θ

How would you go to the very bottom?

More in-depth discussion, see https://web.stanford.edu/~boyd/cvxbook/.

Optimization Problems

(mathematical) optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq b_i, \quad i = 1, \ldots, m \end{array}$$

- $x = (x_1, \ldots, x_n)$: optimization variables
- $f_0 : \mathbf{R}^n \to \mathbf{R}$: objective function
- $f_i : \mathbf{R}^n \to \mathbf{R}$, $i = 1, \ldots, m$: constraint functions

optimal solution $x^\star$ has smallest value of f_0 among all vectors that satisfy the constraints

Optimization Problems

general optimization problem

- very difficult to solve
- methods involve some compromise, $e.g.$, very long computation time, or not always finding the solution

exceptions: certain problem classes can be solved efficiently and reliably

- least-squares problems
- linear programming problems
- convex optimization problems

$$\text{minimize} \quad \|Ax - b\|_2^2$$

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & a_i^T x \leq b_i, \quad i = 1, \ldots, m \end{array}$$

Convex and Non-Convex

Convex optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq b_i, \quad i = 1, \ldots, m \end{array}$$

- objective and constraint functions are convex:

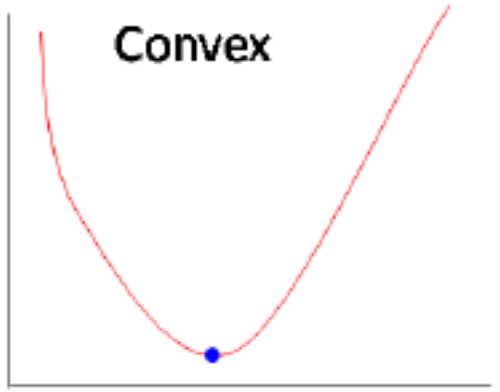
$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y)$$

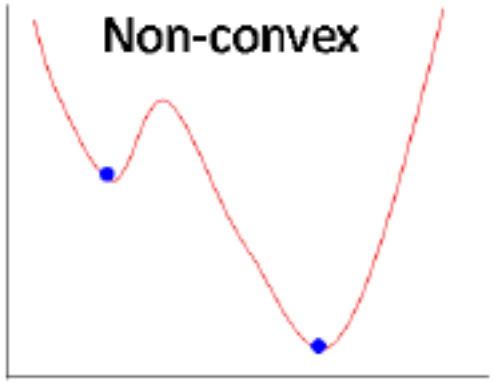
if $\alpha + \beta = 1$, $\alpha \geq 0$, $\beta \geq 0$

- includes least-squares problems and linear programs as special cases

solving convex optimization problems

- no analytical solution
- reliable and efficient algorithms





Gradient Descent

A first-order optimization method: Gradient Descent (GD)

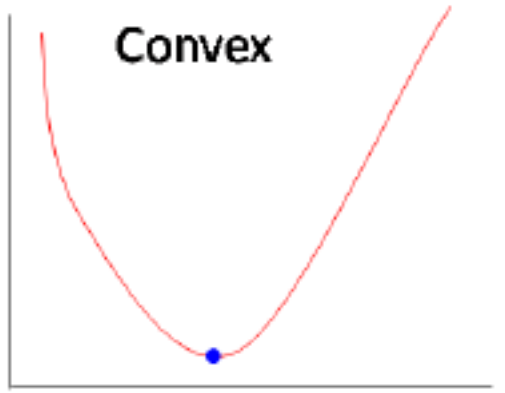
- Update rule for one iteration: $\theta := \theta - \alpha \nabla_\theta \mathscr{L}(\theta)$

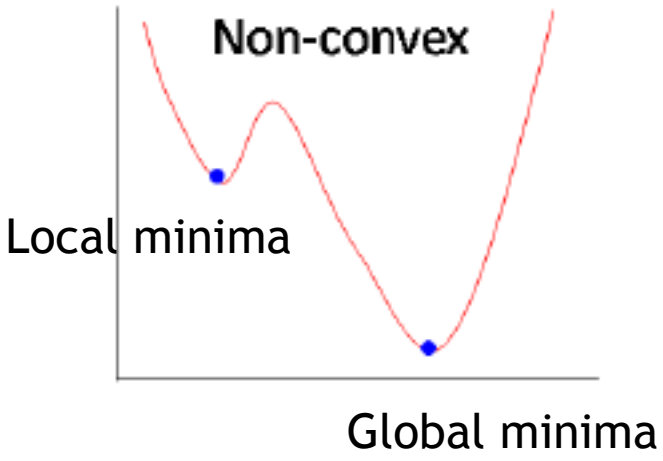
- Learning rate: α
  - If α is small enough, then GD will definitely lead to a smaller loss after the update. However, a too small α needs too many iterations to get the bottom.
  - If α is too big, overshoot! Loss not necessary to decrease.

Local/Global Minima

For convex optimization problem, gradient descent will converge to the global minima.

For general optimization problem, gradient descent will converge to a local minima.





Analytical Gradient

- How to perform GD to minimize NLL loss?

- Derive analytical gradient:
  - For Sigmoid function

$$
\begin{aligned}
g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2}\left(e^{-z}\right) \\
&= \frac{1}{(1+e^{-z})}\cdot\left(1-\frac{1}{(1+e^{-z})}\right) \\
&= g(z)(1-g(z)).
\end{aligned}
$$

Analytical Gradient

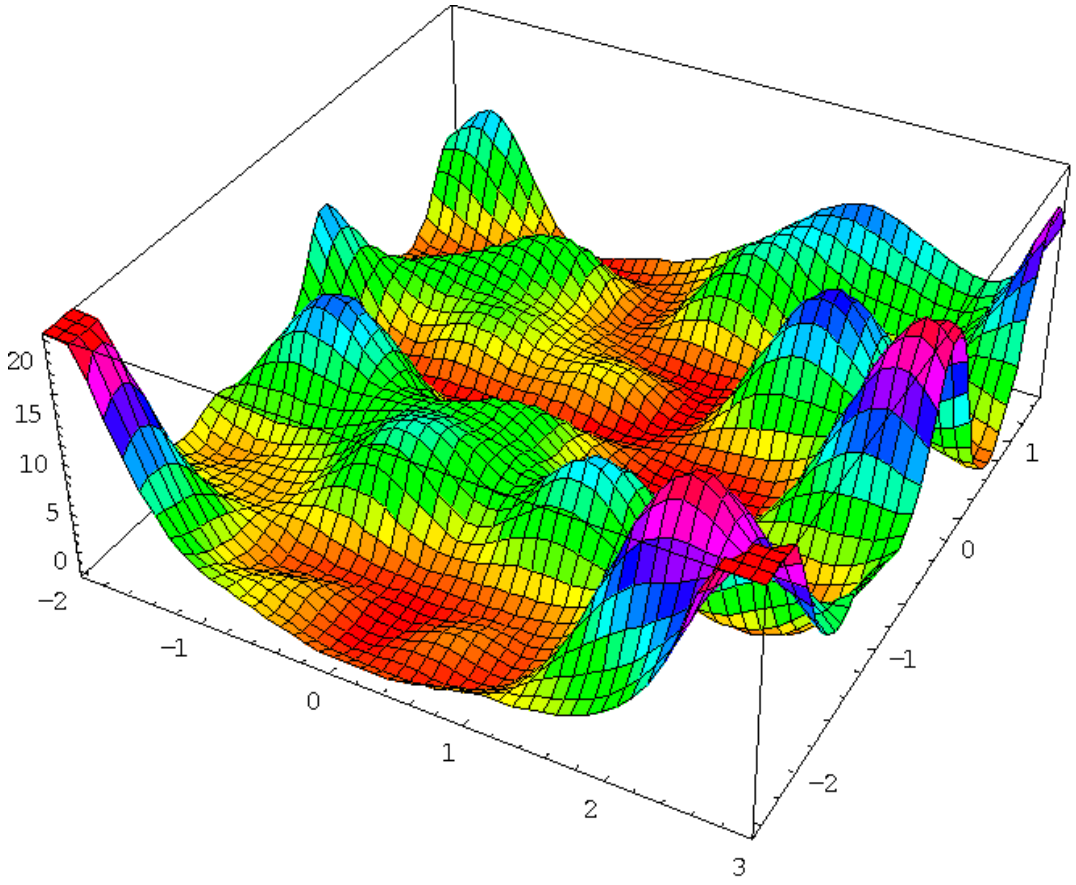
- How to perform GD to minimize NLL loss?

- Derive analytical gradient:

$$\mathscr{L} = -\sum_{i=1}^{n} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

$$\begin{aligned}
\frac{\partial \mathscr{L}}{\partial \theta_j} &= -\sum \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x) \\
&= -\sum \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) g(\theta^T x)(1 - g(\theta^T x)) \frac{\partial}{\partial \theta_j} \theta^T x \\
&= -\sum \left(y(1 - g(\theta^T x)) - (1 - y) g(\theta^T x) \right) x_j \\
&= -\sum \left(y - h_\theta(x) \right) x_j
\end{aligned}$$

Non-Linear and Non-Convex Optimization



Non-convex energy landscape

Naive gradient descent will trap at local minima.

Batch Gradient Descent vs. Stochastic Gradient Descent

- Batch Gradient Descent

Take all data and label pairs in the training set to calculate the gradient.

-: very slow

-: easily get trapped at local minima

- Stochastic Gradient Descent (SGD, or Mini-batch Gradient Descent)

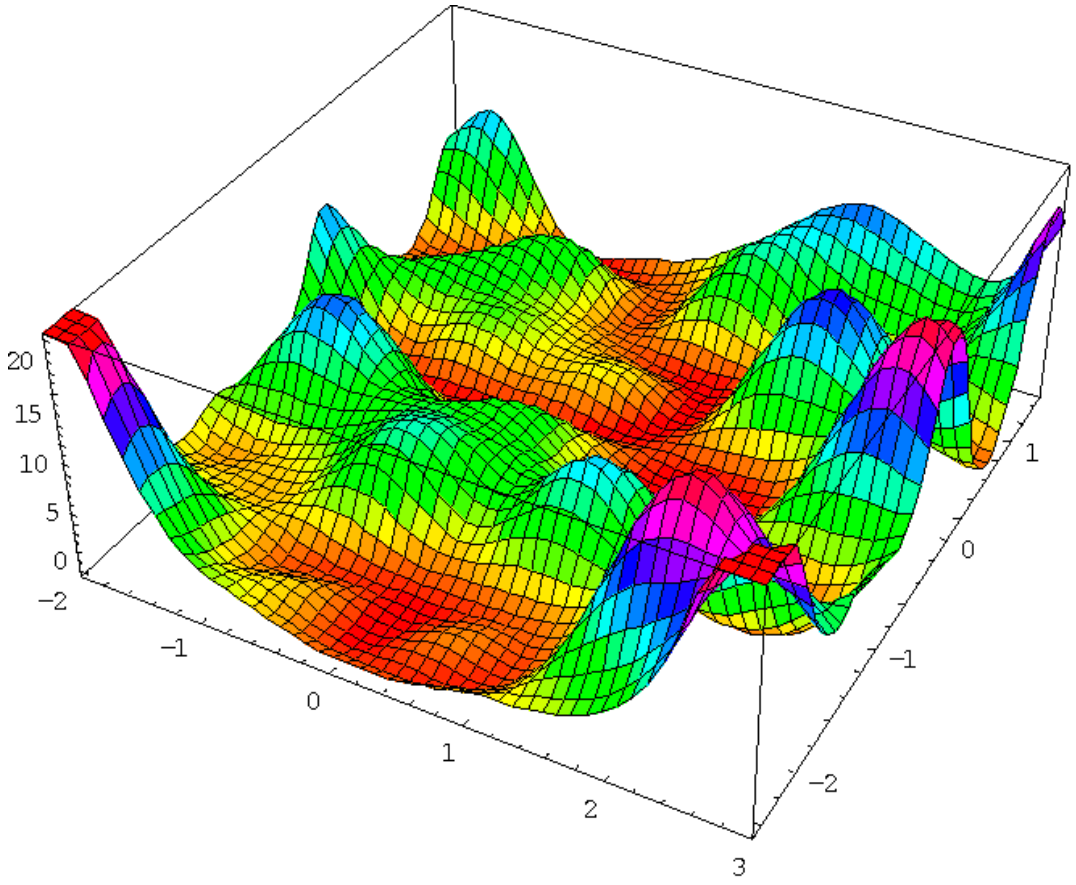
Randomly sample N pairs as a batch from the training data and then compute the average gradient from them.

+: fast

+: can get out of local minima

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

Non-Linear and Non-Convex Optimization



Non-convex energy landscape

SGD has the potential to jump out of a local minima.

Outline

- Set up the task
- Prepare the data —> Need a labeled dataset.
- Built a model —> construct your neural network
- Decide the fitting/training objective —> Loss function
- Perform fitting —> Training by running optimization
- Testing —> Evaluating on test data

Testing and Evaluation

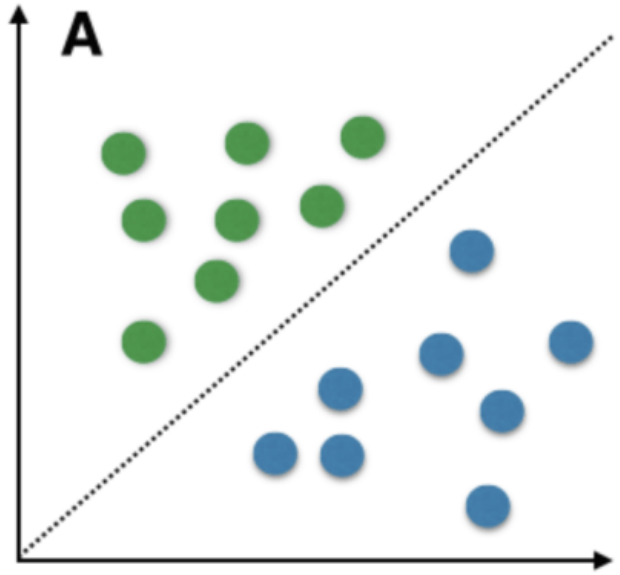
- After training, we need to know how well our model generalizes to unseen data or test data.
- Evaluate the classification accuracy on the test split.
- Will we still work well?

Generalization gap!

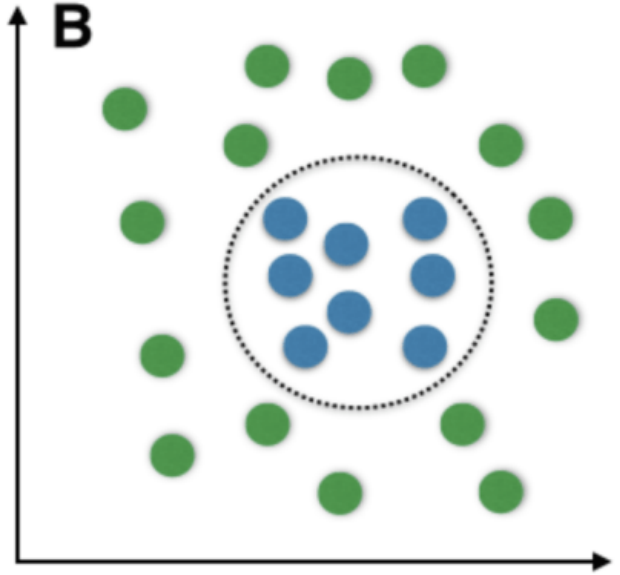
Multilayer Perceptron

Problem with Single-Layer Network

- $g(\theta^T x) = 0$ is a hyperplane in the space of x
- can only handle linear separable cases

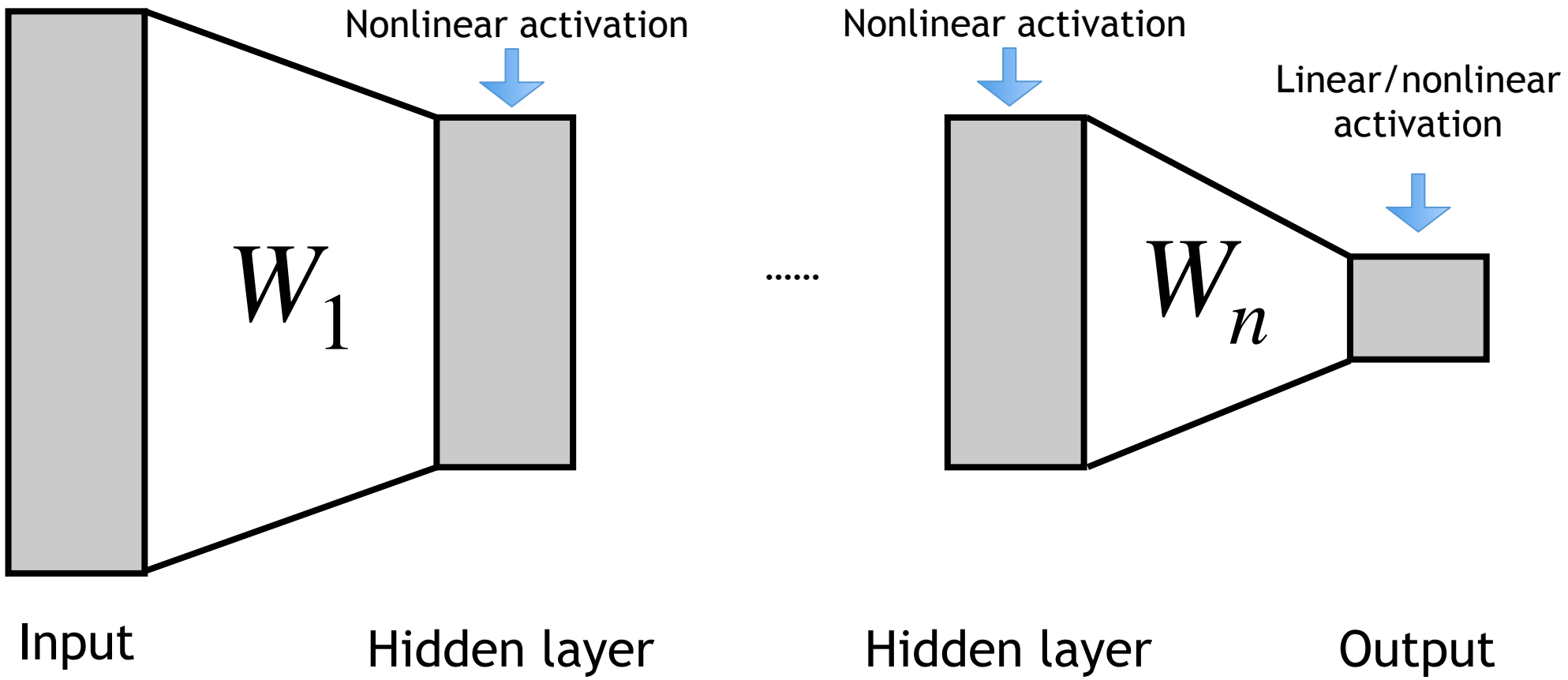


Linear separable



Linear non-separable

Multi-Layer Perceptron (MLP)



- MLP: Stacking linear layer and nonlinear activations.
- Through many non-linear layers, transform a linear non-separable problem to linear separable at the last layer

Classification function with MLP

$$f(x, W) = Wx$$

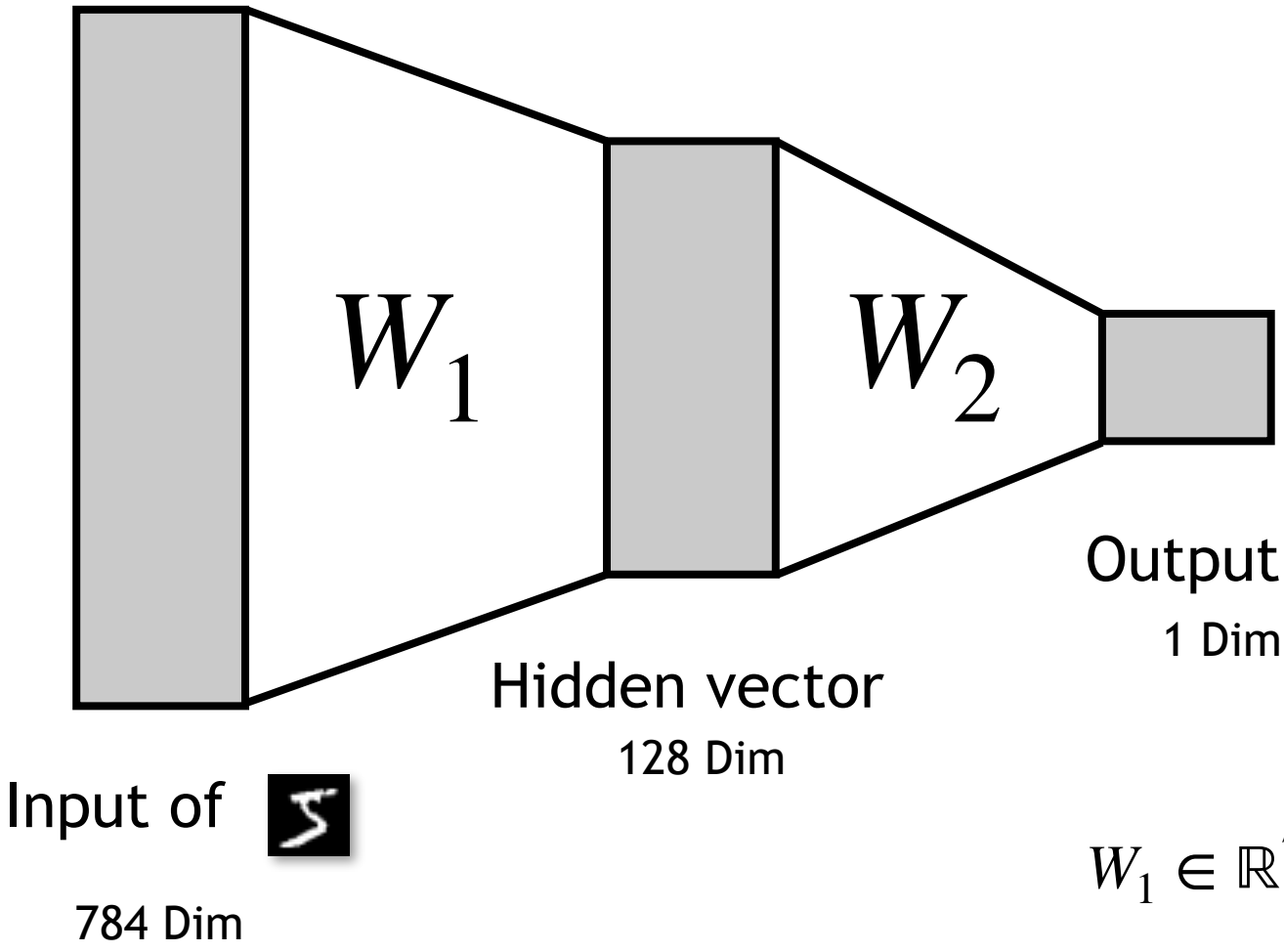
Linear function

$$f(x, W) = g(W_2 g(W_1 x))$$

2-layer MLP,
or fully-conncted layers

In practice, we can concat the input variables with extract 1 for learning bias.

Classification function with MLP



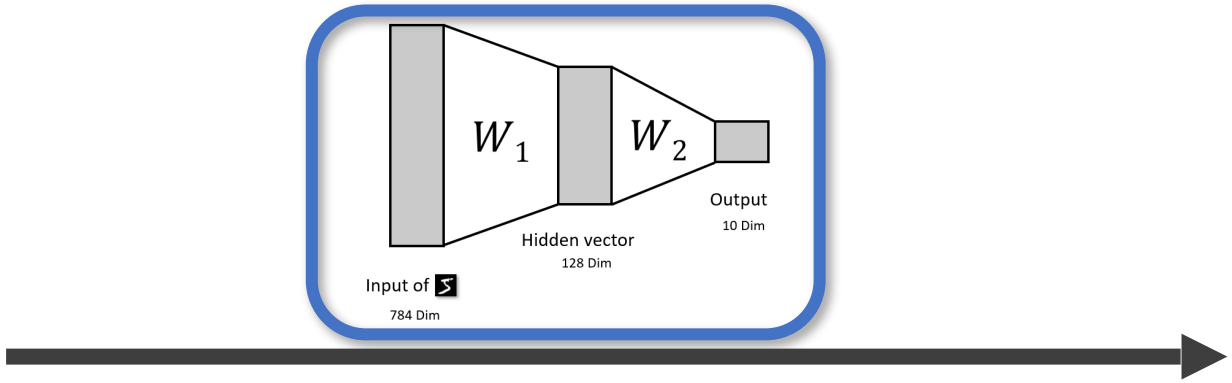
$$W_1 \in \mathbb{R}^{784\times12}, W_2 \in \mathbb{R}^{12\times1}$$

How can we obtain the parameters/weights of the MLP?

Classification function with MLP

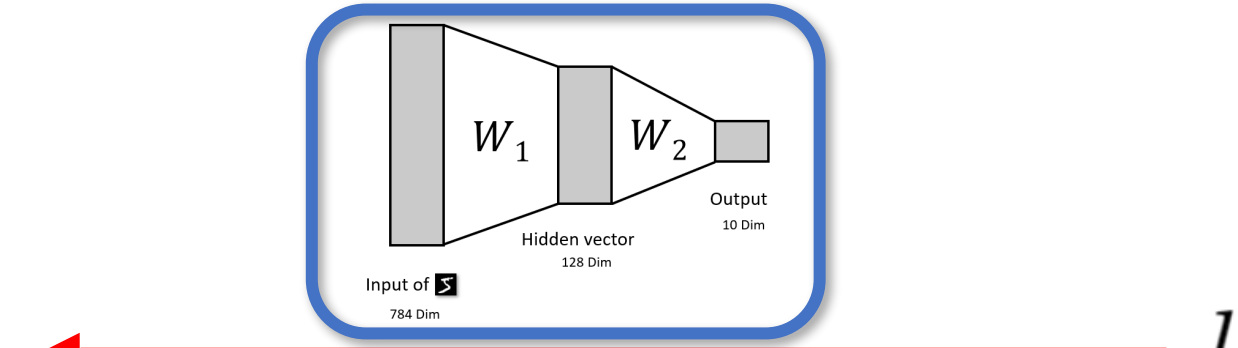
1. Initialization: randomly generate the weights $\mathbf{W}_1 \in \mathbb{R}^{784\times128}, \mathbf{W}_2 \in \mathbb{R}^{128\times10}$

2. Forwarding:



$\mathbf{y}$

3. Gradient decent:



$loss(5, \mathbf{y})$

Update weights

Analytical Gradient?

