Deep Learning for Graphics

Neural Network Basics

Niloy Mitra	Iasonas Kokkinos	Paul Guerrero	Vladimir Kim	Kostas Rematas	Tobias Ritschel
UCL	UCL/Facebook	UCL	Adobe Research	U Washington	UCL

Timetable

	Niloy	Iasonas	Paul	Vova	Kostas	Tobias	Ersin
Introduction	X	X	X			X	
Theory	X						
NN Basics		X	X			X	
Supervised Applications							
Data						X	
Unsupervised Applications			X				
Beyond 2D	X		X	X			X
Outlook	X	X	X	X	X	X	X

Introduction to Neural Networks

Goal: Learn a Parametric Function

$$f_\theta : \mathbb{X} \longrightarrow \mathbb{Y}$$

θ : function parameters, $\mathbb{X}$: source domain $\mathbb{Y}$: target domain
these are learned

Examples:

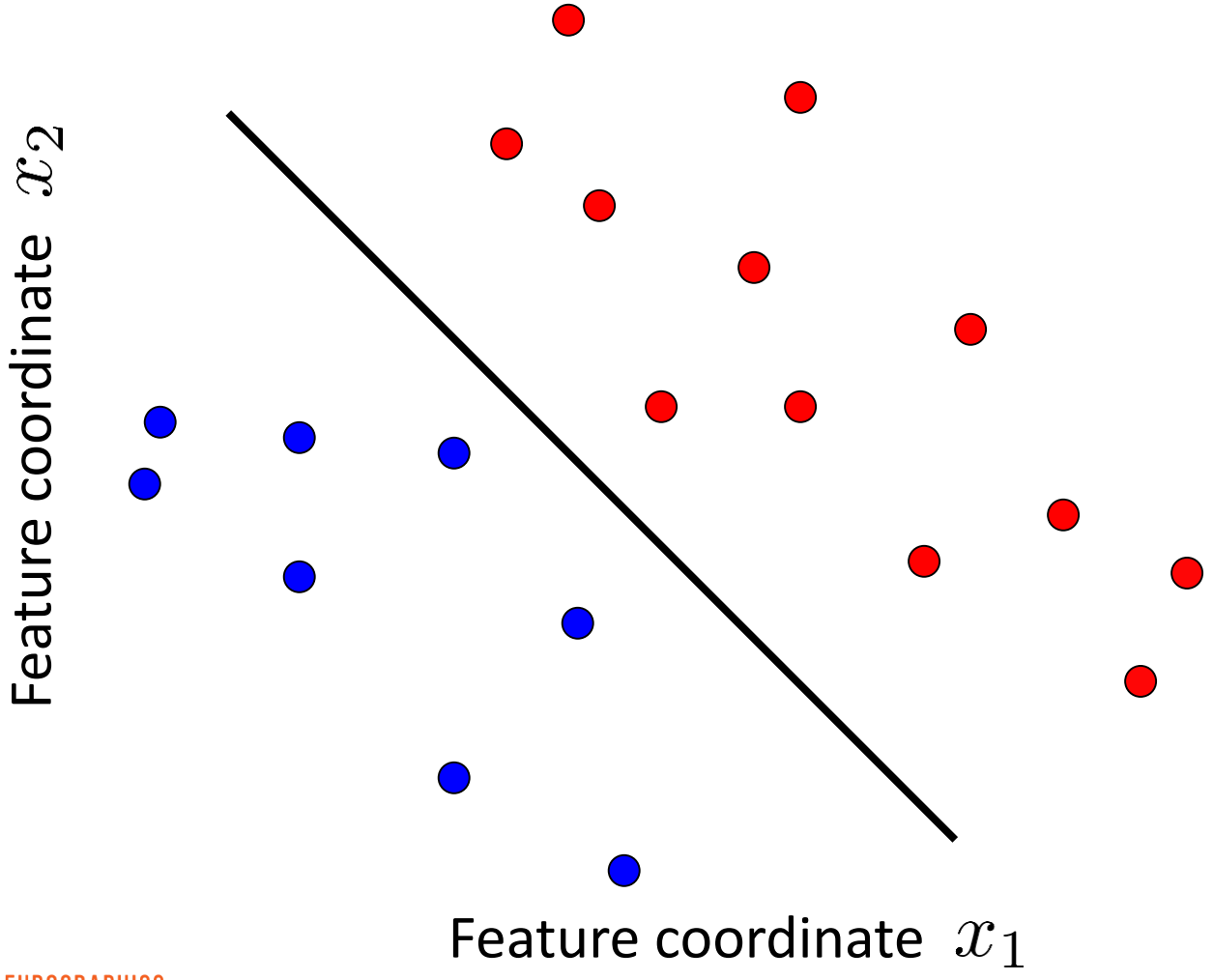
Image Classification: $f_\theta : \mathbb{R}^{w \times h \times c} \longrightarrow \{0, 1, \ldots, k-1\}$

$w \times h \times c$: image dimensions k : class count

Image Synthesis: $f_\theta : \mathbb{R}^n \longrightarrow \mathbb{R}^{w \times h \times c}$

n : latent variable count $w \times h \times c$: image dimensions

Machine Learning 101: Linear Classifier



$$f_\theta : \mathbb{R}^n \longrightarrow \{0, 1\}$$

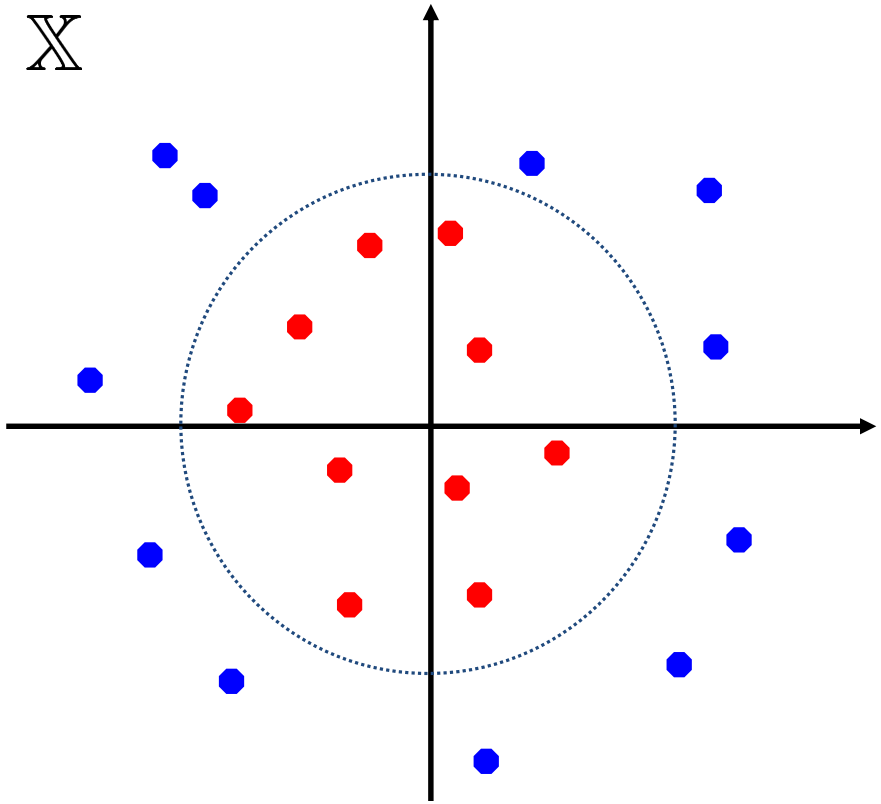
$$f_\theta(x) = \begin{cases} 1 & \text{if } wx + b \geq 0 \\ 0 & \text{if } wx + b < 0 \end{cases}$$

$$\theta = \{w, b\}$$

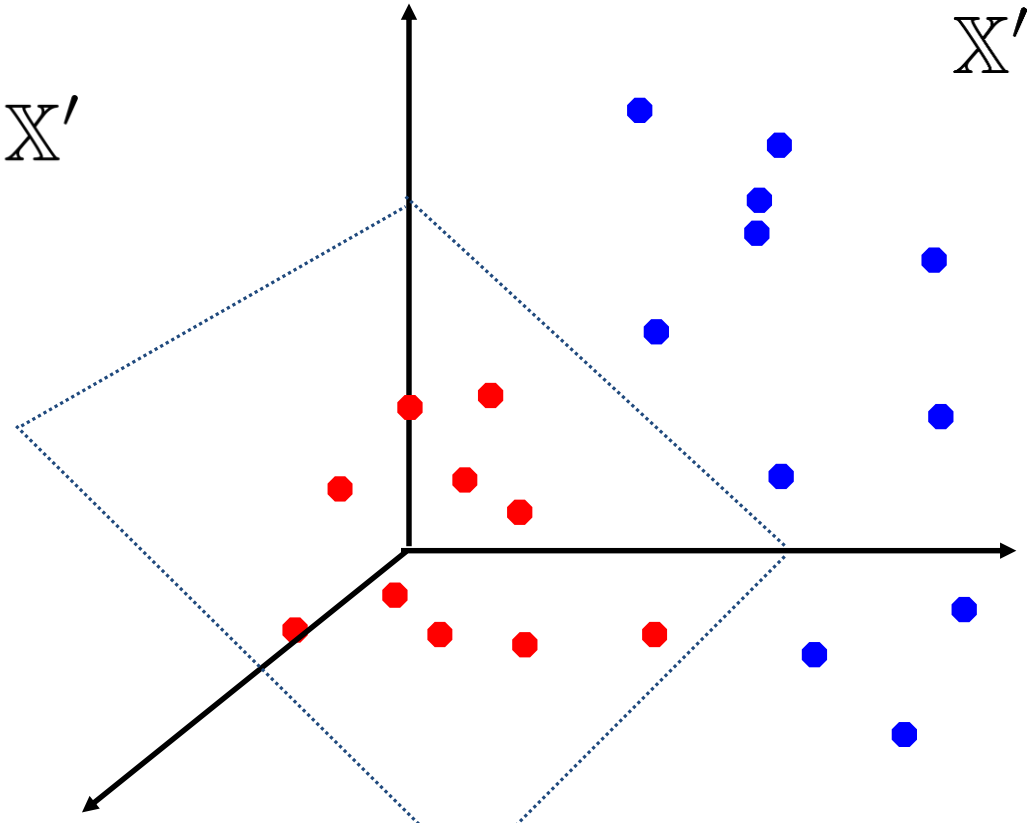
Each data point has a class label:

$$y^i = \begin{cases} 1 & (\bullet) \\ 0 & (\bullet) \end{cases}$$

Nonlinear decision boundaries



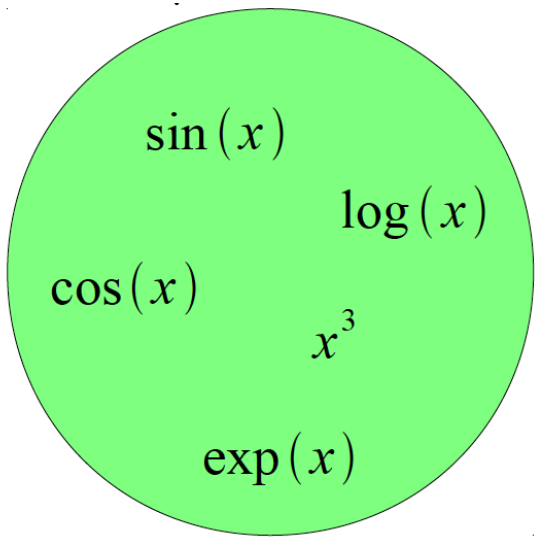
$$g : \mathbb{X} \longrightarrow \mathbb{X}'$$



$$f_\theta(x) = \begin{cases} 1 & \text{if } w\ g(x) + b \geq 0 \\ 0 & \text{if } w\ g(x) + b < 0 \end{cases}$$

Building A Complicated Function

Given a library of simple functions



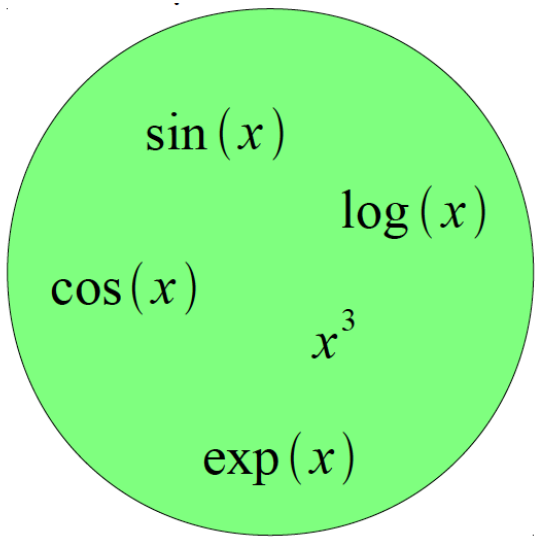
Compose into a

complicated function

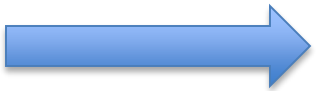
Slide Credit: Marc'Aurelio Ranzato, Yann LeCun

Building A Complicated Function

Given a library of simple functions



Compose into a

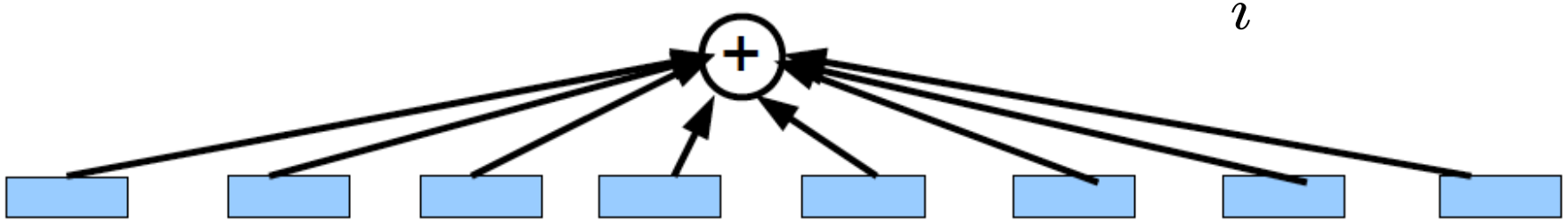


complicated function

Idea 1: Linear Combinations

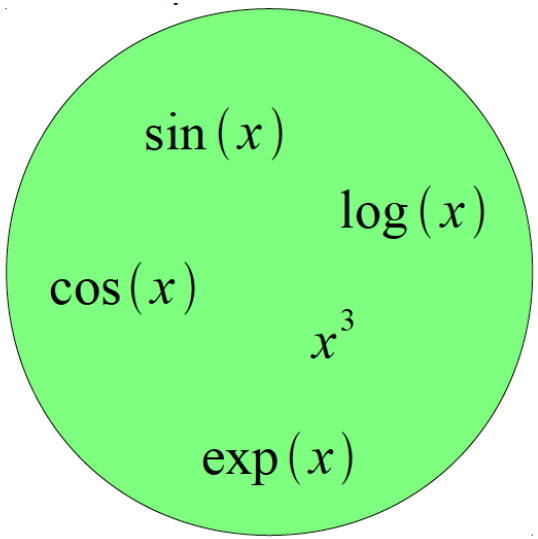
- Boosting
- Kernels
- …

$$f(x) = \sum_i \alpha_i g_i(x)$$



Building A Complicated Function

Given a library of simple functions



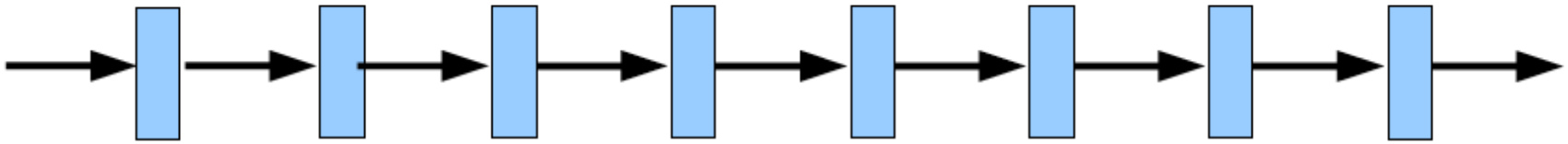
Compose into a

complicated function

Idea 2: Compositions

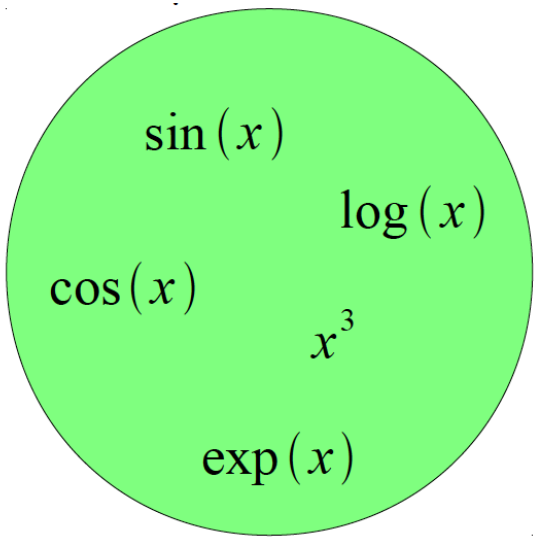
- Decision Trees
- Deep Learning

$$f(x) = g_1(g_2(\ldots(g_n(x)\ldots))$$

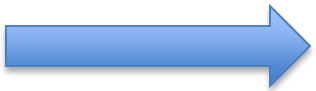


Building A Complicated Function

Given a library of simple functions



Compose into a

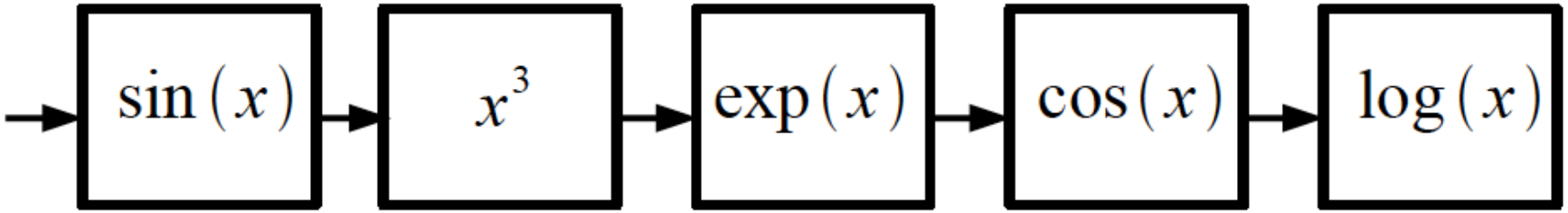


complicated function

Idea 2: Compositions

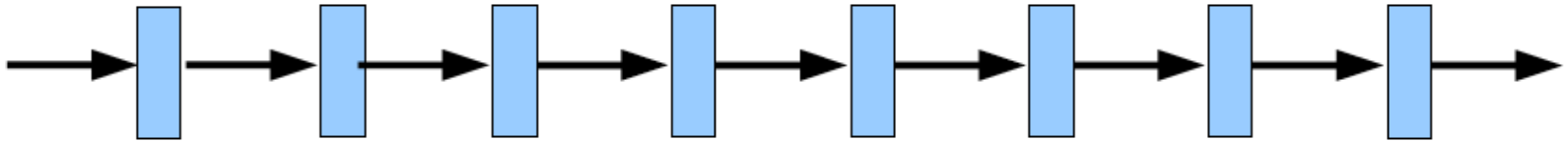
- Decision Trees
- Grammar models
- Deep Learning