So, if we can compute the gradient $\frac{\partial \mathbf{L}}{\partial \mathbf{W}_1}, \frac{\partial \mathbf{L}}{\partial \mathbf{W}_2}$, then we can learn the $\mathbf{W}_1, \mathbf{W}_2$

An intuitive idea : Derive $\frac{\partial \mathbf{L}}{\partial \mathbf{W}}$ by hand

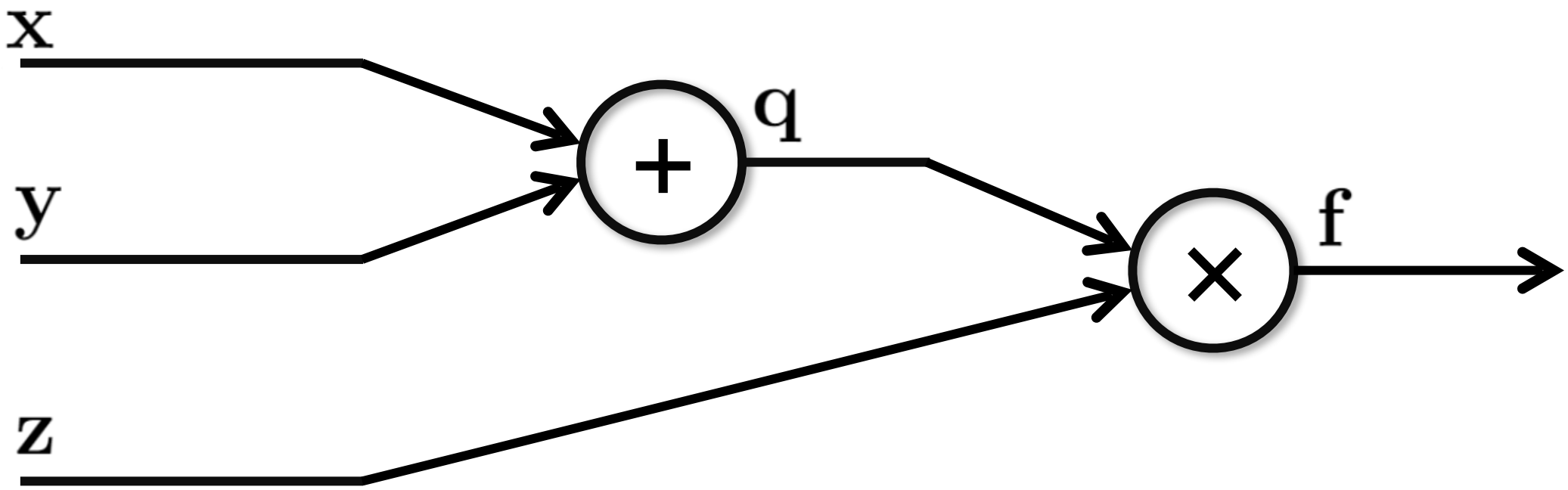
Cons:

- Lots of matrix calculus
- Infeasible: any modification requires re-derivation

Backpropagation with a toy example

Let we consider a toy example:
(Computational graph)

$$f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$$

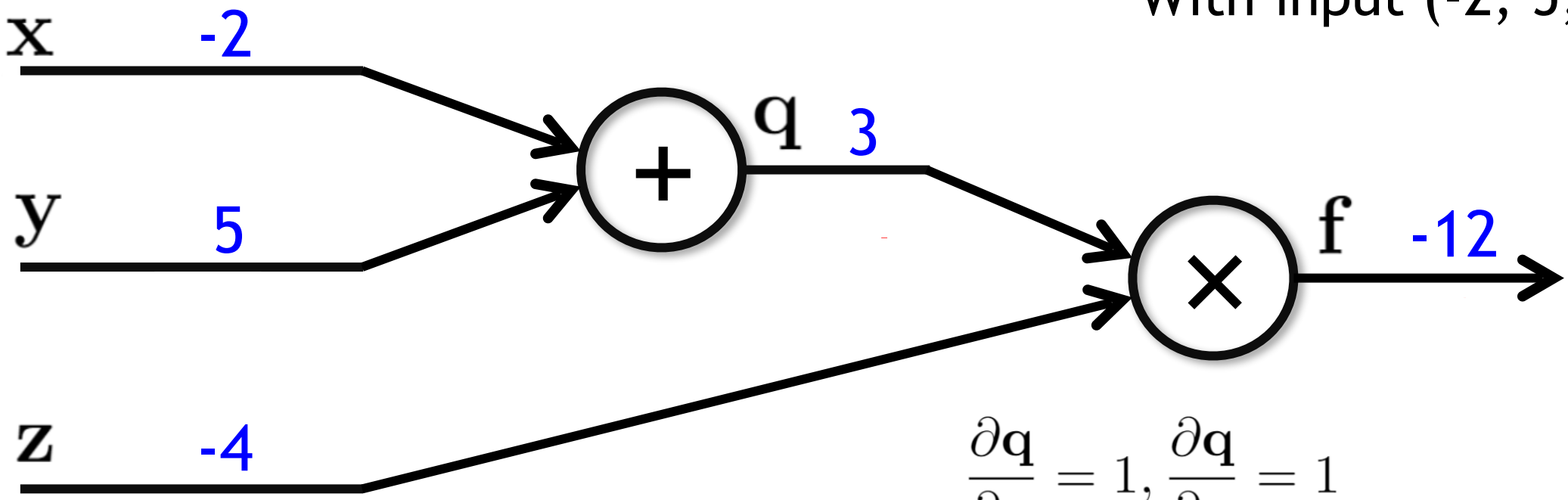


And we want $\frac{\partial f}{\partial \mathbf{x}}, \frac{\partial f}{\partial \mathbf{y}}, \frac{\partial f}{\partial \mathbf{z}}$

Backpropagation with a toy example

Let we consider a toy example: $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$

With input (-2, 5, -4)



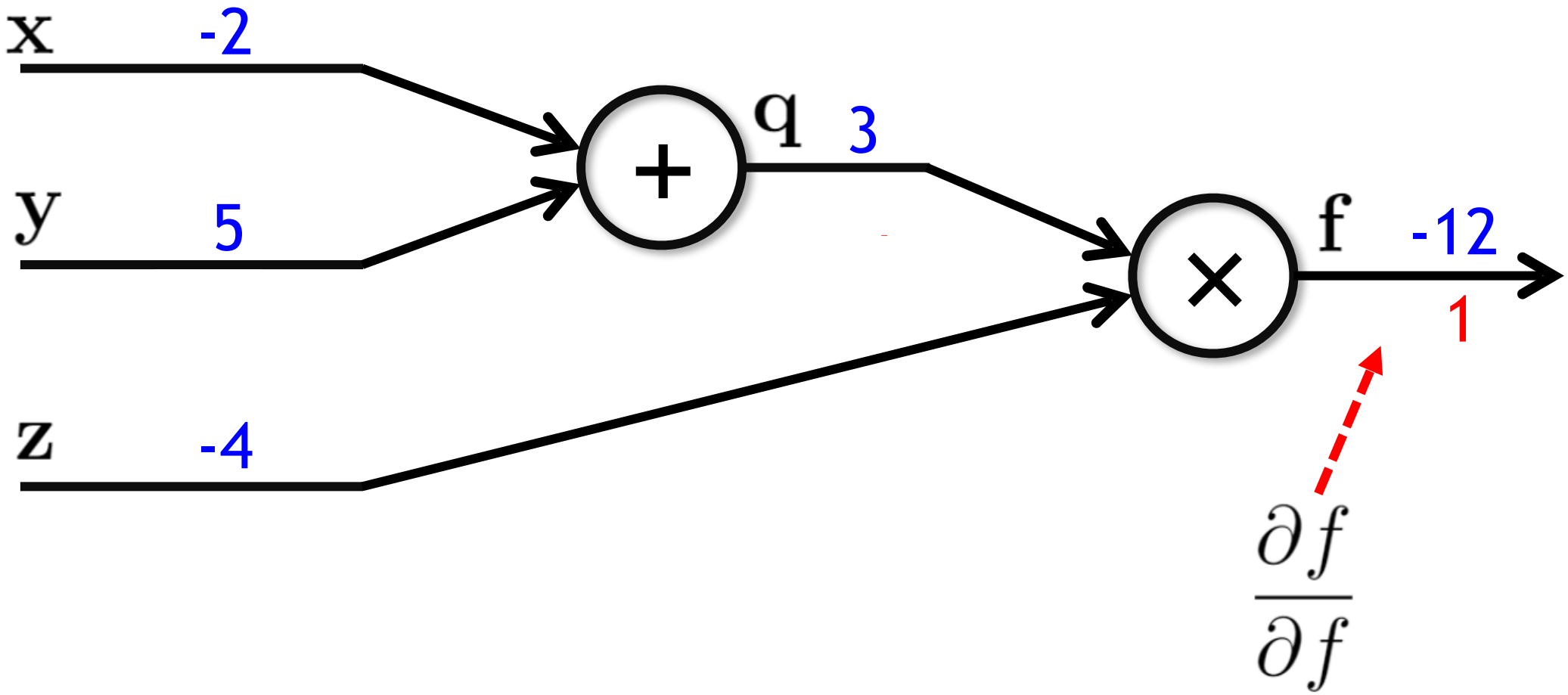
And we have the derivation

$$\frac{\partial \mathbf{q}}{\partial \mathbf{x}} = 1, \frac{\partial \mathbf{q}}{\partial \mathbf{y}} = 1$$

$$\frac{\partial f}{\partial \mathbf{q}} = \mathbf{z}, \frac{\partial f}{\partial \mathbf{z}} = \mathbf{q}$$

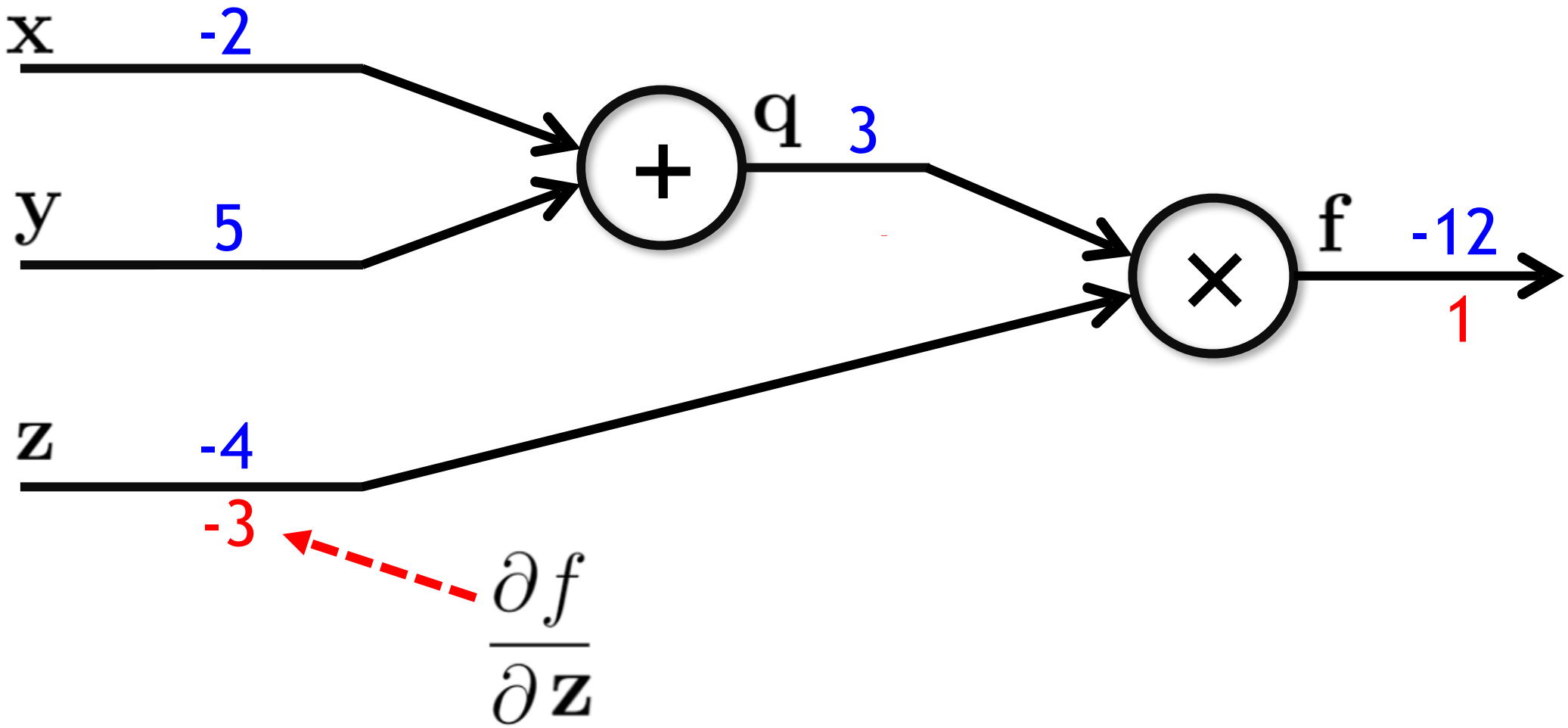
Backpropagation with a toy example

Let we consider a toy example: $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$



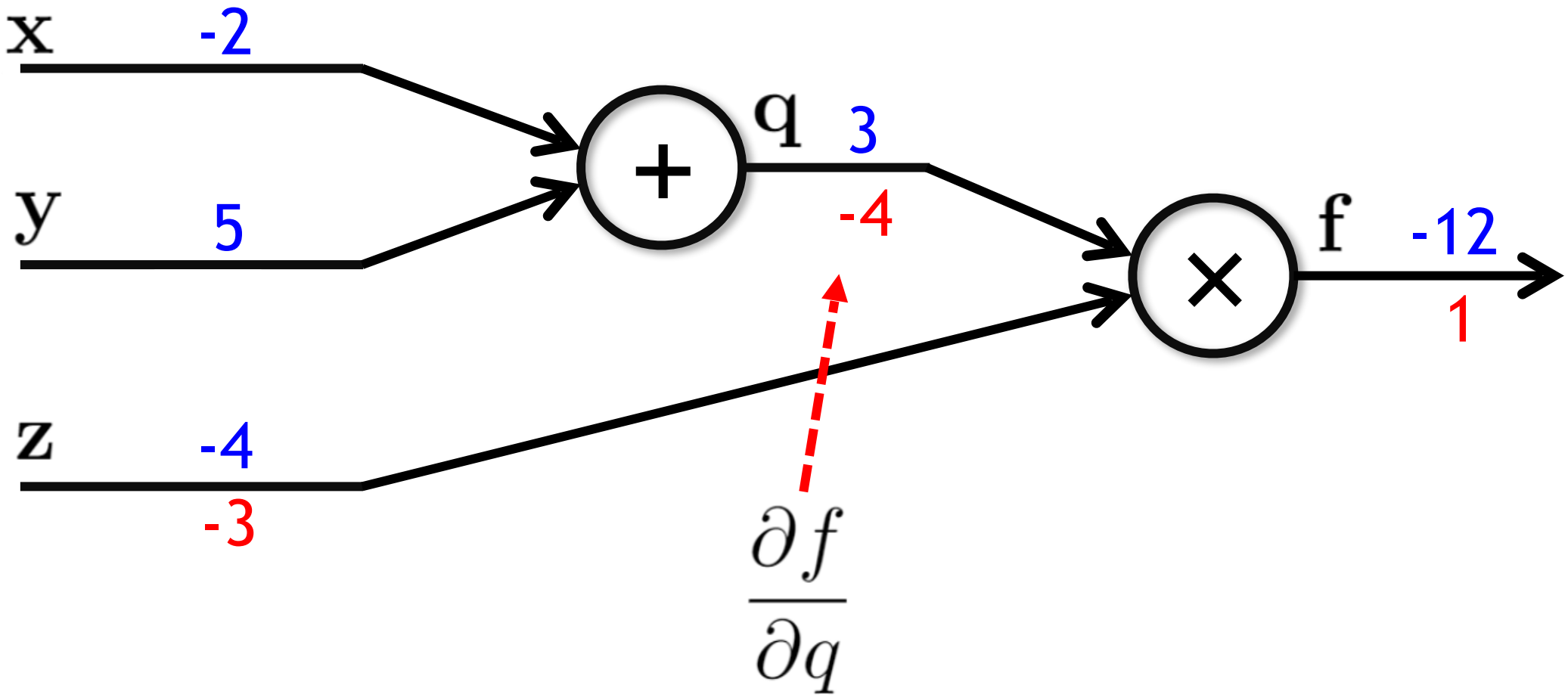
Backpropagation with a toy example

Let we consider a toy example: $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$



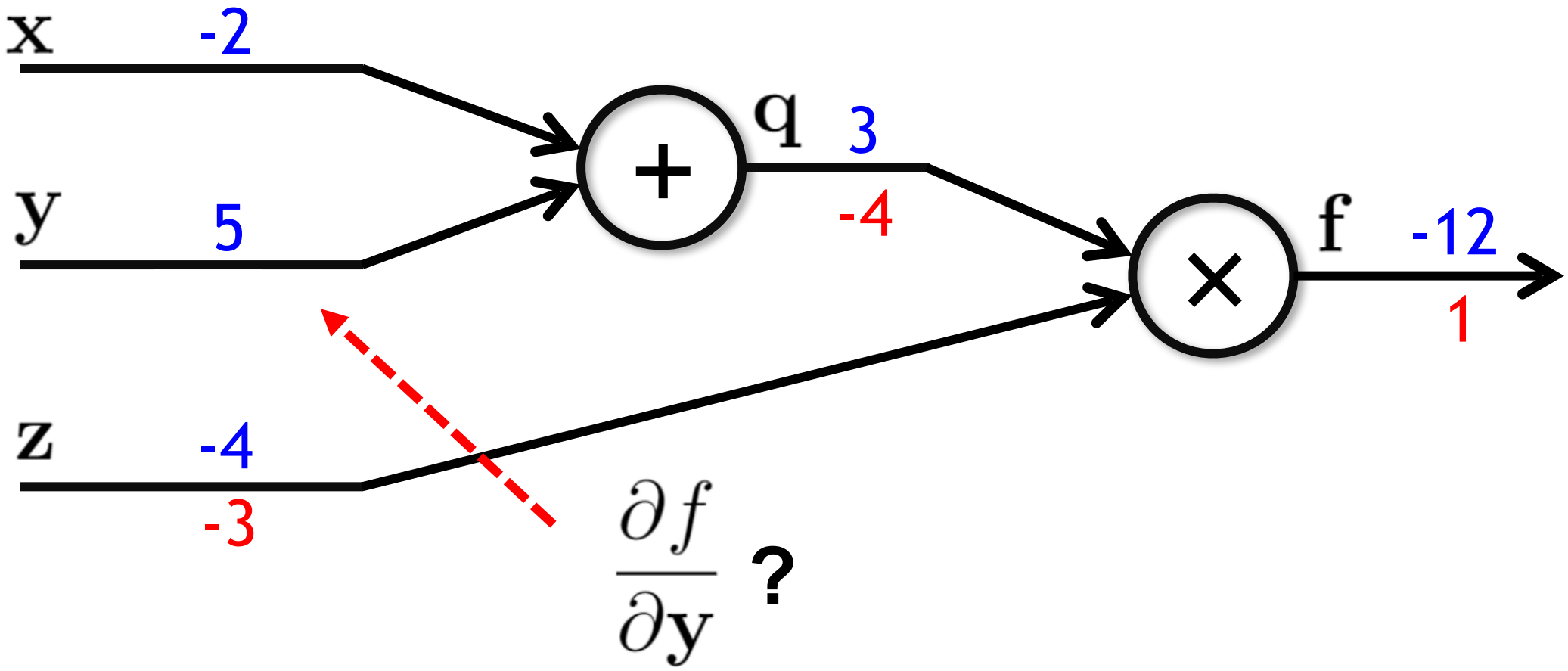
Backpropagation with a toy example

Let we consider a toy example: $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$



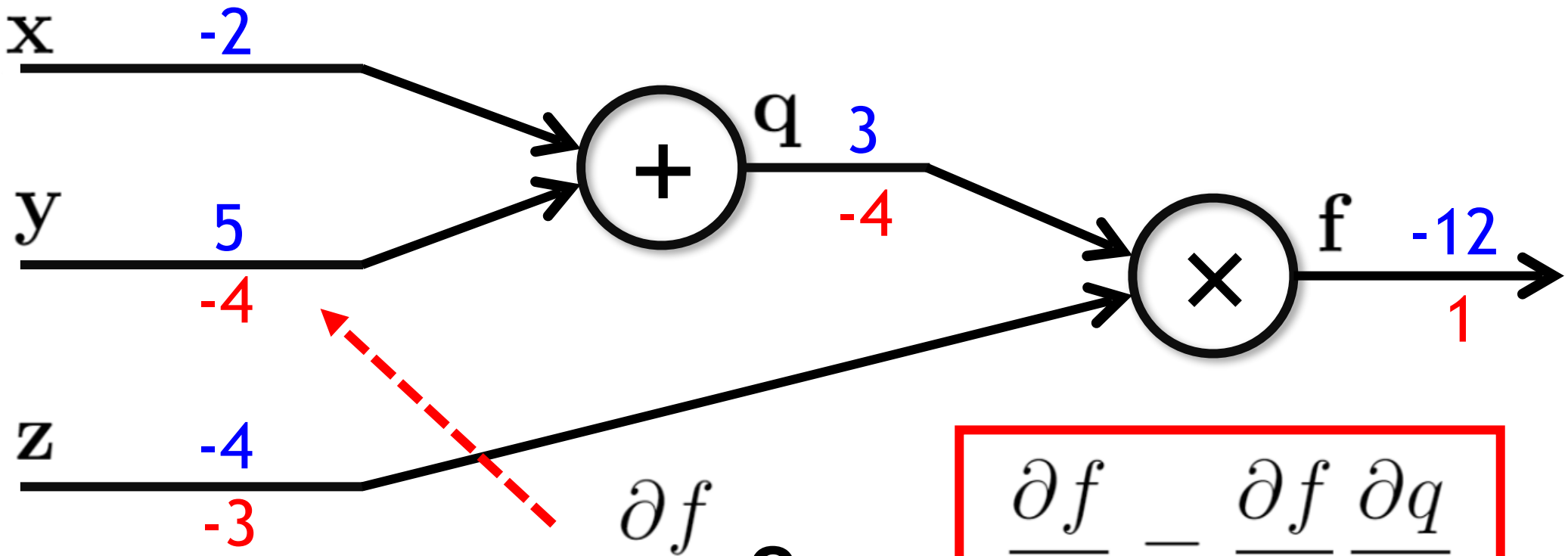
Backpropagation with a toy example

Let we consider a toy example: $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$



Backpropagation with a toy example

Let we consider a toy example: $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$



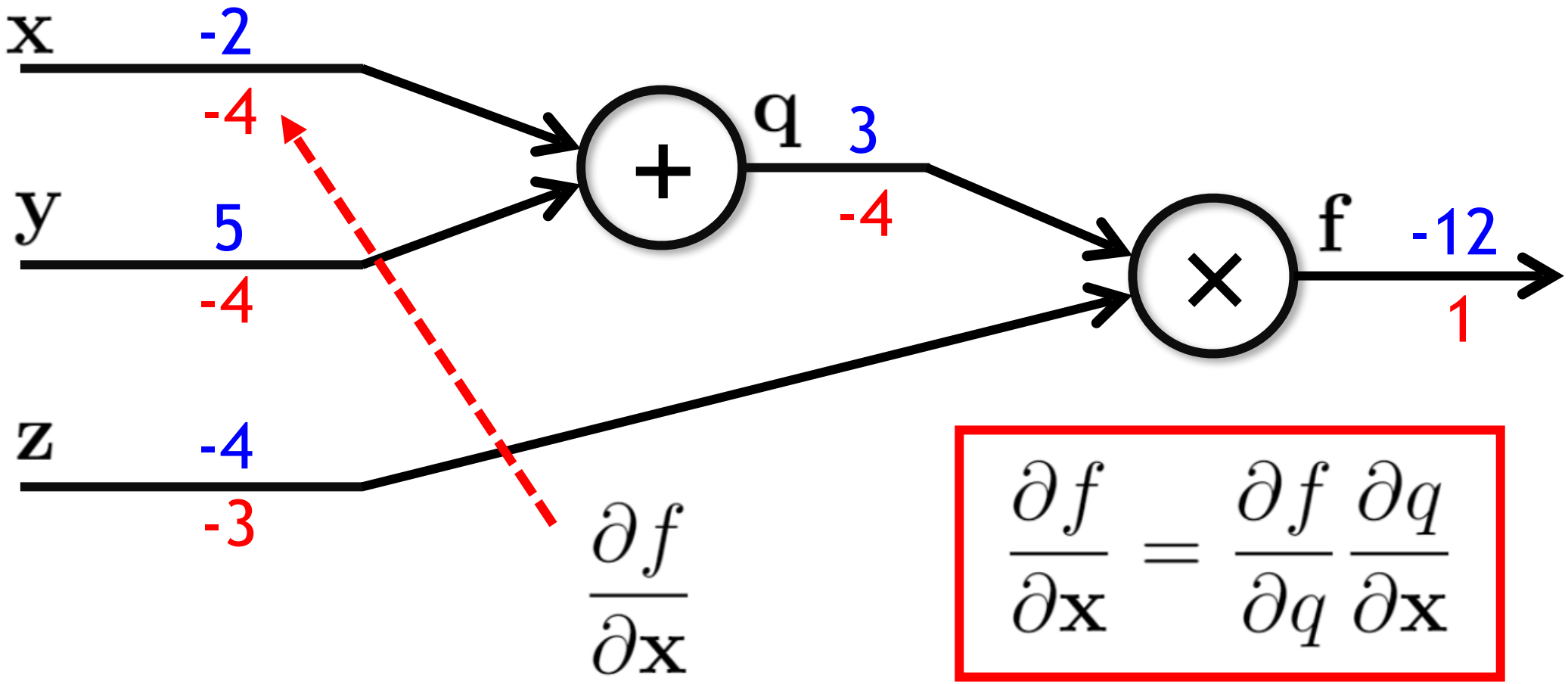
$$\frac{\partial f}{\partial \mathbf{y}} \; ?$$

$$\frac{\partial f}{\partial \mathbf{y}} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial \mathbf{y}}$$

Chain rule

Backpropagation with a toy example

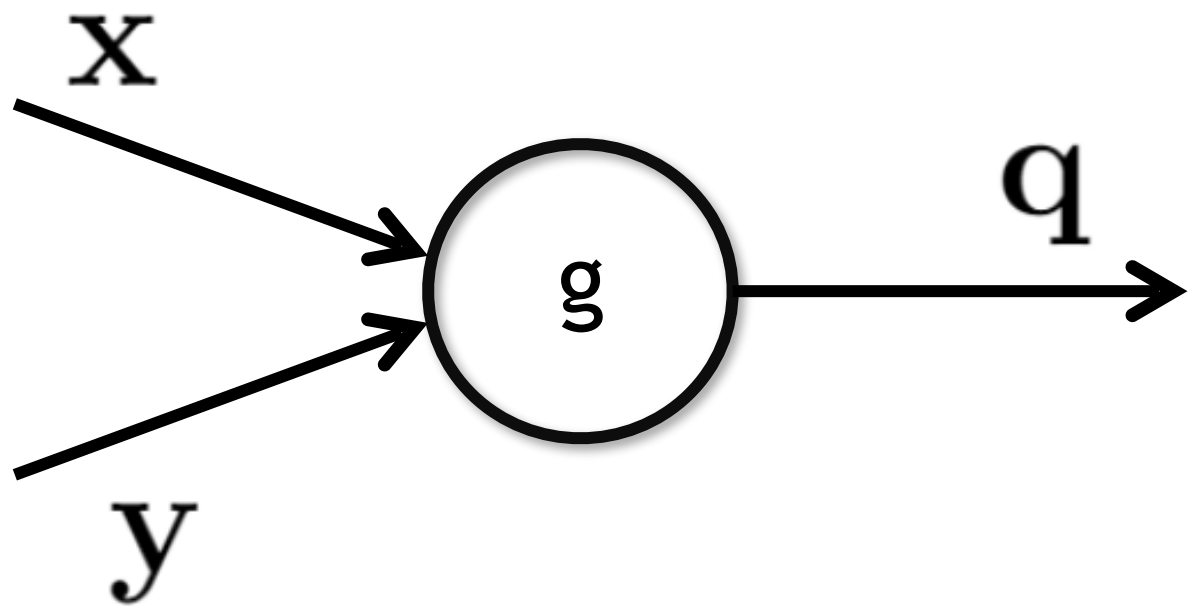
Let we consider a toy example: $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x} + \mathbf{y}) \cdot \mathbf{z}$



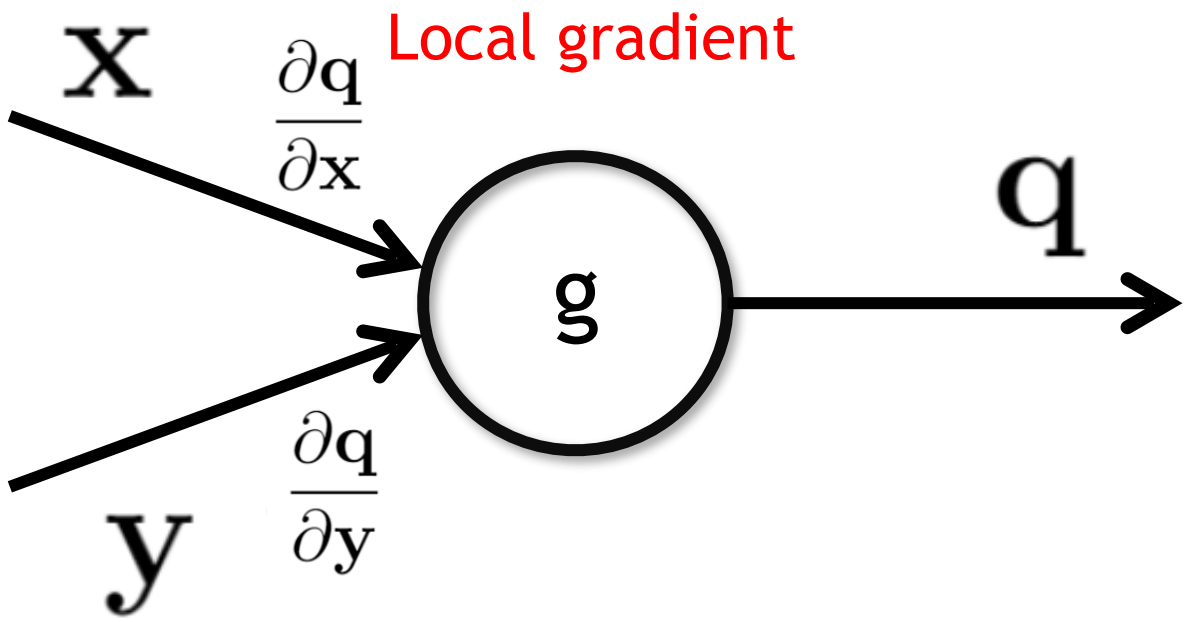
$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial \mathbf{x}}$$

Chain rule

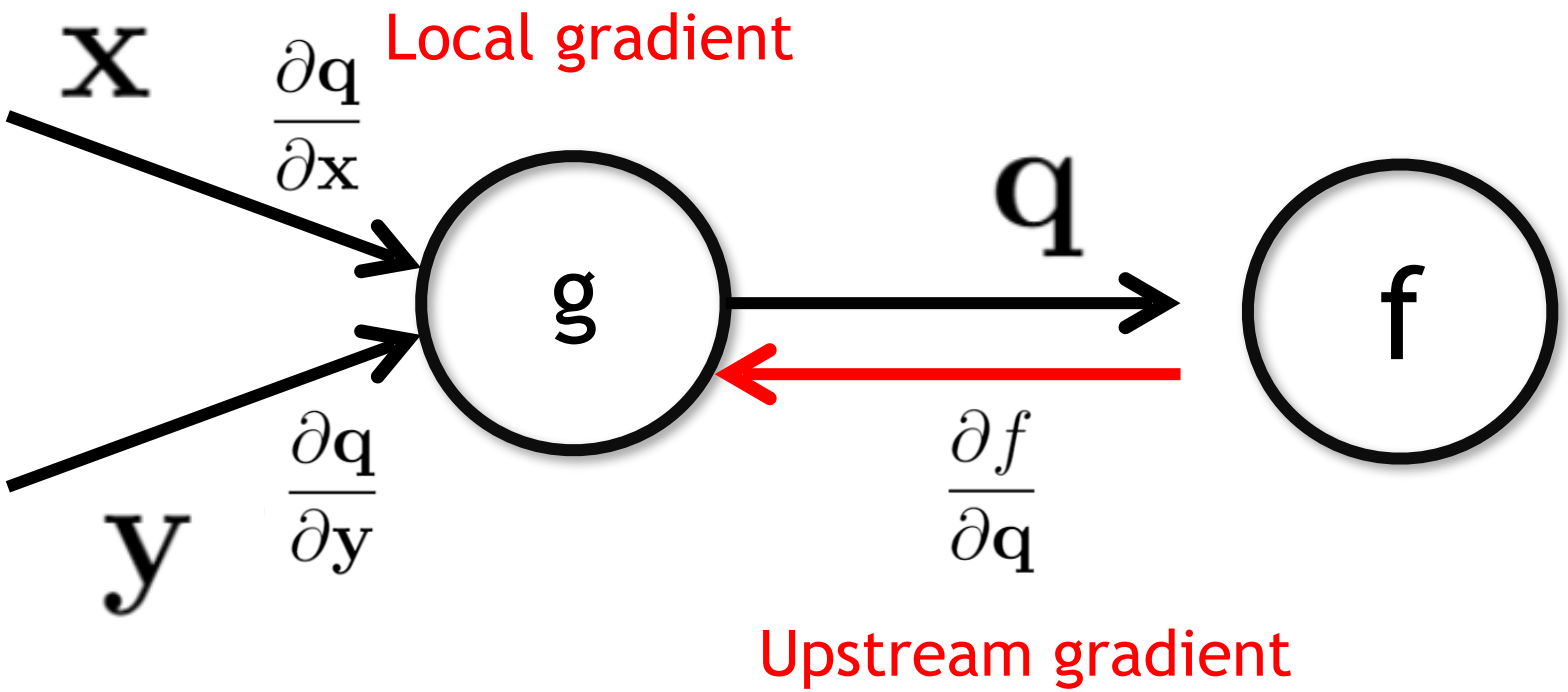
Chain rule



Chain rule

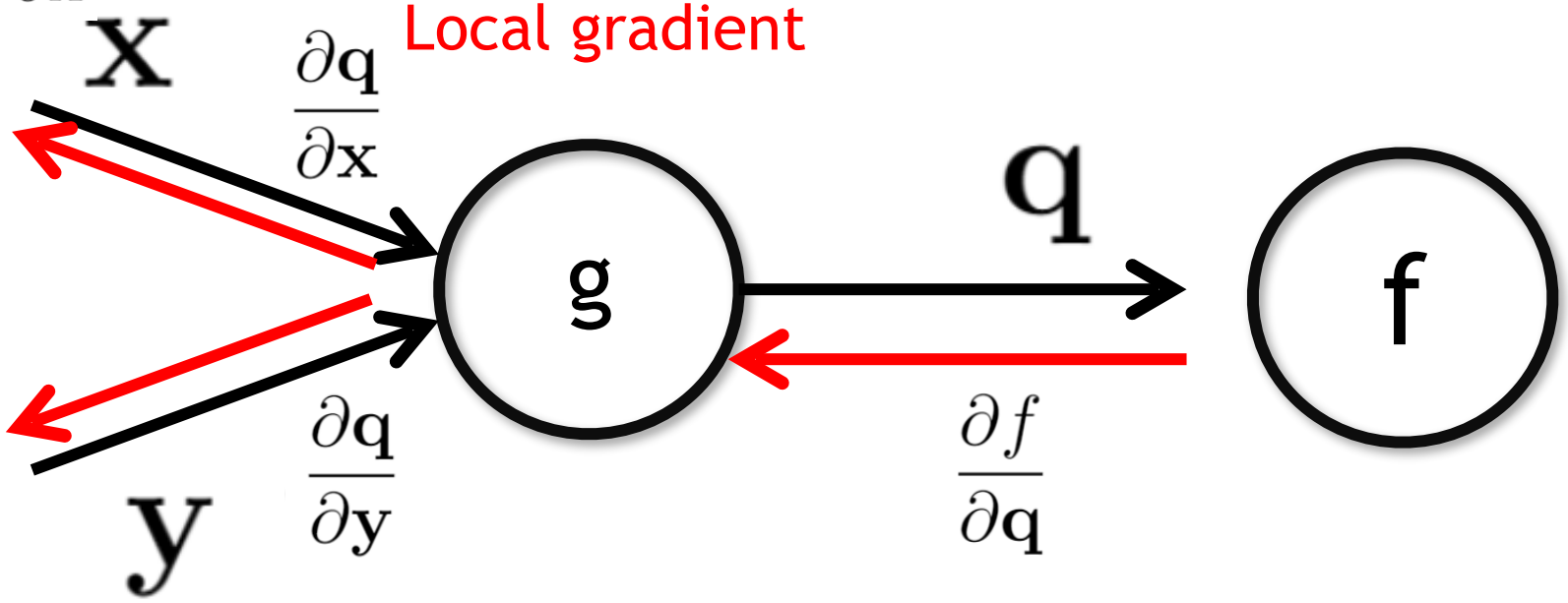

x
$\frac{\partial q}{\partial x}$
Local gradient
g
q
y
$\frac{\partial q}{\partial y}$

Chain rule


x
y
$\frac{\partial q}{\partial x}$
$\frac{\partial q}{\partial y}$
Local gradient
g
q
f
$\frac{\partial f}{\partial q}$
Upstream gradient

Chain rule

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial \mathbf{x}}$$

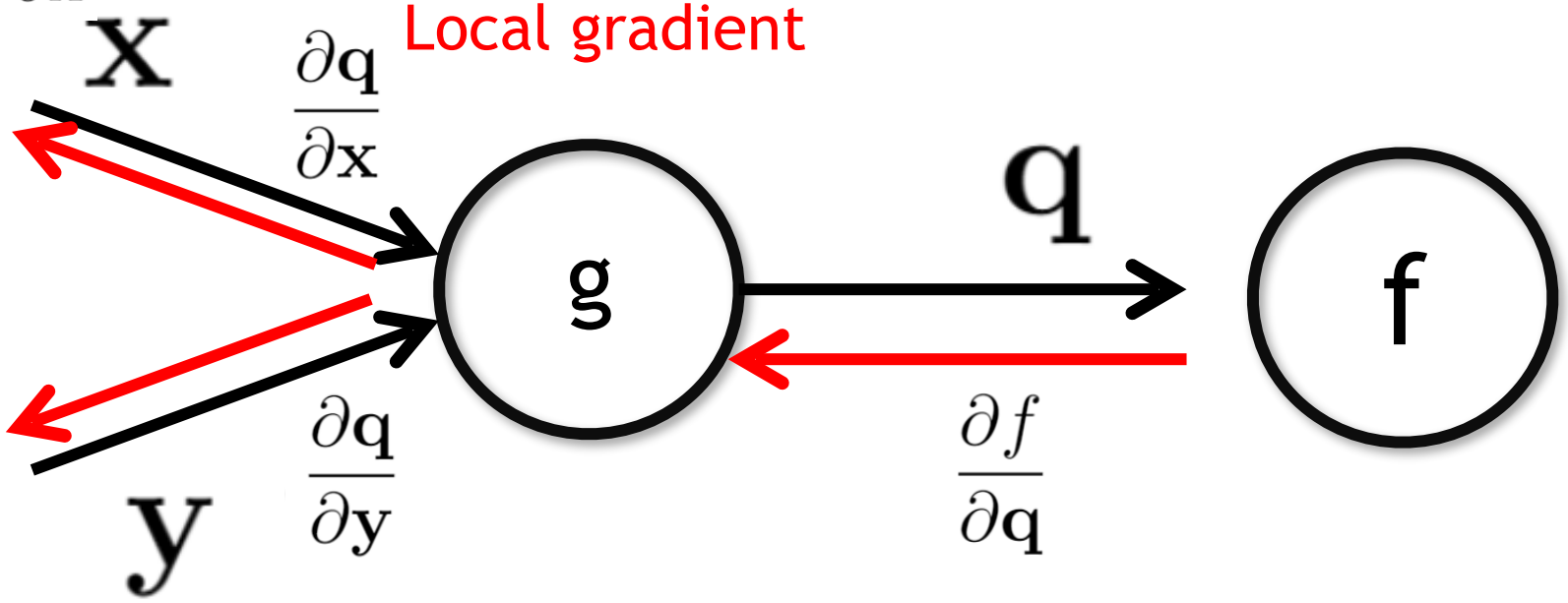


$$\frac{\partial f}{\partial \mathbf{y}} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial \mathbf{y}}$$

Downstream gradient

Chain rule

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial \mathbf{x}}$$



$$\frac{\partial f}{\partial \mathbf{y}} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial \mathbf{y}}$$

Downstream gradient

The backpropagation can be efficiently implemented with simple matrix operations

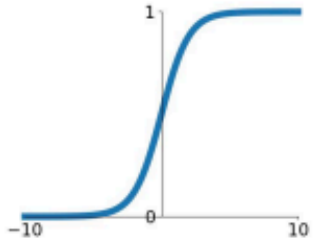
"Backpropagation for a Linear Layer" from Justin 2017

Activation Function

Activation functions

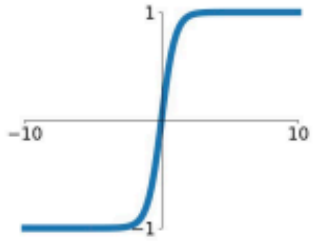
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



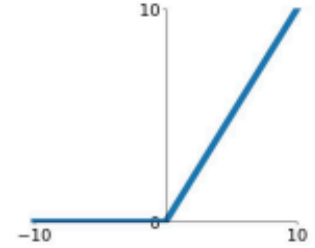
tanh

$$\tanh(x)$$



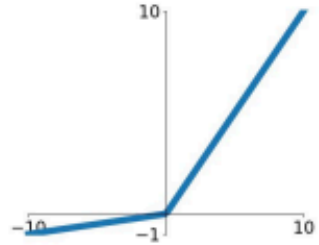
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

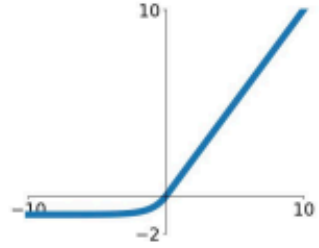


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Slides credit: Stanford CS231N

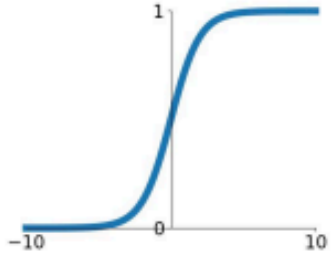
Activation Function

Activation functions

ReLU is a good default choice for most problems

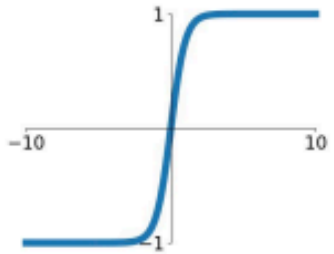
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



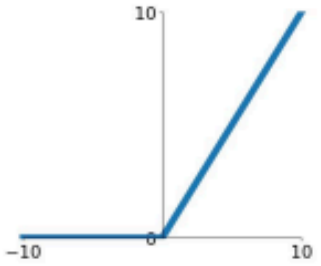
tanh

$$\tanh(x)$$



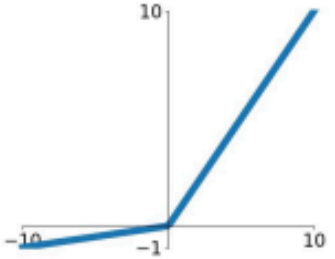
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

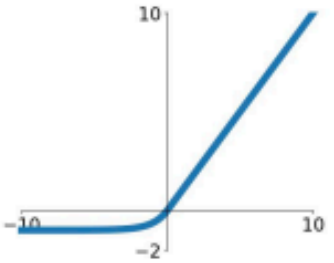


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

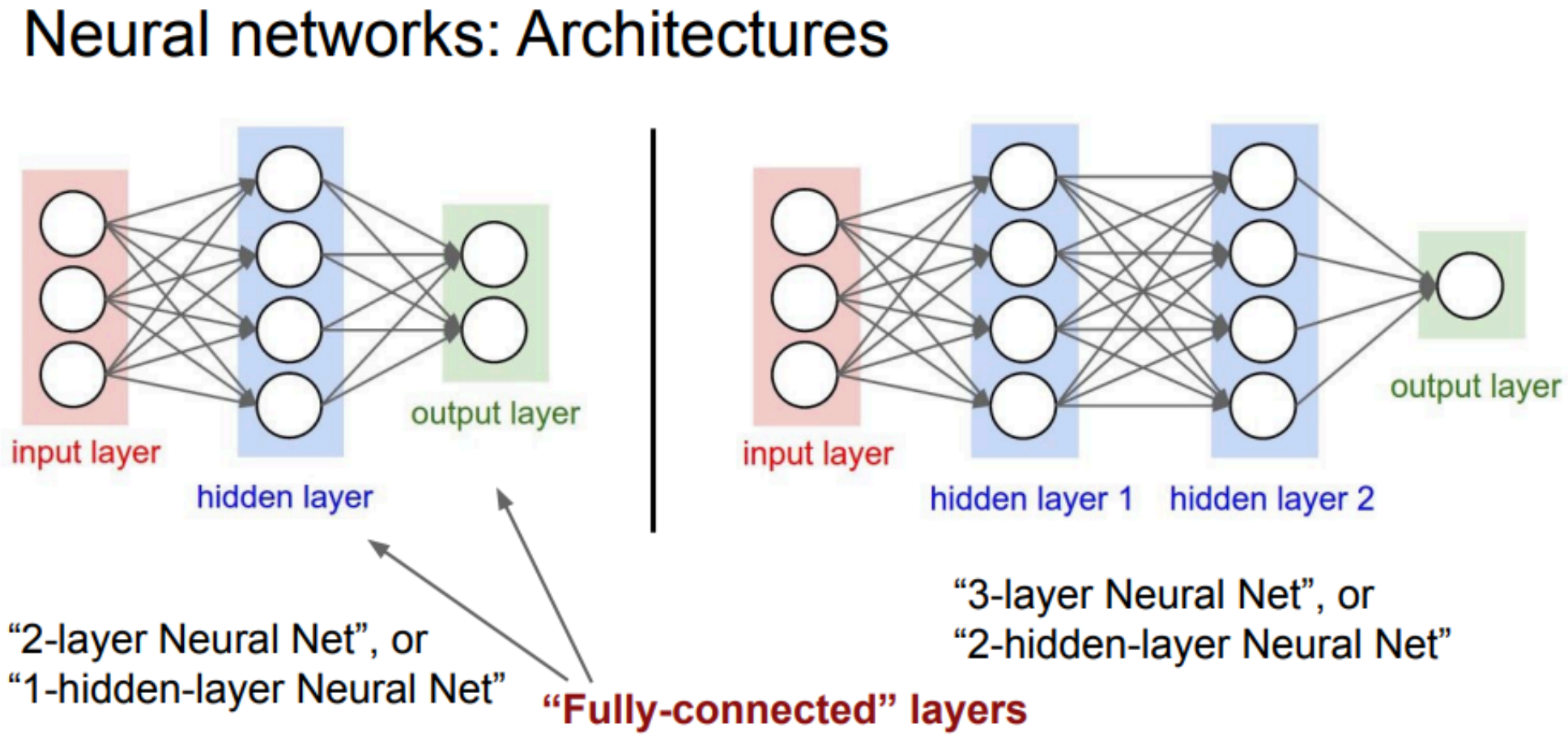
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Slides credit: Stanford CS231N

Review



Slides credit: Stanford CS231N

Problem for Using MLP to Process Vision Signals

- Flatten an image into a vector would be very expensive for high resolution images

- Flattening operation breaks the local structure of an image.

Convolutional Neural Network

Slides are borrowed from Stanford CS231N.

Convolutional Neural Network

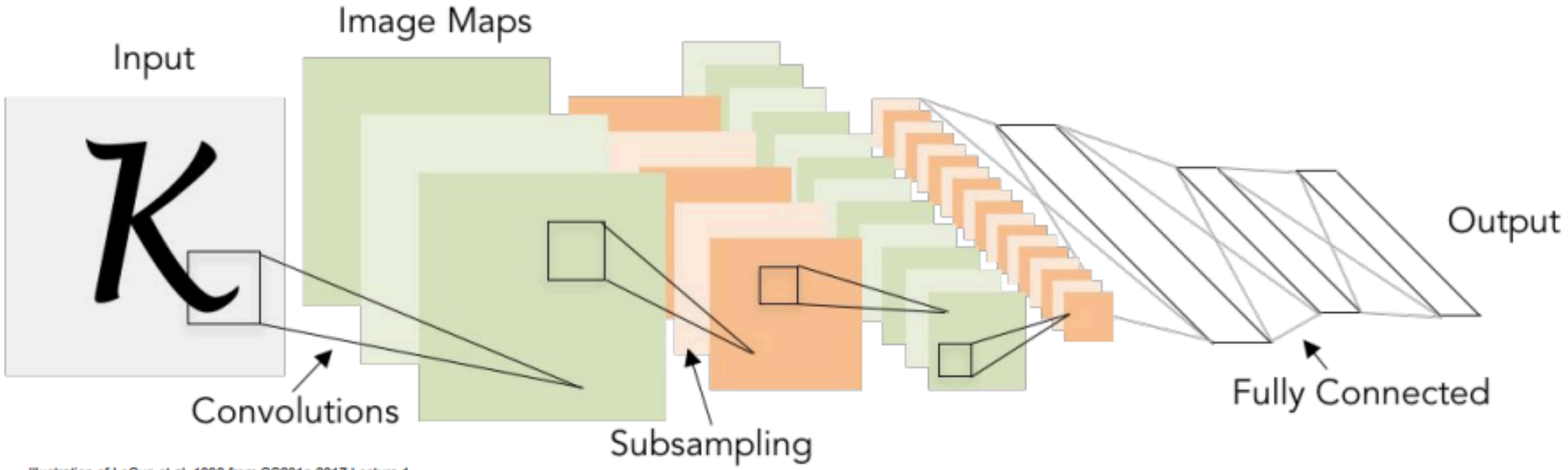
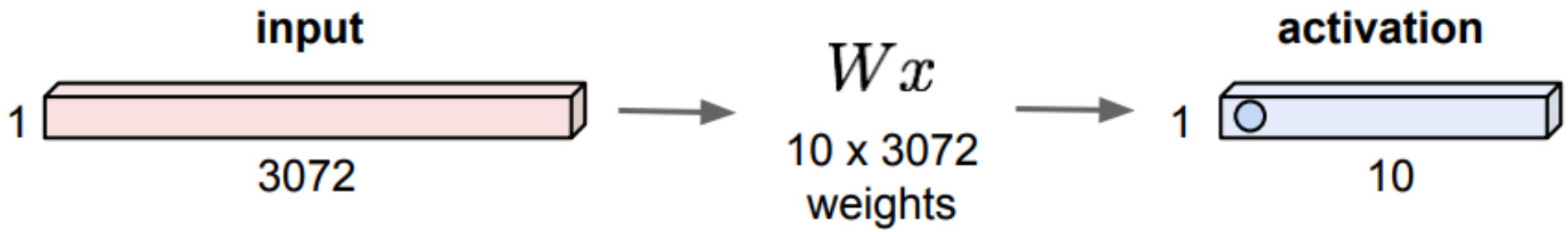


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Convolutional Neural Network

Recap: Fully Connected Layer

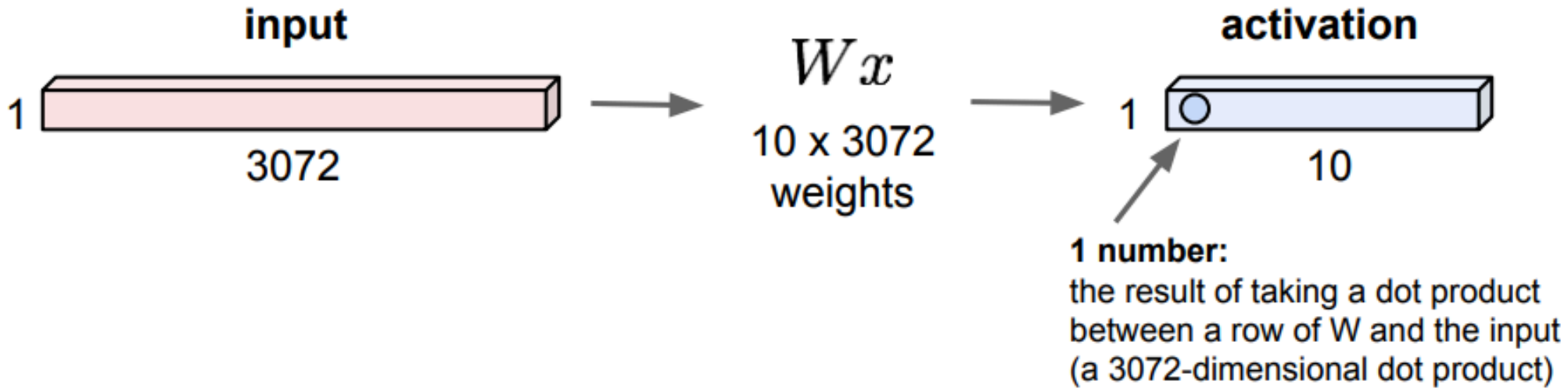
32x32x3 image -> stretch to 3072 x 1



Convolutional Neural Network

Fully Connected Layer

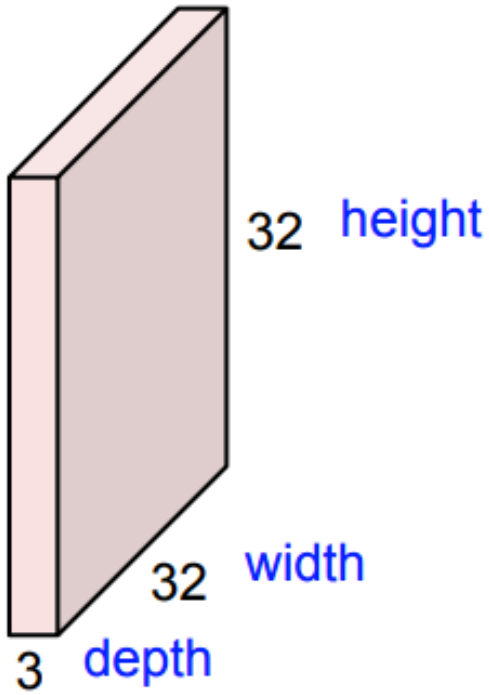
32x32x3 image -> stretch to 3072 x 1



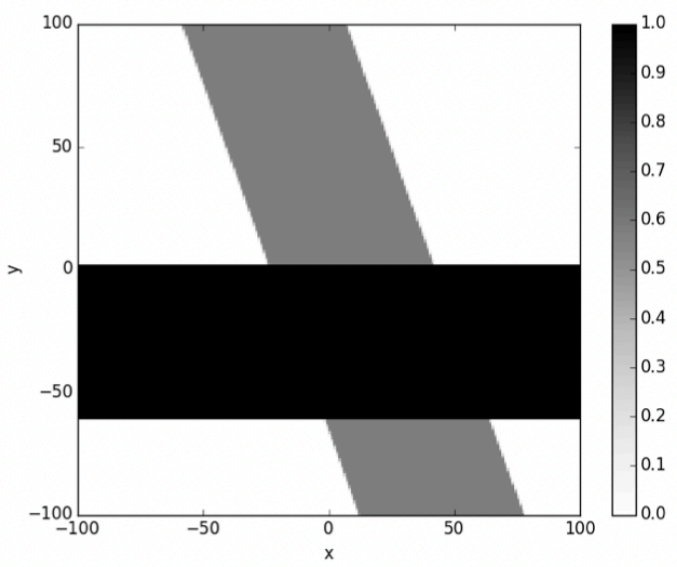
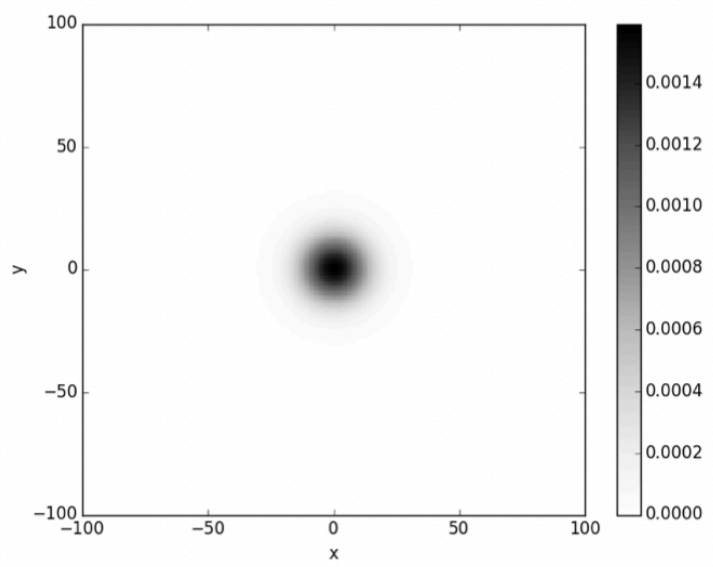
Convolutional Neural Network

Convolution Layer

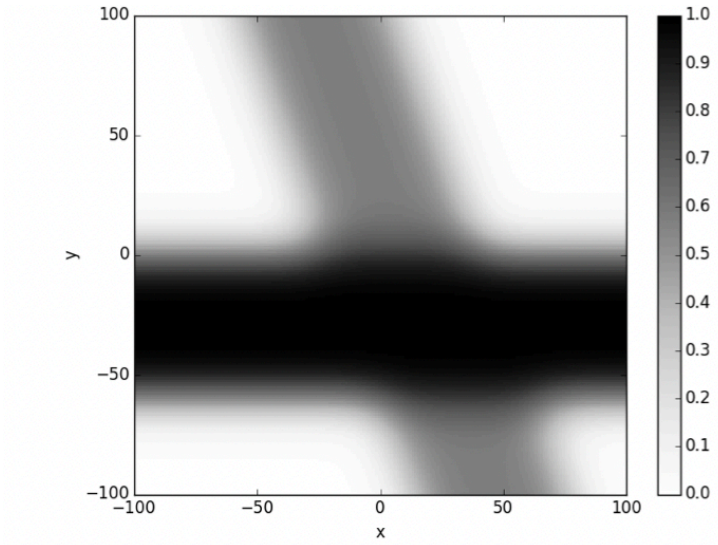
32x32x3 image -> preserve spatial structure