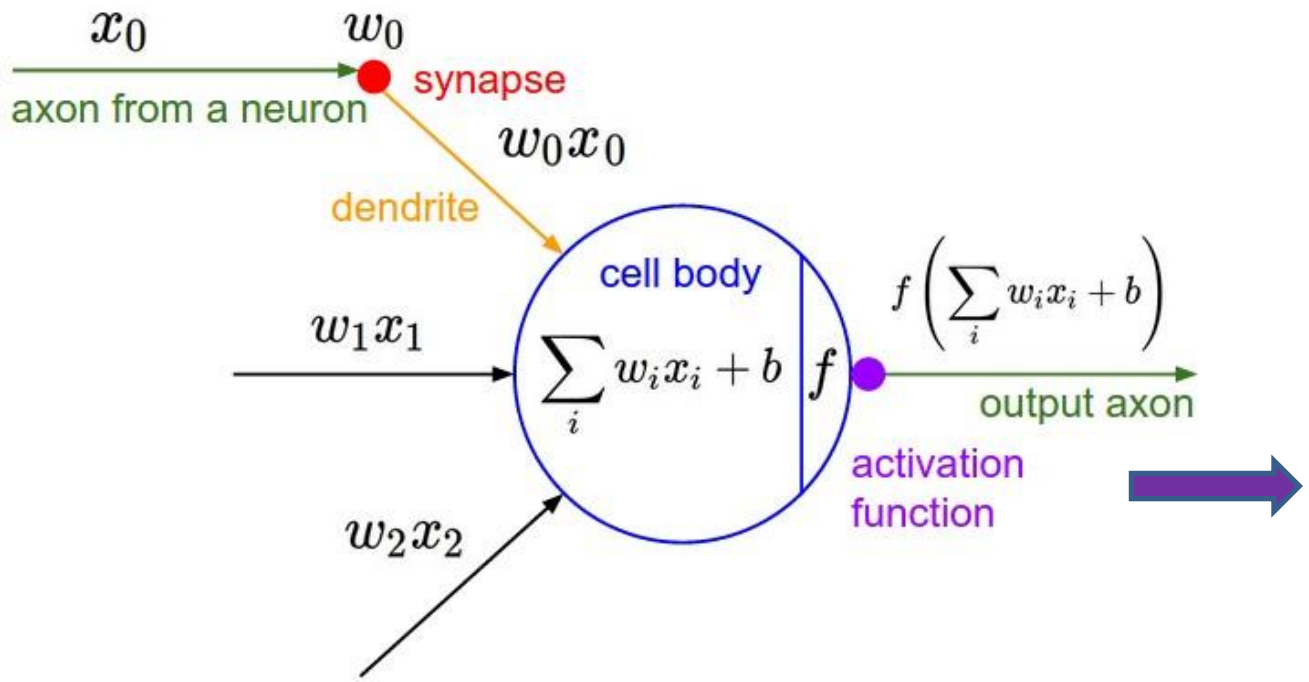
$$f(x) = \log(\cos(\exp(\sin^3(x))))$$



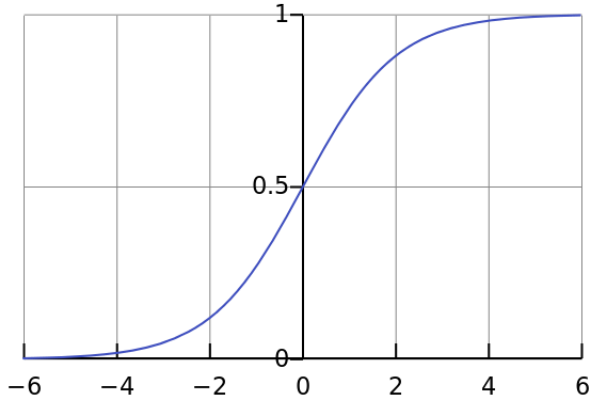
‘Neuron’: Cascade of Linear and Nonlinear Function

basic building block

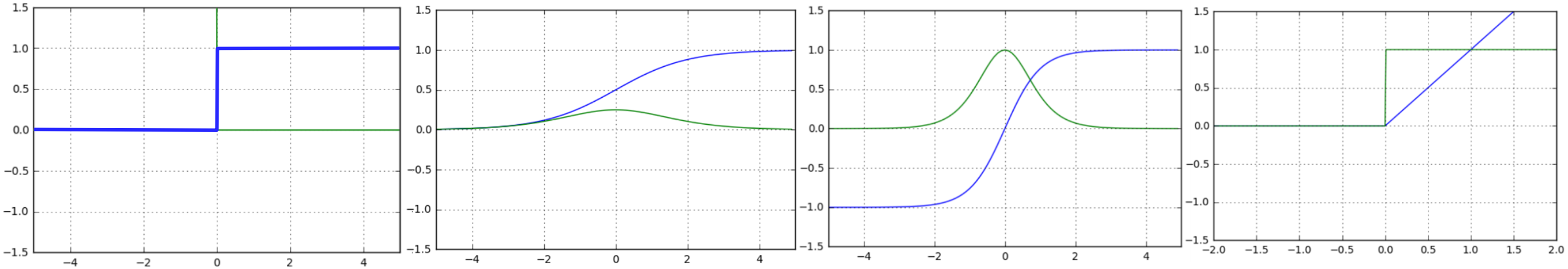




Sigmoidal activation



Activation functions



Step (“perceptron”)

$$g(a) = \begin{cases} 0 & a < 0 \\ 1 & a \geq 0 \end{cases}$$

Sigmoidal (“logistic”)

$$g(a) = \frac{1}{1 + \exp(-a)}$$

Hyperbolic tangent

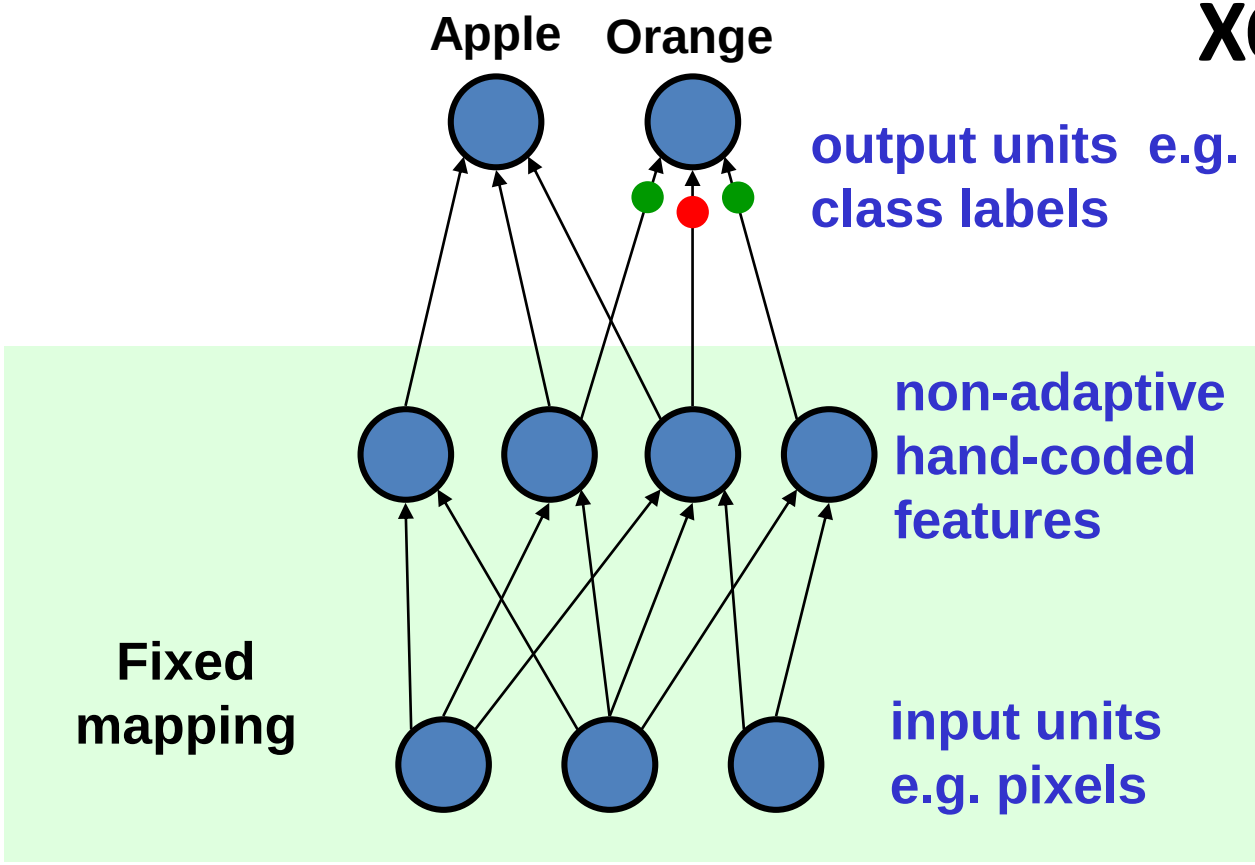
$$g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$

Rectified Linear Unit (RELU)

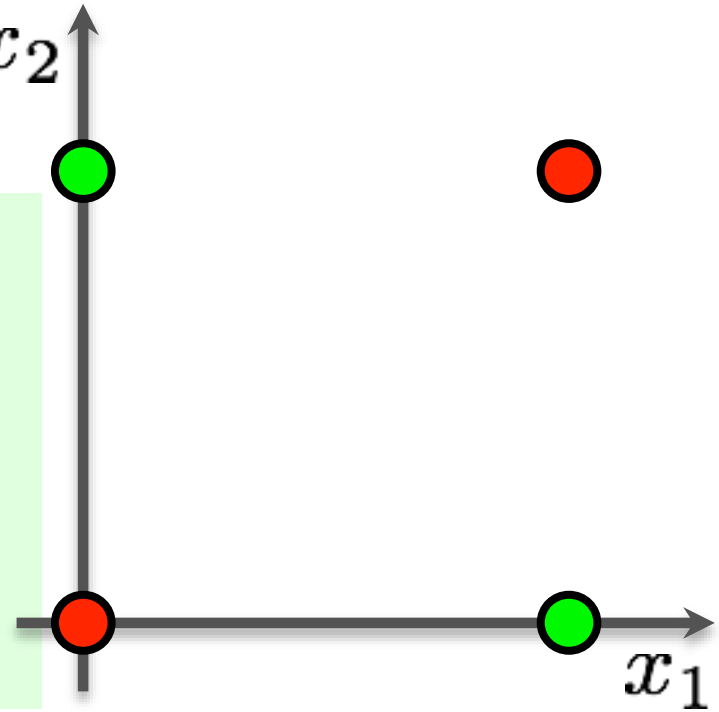
$$g(a) = \max(0, a)$$

Image Credit: Olivier Grisel and Charles Ollion

Perceptrons (60's)



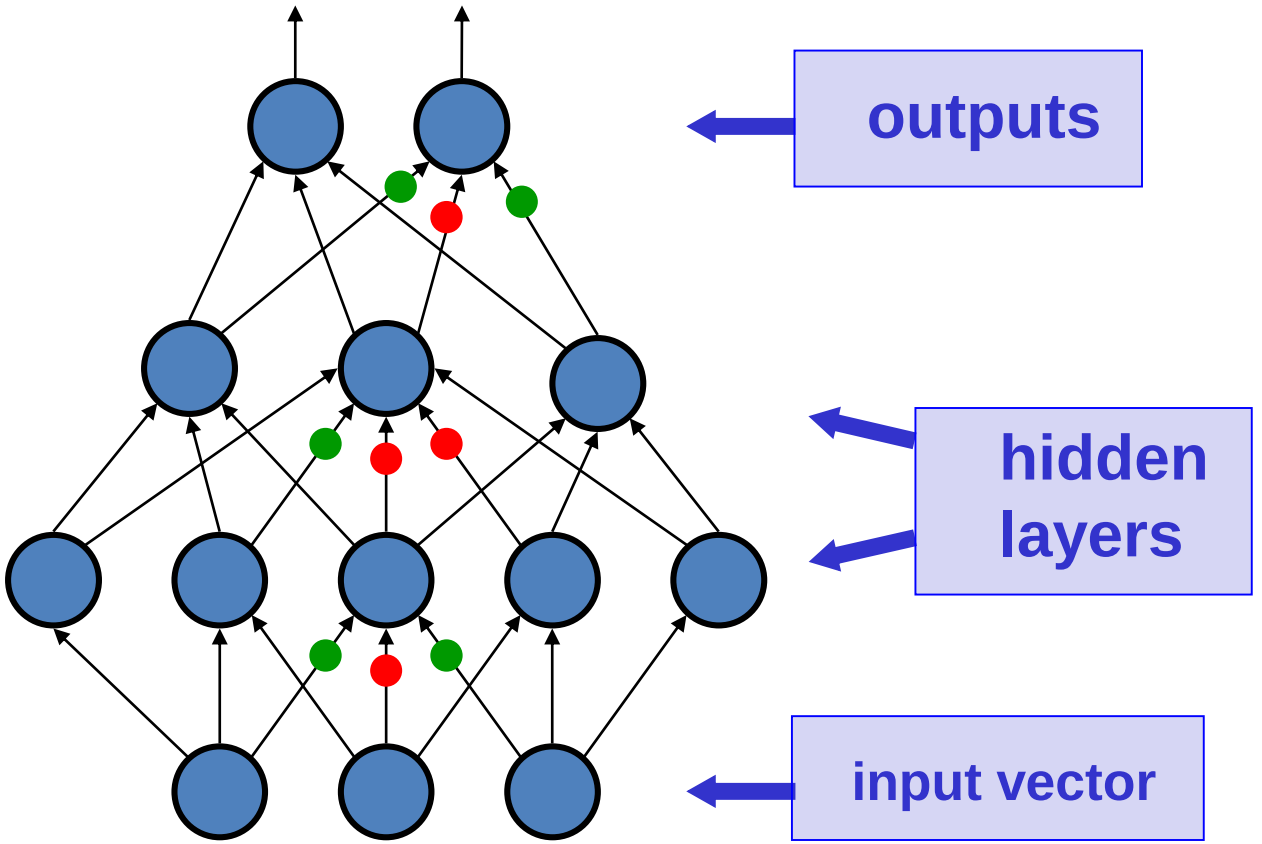
XOR: perceptron killer



Slide credit: G. Hinton

Multi-Layer Perceptrons (~1985)

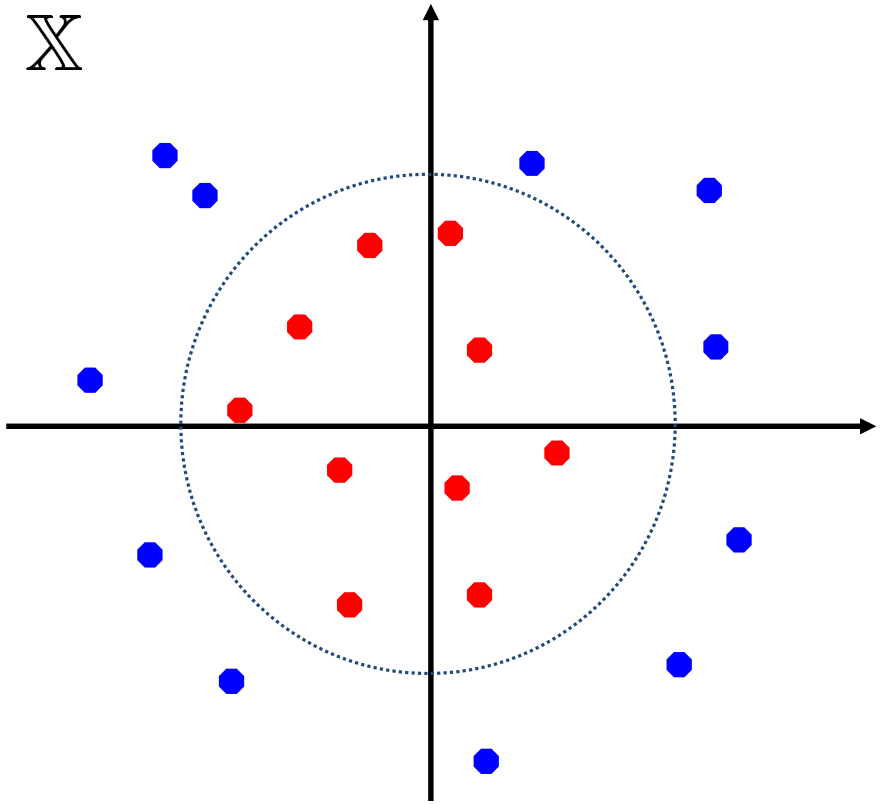
$$u_i = g\left(\sum_{k\in\mathcal{N}(i)} w_{k,i}\, g\left(\sum_{m\in\mathcal{N}(k)} w_{m,k}u_m + b_k\right) + b_i\right)$$



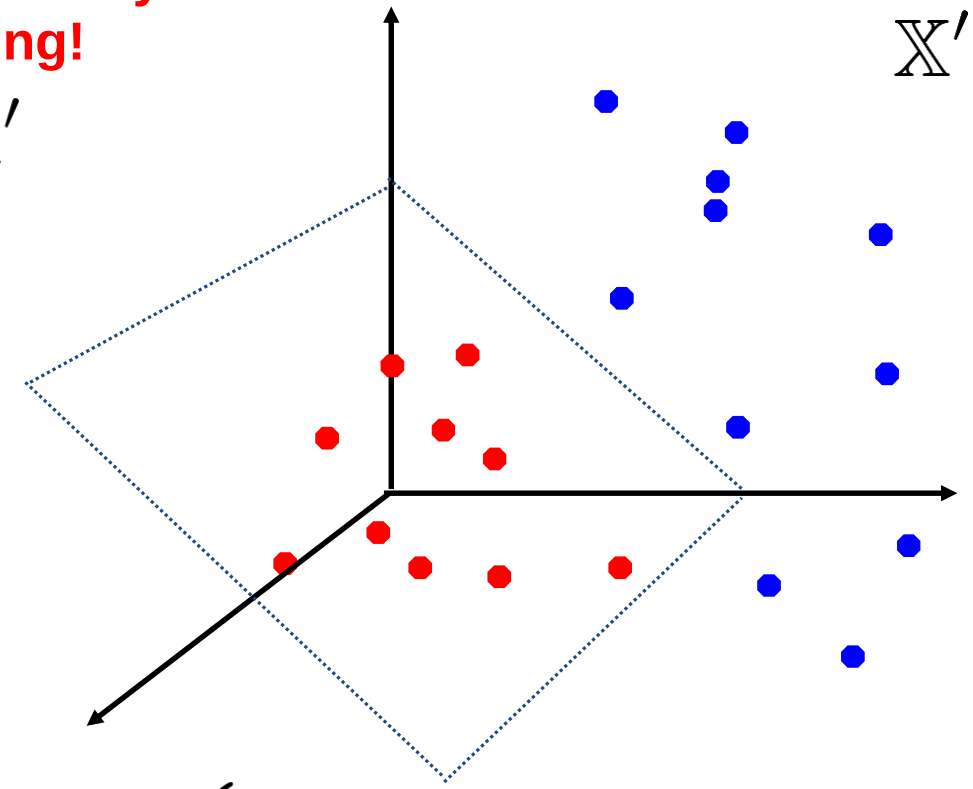
Slide credit: G. Hinton

Reminder: Non-linear decision boundaries

This is what the hidden layers should be doing!