Two-Dimensional Convolution

f  $*$ 

$$g = \frac{1}{2\pi\sigma^2} \exp - \frac{x^2 + y^2}{2\sigma^2}$$

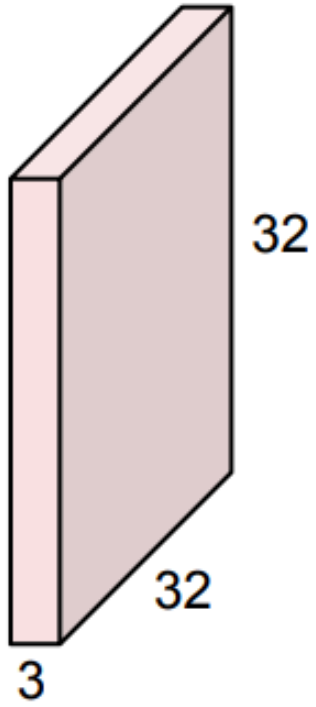
$=$ 

$$(f * g)[m, n] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k, l] g[m - k, n - l]$$

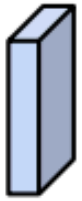
Convolutional Neural Network

Convolution Layer

32x32x3 image



5x5x3 filter



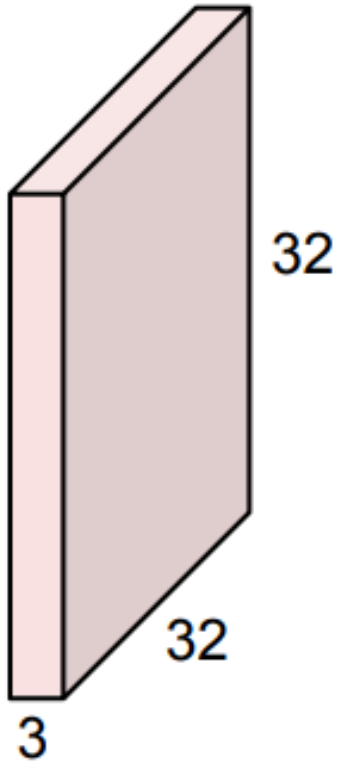
Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolutional Neural Network

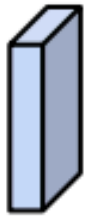
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image



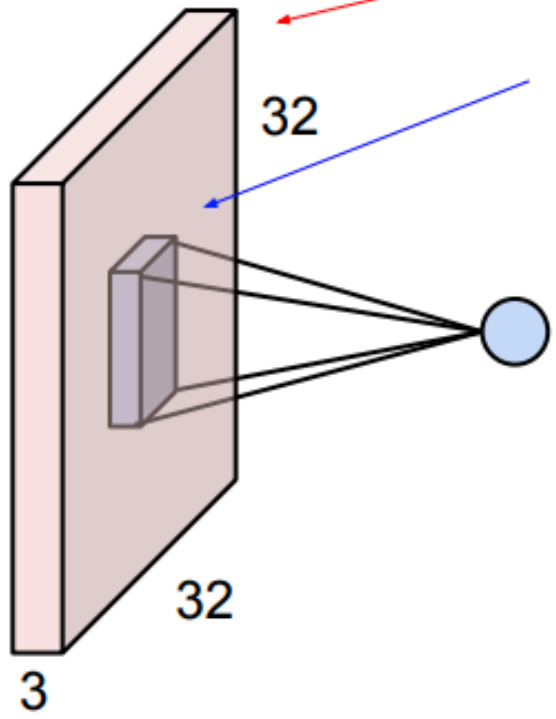
5x5x3 filter



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

Convolutional Neural Network

Convolution Layer



32x32x3 image

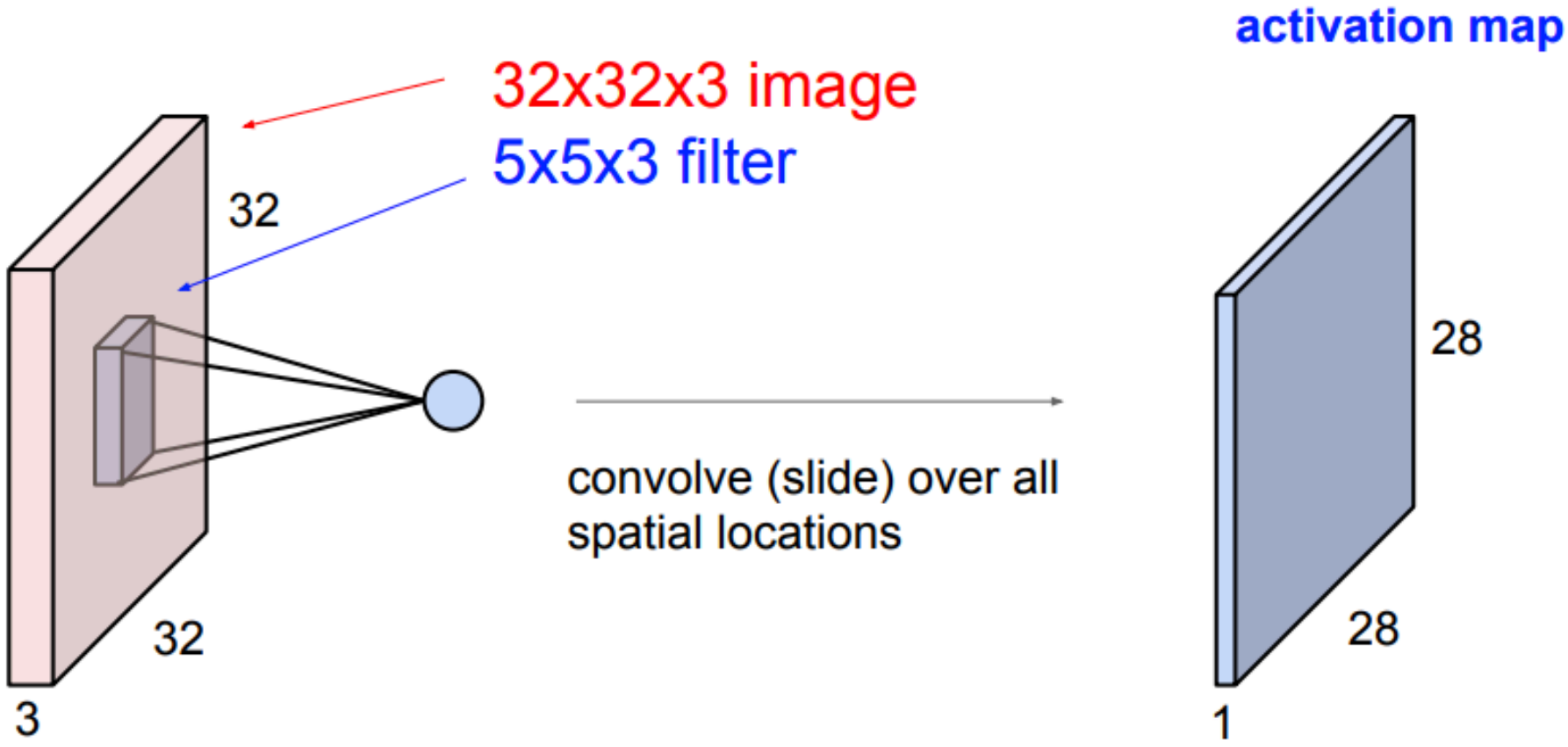
5x5x3 filter w

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)

$$w^T x + b$$

Convolutional Neural Network

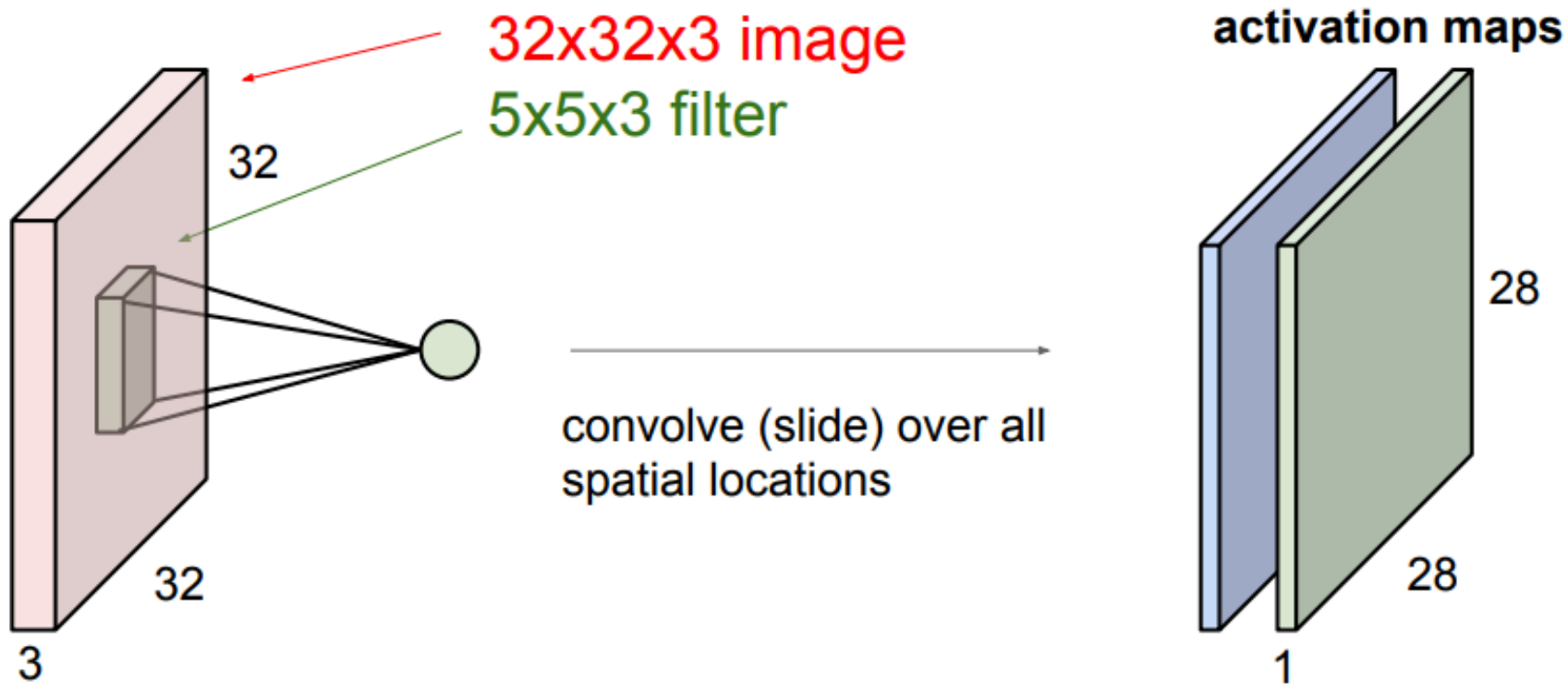
Convolution Layer



Convolutional Neural Network

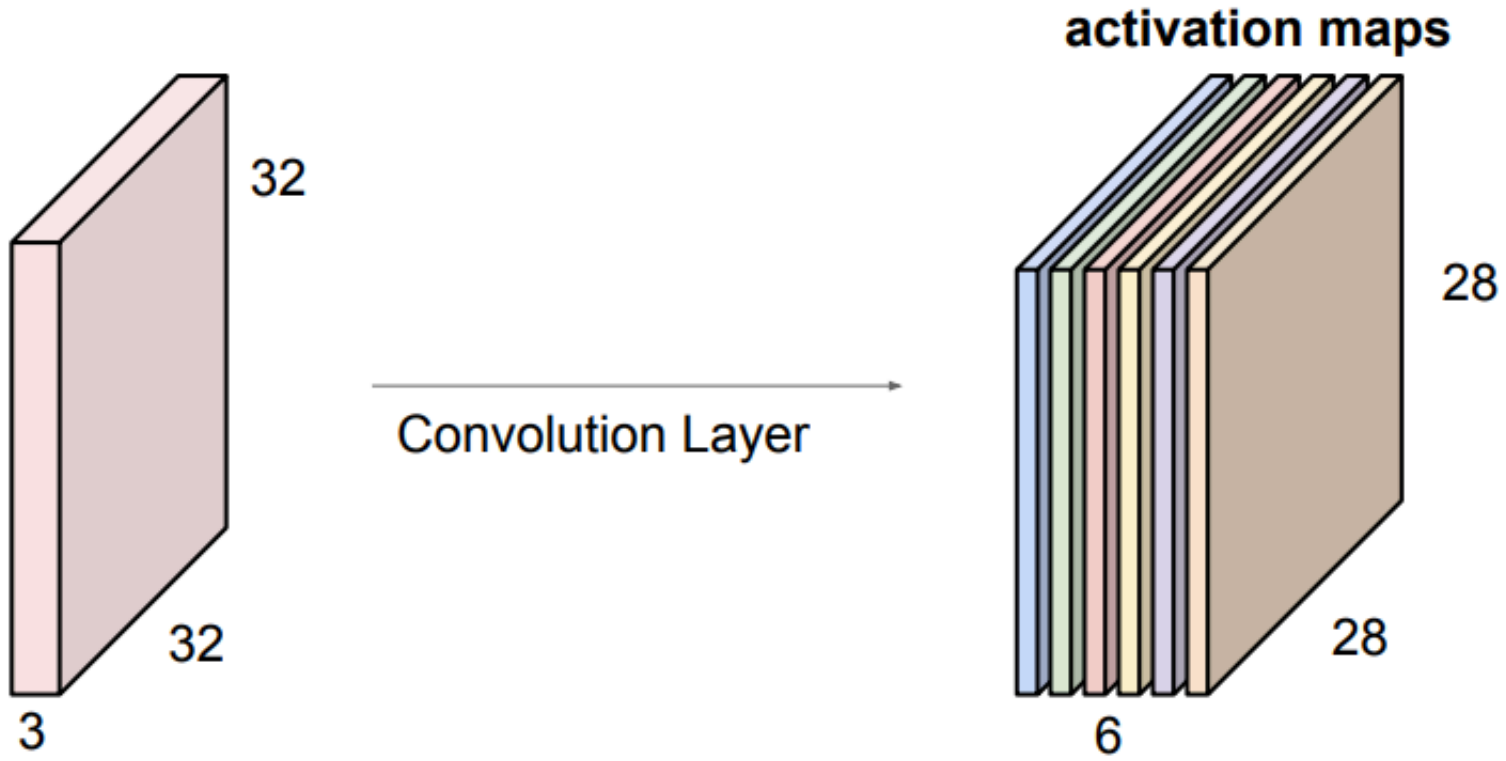
Convolution Layer

consider a second, green filter



Convolutional Neural Network

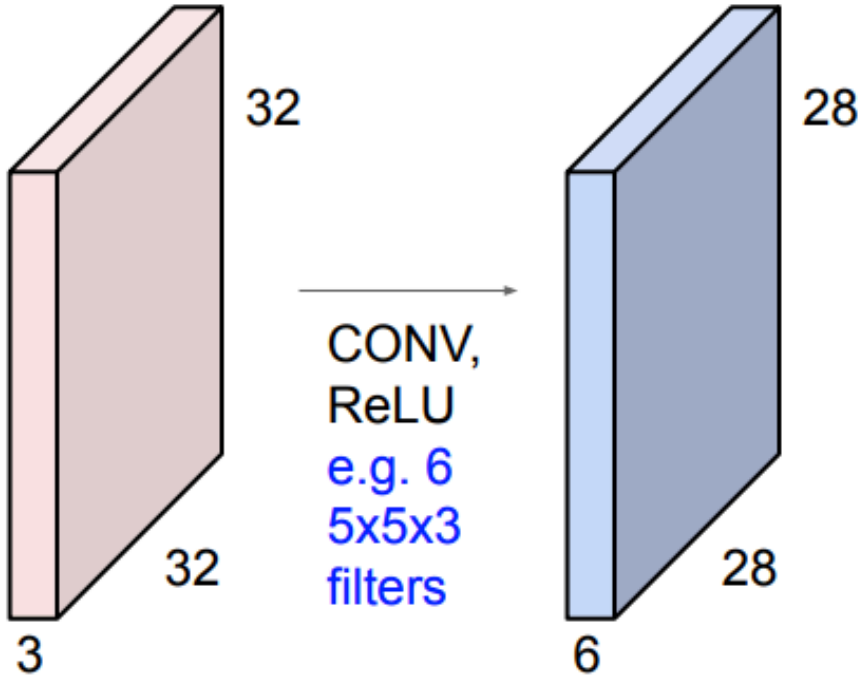
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



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

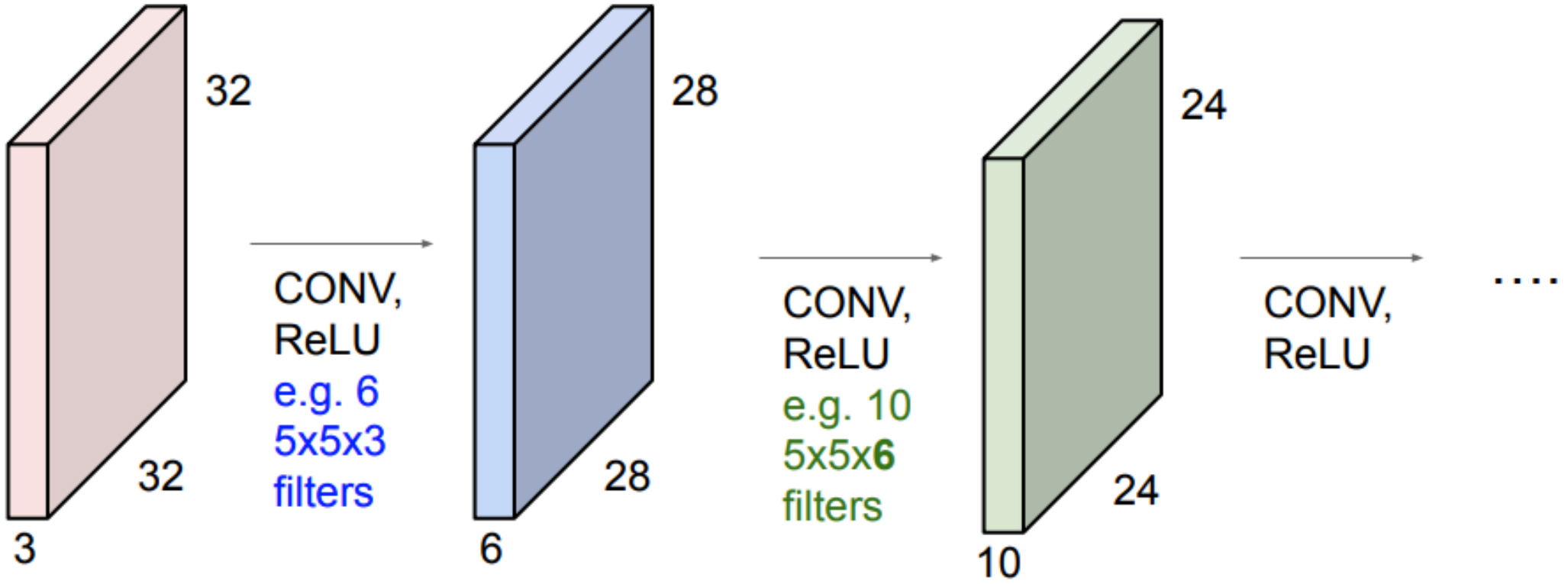
Convolutional Neural Network

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

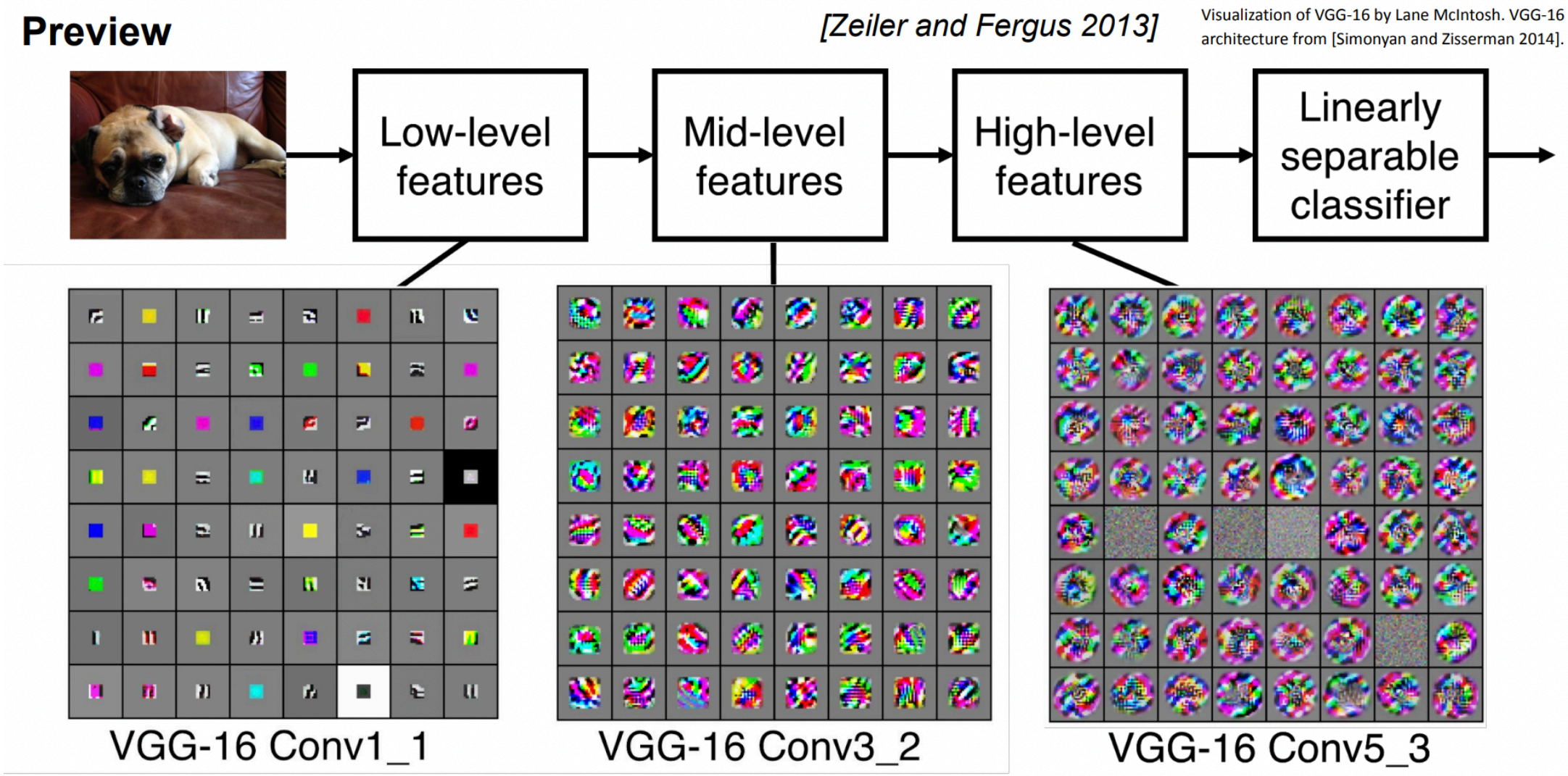


Convolutional Neural Network

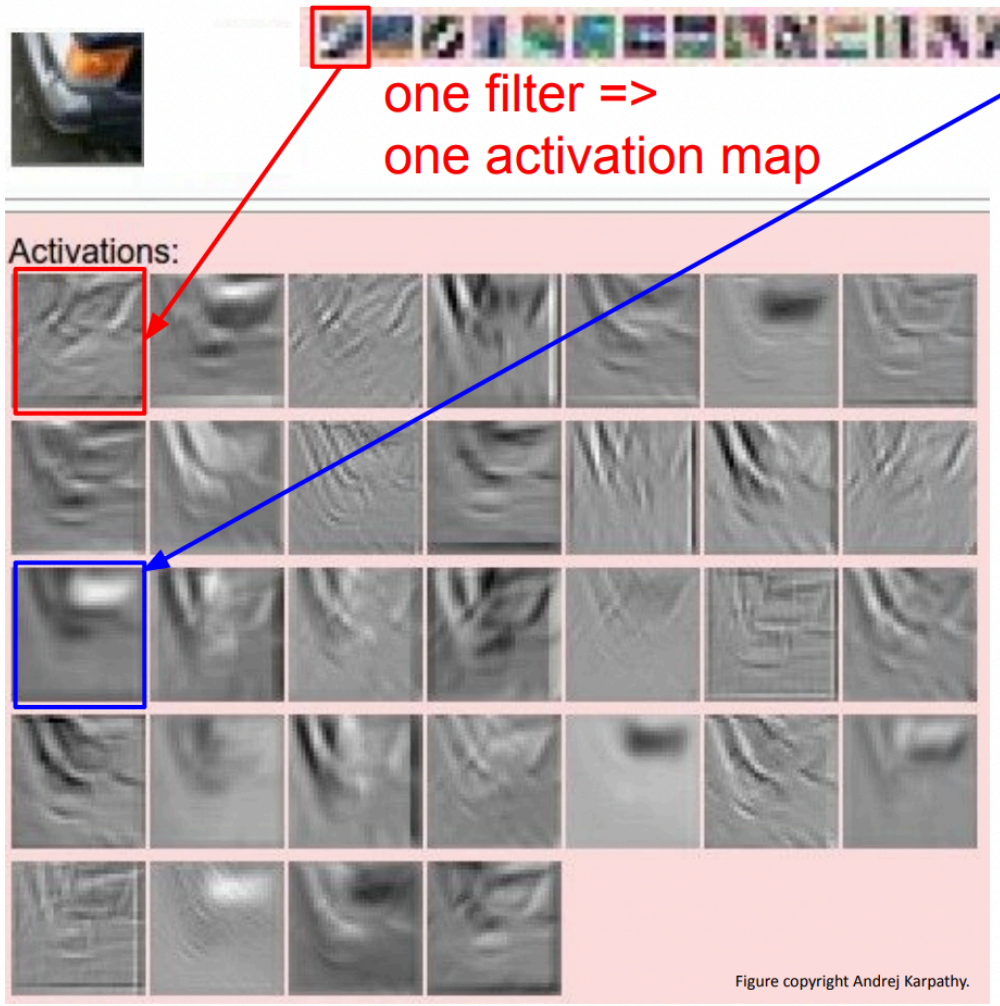
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions

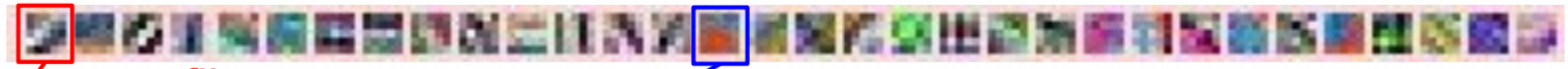


Feature Visualization



Feature Visualization





example 5x5 filters
(32 total)

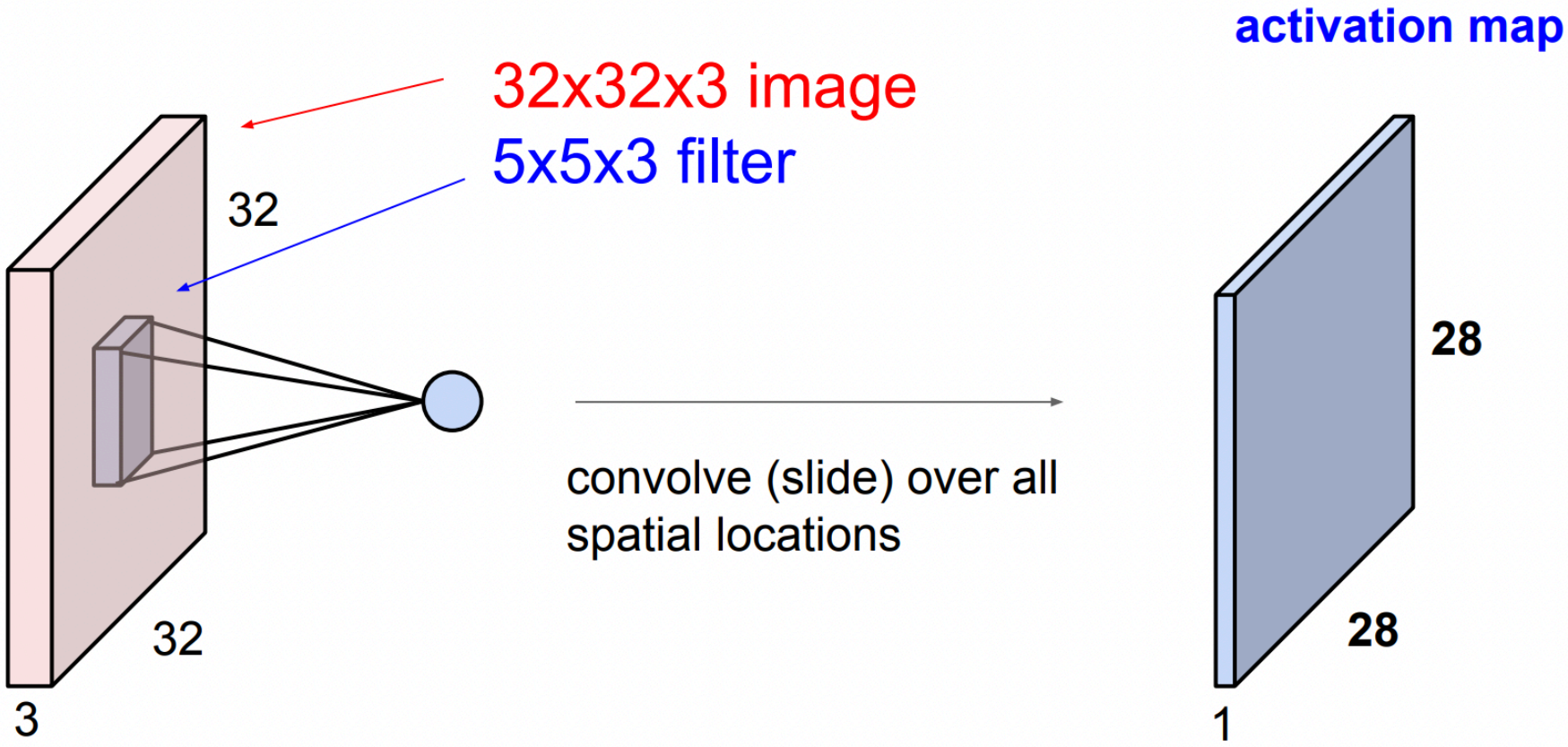
We call the layer convolutional because it is related to convolution of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

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

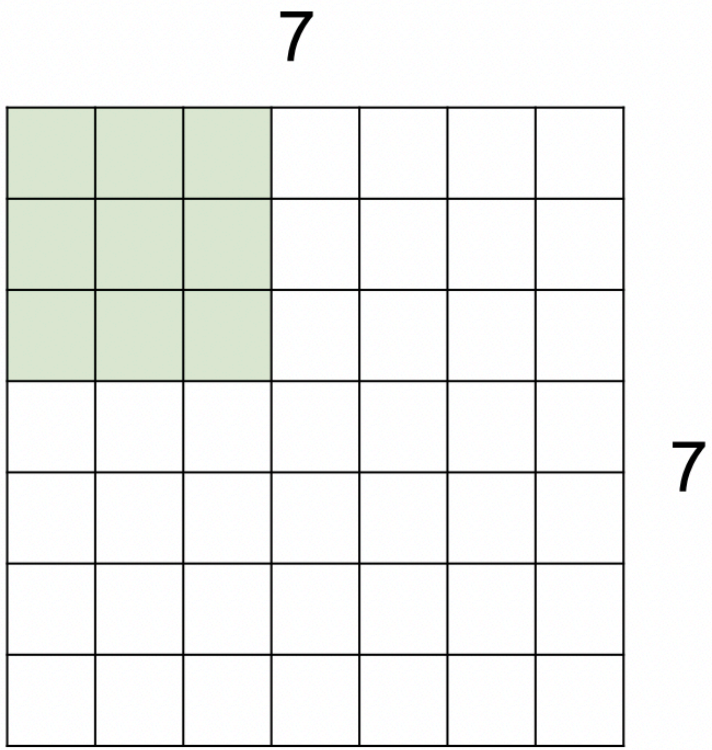
Convolutional Layer

A closer look at spatial dimensions:



Convolutional Layer

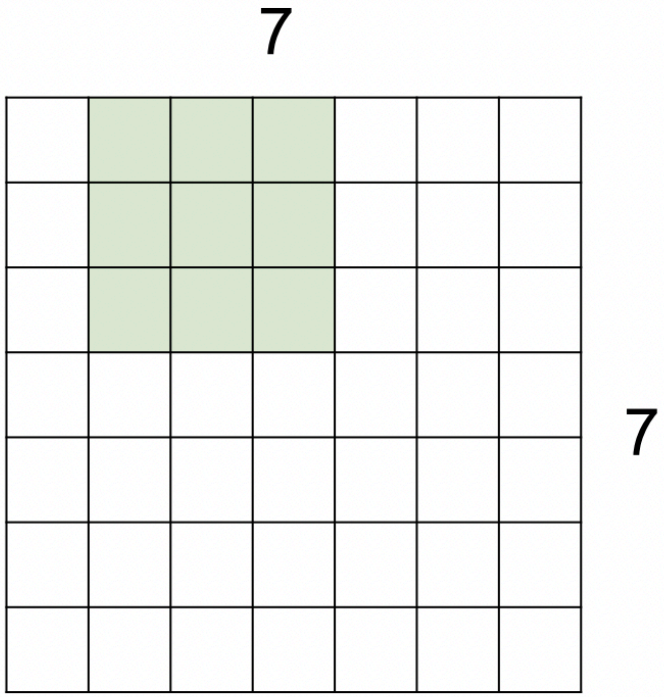
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutional Layer

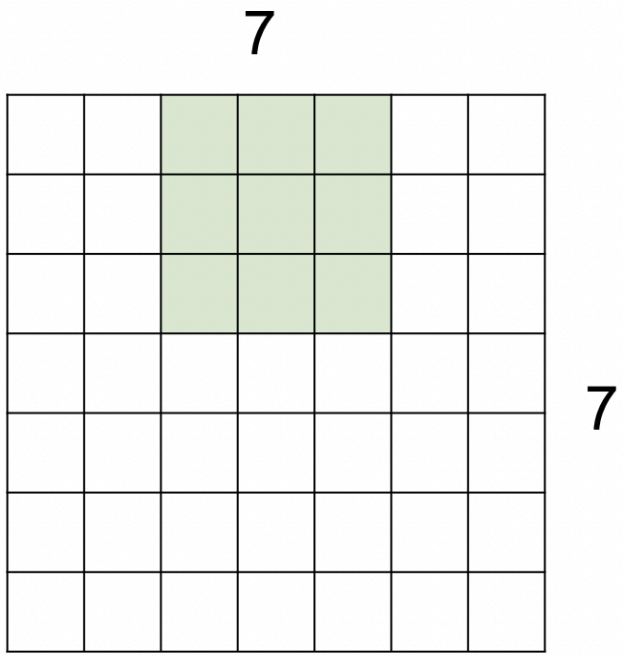
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutional Layer

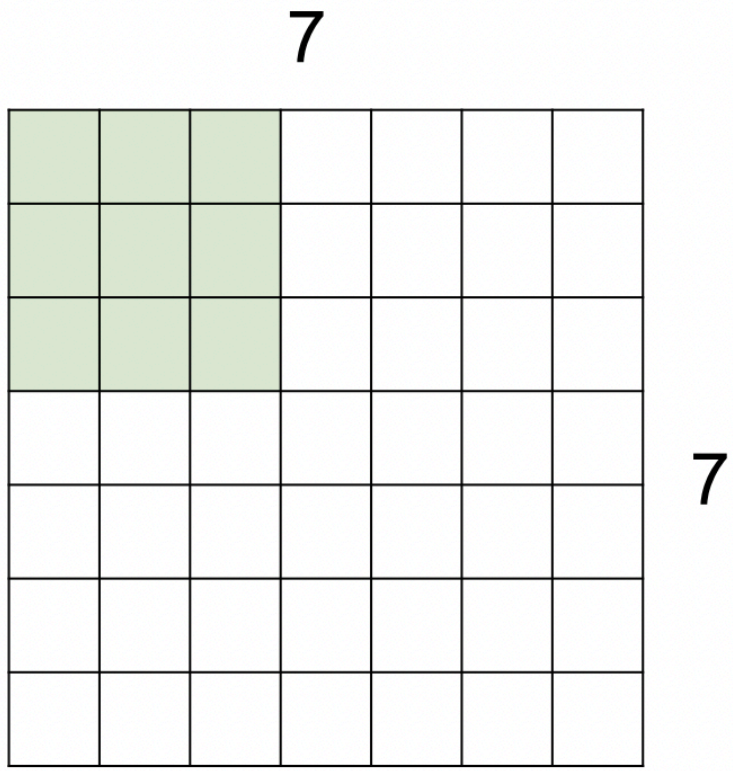
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutional Layer

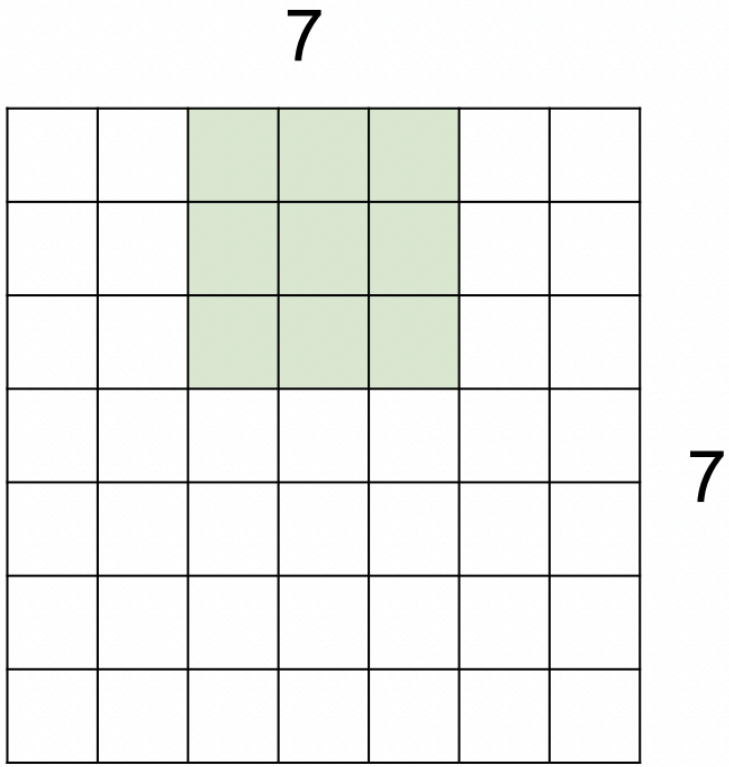
A closer look at spatial dimensions:



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

Convolutional Layer

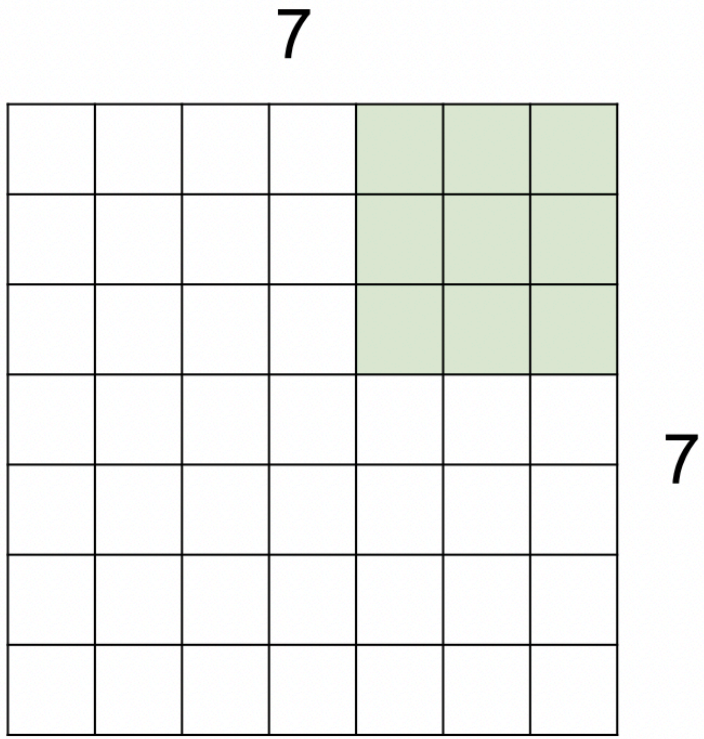
A closer look at spatial dimensions:



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

Convolutional Layer

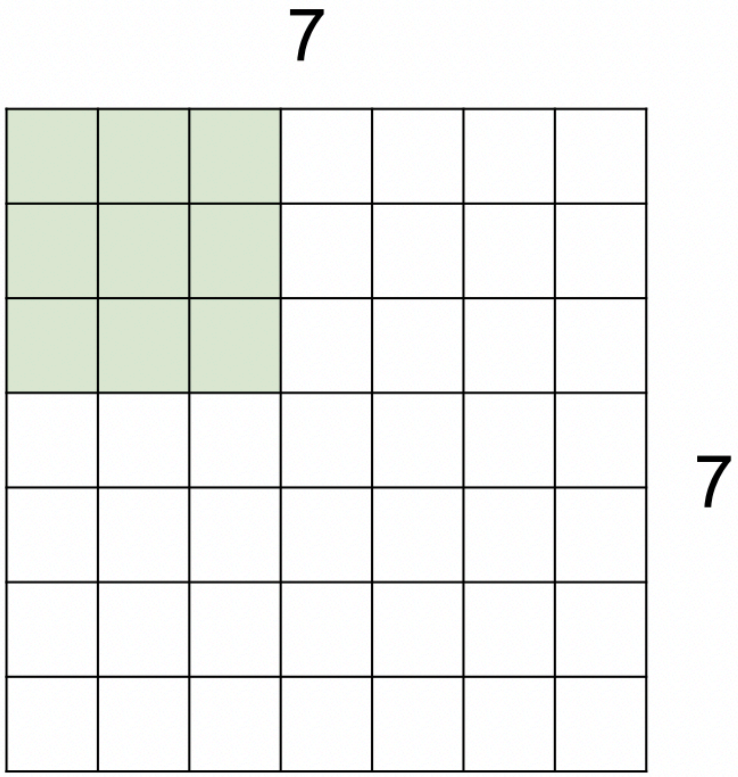
A closer look at spatial dimensions:



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

Convolutional Layer

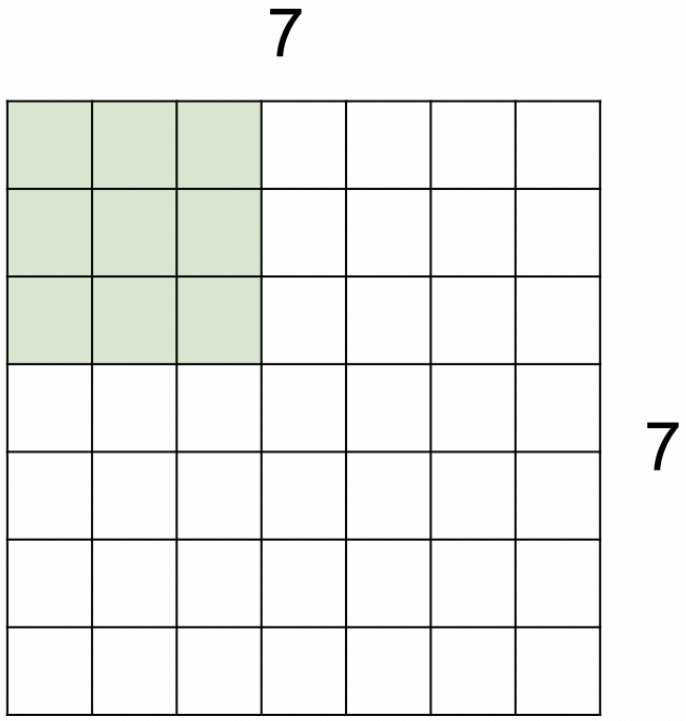
A closer look at spatial dimensions:



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

Convolutional Layer

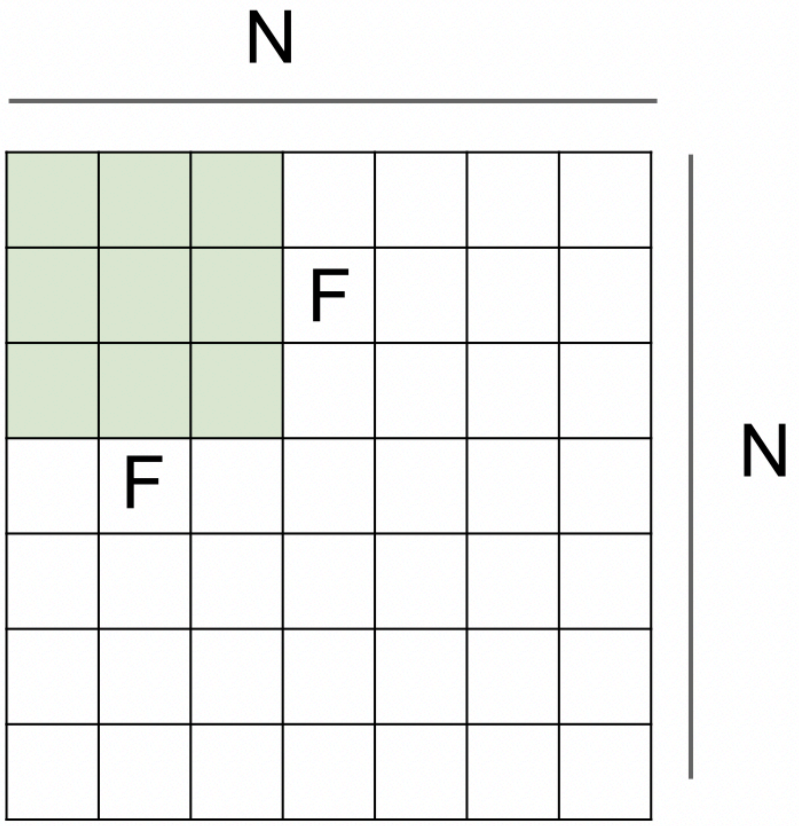
A closer look at spatial dimensions:



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

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

Convolutional Layer

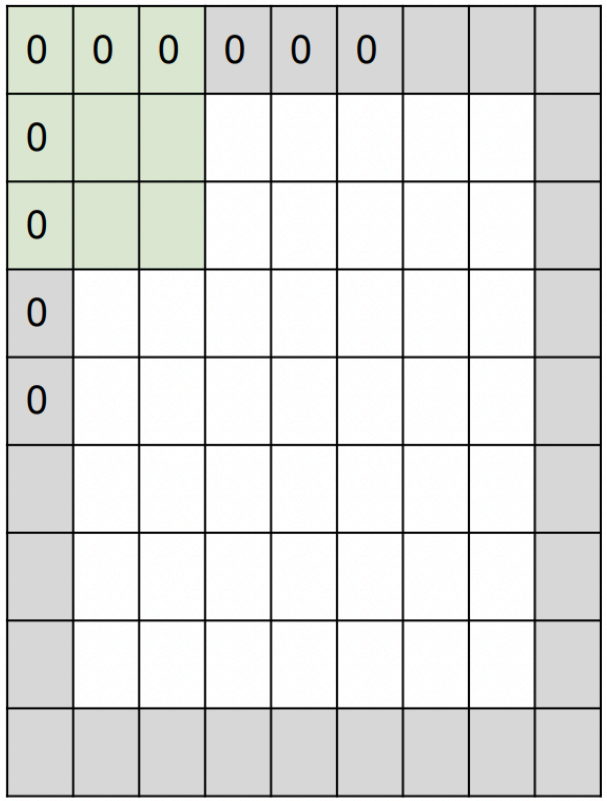


Output size:
(N - F) / stride + 1

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 :\

Padding

In practice: Common to zero pad the border

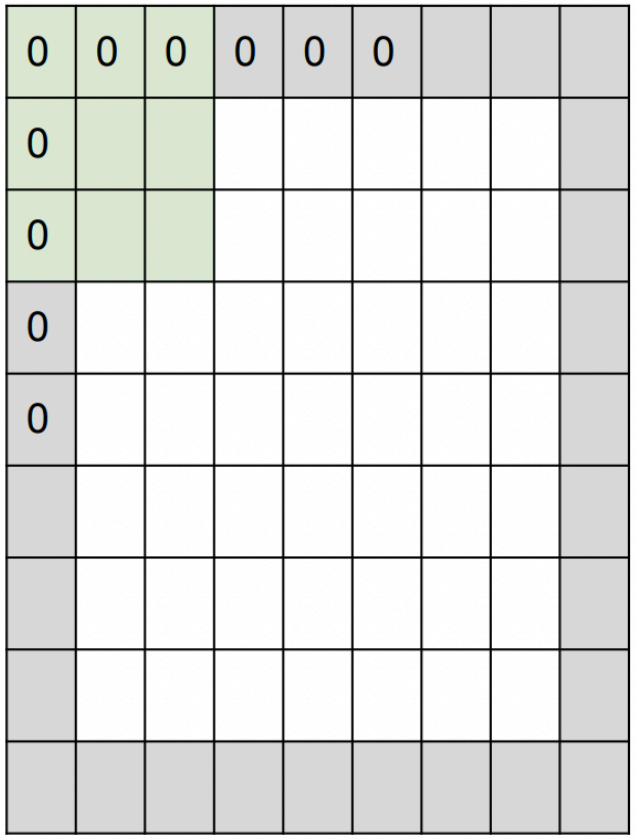


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

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

Padding

In practice: Common to zero pad the border



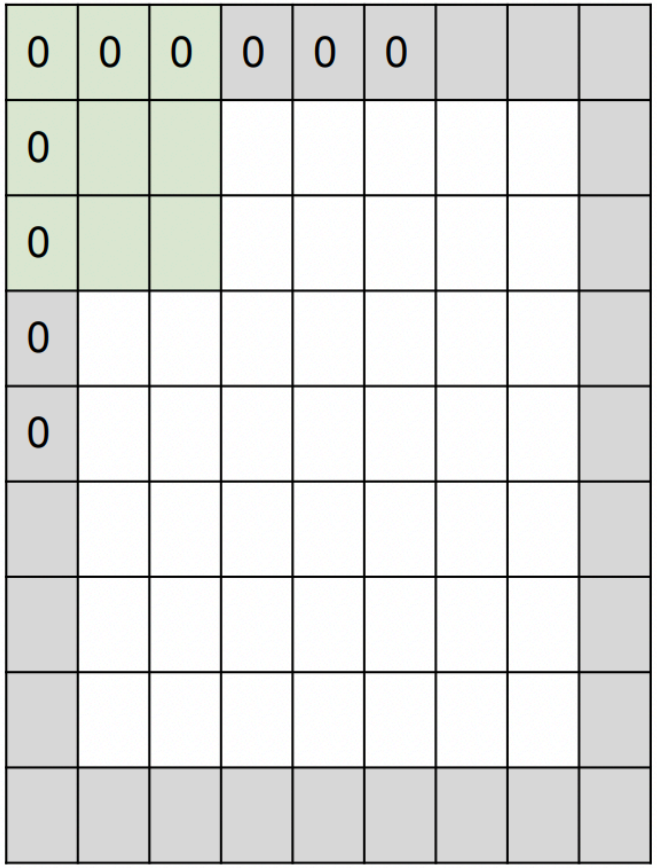
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

(recall:)
(N + 2P - F) / stride + 1

Padding

In practice: Common to zero pad the border

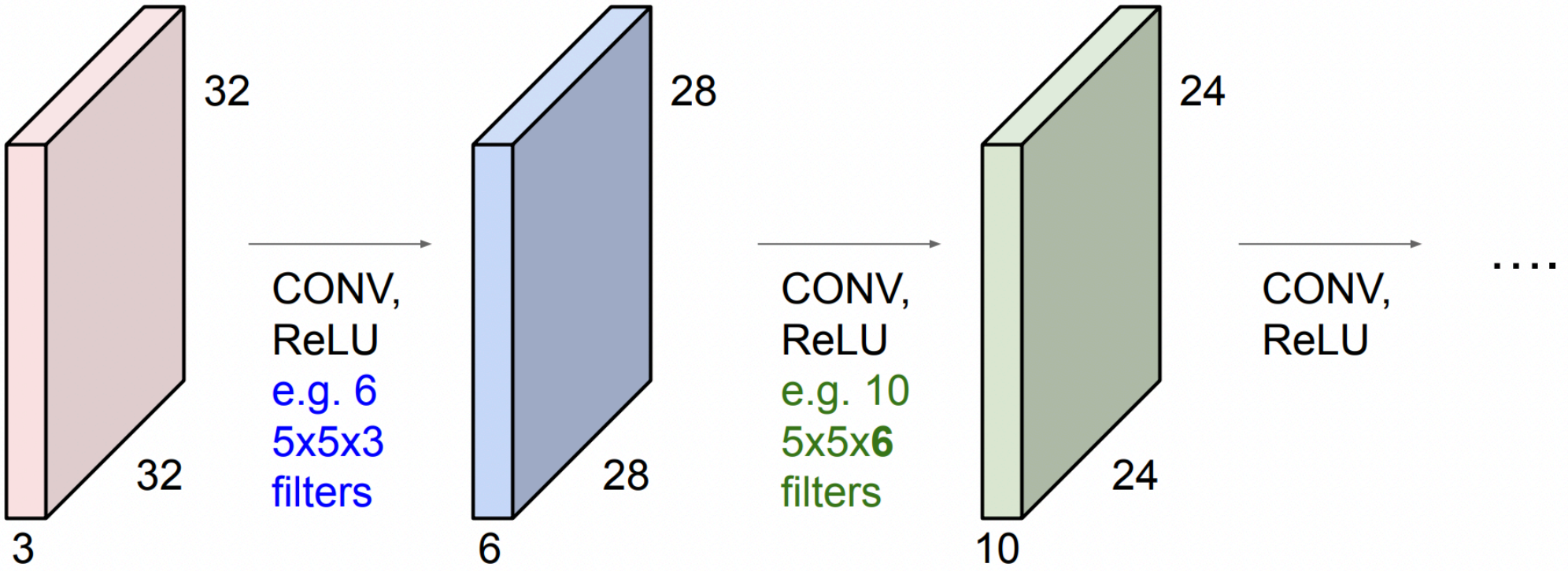


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

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

Convolutional Layer

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.

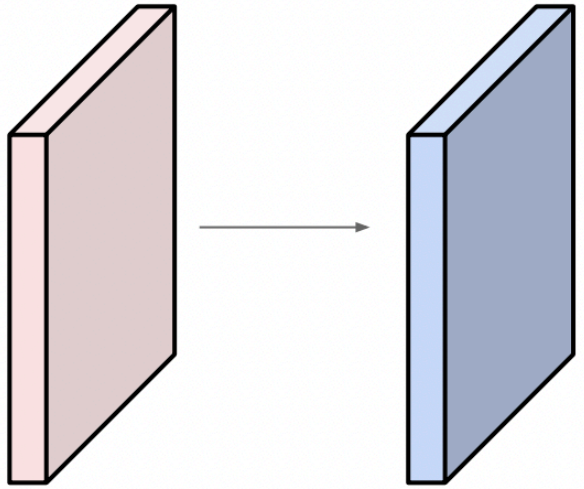


Convolutional Layer

Examples time:

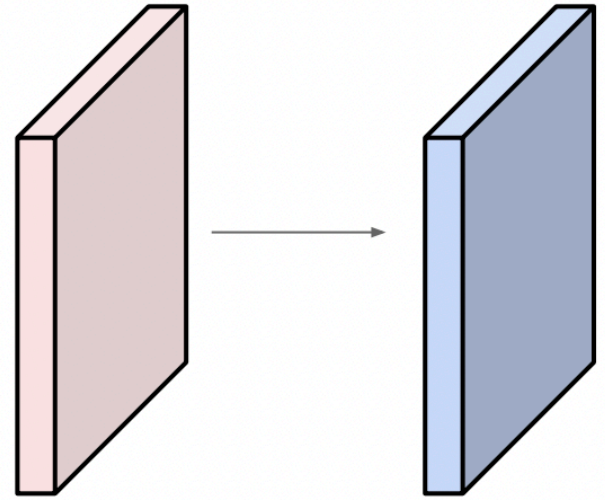
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?



Convolutional Layer

Examples time:



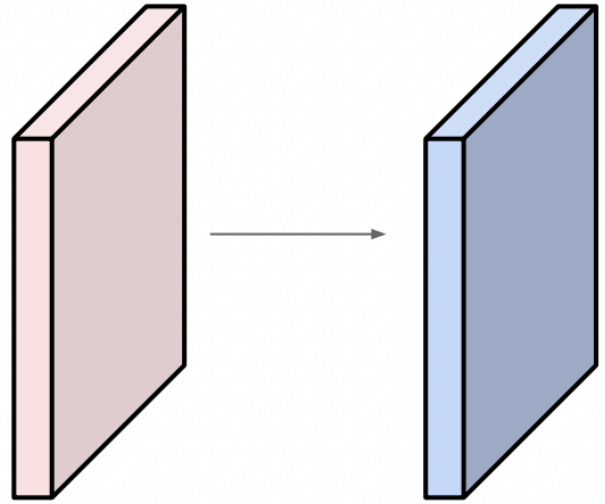
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

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

Convolutional Layer

Examples time:

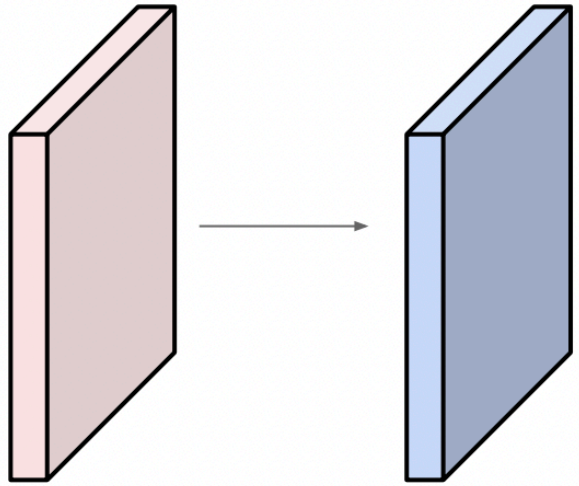
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2



Number of parameters in this layer?