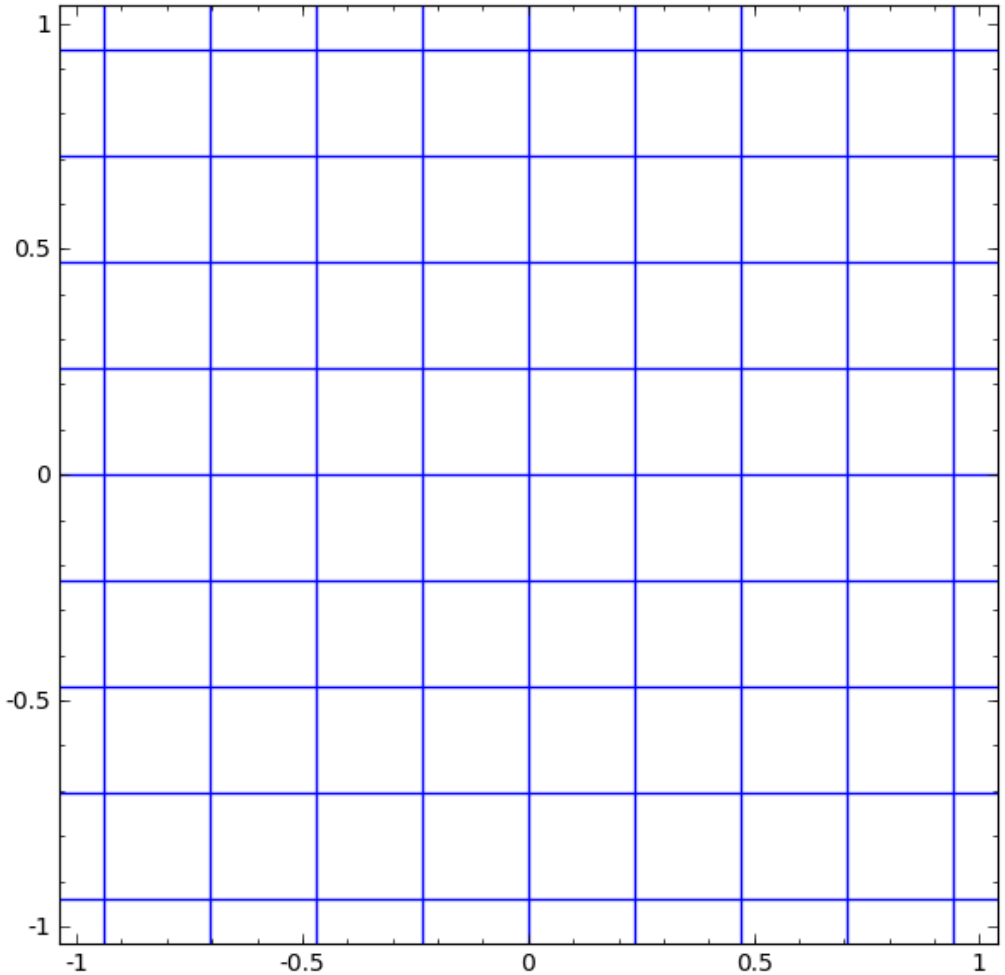
$$g : \mathbb{X} \longrightarrow \mathbb{X}'$$



$$f_\theta(x) = \begin{cases} 1 & \text{if } w\ g(x) + b \geq 0 \\ 0 & \text{if } w\ g(x) + b < 0 \end{cases}$$

Nonlinear mapping $\mathbb{R}^2 \rightarrow \mathbb{R}^2$

Evolution of isocontours as parameters change



$$y_1 = g(w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3})$$
$$y_2 = g(w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3})$$

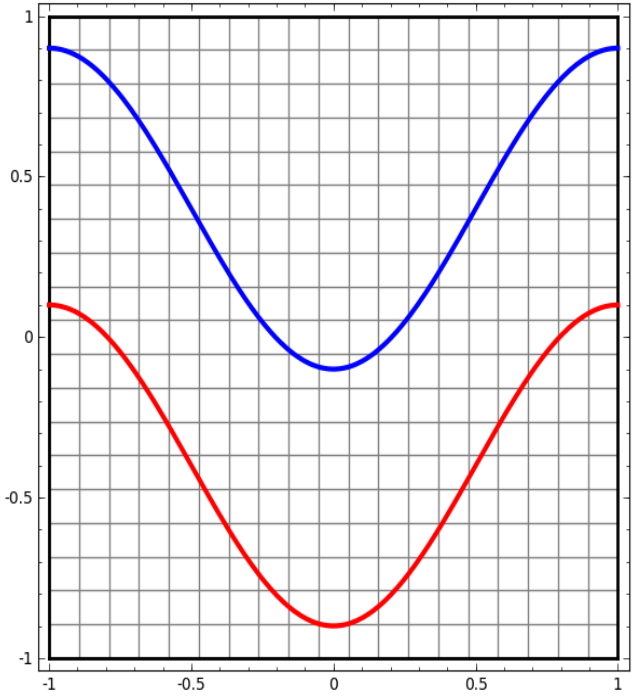
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

$$\mathbf{y} = g(\mathbf{W}\mathbf{x})$$

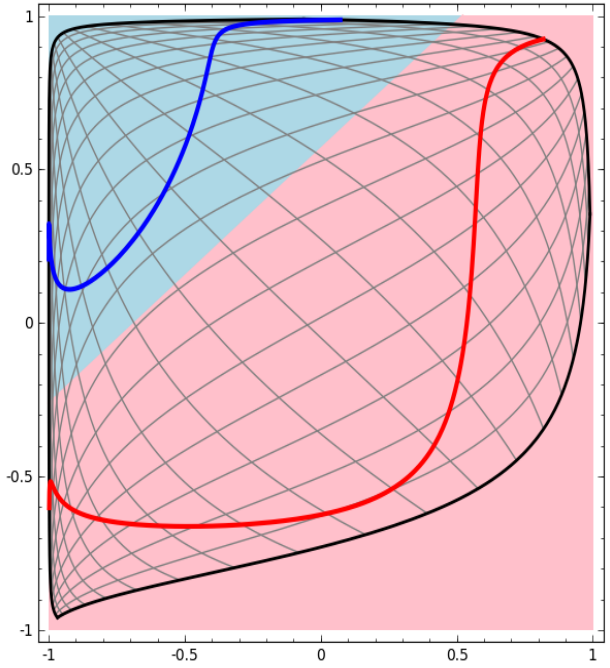
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

From non-separable to linearly separable

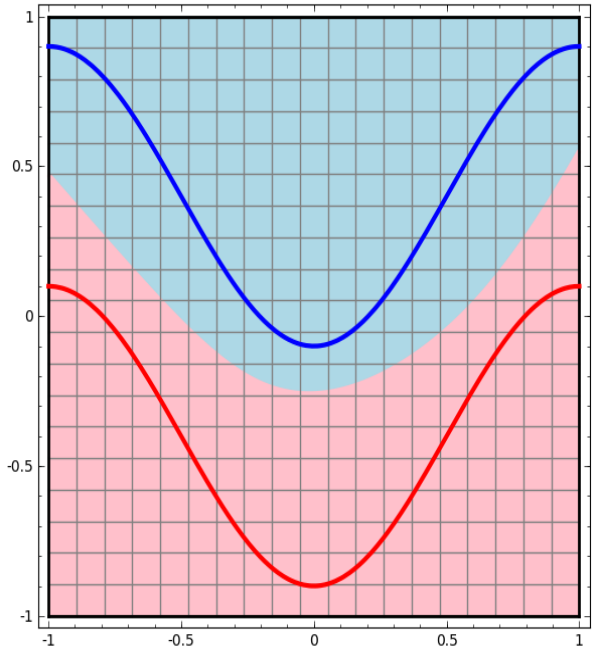
Non-linearly separable data



Data mapped to learned space

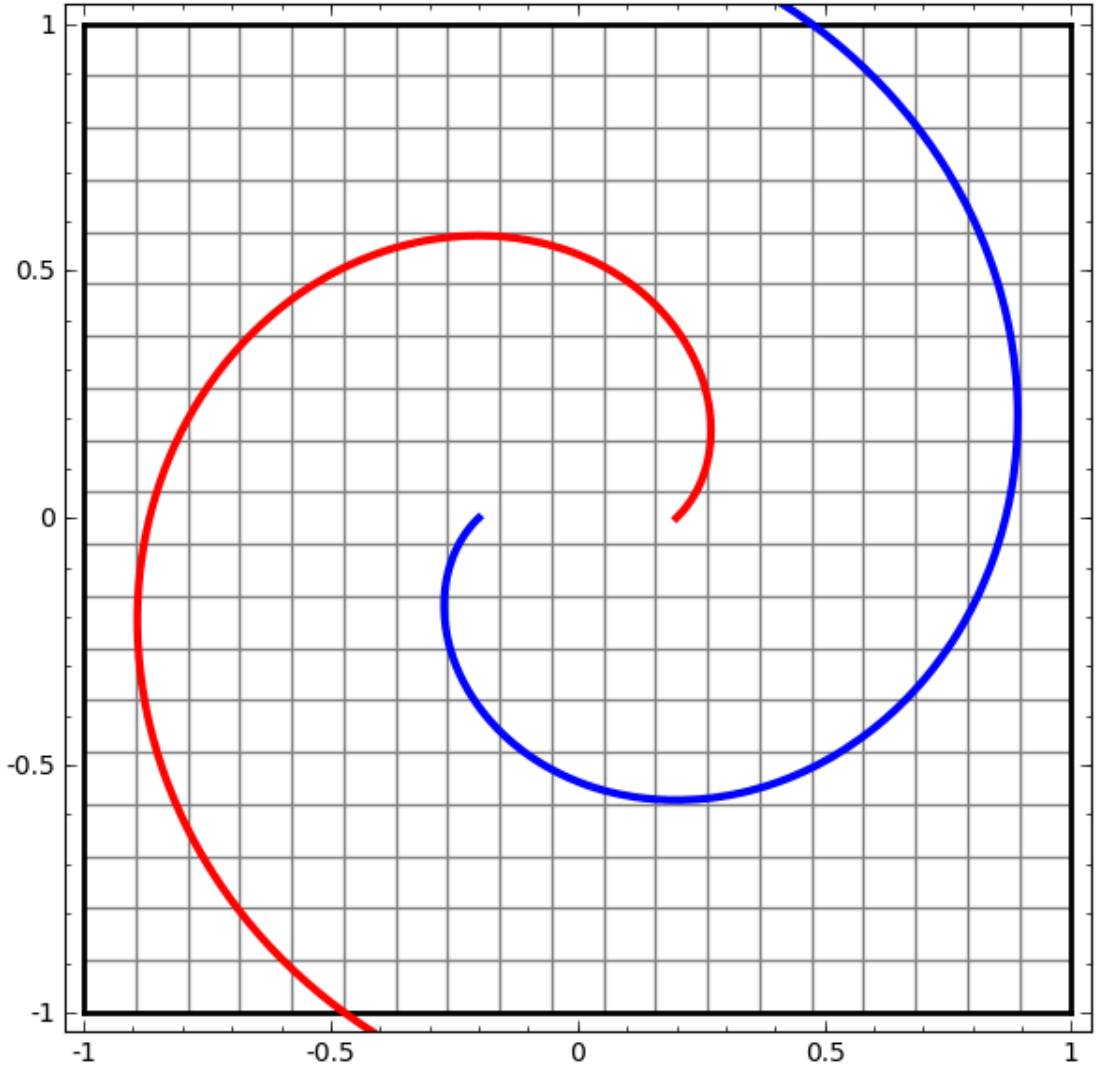


Decision function



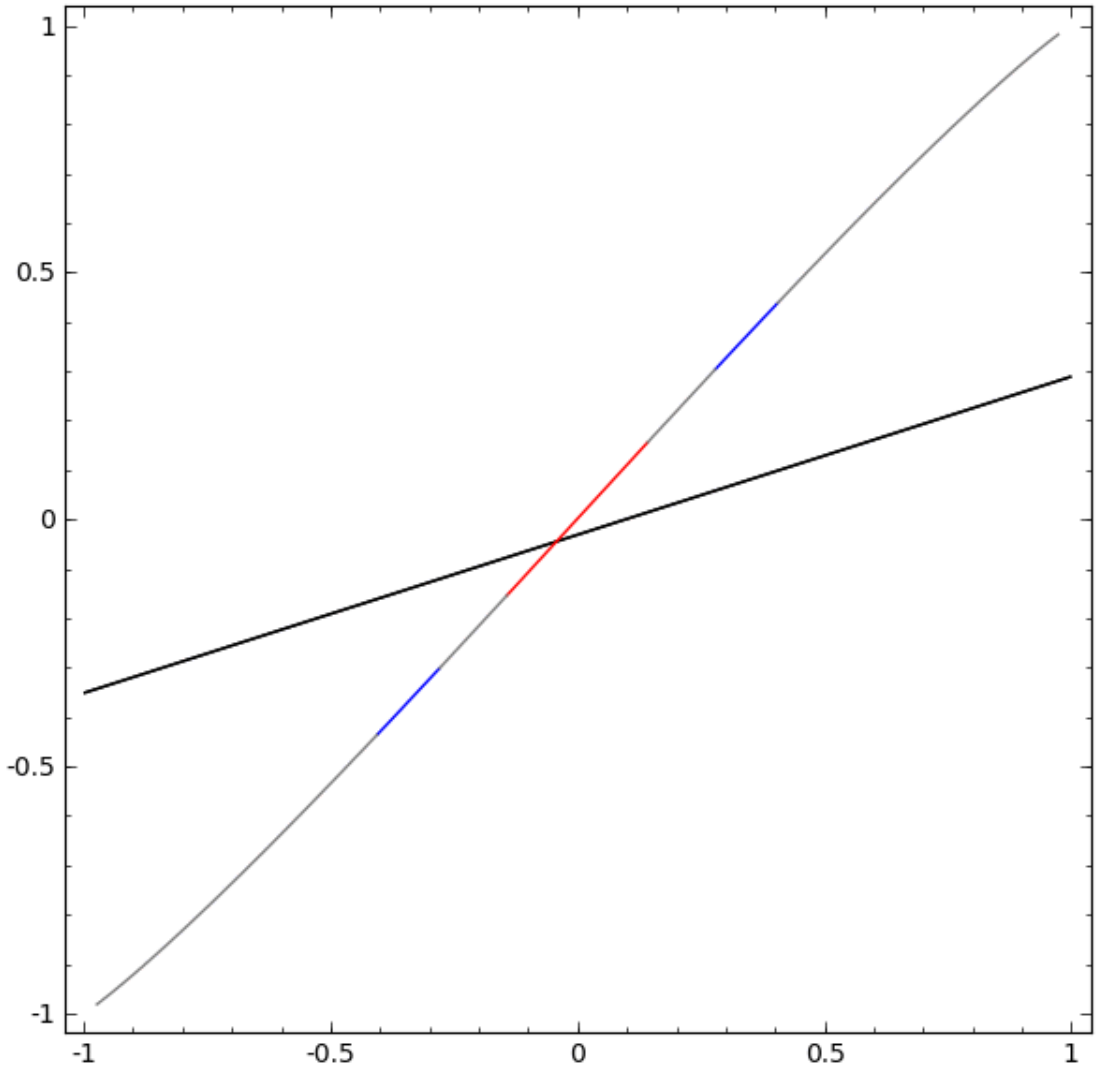
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Linearizing a 2D classification task (4 hidden layers)



http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

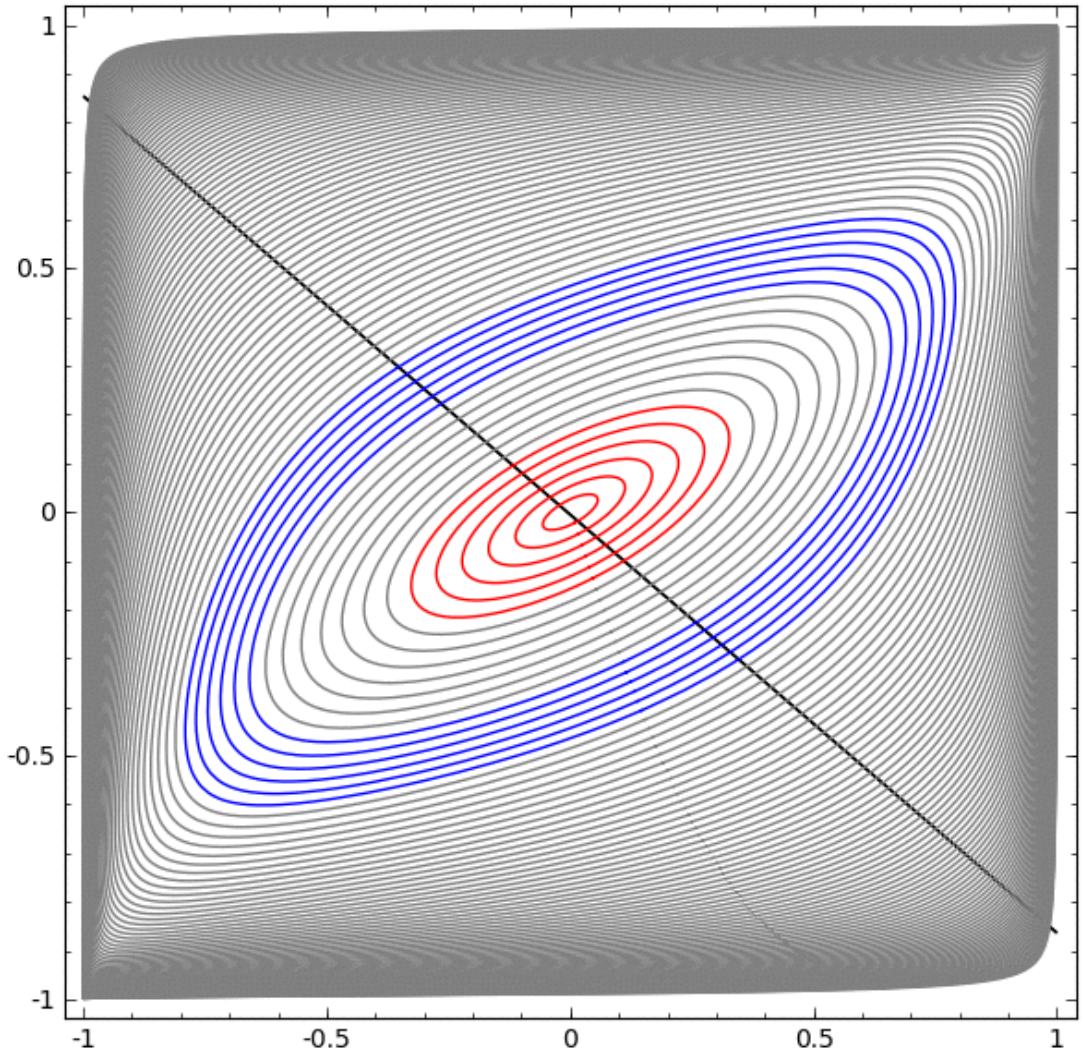
Linearization: may need higher dimensions



Points in 1D,
Decision in 2D

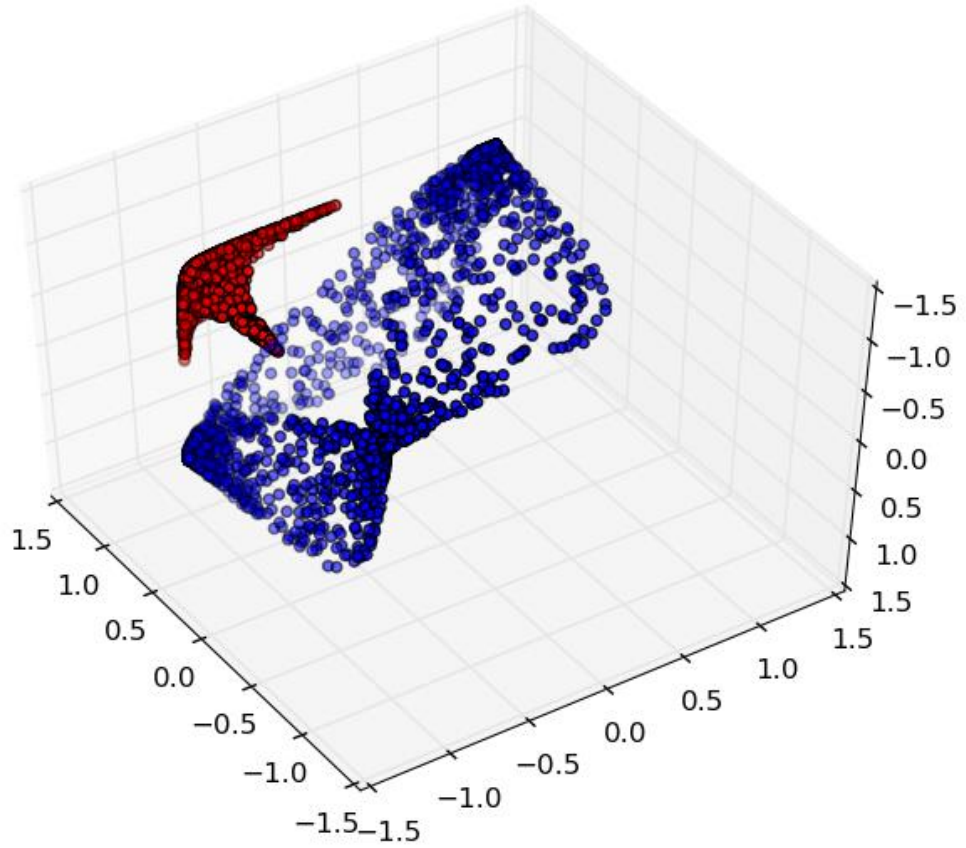
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Linearization: may need higher dimensions



http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Linearization: may need higher dimensions



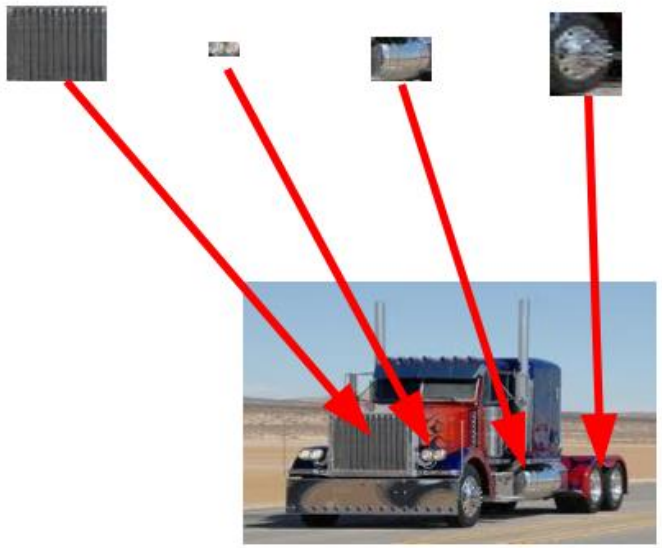
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Hidden Layers: intuitively, what do they do?