Convolutional 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**

Convolutional Layer

Convolution layer: summary

Let's assume input is W_1 x H_1 x C
Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of W_2 x H_2 x K
where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters: F^2CK and K biases

Convolutional Layer

Convolution layer: summary

Let's assume input is W_1 x H_1 x C
Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of W_2 x H_2 x K
where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters: F^2CK and K biases

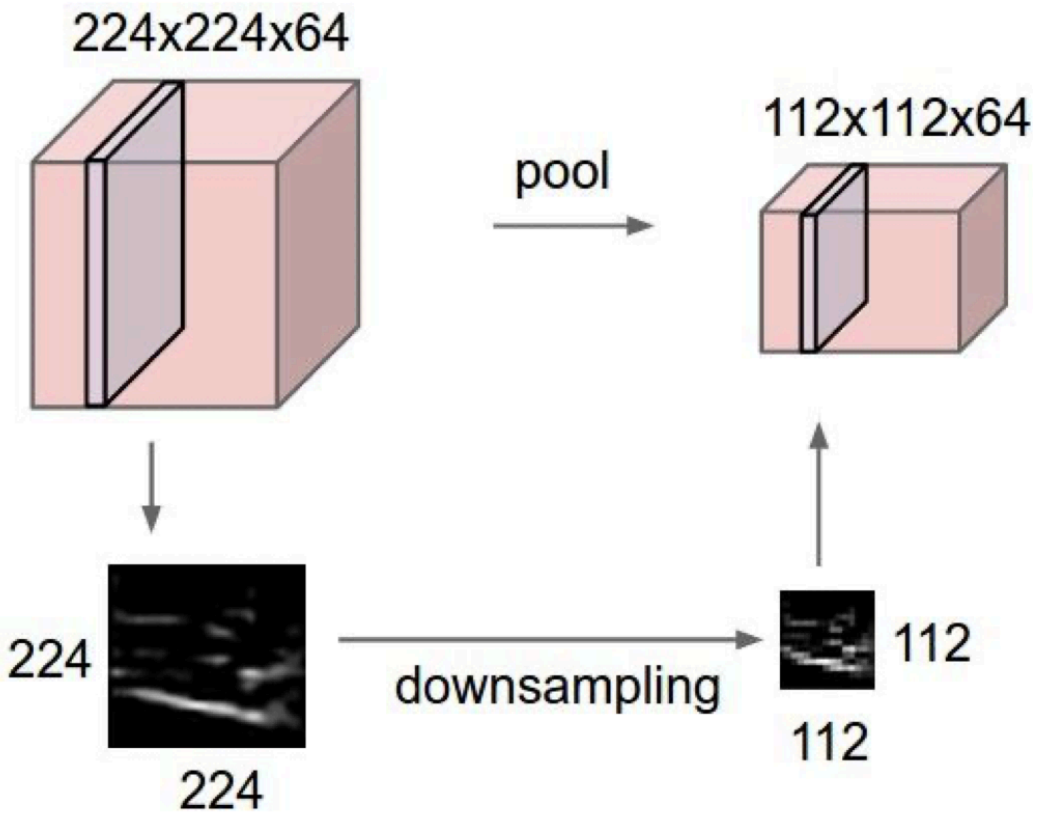
Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512)

- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

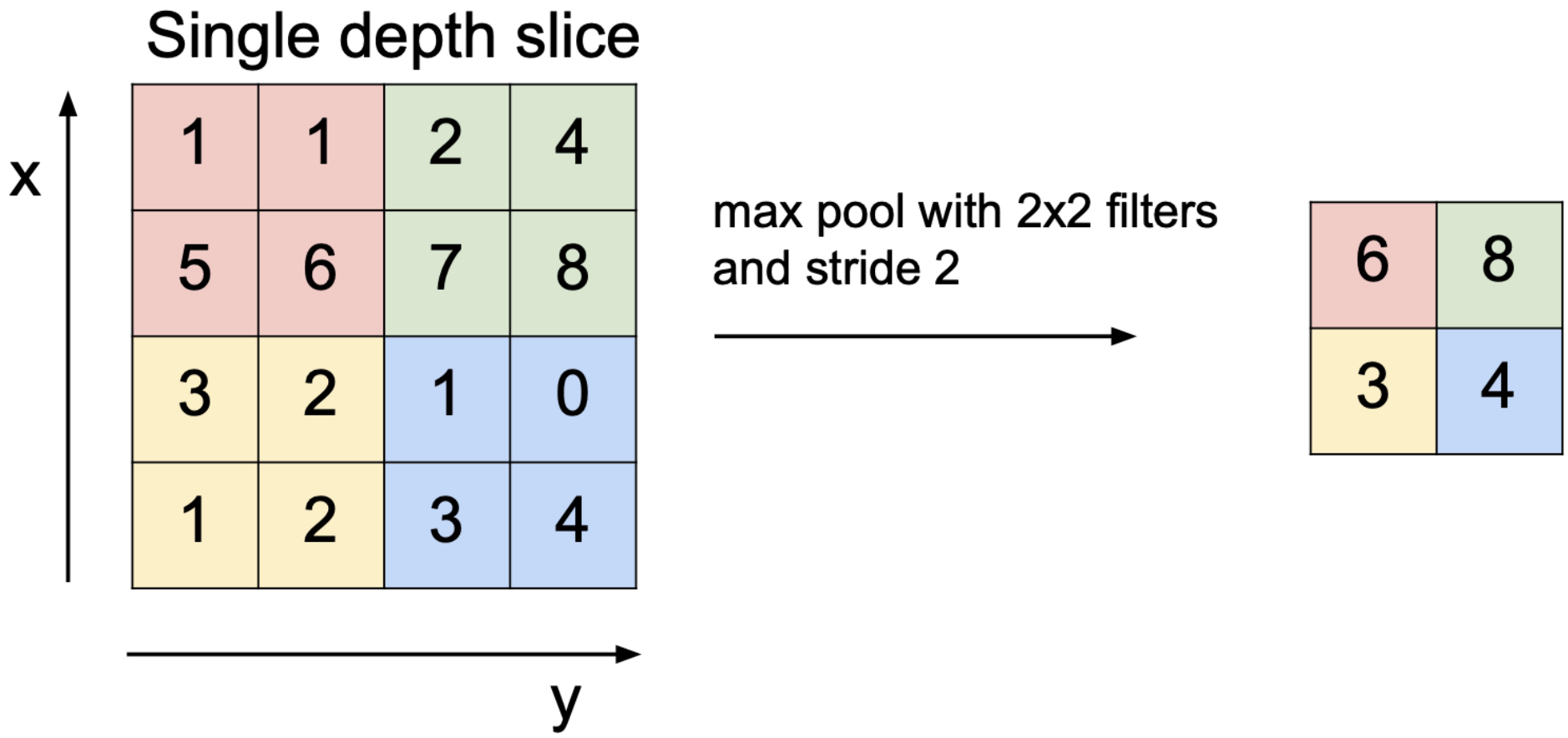
Pooling

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



Slide credit: Stanford CS231N

Max Pooling



Slide credit: Stanford CS231N

Pooling

- Average pooling
- Max pooling
- Sum pooling (rarely)

Pooling Layer: Summary

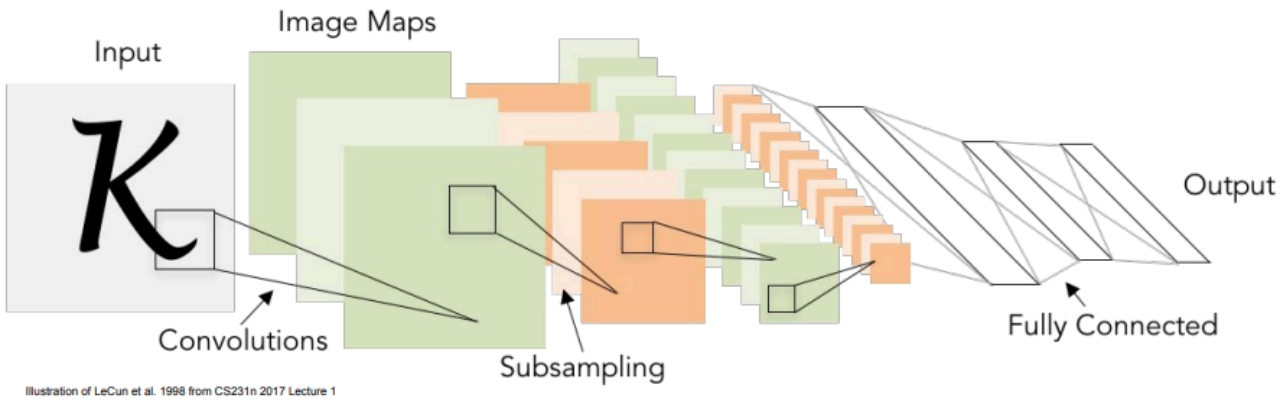
- Let's assume input is $W_1 \times H_1 \times C$
- 2×2 pooling will result in
  - $W_2 = \lceil W_1/2 \rceil$
  - $H_2 = \lceil H_2/2 \rceil$
- Number of parameters: 0

Summary of CNN-based Classification Network

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically architectures looked like

  [(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX

  where N is usually up to ~5, M is large, 0 <= K <= 2.
  - but recent advances such as ResNet/GoogLeNet have challenged this paradigm



Slides are borrowed from Stanford CS231N.

A Comparison between MLP and CNN

- Input: $W_1 \times H_1 \times C$
- Output: $W_2 \times H_2 \times K$

  - Fully connected layer (FC)
    - Densely connected
    - Parameters: $W_1 W_2 H_1 H_2 C K$
  - CNN layer
    - Filter size: F
    - Parameters: $F^2 C K$

Toeplitz Matrix for 1D convolution

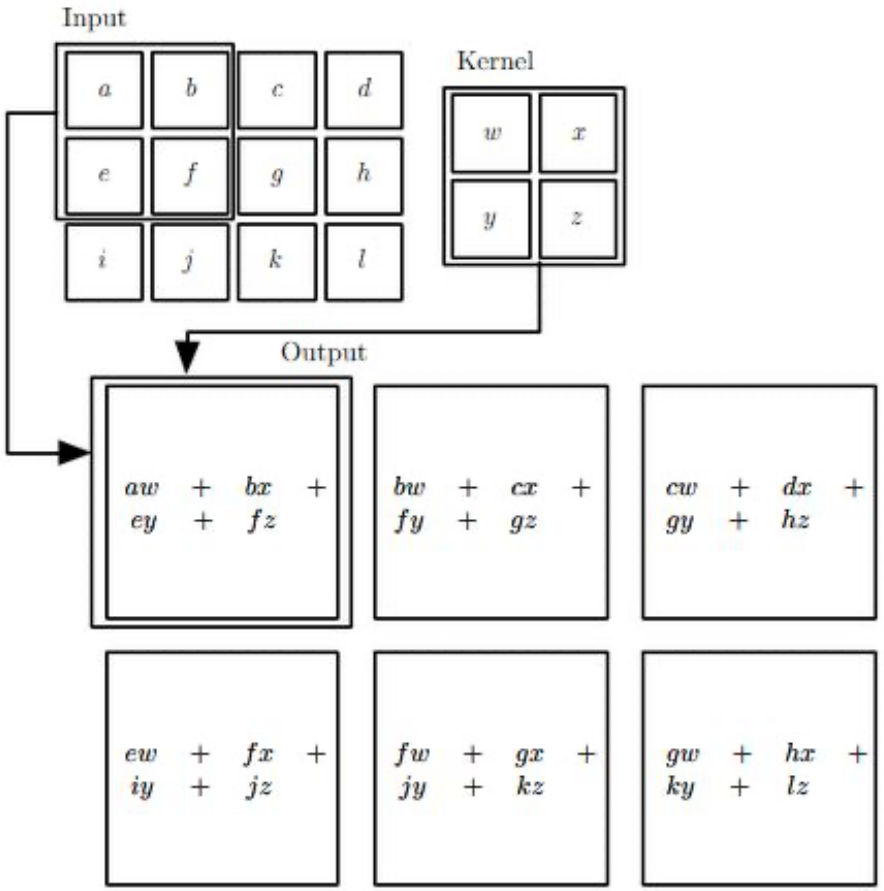
For 1D convolution, $h \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, the convolution operation $y = h * m$ can be constructed as a matrix multiplication:

$$y = h * x = \begin{bmatrix} h_1 & 0 & \cdots & 0 & 0 \\ h_2 & h_1 & & \vdots & \vdots \\ h_3 & h_2 & \cdots & 0 & 0 \\ \vdots & h_3 & \cdots & h_1 & 0 \\ h_{m-1} & \vdots & \ddots & h_2 & h_1 \\ h_m & h_{m-1} & & \vdots & h_2 \\ 0 & h_m & \ddots & h_{m-2} & \vdots \\ 0 & 0 & \cdots & h_{m-1} & h_{m-2} \\ \vdots & \vdots & & h_m & h_{m-1} \\ 0 & 0 & 0 & \cdots & h_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

- A sparse matrix
- A lot of equal constraints

a Toeplitz matrix or diagonal-constant matrix, named after Otto Toeplitz, is a matrix in which each descending diagonal from left to right is constant.

Matrix for 2D convolution



A very sparse
doubly block circulant matrix

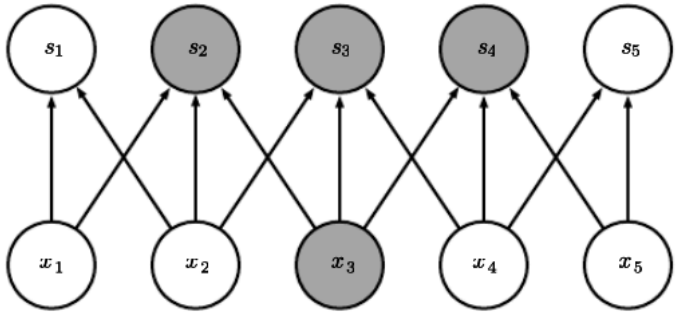
Figure from Deep learning (MIT press, 2016).

Sparse Connectivity

- Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input.

CNN

1D Conv with filter size=3



FC

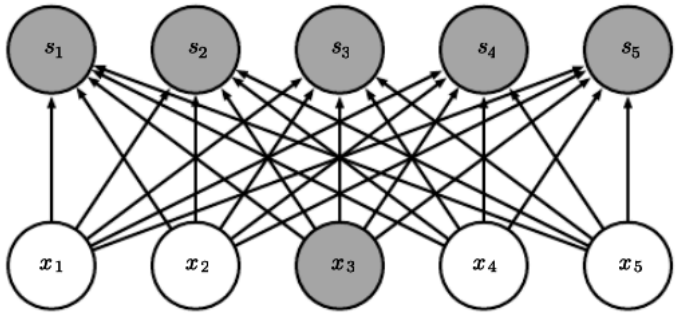


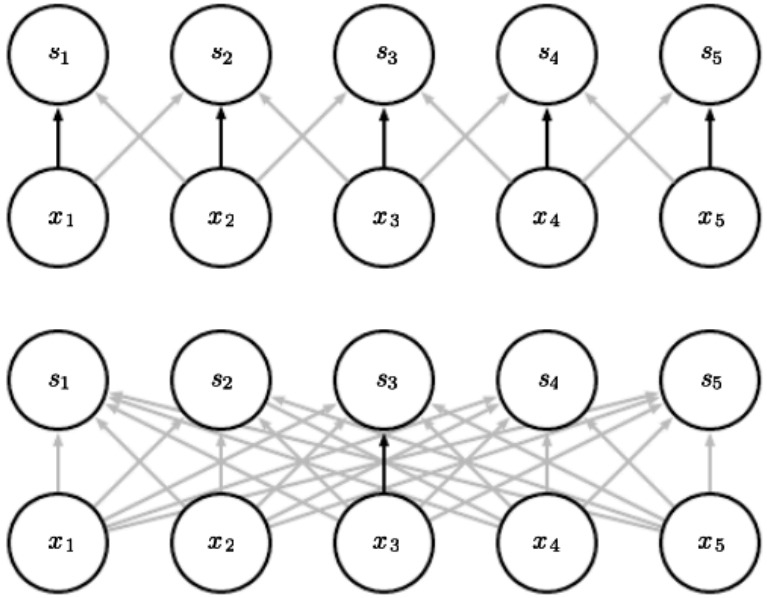
Figure from Deep learning (MIT press, 2016).

Parameter Sharing

- refers to using the same parameter for more than one function in a model.

CNN

FC



Parameter sharing. Black arrows indicate the connections that use a particular parameter in two different models.

Figure from Deep learning (MIT press, 2016).

Why CNN?

- Which is more expressive?
  - CNN
  - FC

Why CNN?

- Which is more expressive?
  - CNN
  - FC is a super set of CNN (without sparse and parameter sharing constraints.)

- What is the problem of FC/MLP?

Why CNN?

- Which is more expressive?
  - CNN
  - FC is a super set of CNN (without sparse and parameter sharing constraints.)

- What is the problem of FC/MLP?
  - FC needs too many parameters.
  - What else?

Image Perturbations

Applying a small translation

https://greatergood.berkeley.edu/article/item/the_science_backed_benefits_of_being_a_dog_owner

Image Perturbations

Applying a small rotation

https://greatergood.berkeley.edu/article/item/the_science_backed_benefits_of_being_a_dog_owner

The behavior of Fully Connect Layer

- FC will change dramatically even if you just shift one row or one column. Same happens for a 5° rotation.
- A huge problem during network training or optimization!
- What about CNN?

Equivariance

General definition of equivariance:

$$S_A[\phi(X)] = \phi(T_A(X))$$

Here A is an operation, T_A and S_A are their transformation function in the space of X and $\phi(X)$.

Equivariance

General definition of equivariance:

$$S_A[\phi(X)] = \phi(T_A(X))$$

Here A is an operation, T_A and S_A are their transformation function in the space of X and $\phi(X)$.

Invariance is a special case of equivariance,
When $S_A = I, \phi(X) = \phi(T_A(X))$.

Parameter Sharing = Equivariance with Translation

- Ignoring the values at the boundary, 2D Conv is equivariant with translation.
- We use the same way to process all local regions due to parameter sharing.
- For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image.

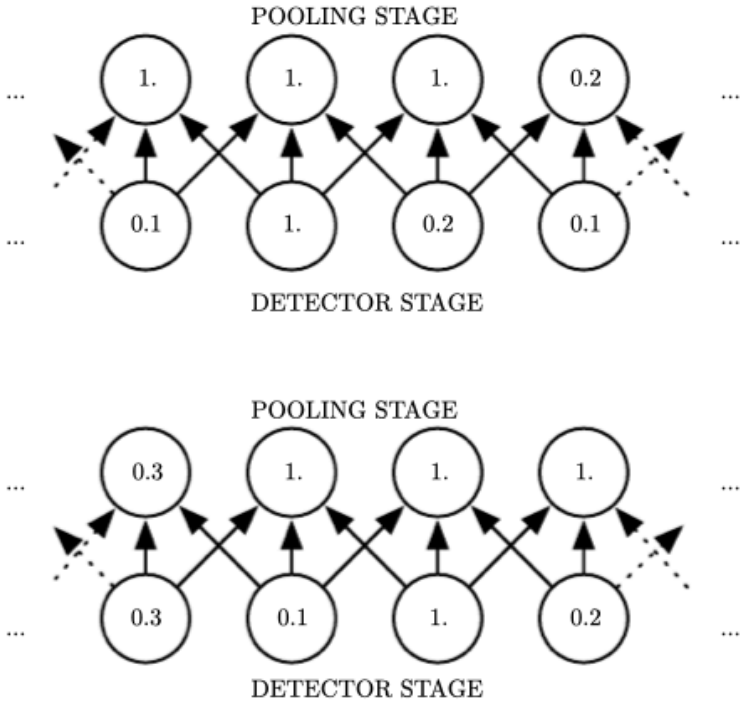
CNN

- Convolution is not naturally equivariant with some other transformations
  - changes in the scale
  - rotation of an image.

- Other mechanisms are necessary for handling these kinds of transformations.

Pooling

- Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is.

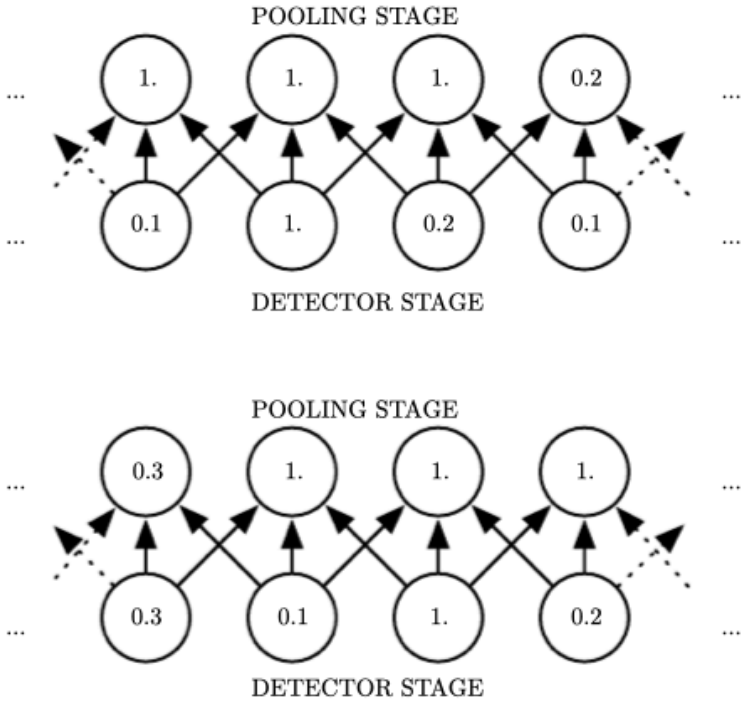


Max pooling introduces invariance.

Figure from Deep learning (MIT press, 2016).

Pooling

- Applying pooling induces invariance to small translations and rotations.



Max pooling introduces invariance.

Figure from Deep learning (MIT press, 2016).

Convolution and Pooling as a Prior

- CNN stacks Conv layer and pooling layer.
- Conv layer + pooling layer is invariant with small translation and rotation.
- Thus CNN is much easier to optimize than a FC for an image.
- Loss landscape:

CNN Training

To Train a CNN

Mini-batch SGD

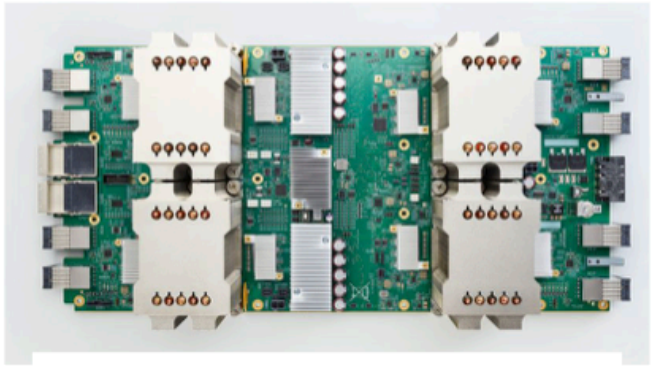
Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Slide credit: Stanford CS231N

To Train a CNN

Hardware + Software



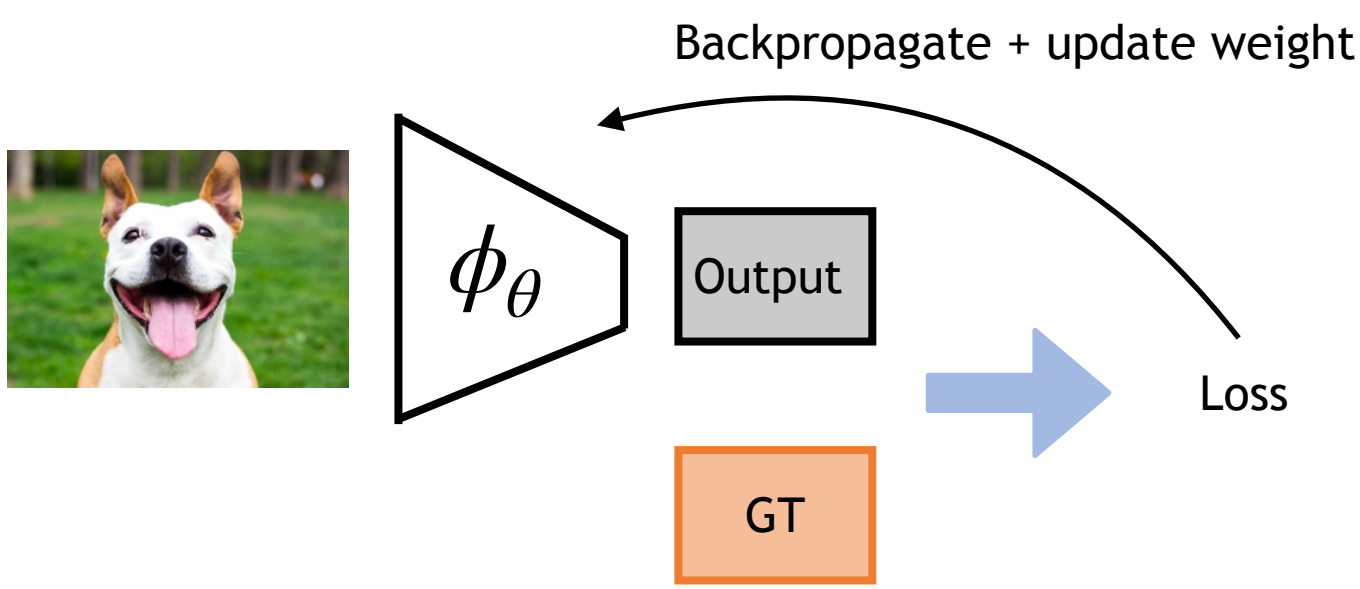
PyTorch

TensorFlow

Slide credit: Stanford CS231N

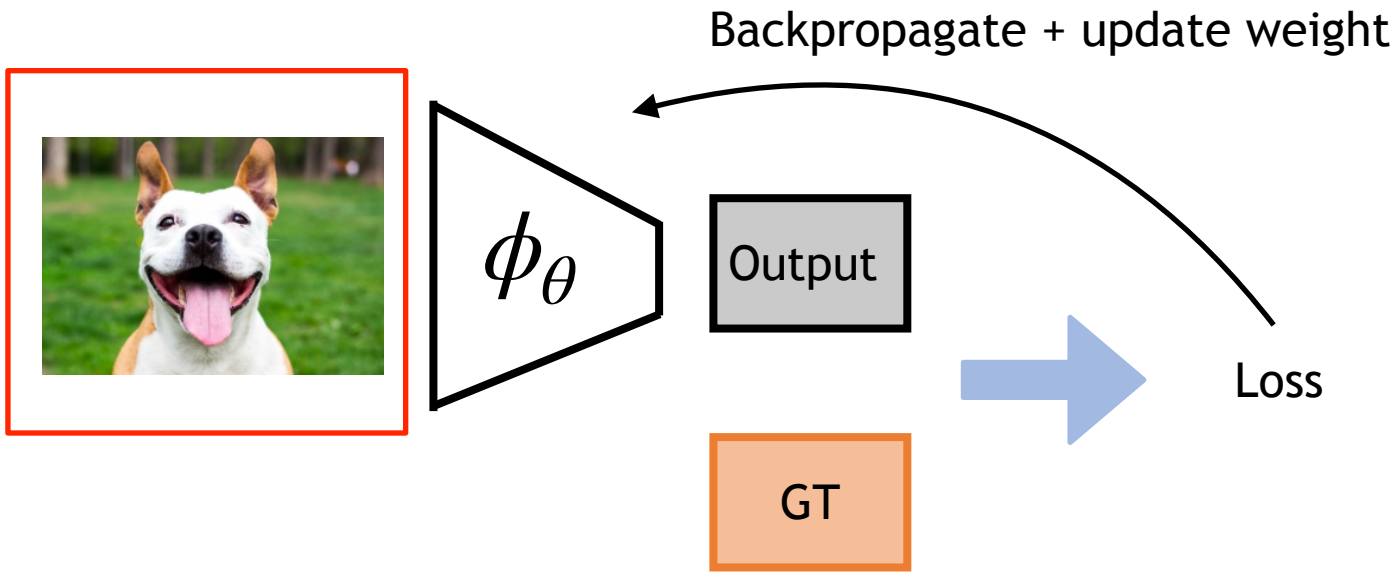
Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?