Intuition: learn “dictionary” for objects

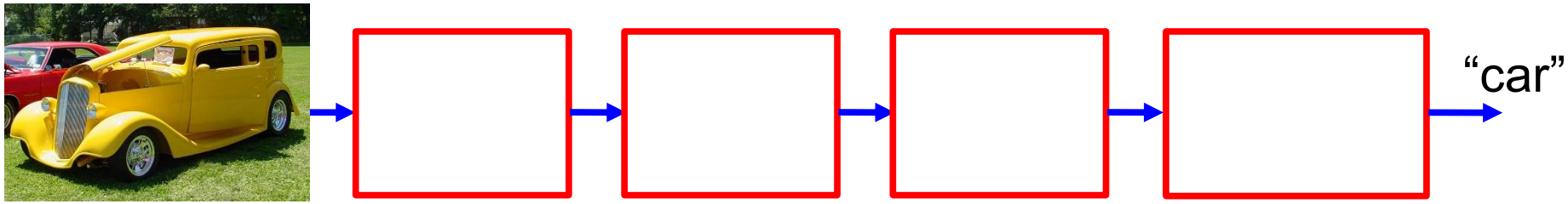
“Distributed representation”:
represent (and classify) objects by mixing & mashing reusable parts

[0 0 1 0 0 0 0 1 0 0 1 1 0 0 1 0 …] truck feature

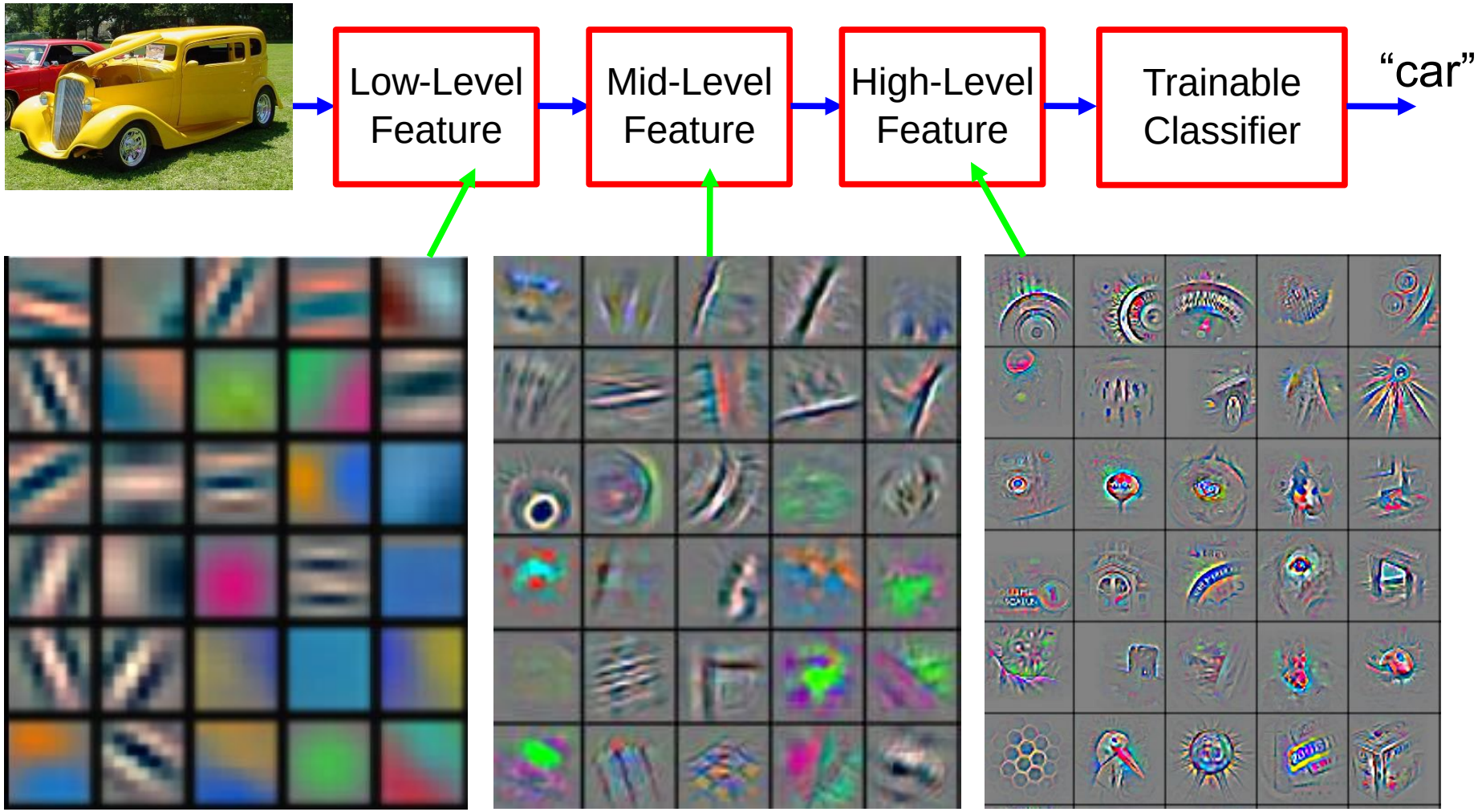


Slide Credit: Marc'Aurelio Ranzato, Yann LeCun

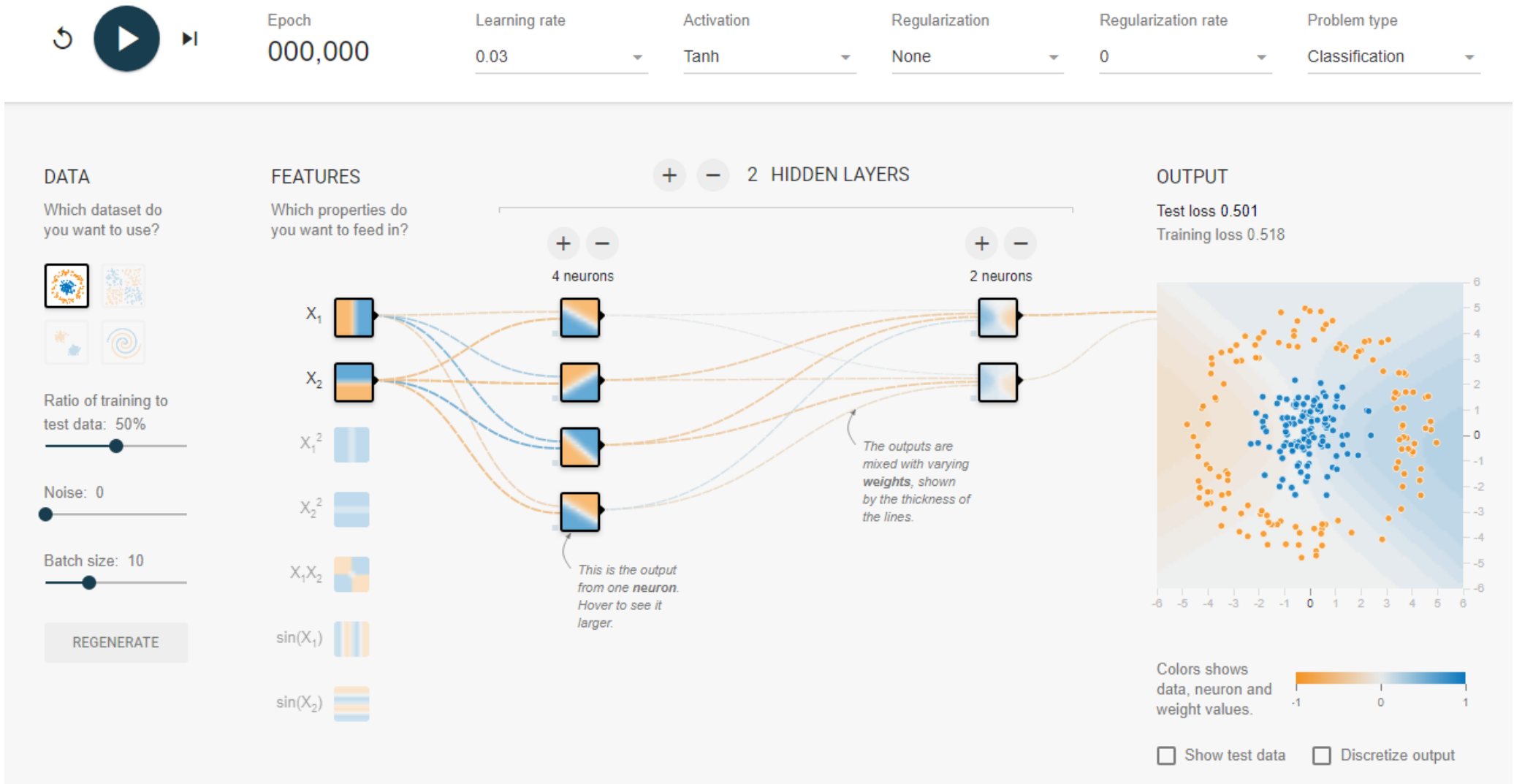
Deep Learning = Hierarchical Compositionality



Deep Learning = Hierarchical Compositionality



MLP Demo: playground.tensorflow.org



Training and Optimization

Neural Network Training: Old & New Tricks

Old:

- Back-propagation algorithm
- Stochastic Gradient Descent, Momentum, “weight decay”

New: (last 5-6 years)

- Dropout
- ReLUs
- Batch Normalization
- Residual Networks

Training Goal

Our network implements a parametric function:

$$f_\theta : \mathbb{X} \longrightarrow \mathbb{Y} \qquad \hat{y} = f(x;\theta)$$

During training, we search for parameters that minimize a loss:

$$\min_\theta L_f(\theta)$$

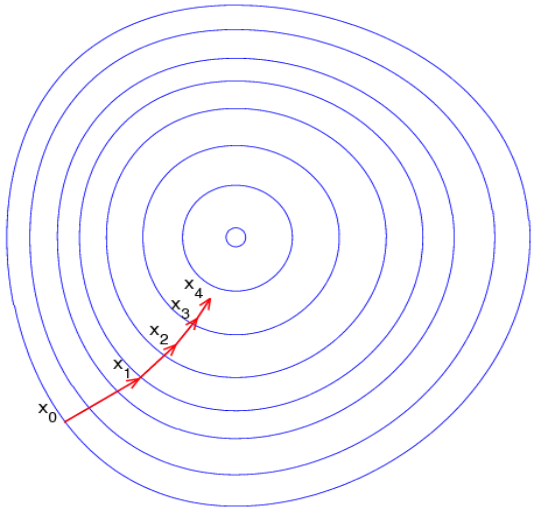
Example: L2 regression loss given target (x^i, y^i) pairs :

$$L_f(\theta) = \sum_i \| f(x^i;\theta) - y^i \|_2^2$$

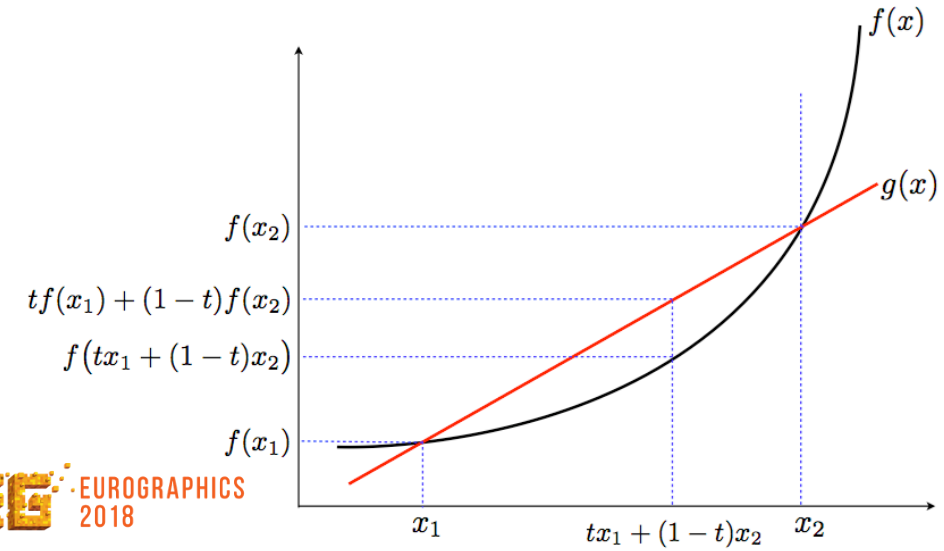
Gradient Descent Minimization Method

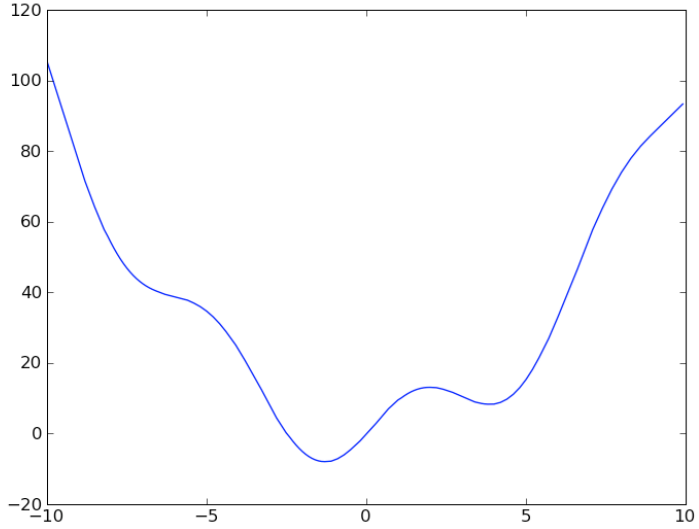
Initialize: θ_0

Update: $\theta_{i+1} = \theta_i - \alpha \nabla f(\theta_i)$

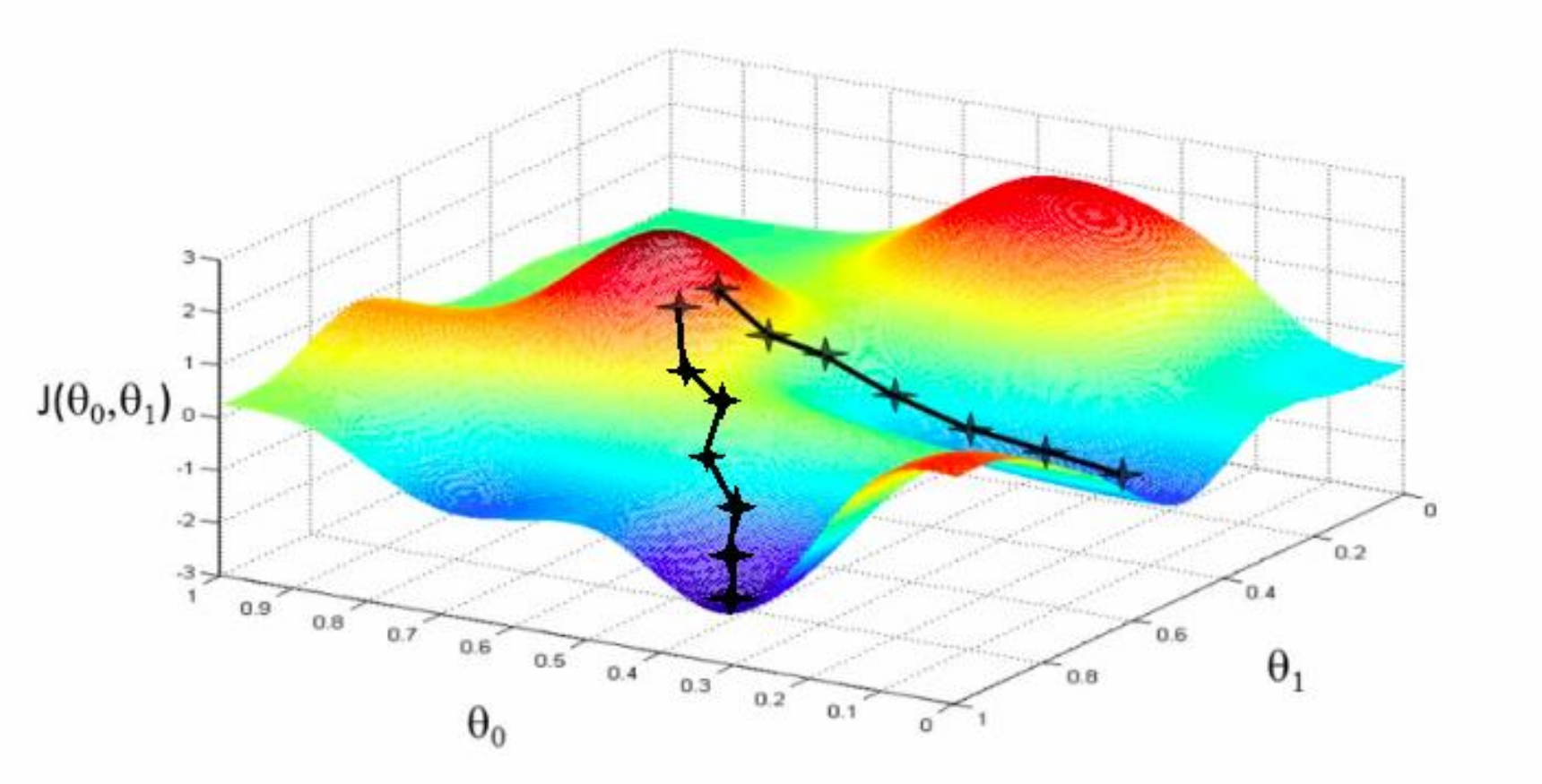


We can always make it converge for a **convex** function





Multiple Local Minima, based on initialization

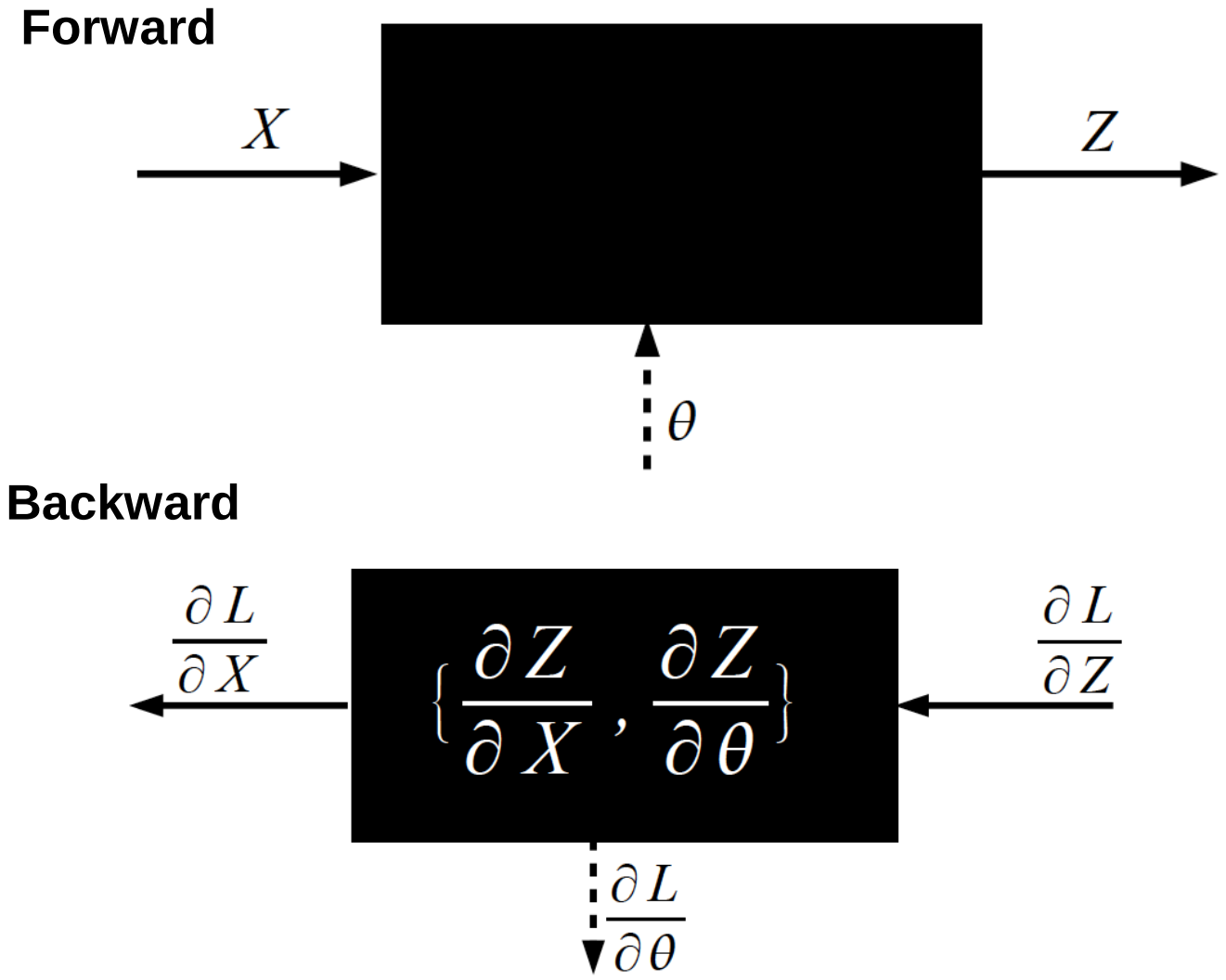


Empirically all are almost equally good

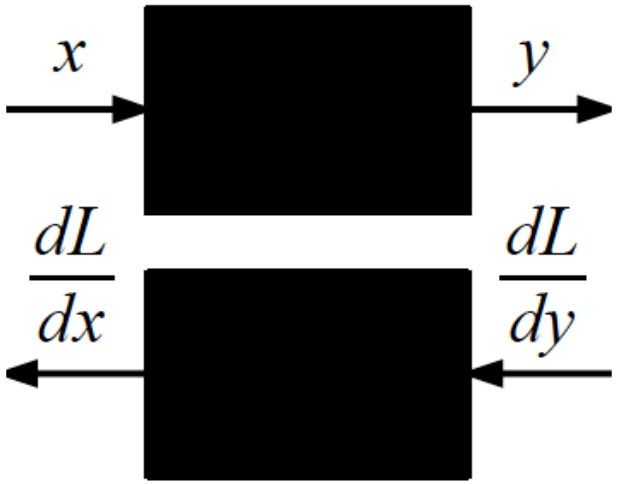
Central research topic: how can this happen?

On to the gradients!

All you need is gradients



Chain Rule

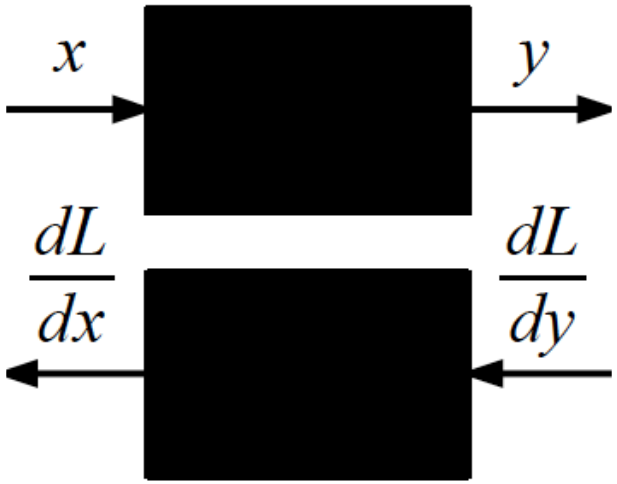


Given $y(x)$ and dL/dy,

What is dL/dx ?

Slide Credit: Marc'Aurelio Ranzato, Yann LeCun

Chain Rule



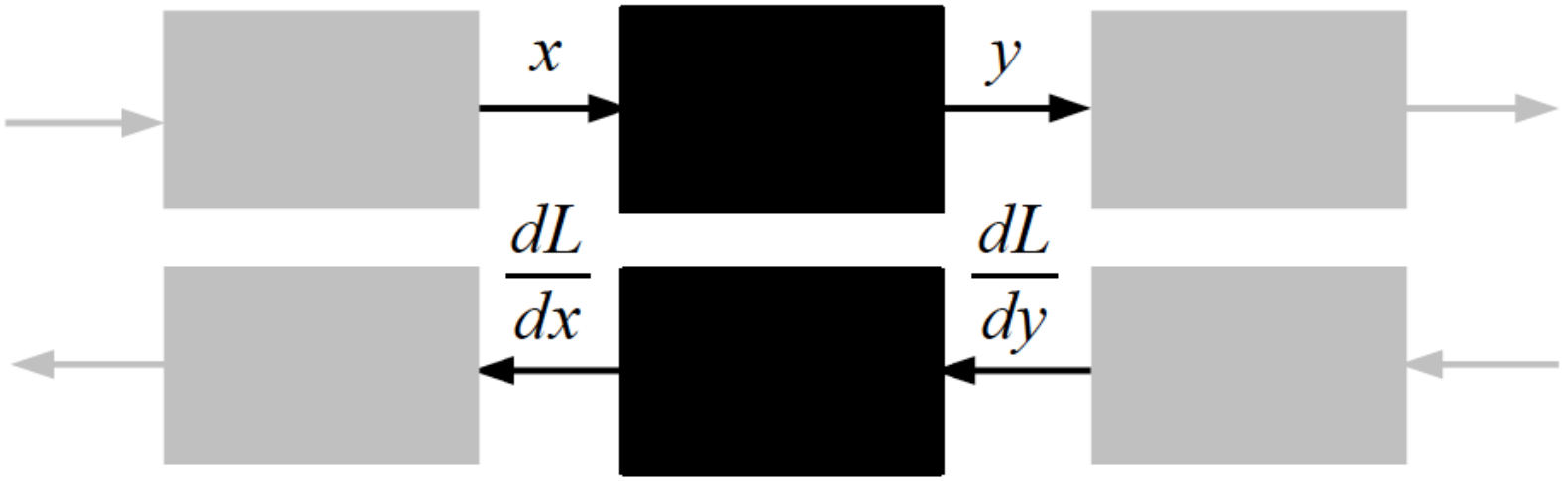
Given $y(x)$ and dL/dy,

What is dL/dx ?

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

Slide Credit: Marc'Aurelio Ranzato, Yann LeCun

‘Another Brick in the Wall’



Given $y(x)$ and dL/dy,

What is dL/dx ?

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

Slide Credit: Marc'Aurelio Ranzato, Yann LeCun

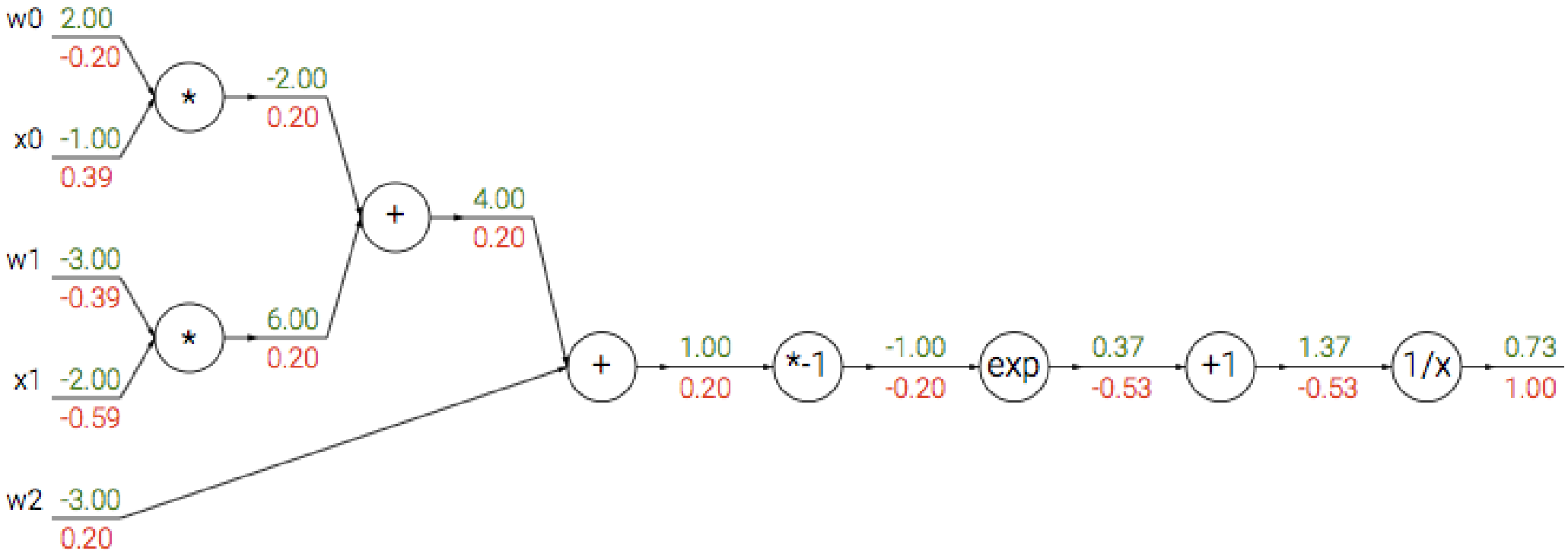
Toy example: single sigmoidal unit

$$f(w,x) = \frac{1}{1+\exp(-(w_0x_0 + w_1x_1 + w_2))}$$

Composition of differentiable blocks:

$$\begin{aligned} f(x) = \frac{1}{x} \quad &\rightarrow \quad f'(x) = -\frac{1}{x^2} \\ f_c(x) = c + x \quad &\rightarrow \quad f'(x) = 1 \\ f(x) = e^x \quad &\rightarrow \quad f'(x) = e^x \\ f_a(x) = ax \quad &\rightarrow \quad f'(x) = a \end{aligned}$$

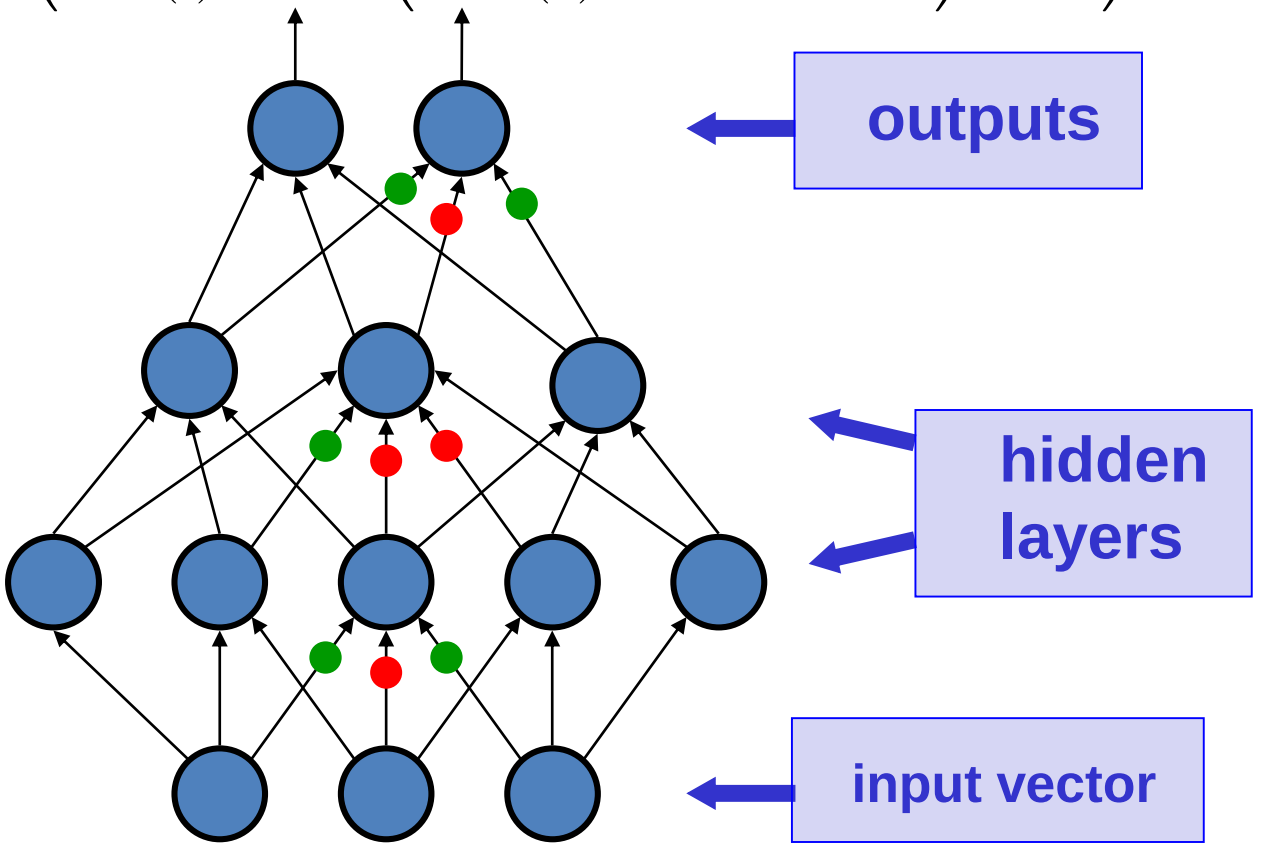
Computation graph & automatic differentiation



Slide Credit: Justin Johnson

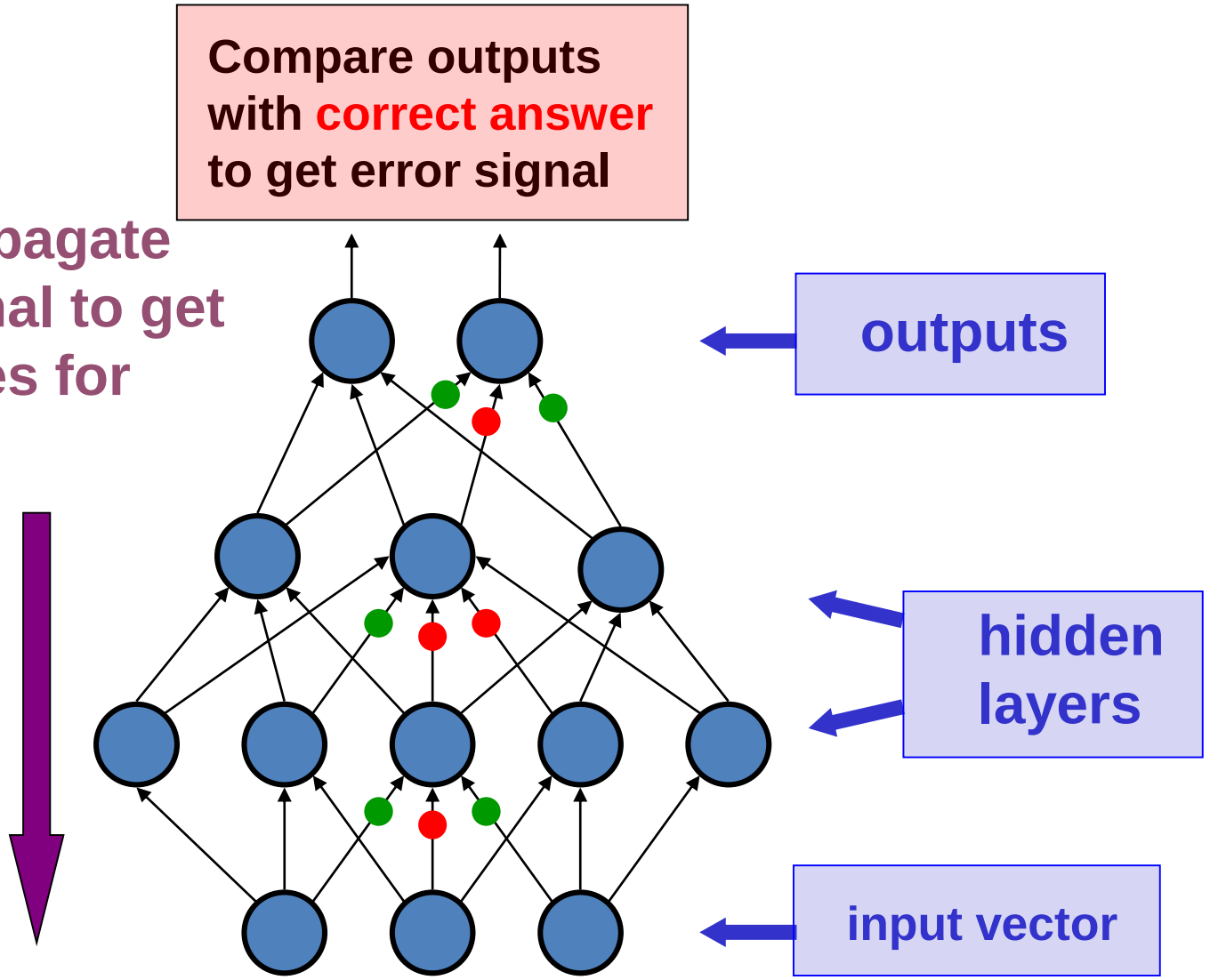
Multi-Layer Perceptrons

$$u_i = g\left(\sum_{k\in\mathcal{N}(i)} w_{k,i}\, g\left(\sum_{m\in\mathcal{N}(k)} w_{m,k}u_m + b_k\right) + b_i\right)$$