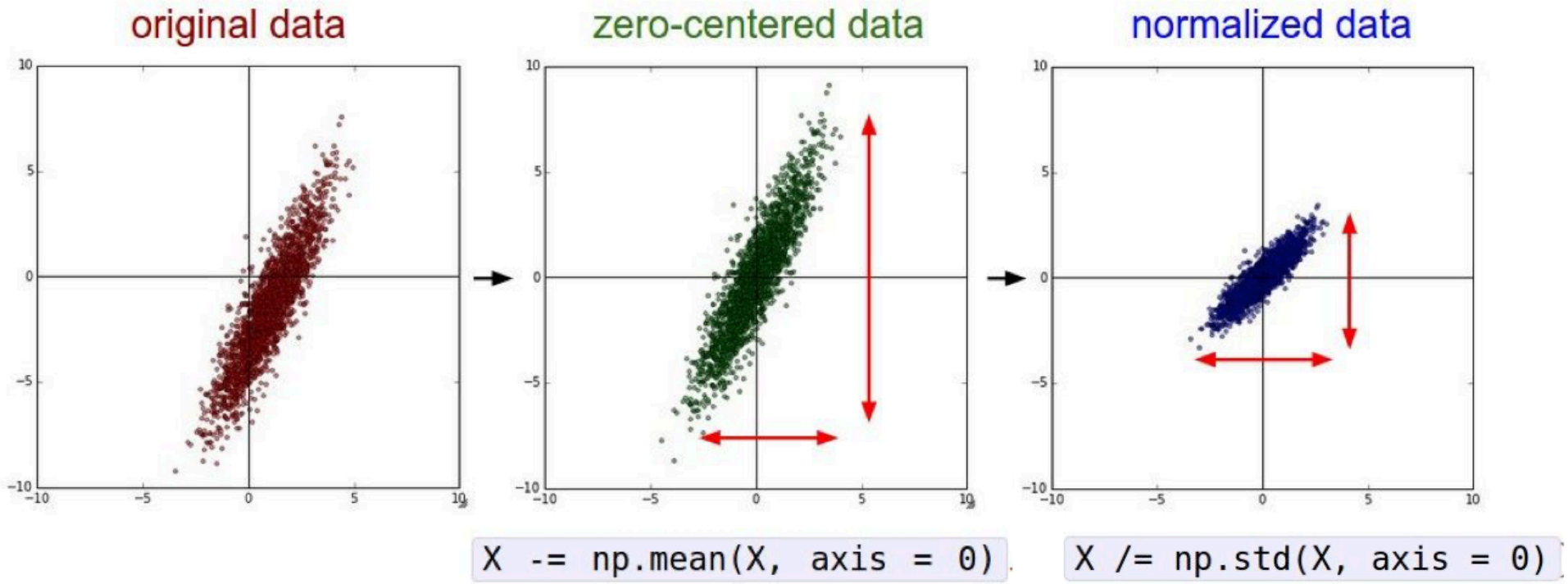


Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



Data Preprocessing



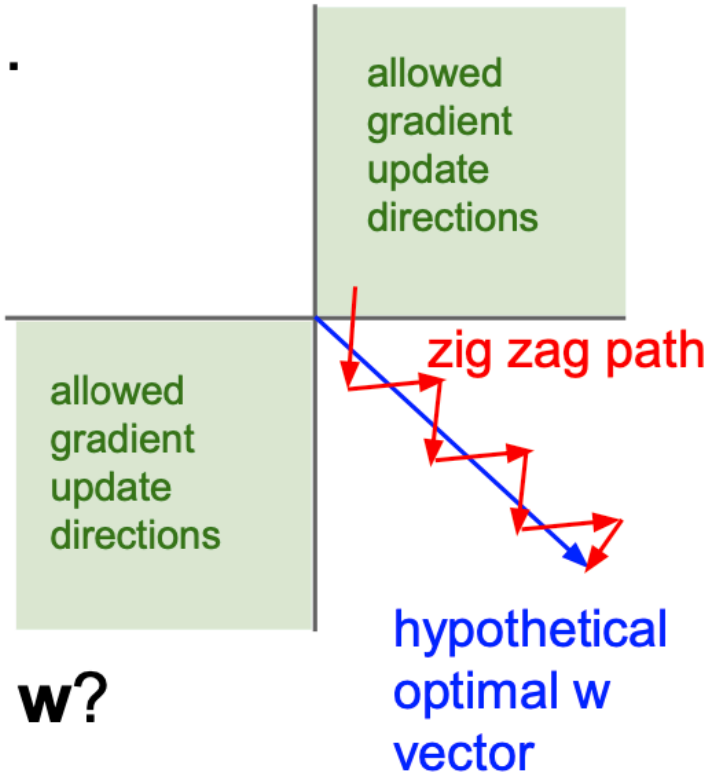
(Assume X [NxD] is data matrix,
each example in a row)

Slide credit: Stanford CS231N

Data Preprocessing

Remember: Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on **w**?

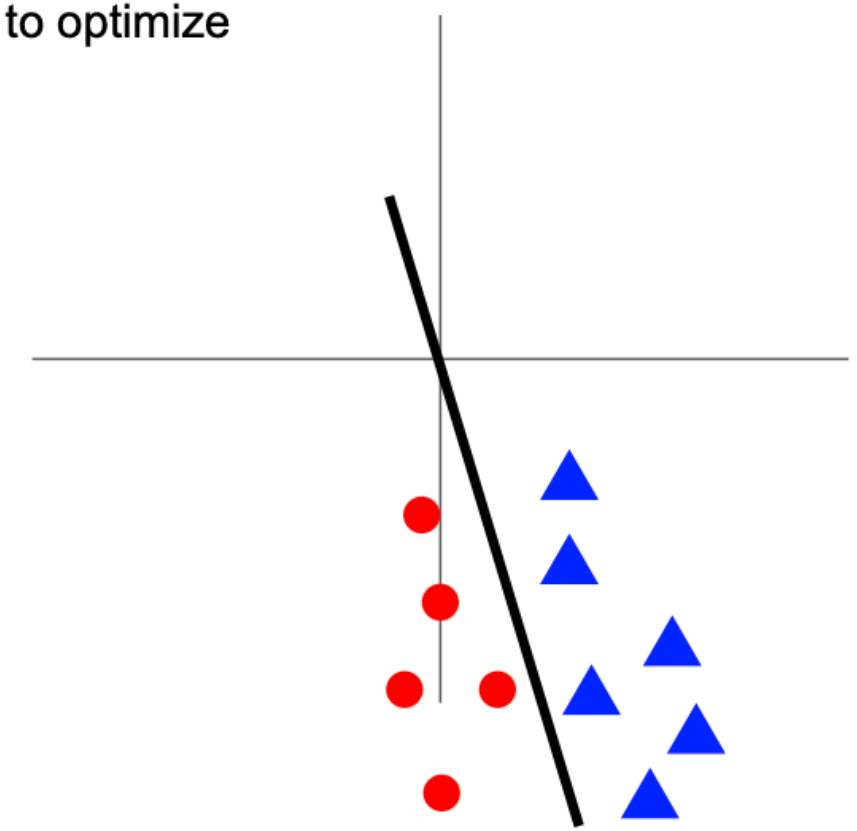
Always all positive or all negative :(

(this is also why you want zero-mean data!)

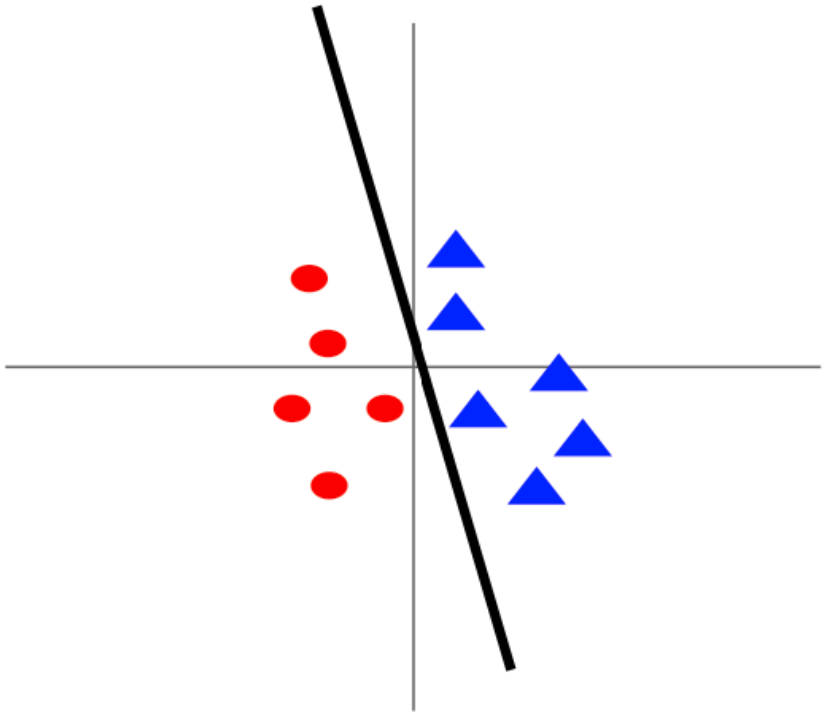
Slide credit: Stanford CS231N

Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Slide credit: Stanford CS231N

Summary of Data Preprocessing

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

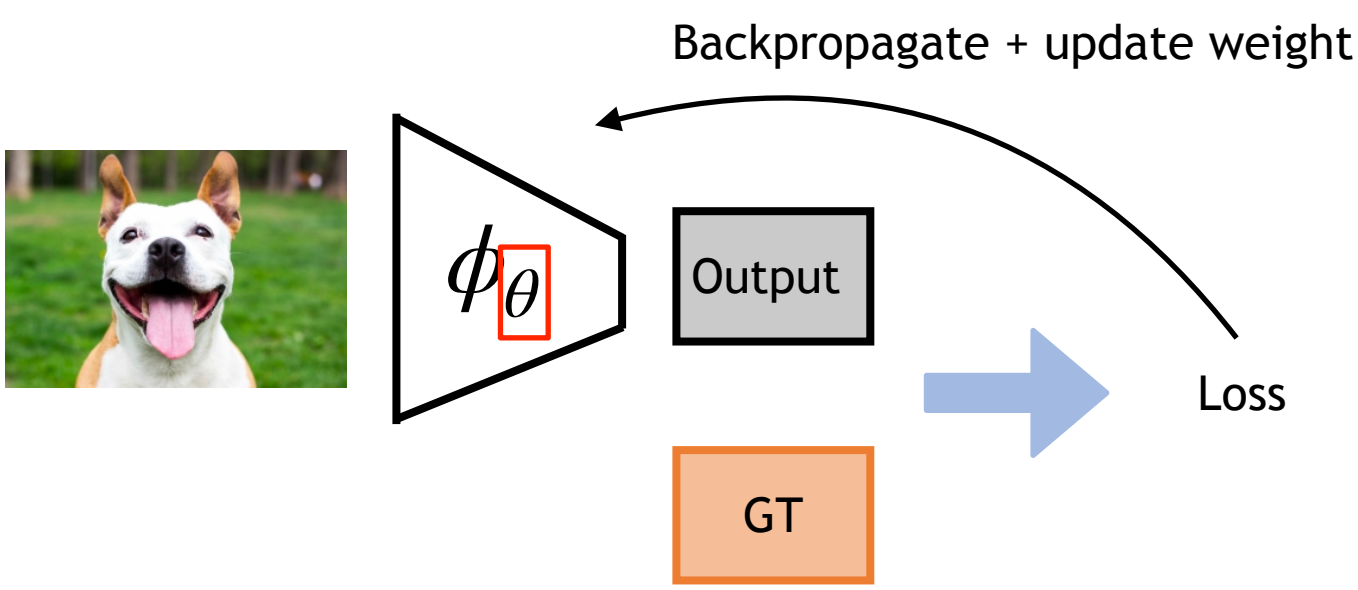
Slide credit: Stanford CS231N

Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?

Outline

- Data preparation
- **Weight Initialization**
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



Weight Initialization

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(Din, Dout)
```

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

Slide credit: Stanford CS231N

Weight Initialization

```
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

What will happen to the activations for the last layer?

Slide credit: Stanford CS231N

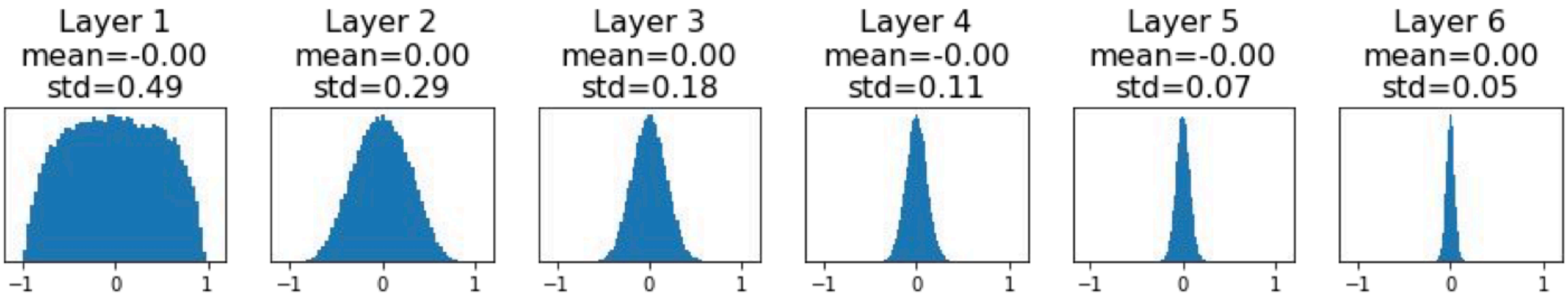
Weight Initialization: Activation Statistics

```
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?



Slide credit: Stanford CS231N

Weight Initialization: Activation Statistics

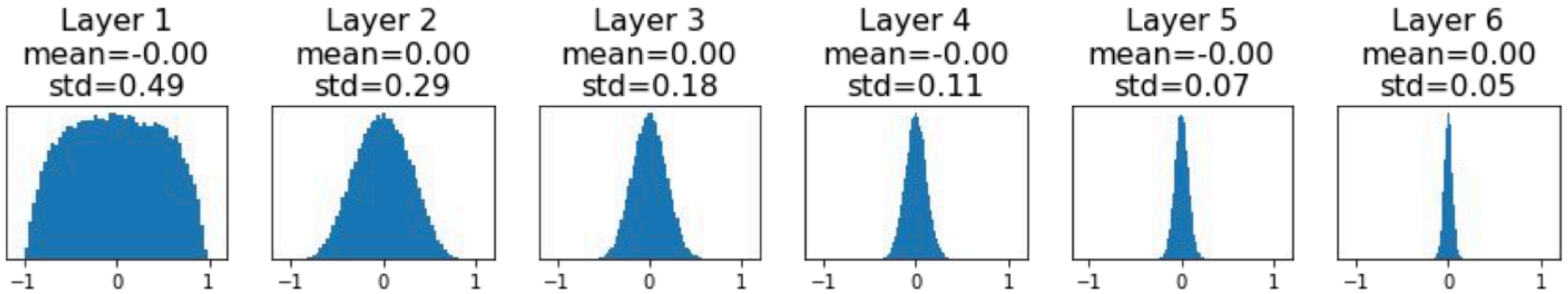
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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 =(



Slide credit: Stanford CS231N

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

What will happen to the activations for the last layer?

Slide credit: Stanford CS231N

Weight Initialization: Activation Statistics

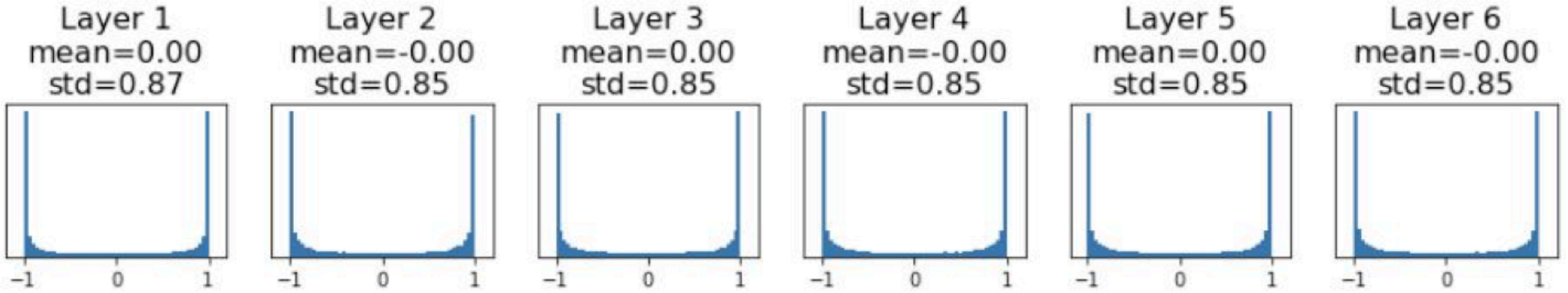
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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 =(



Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
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)

Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
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 filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2)= \ldots=Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$$\begin{aligned} Var(y) &= Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}) \\ &= Din\ Var(x_iw_i) \\ &= Din\ Var(x_i)\ Var(w_i) \end{aligned}$$

[Assume all x_i, w_i are iid]

So, $Var(y) = Var(x_i)$ only when $Var(w_i) = 1/Din$

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

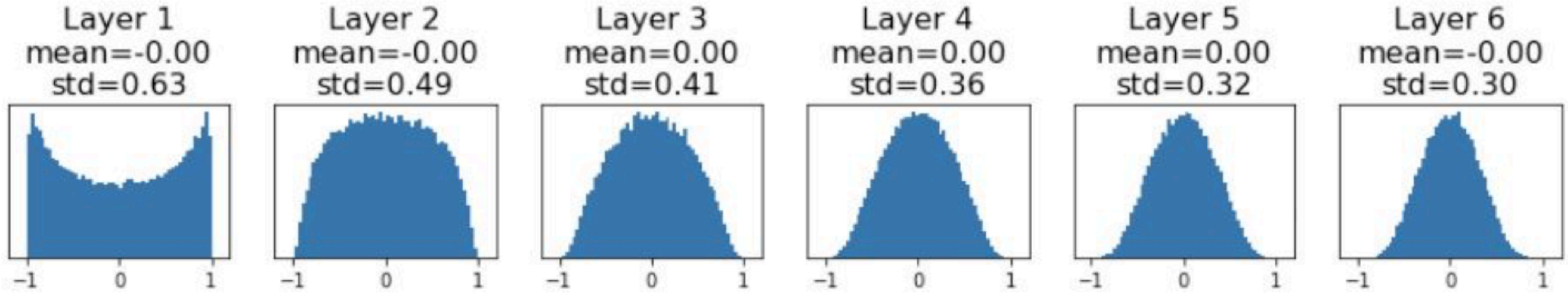
Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
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

Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
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

Slide credit: Stanford CS231N

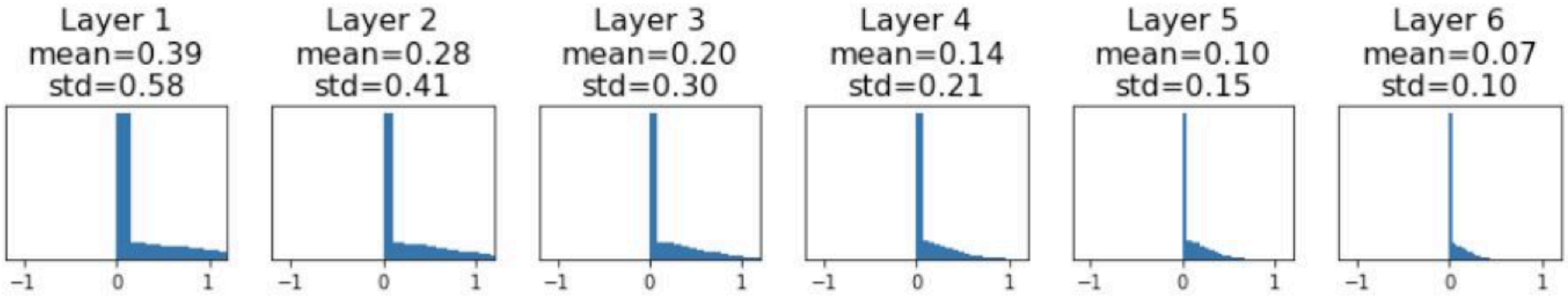
Weight Initialization: Xavier Inititalization