Slide Credit: G. Hinton

Multi-Layer Perceptrons



Slide Credit: G. Hinton

Back-propagation Algorithm

Training Goal

Our network implements a parametric function:

$$f_\theta : \mathbb{X} \longrightarrow \mathbb{Y} \qquad \hat{y} = f(x;\theta)$$

During training, we search for parameters that minimize a loss:

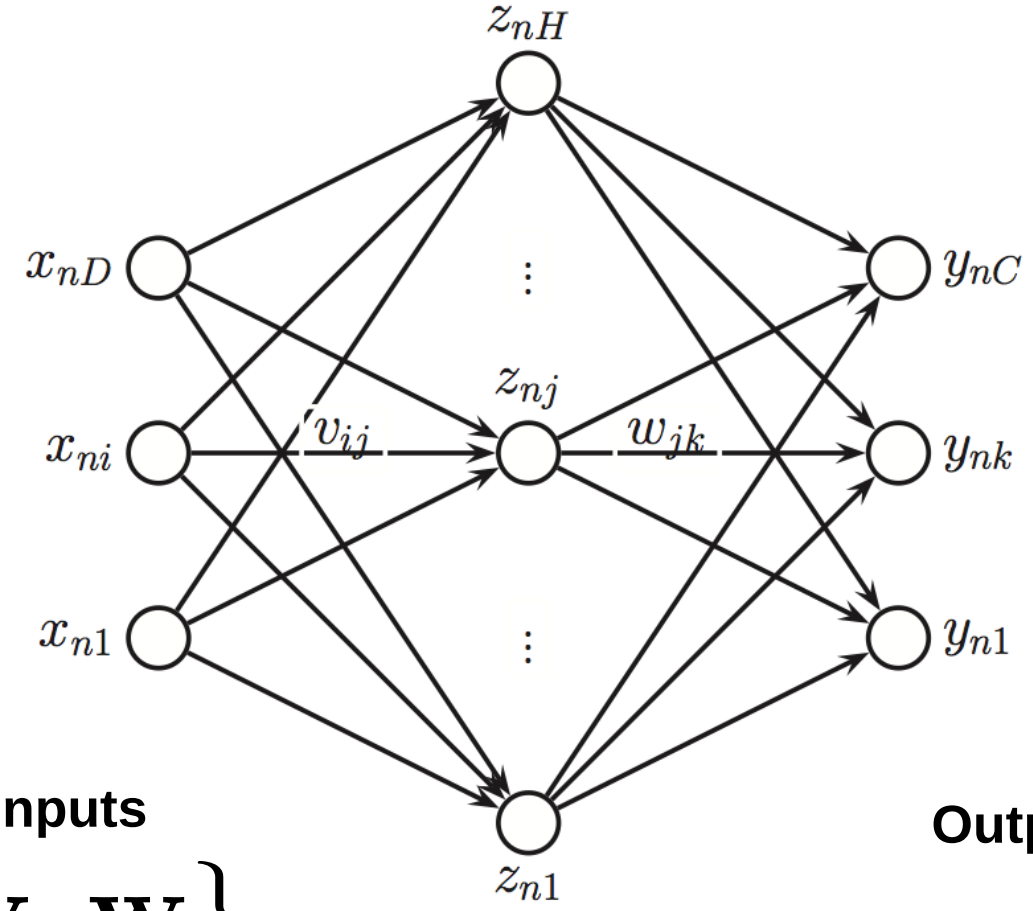
$$\min_\theta L_f(\theta)$$

Example: L2 regression loss given target (x^i, y^i) pairs :

$$L_f(\theta) = \sum_i \| f(x^i;\theta) - y^i \|_2^2$$

A Neural Network for Multi-way Classification

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n$$



Inputs

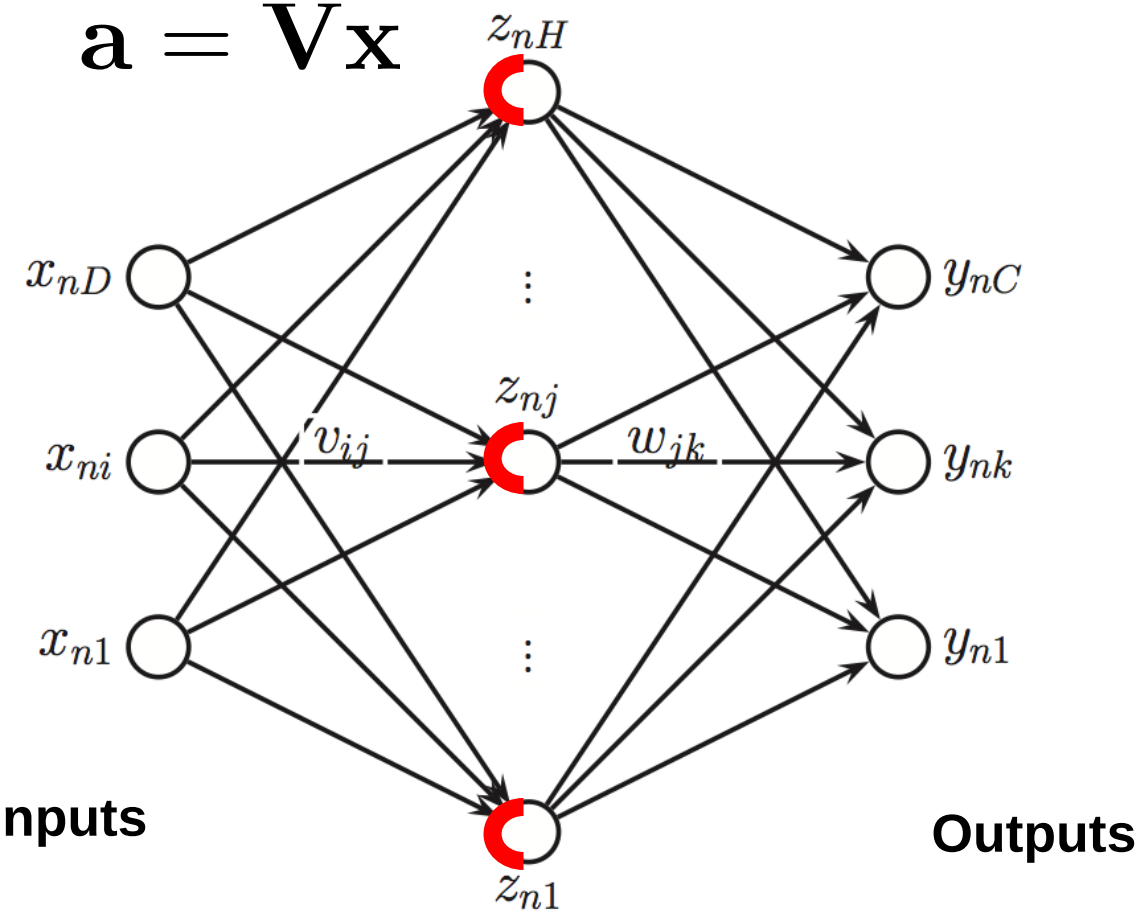
Outputs

Hidden layer

Parameters: $\theta = \{\mathbf{V}, \mathbf{W}\}$

A Neural Network in Forward Mode

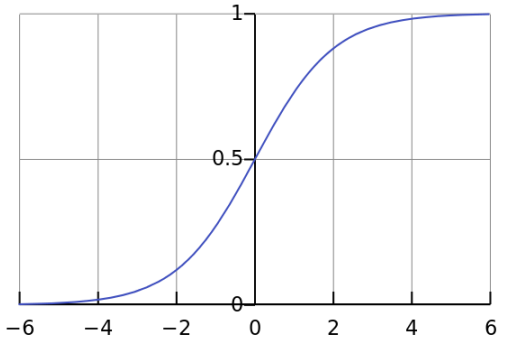
$$\mathbf{a} = \mathbf{V}\mathbf{x}$$

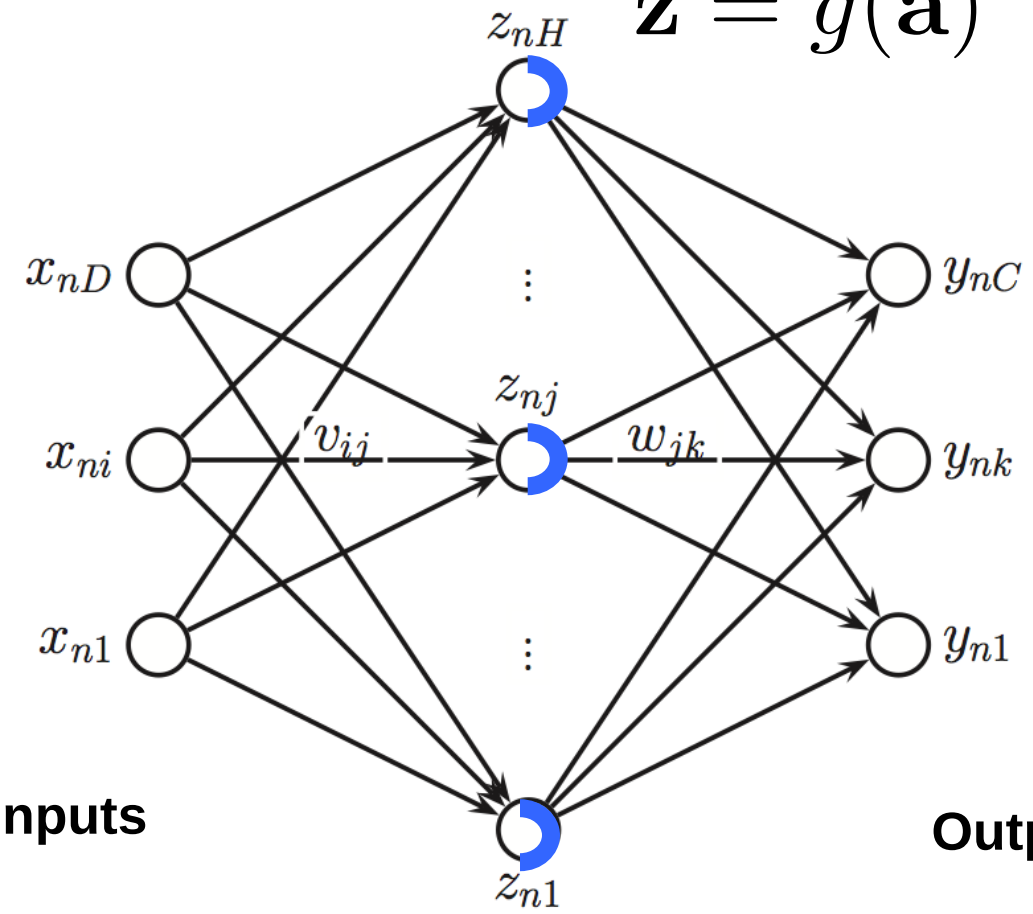


A Neural Network in Forward Mode ⏩

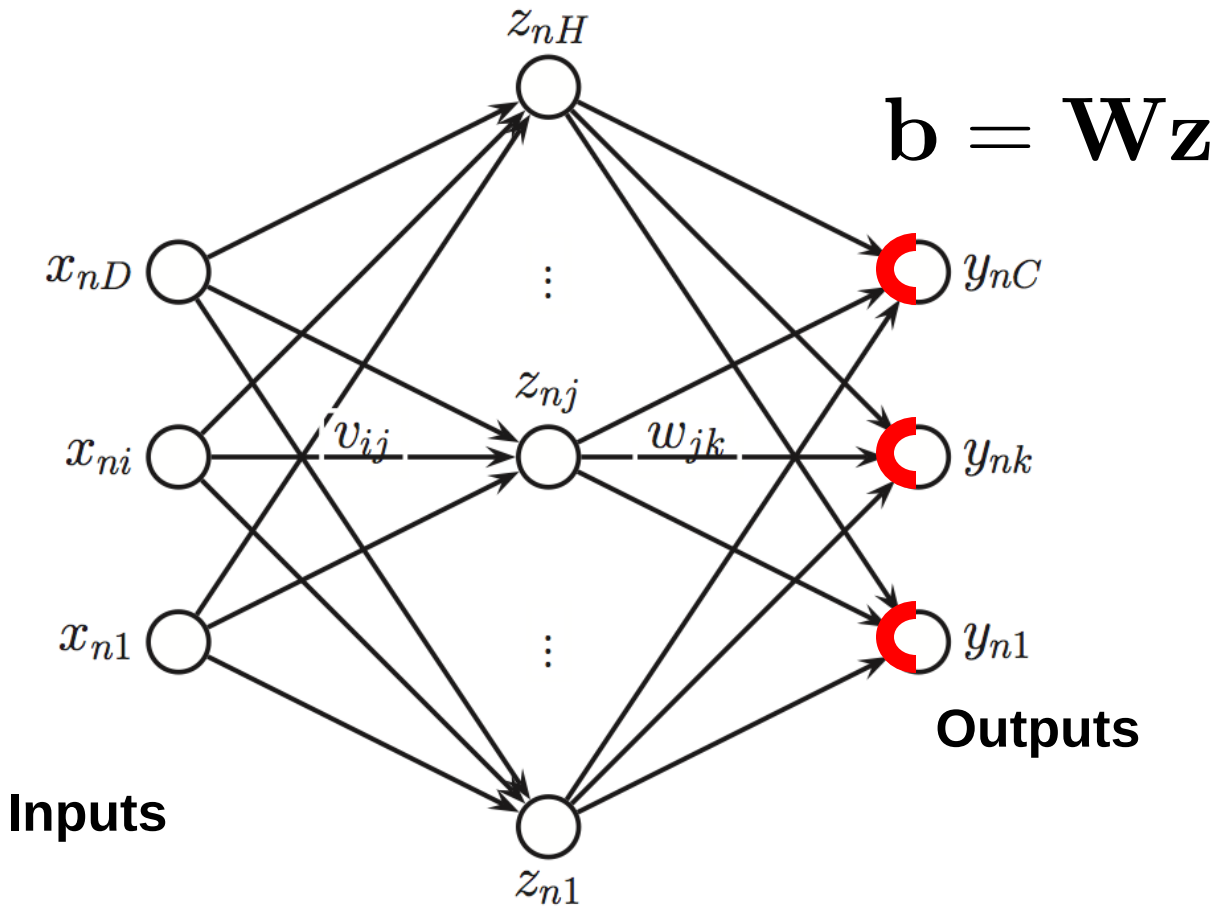
$$\mathbf{z} = g(\mathbf{a})$$

$$z_k = \frac{1}{1 + \exp(-a_k)}$$

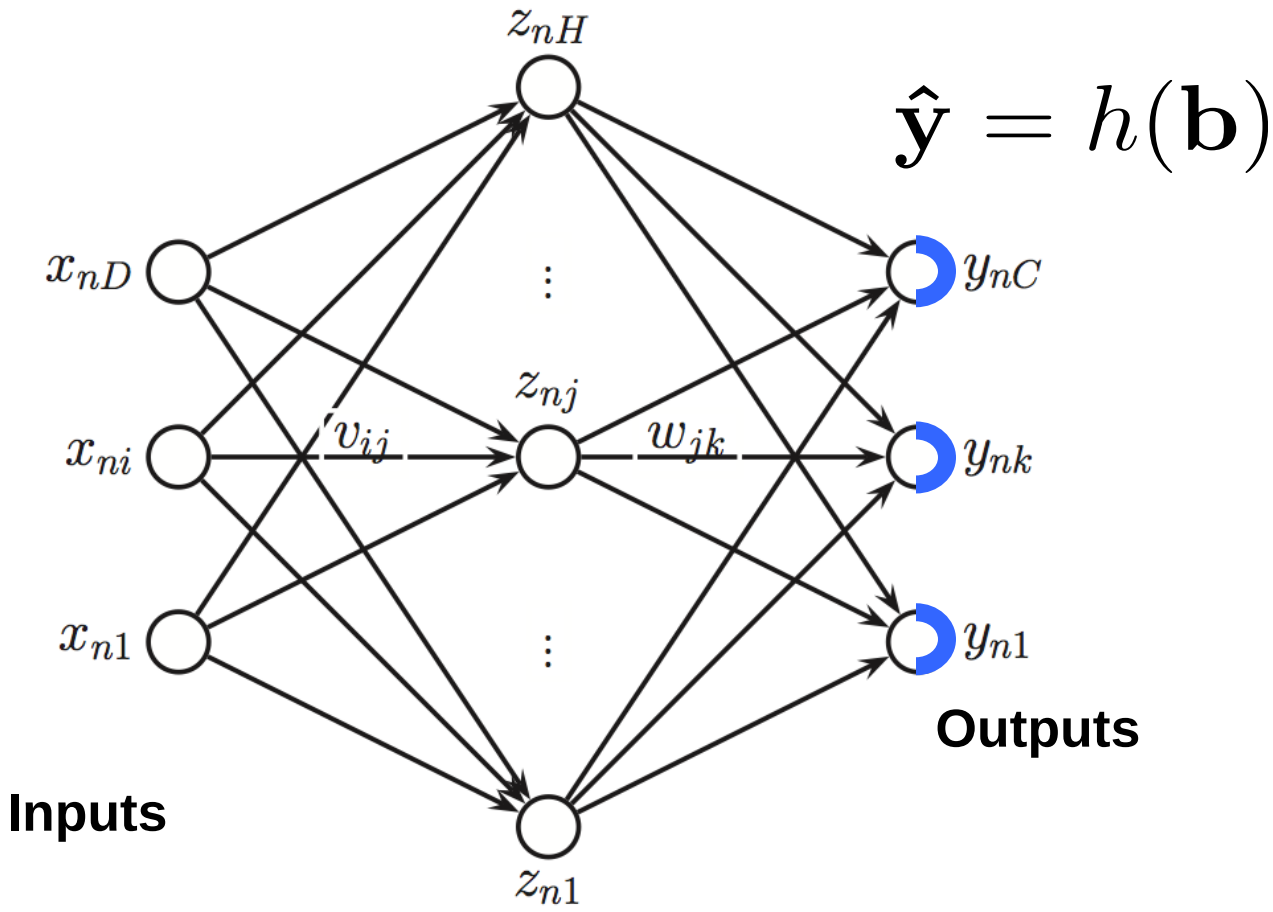




A Neural Network in Forward Mode ⏩

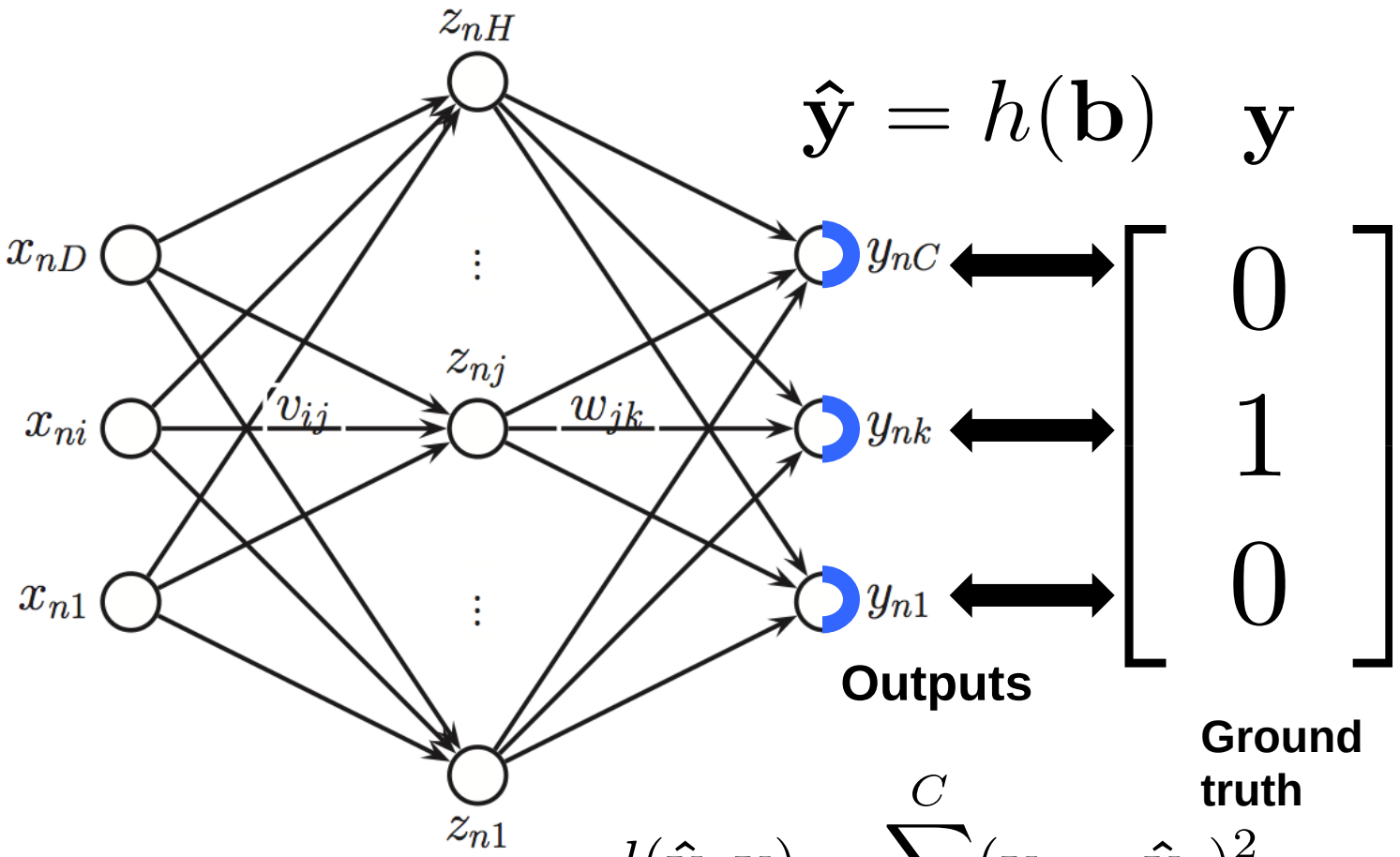


A Neural Network in Forward Mode ⏩



Objective for linear regression

$$h(\mathbf{b}) = \mathbf{b}$$



$$\hat{\mathbf{y}} = h(\mathbf{b}) \qquad \mathbf{y}$$

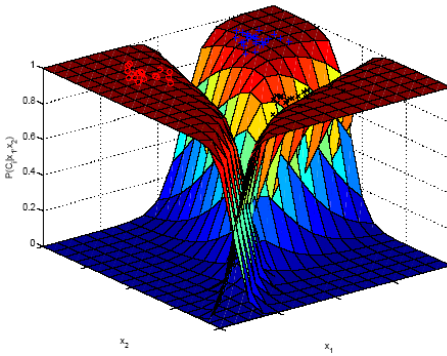
$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Outputs

Ground truth

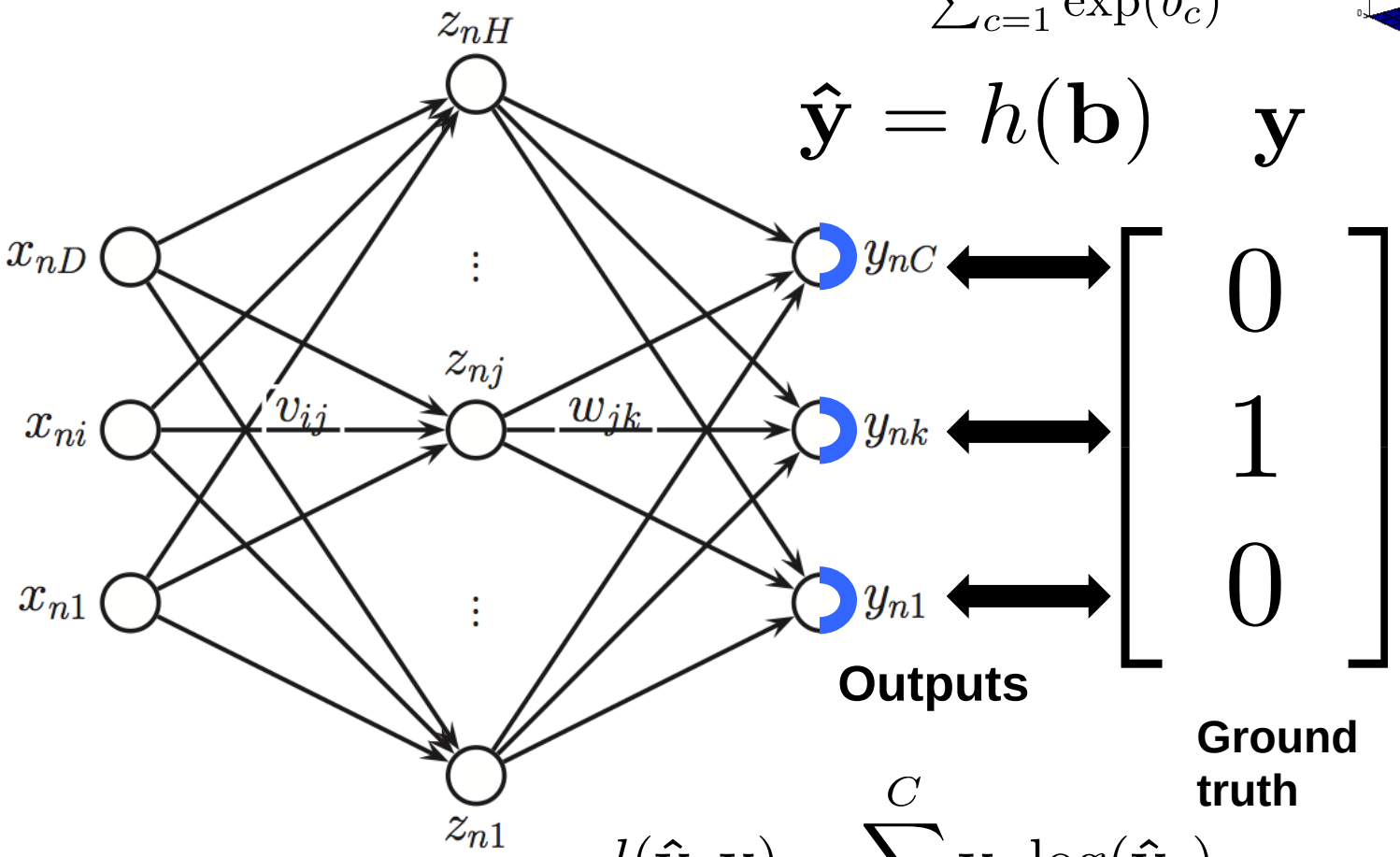
$$l(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{c=1}^{C} (\mathbf{y}_c - \hat{\mathbf{y}}_c)^2$$

Objective for multi-class classification



Softmax unit $\hat{y}_k = \frac{\exp(b_k)}{\sum_{c=1}^{C} \exp(b_c)}$

$\hat{\mathbf{y}} = h(\mathbf{b})$ $\mathbf{y}$



Outputs

Ground truth

$l(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{c=1}^{C} \mathbf{y}_c \log(\hat{\mathbf{y}}_c)$ `Cross-entropy' loss

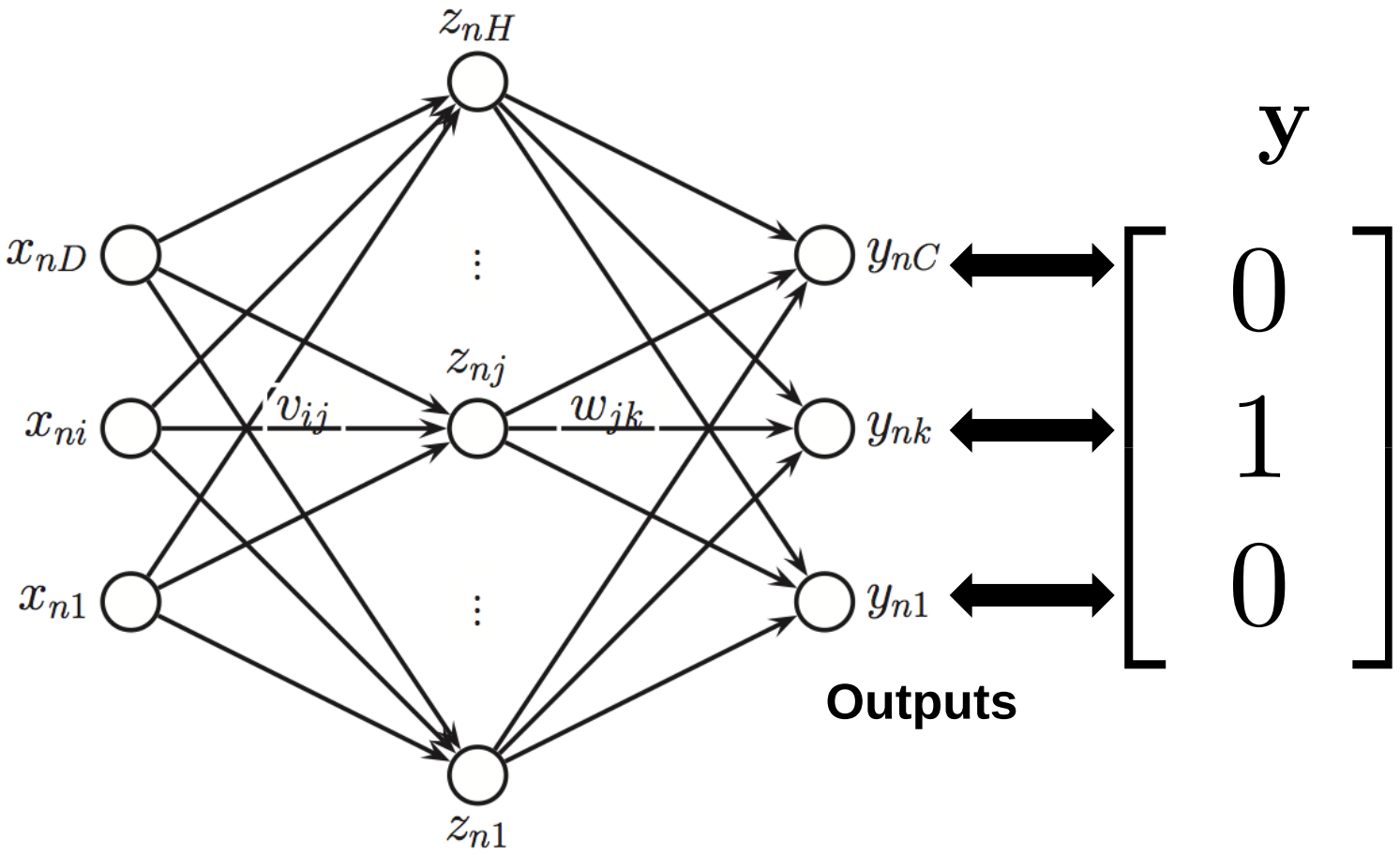
Neural network in forward mode: recap

Network output: $\hat{\mathbf{y}} = f(\mathbf{x}; \mathbf{v}, \mathbf{w})$

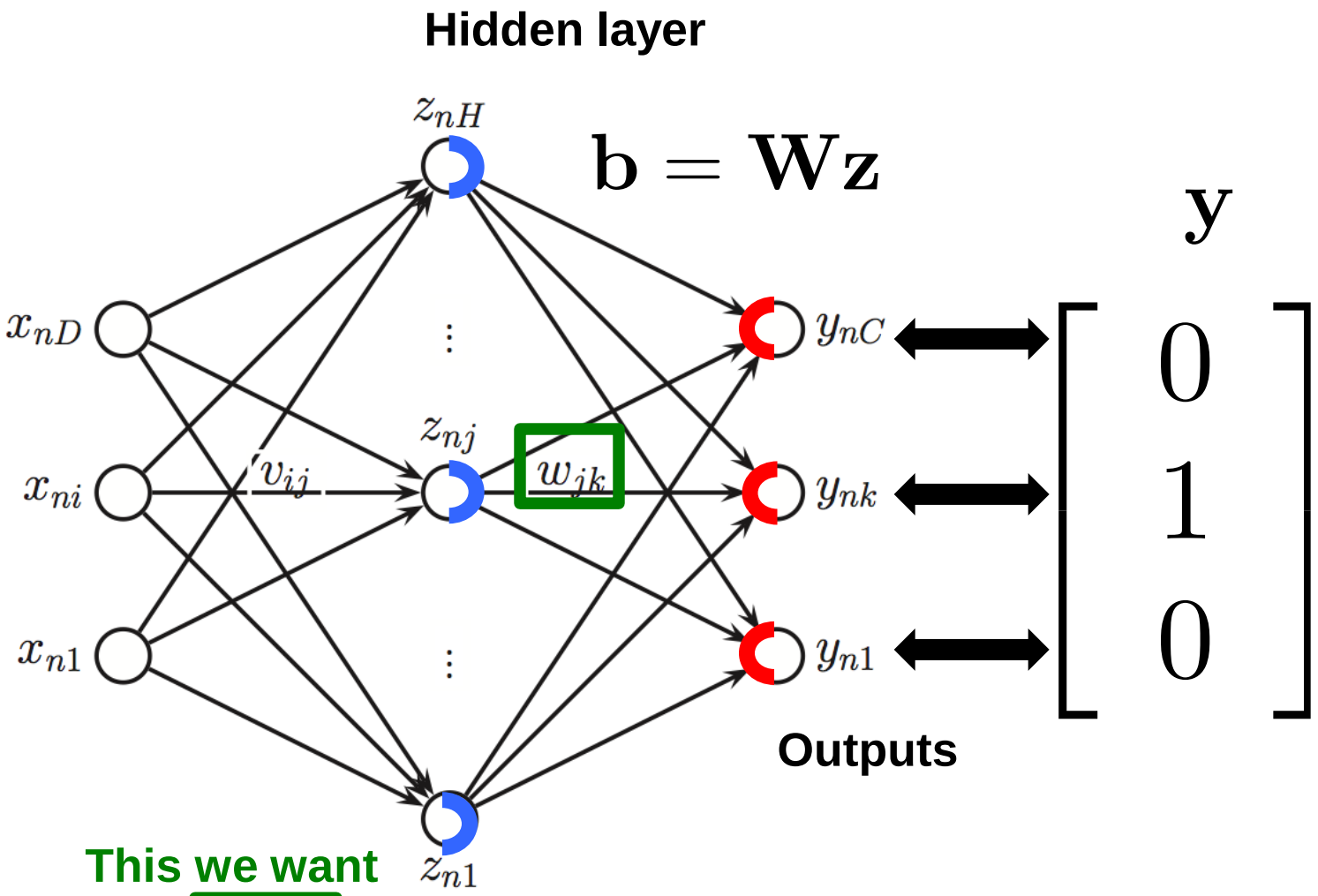
Loss (prediction error): $l(\hat{\mathbf{y}}, \mathbf{y})$

What we need to compute for gradient descent: $\frac{\partial l(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{v}_i}$ $\frac{\partial l(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{w}_j}$

A Neural Network in Backward Mode



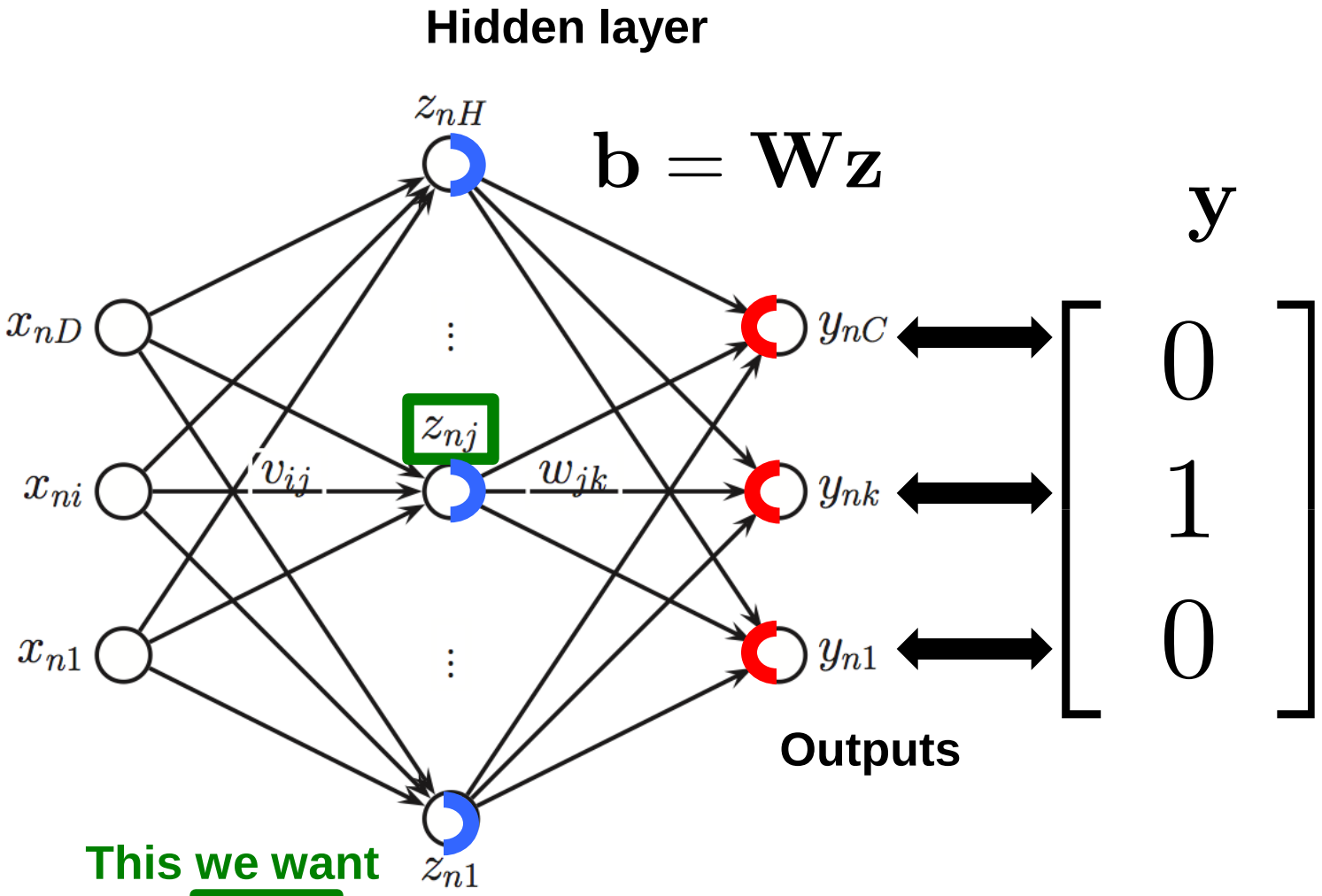
A Neural Network in Backward Mode



This we want

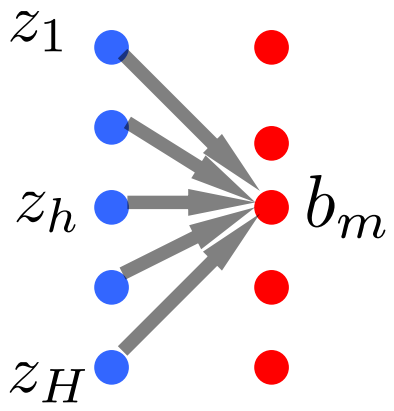
$$\boxed{\frac{\partial l}{\partial w_{jk}}} = \; ?$$