```
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 =(



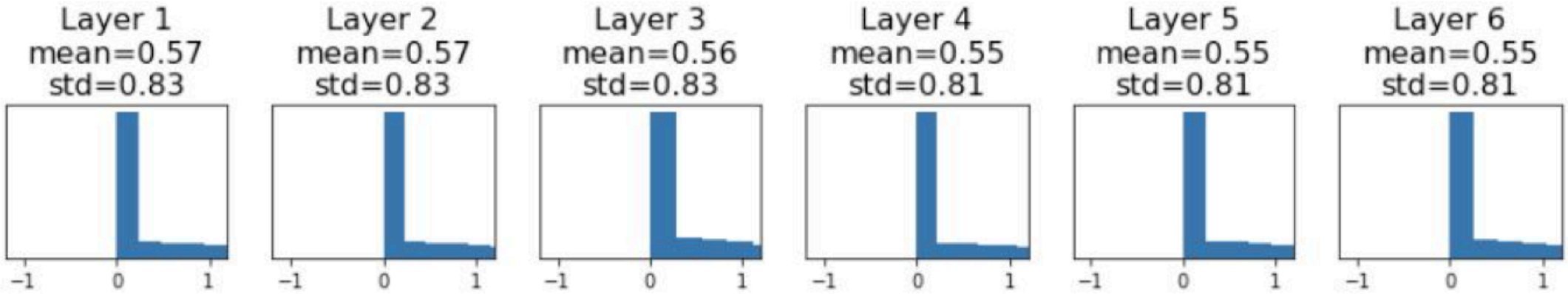
Slide credit: Stanford CS231N

Weight Initialization: He Inititalization

```
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(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

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



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

Slide credit: Stanford CS231N

Initialization is still an Active Research Area

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

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

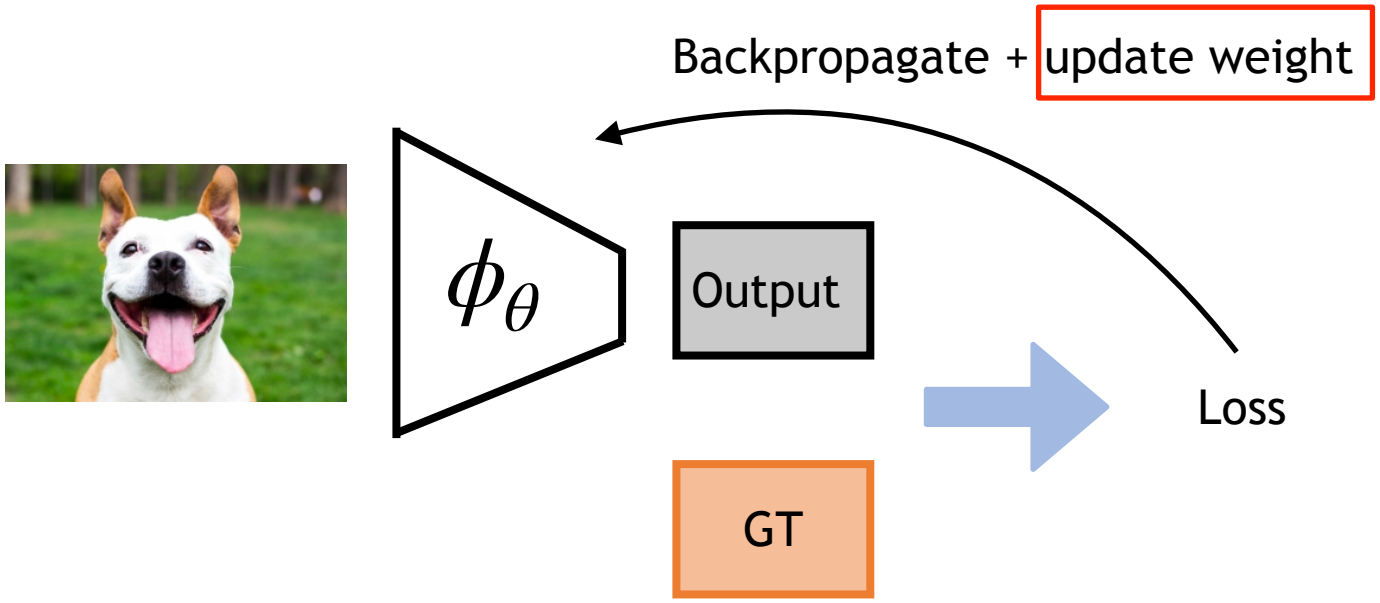
Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019

Slide credit: Stanford CS231N

Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



SGD

Update rule:

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

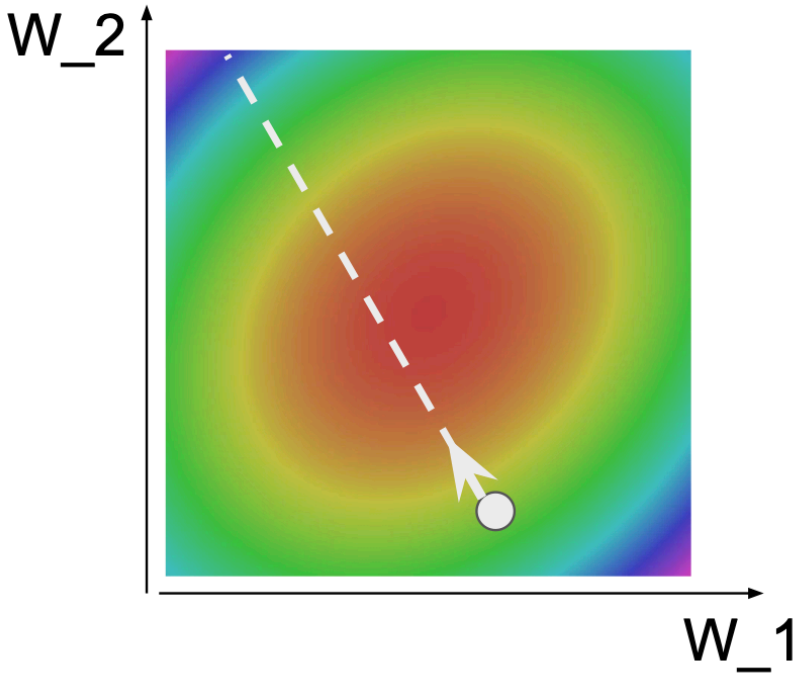
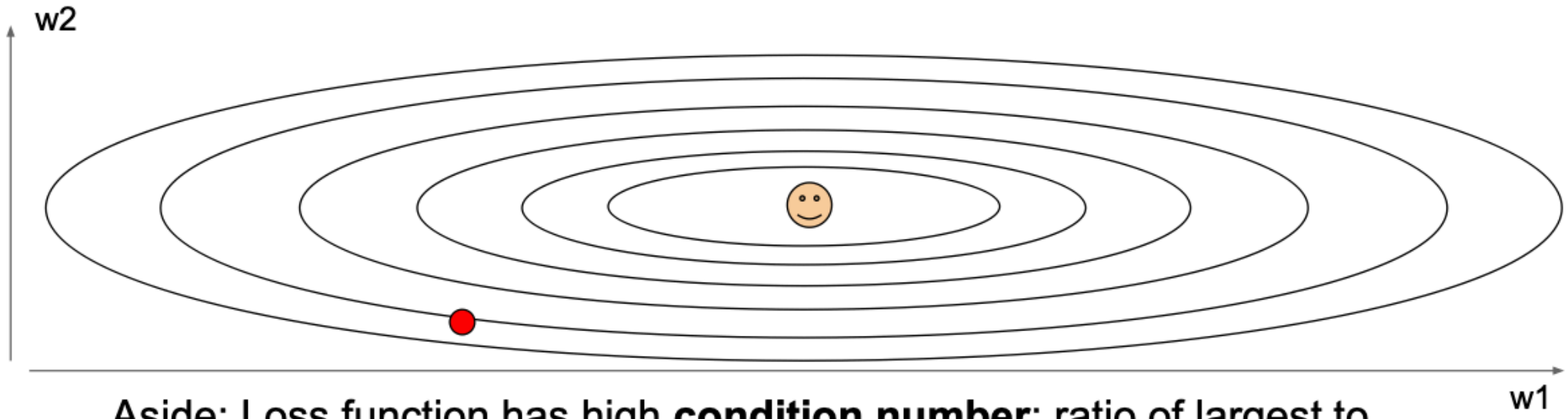


Figure credit: Stanford CS231N

Problems with SGD: #1

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?

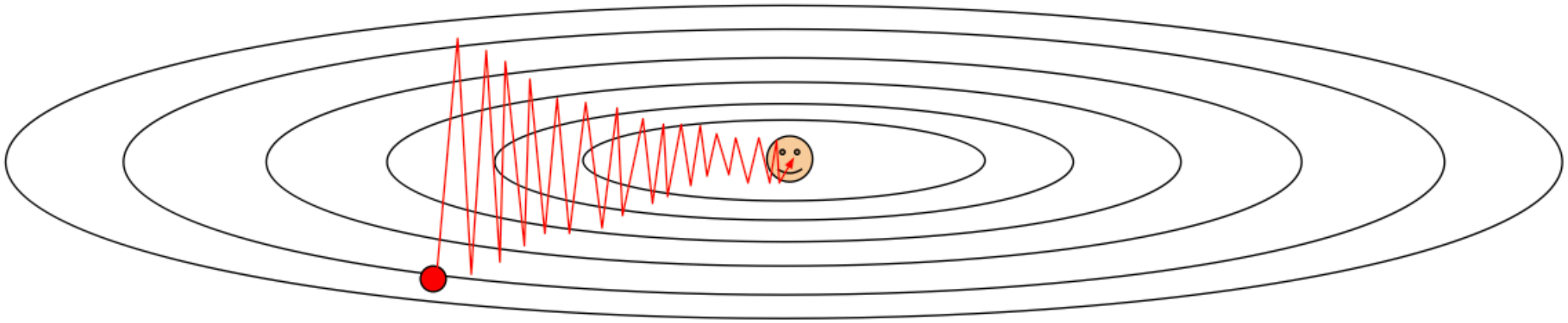


Aside: Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Slide credit: Stanford CS231N

Problems with SGD: #1

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

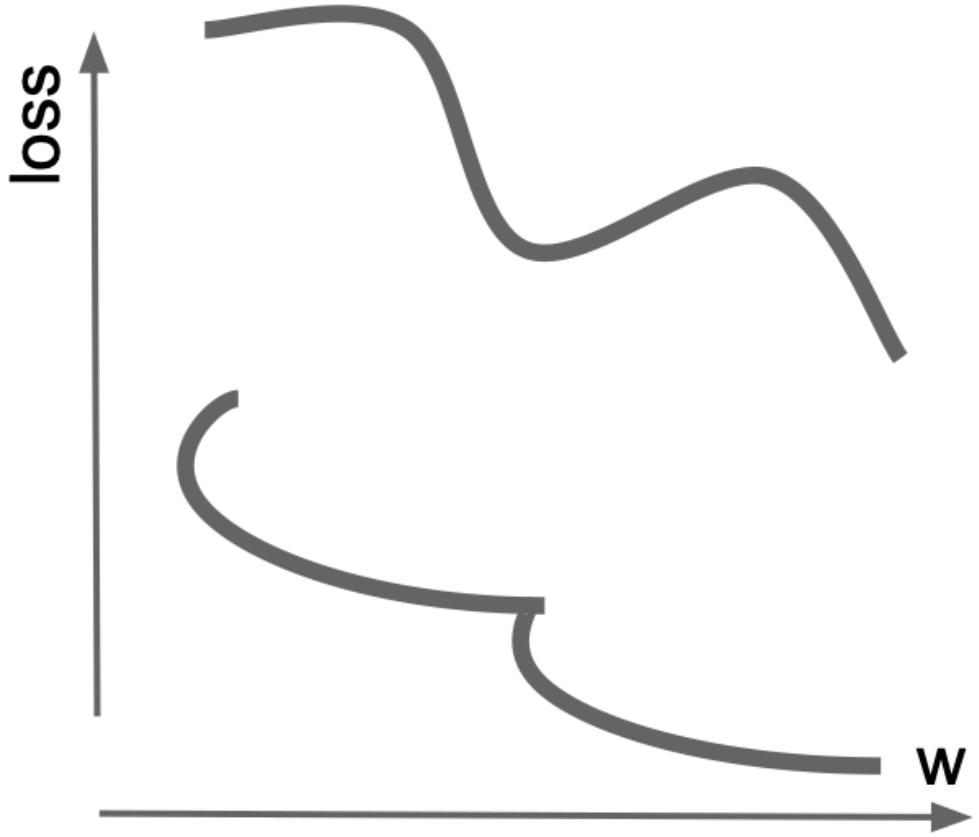


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Slide credit: Stanford CS231N

Problems with SGD: #2

What if the loss function has a **local minima** or **saddle point**?

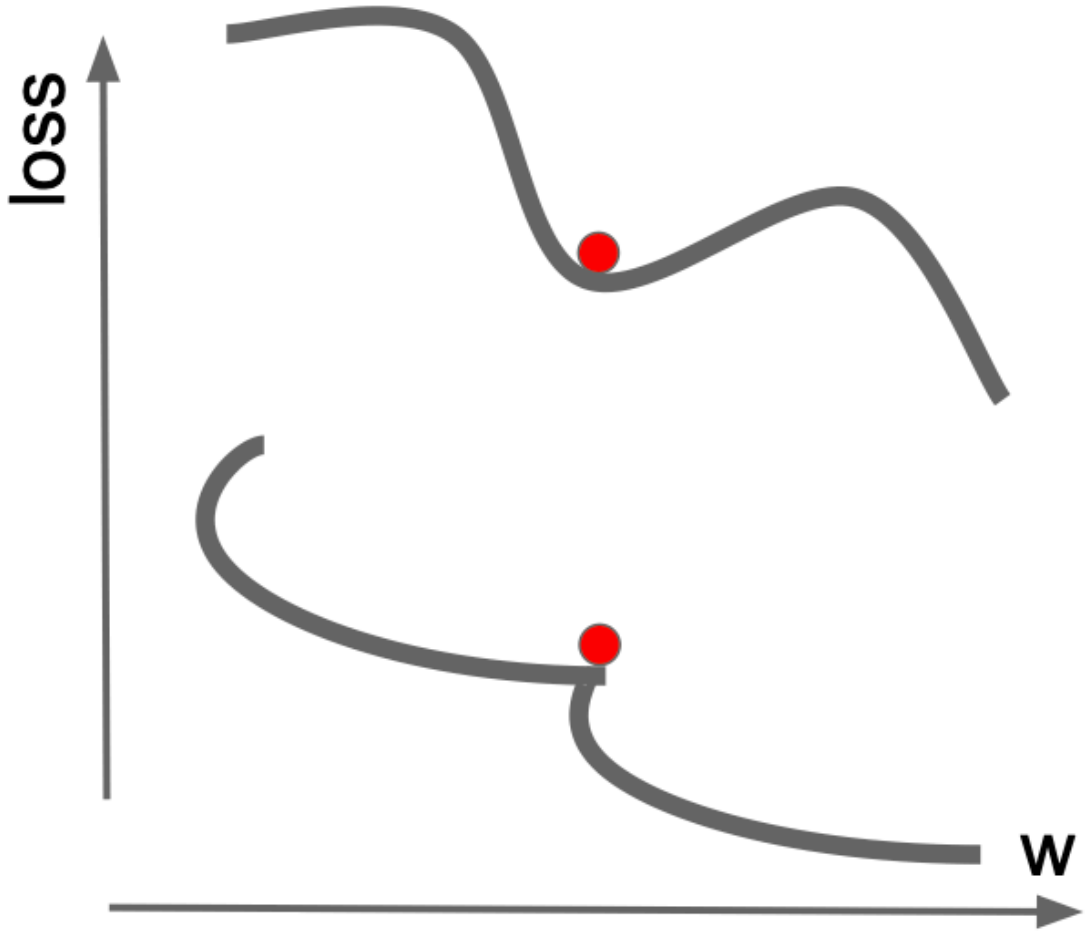


Slide credit: Stanford CS231N

Problems with SGD: #2

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck

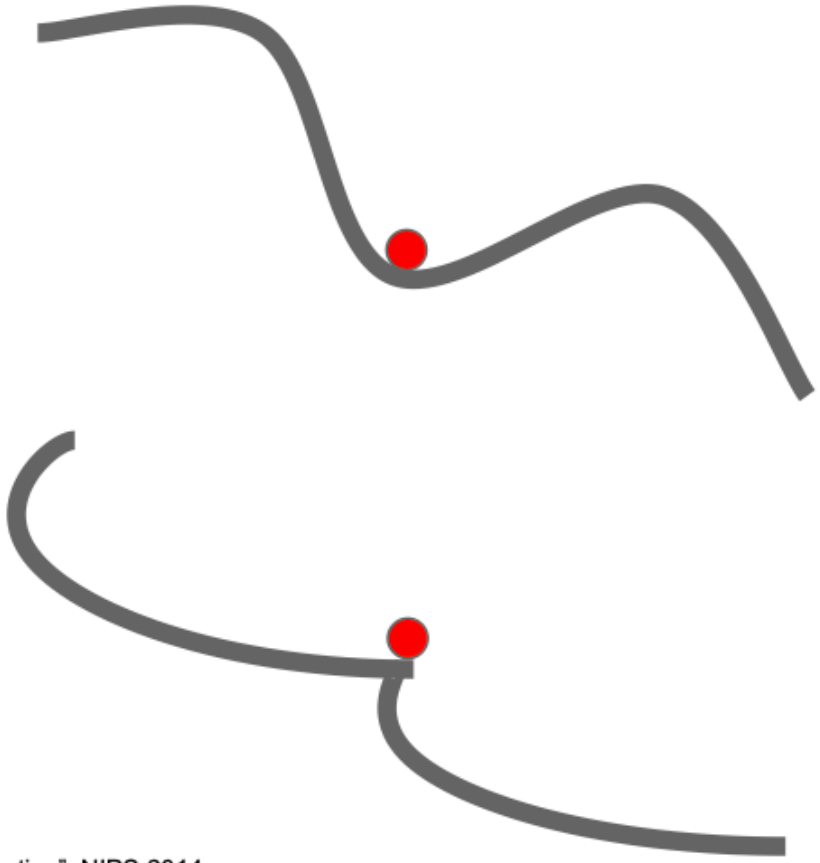


Slide credit: Stanford CS231N

Problems with SGD: #2

What if the loss function has a **local minima** or **saddle point**?

Saddle points much more common in high dimension



Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

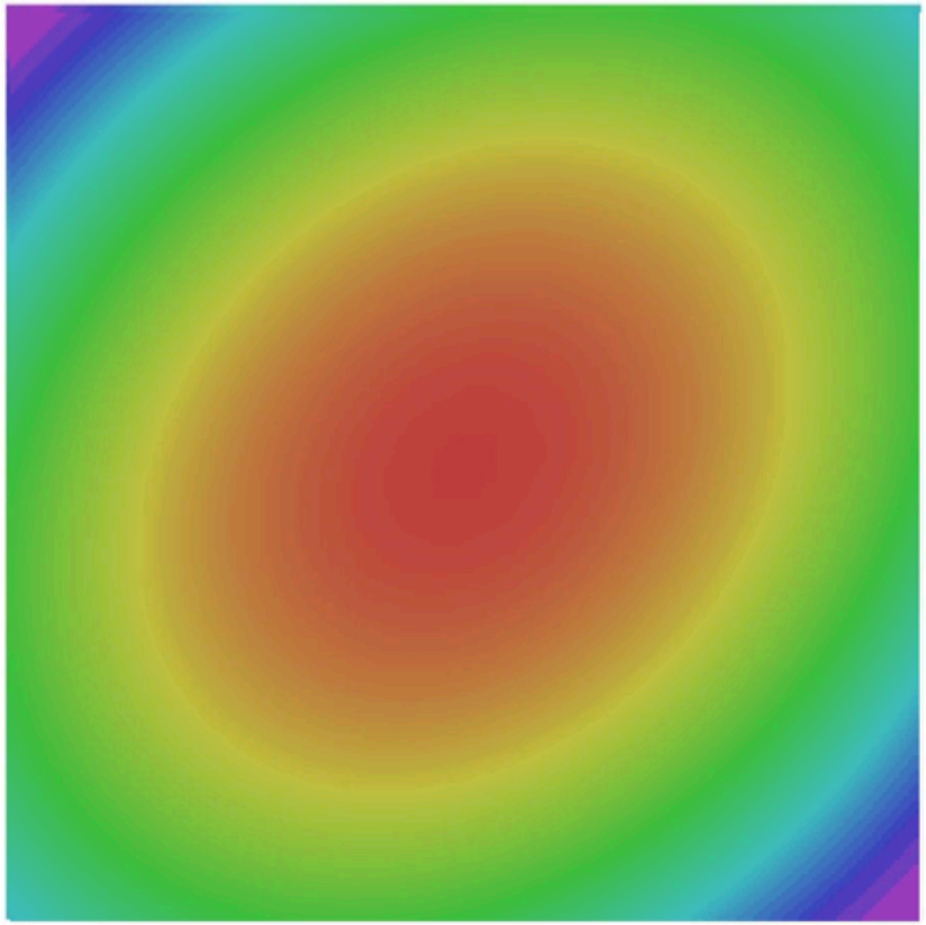
Slide credit: Stanford CS231N

Problems with SGD: #3

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Slide credit: Stanford CS231N

SGD + Momentum

continue moving in the general direction as the previous iterations

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Slide credit: Stanford CS231N

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

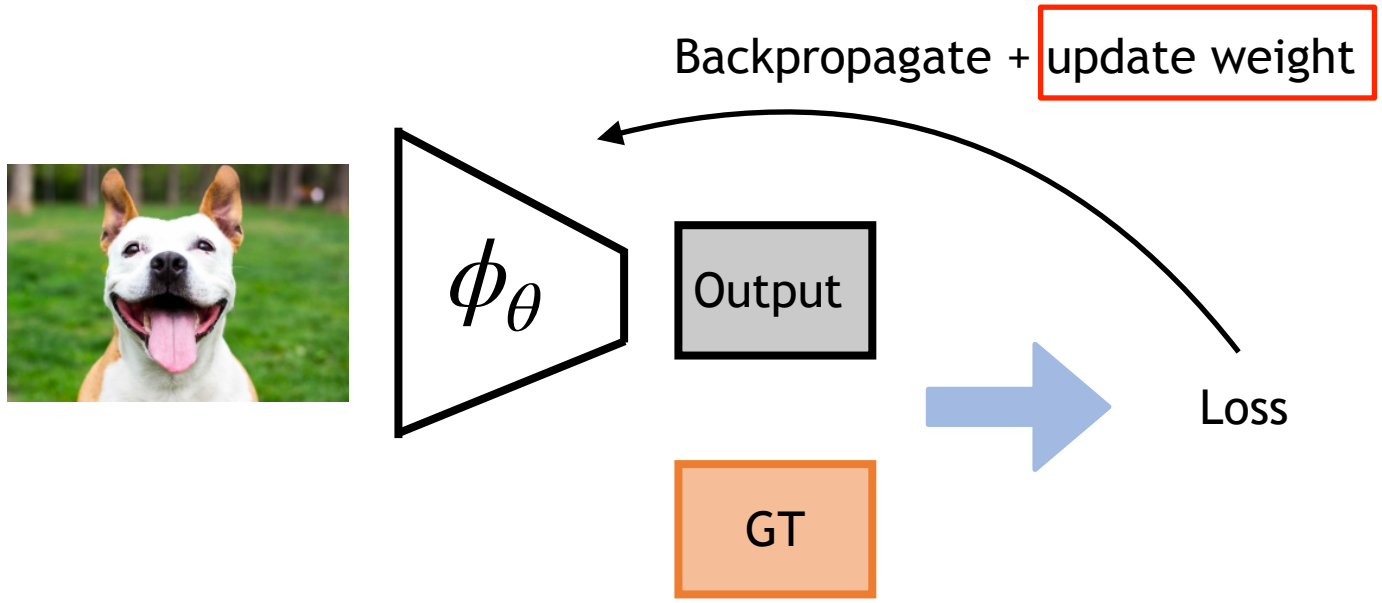
Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

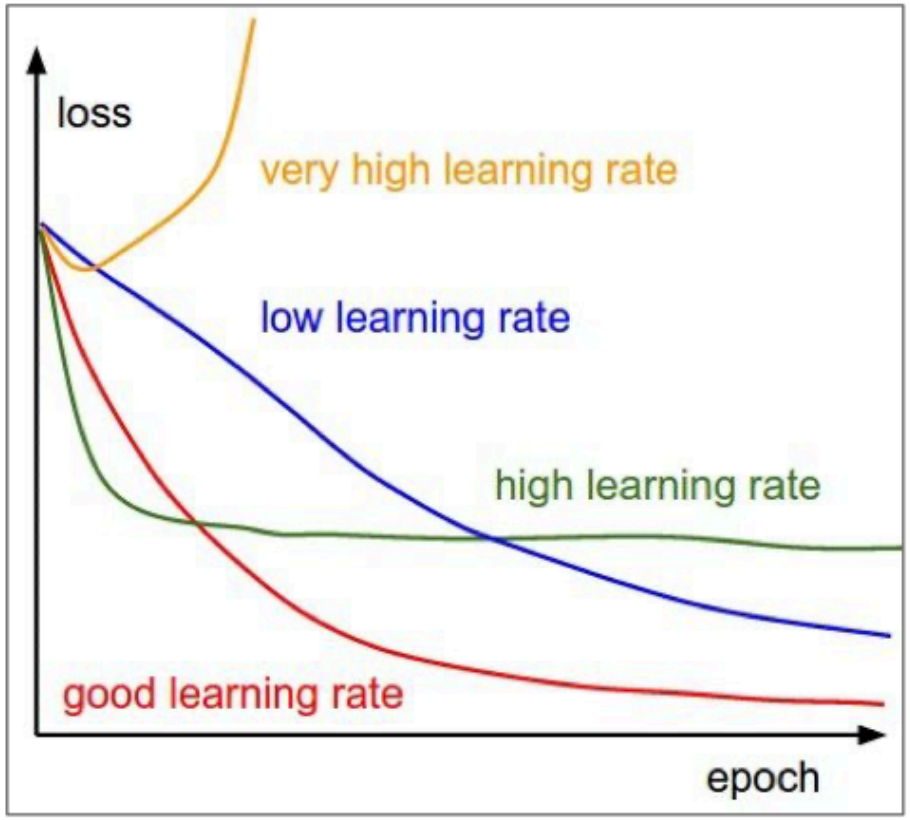
Slide credit: Stanford CS231N

Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



Loss Curves for Different Learning Rates



- An appropriate learning rate for classification: 1e-6 ~ 1e-3
- Low learning rate: undershoot
- High learning rate: overshoot

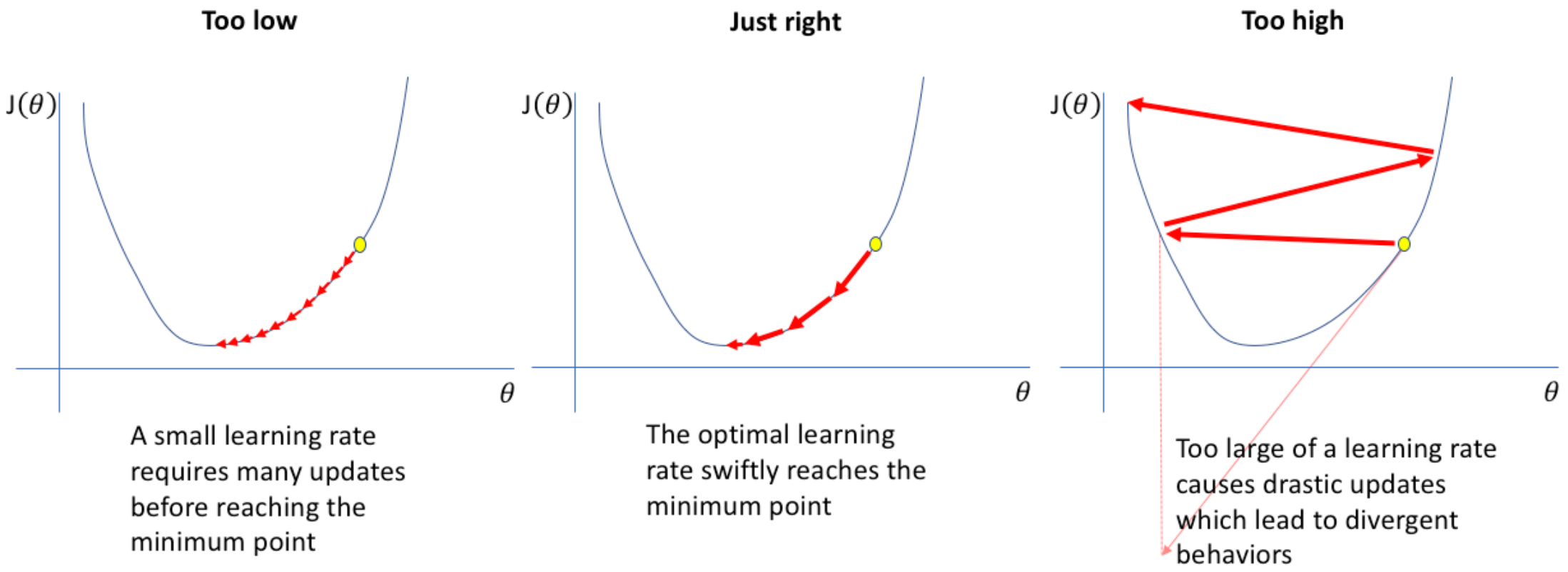
Figure credit: Stanford CS231N.

Iteration and Epoch

- Iteration:
  - One batch (whose size is called batch size)
  - A gradient descent step

- Epoch
  - Contains many iterations that go over the training data for one complete pass
  - After a epoch, plot train curve, evaluate on val, save model…

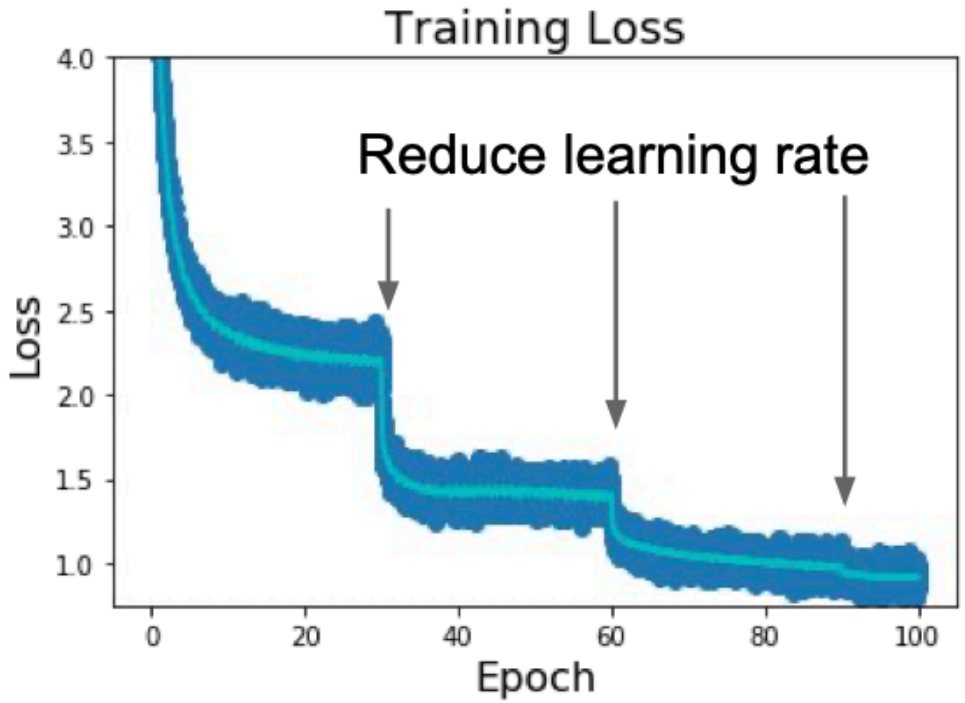
Learning Rate



https://www.jeremyjordan.me/content/images/2018/02/Screen-Shot-2018-02-24-at-11.47.09-AM.png

Learning Rate Schedule

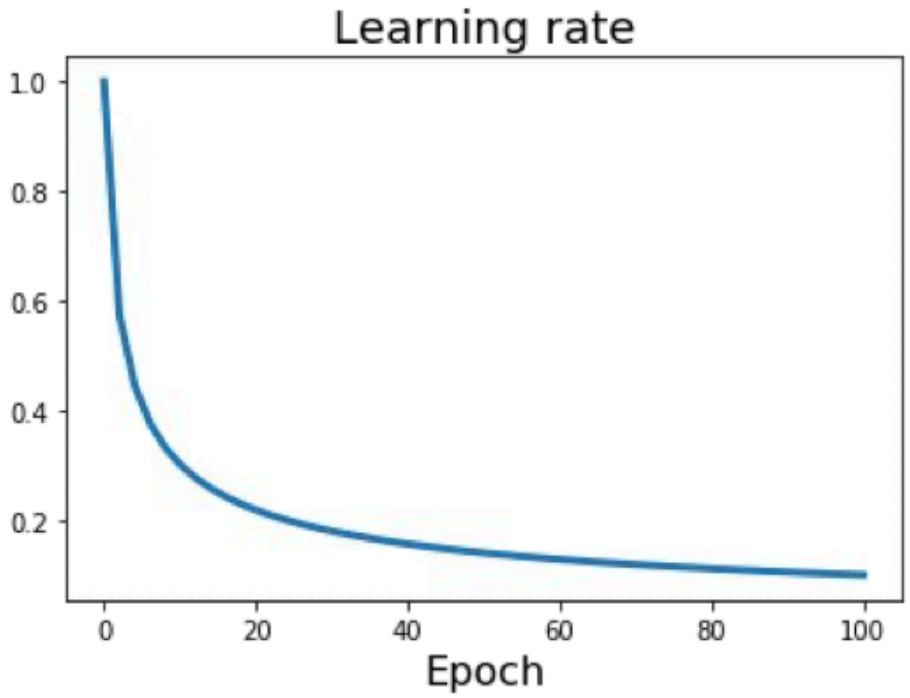
Idea: high learning rate at the beginning, decay it later



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Slide credit: Stanford CS231N

Learning Rate Schedule



Vaswani et al, "Attention is all you need", NIPS 2017

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

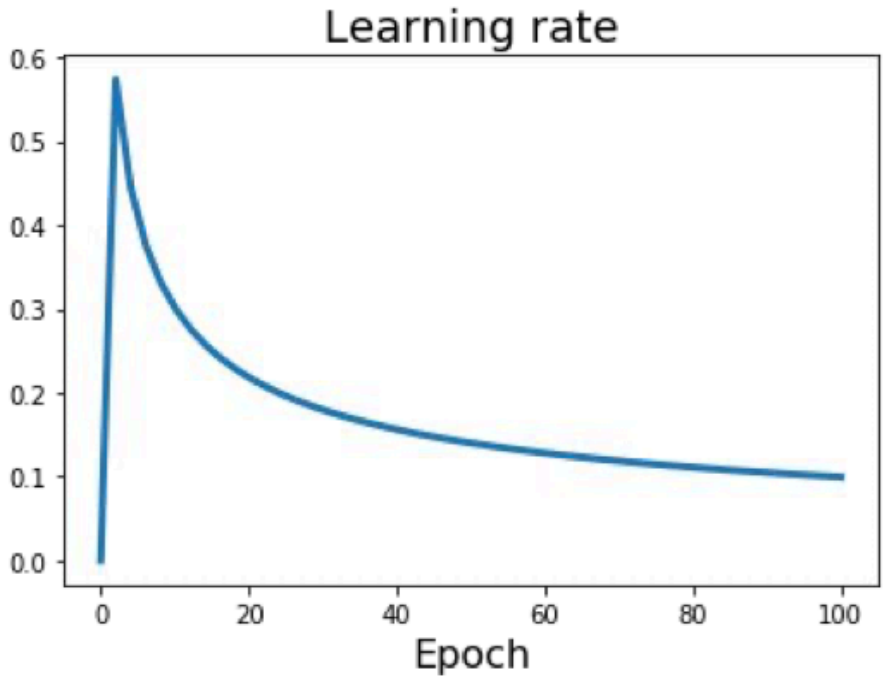
Cosine: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

Linear: $\alpha_t = \alpha_0(1 - t/T)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Learning Rate Schedule: Linear Warmup



High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

Batch Size and Learning Rate

- An empirical rule of thumb: if you increase the batch size by N, also scale the initial learning rate by N.

- Why? Suggested reading: visualizing learning rate vs. batch size.

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

Summary of Learning Rate Schedule

- **Adam** is a good default choices working okay with constant learning rate.
- **SGD + Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule: very few hyper parameters.
- If you are new to a dataset, use Adam with constant learning rate until you see it converges and then modify the learning rate.

Introduction to Computer Vision

Next Week: Lecture 5,
Deep Learning III and 3D Vision I