A Neural Network in Backward Mode



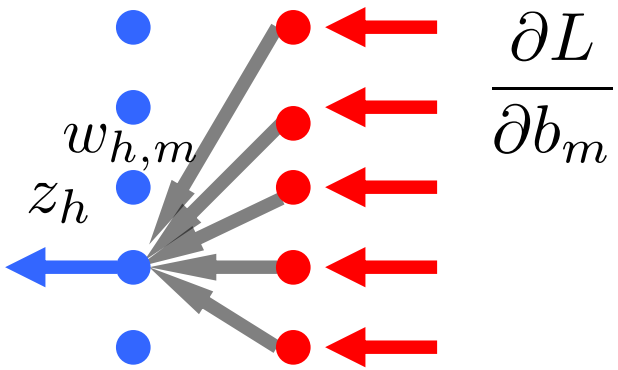
Linear Layer in Forward Mode: All For One

$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



Linear Layer in Backward Mode: One From All

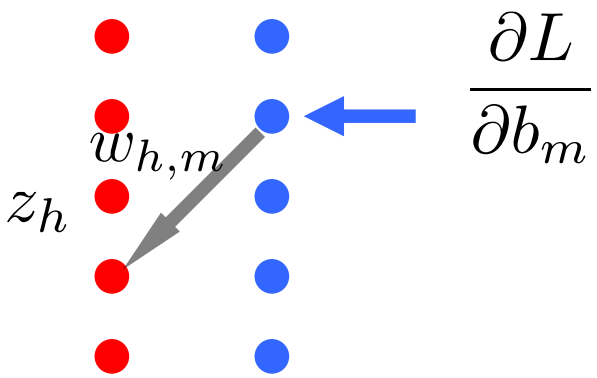
$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



$$\frac{\partial L}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} w_{h,c}$$

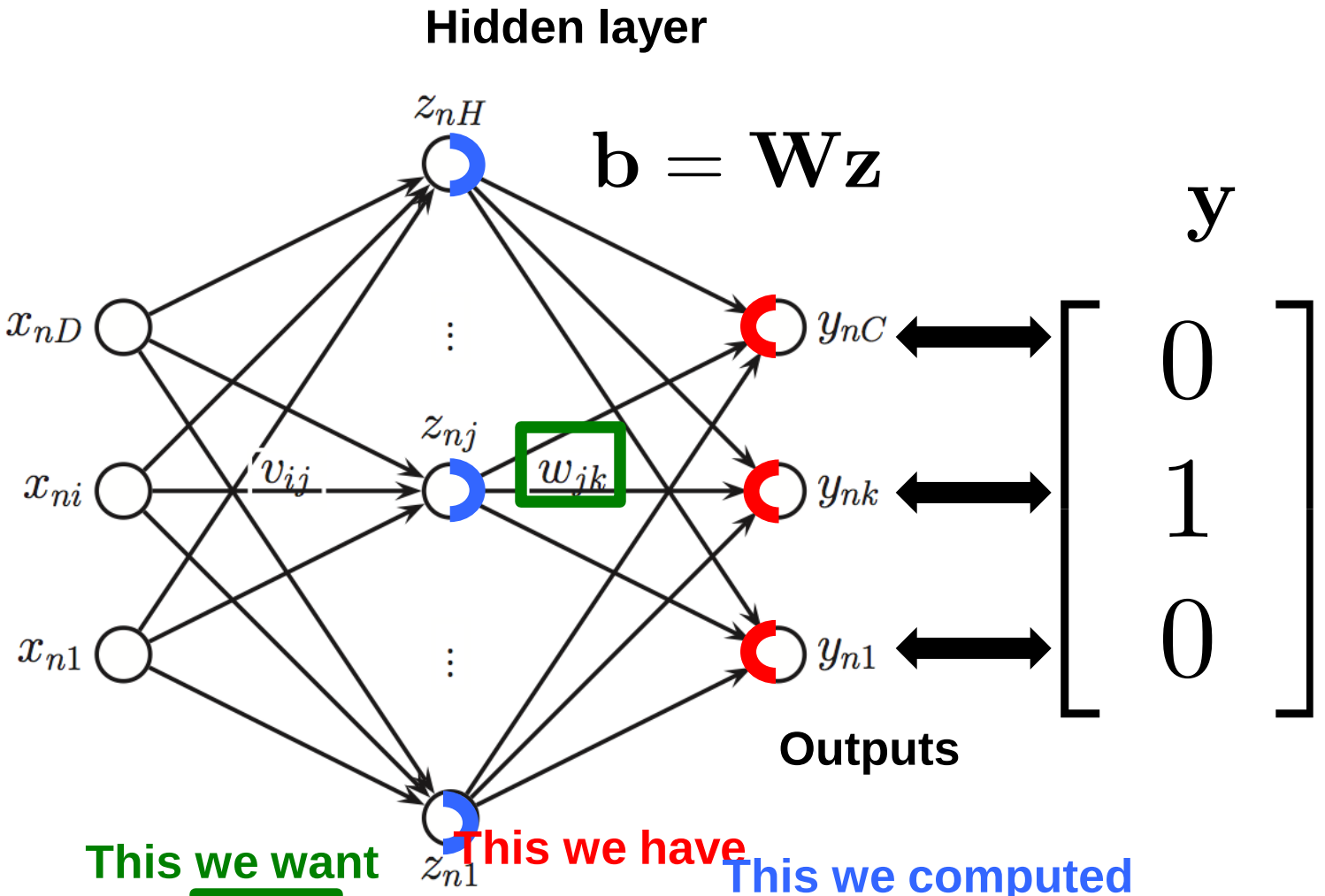
Linear Layer Parameters in Backward: 1-to-1

$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



$$\frac{\partial L}{\partial w_{h,m}} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial w_{h,m}} = \frac{\partial L}{\partial b_m} z_h$$

A Neural Network in Backward Mode



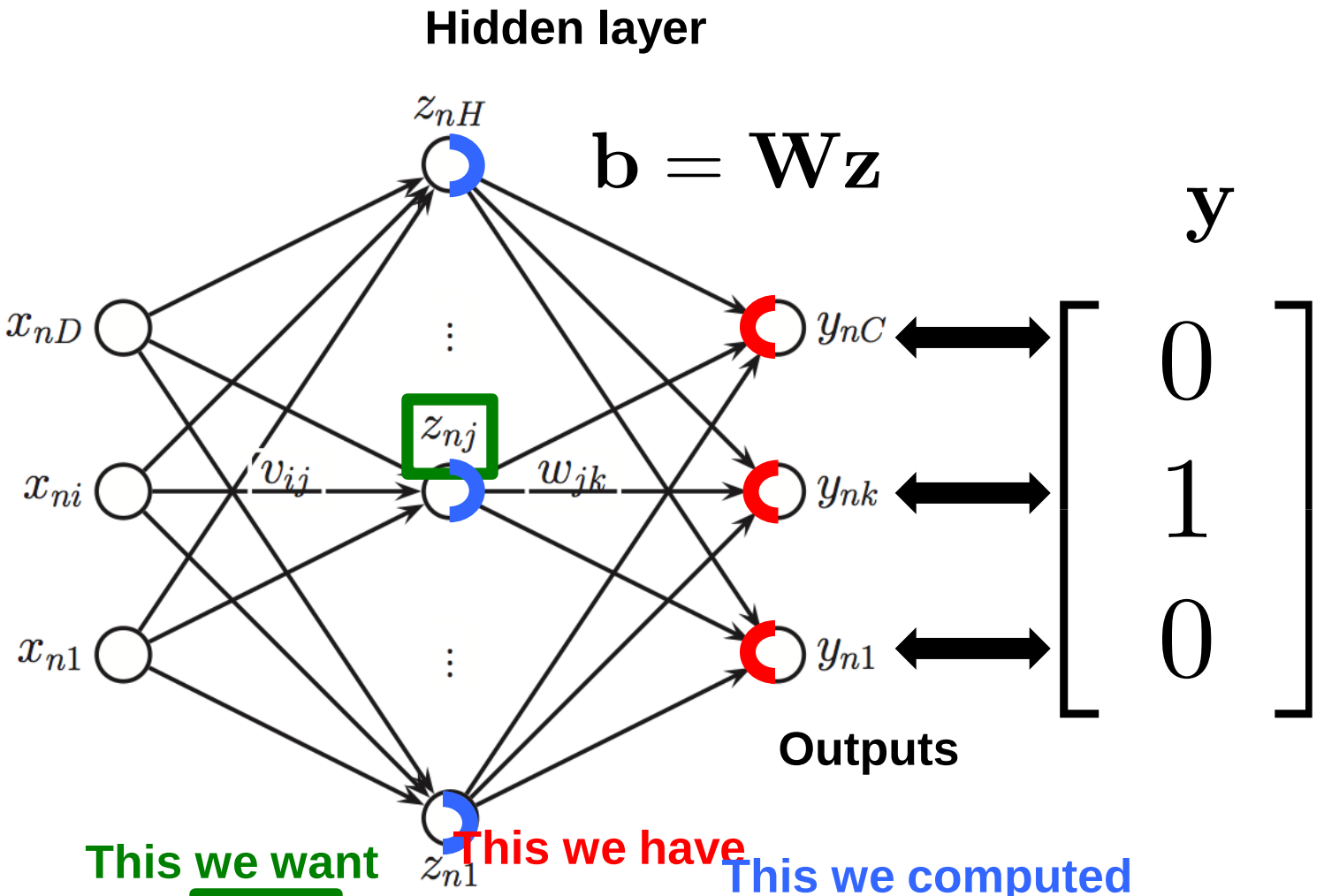
This we want

This we have

This we computed

$$\frac{\partial l}{\partial w_{jk}} = \sum_m \frac{\partial l}{\partial b_m} \frac{\partial b_m}{\partial w_{jk}} = \frac{\partial l}{\partial b_m} z_j$$

A Neural Network in Backward Mode



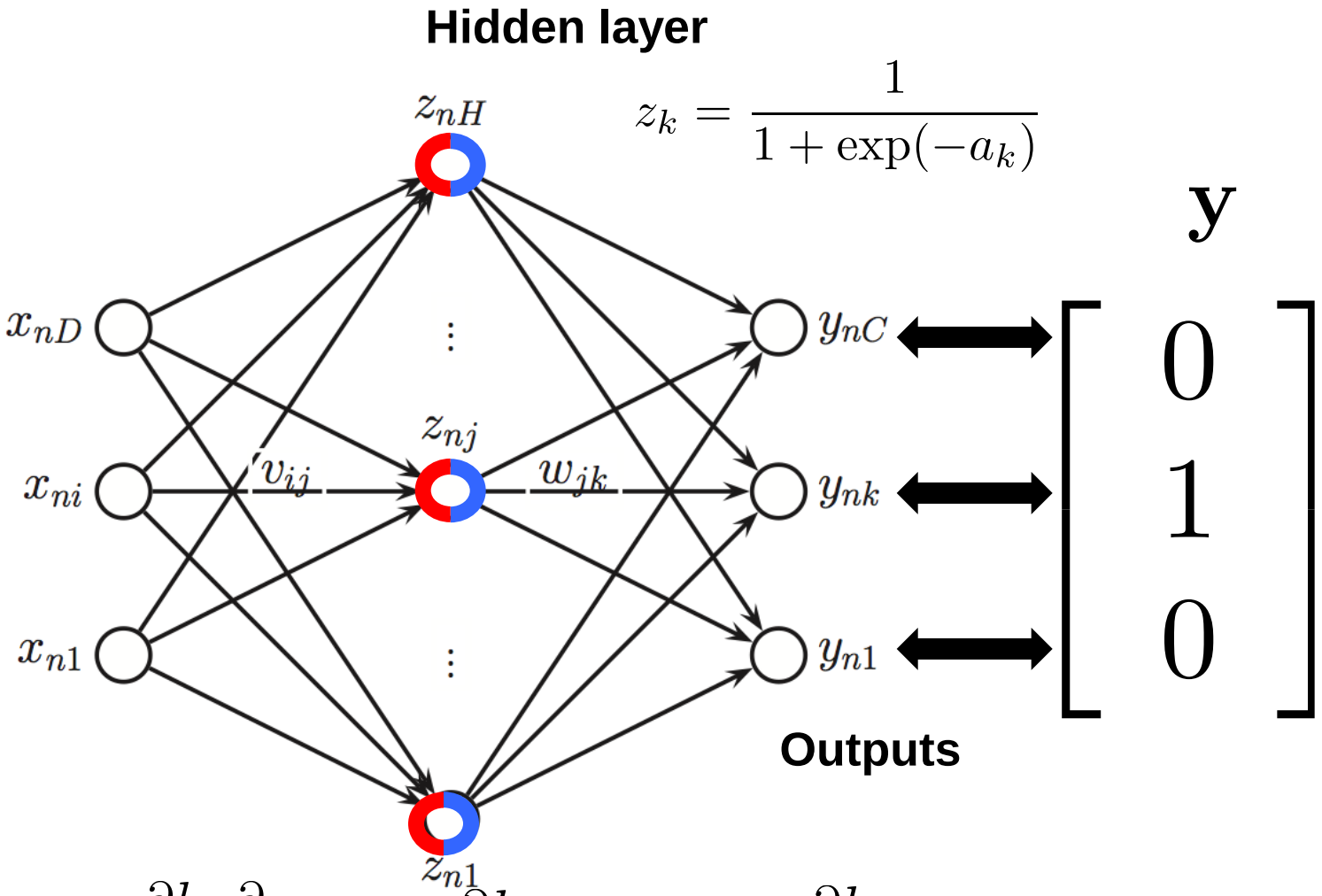
This we want

This we have

This we computed

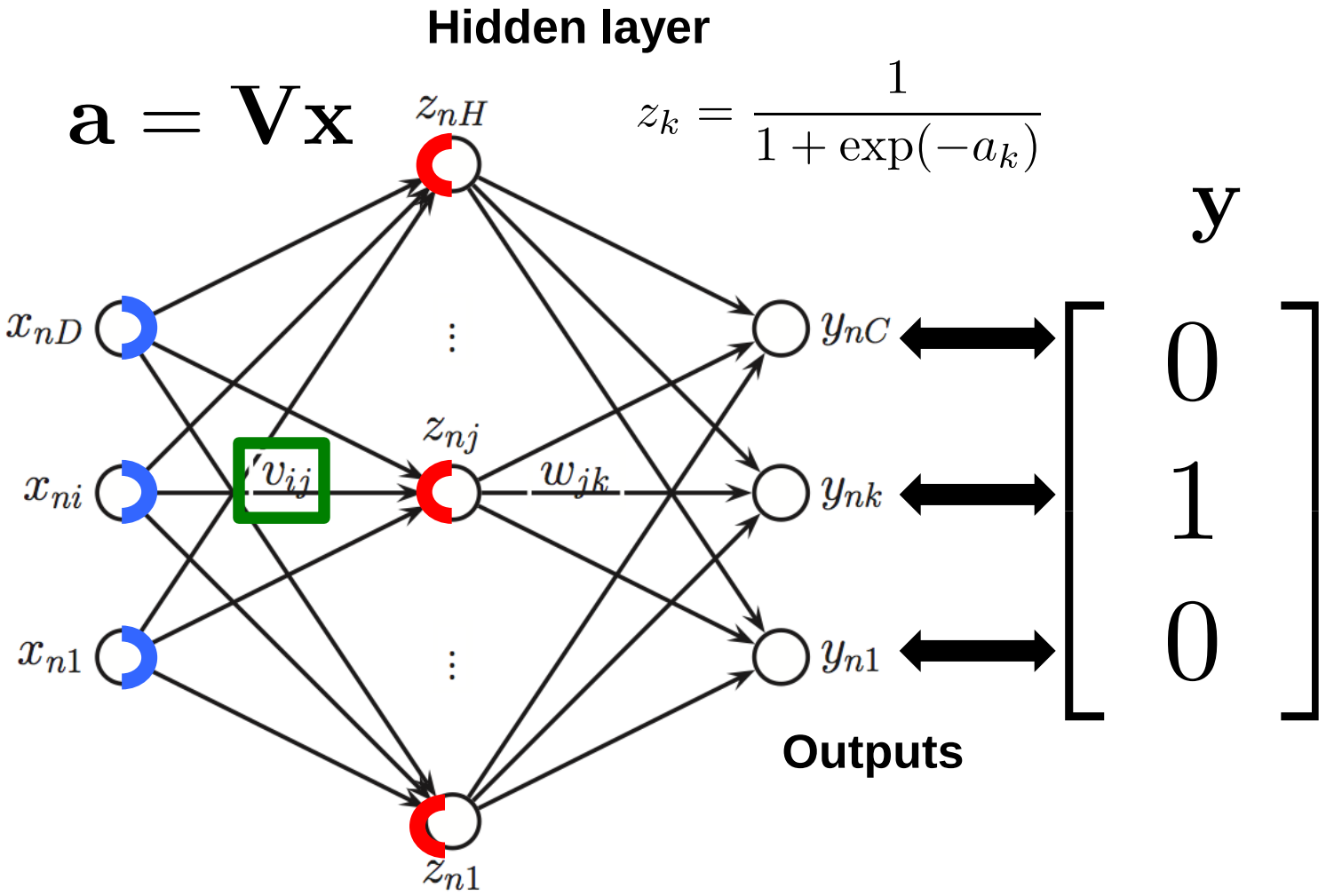
$$\boxed{\frac{\partial l}{\partial z_j}} = \sum_m \boxed{\frac{\partial l}{\partial b_m}} \boxed{\frac{\partial b_m}{\partial z_j}} = \sum_m \frac{\partial l}{\partial b_m} w_{j,m}$$

A Neural Network in Backward Mode



$$\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m}\frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_k} g'(a_k) = \frac{\partial l}{\partial z_k} g(a_k)(1-g(a_k))$$

A Neural Network in Backward Mode



$$\frac{\partial l}{\partial v_{ij}} = \sum_k \frac{\partial l}{\partial a_k}\frac{\partial a_k}{\partial v_{ij}} = \frac{\partial l}{\partial a_j} x_i$$

Neural Network Training: Old & New Tricks

Old:

- **Back-propagation algorithm**
- **Stochastic Gradient Descent, Momentum, “weight decay”**

New: (last 5-6 years)

- **Dropout**
- **ReLUs**
- **Batch Normalization**

Training Objective for N training samples

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}^l_{k,m})^2$$

Per-sample loss

Per-layer regularization

Gradient descent: $\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$

(l,k,m) element of gradient vector:

$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Back-prop for i-th example

If N=10^6 , we will need to run back-prop 10^6 times to update **W** once!

Stochastic Gradient Descent (SGD)

Gradient: **Batch:** [1..N]

$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Noisy ('Stochastic') Gradient: **Minibatch:** B elements b(1), b(2),…, b(B): sampled from [1,N]

$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} \simeq \frac{1}{B}\sum_{i=1}^{B}\frac{\partial l(\mathbf{y}^{b(i)}, \hat{\mathbf{y}}^{b(i)})}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Epoch: N samples, N/B batches

Code example

Gradient Descent

vs

Stochastic Gradient Descent

Regularization in SGD: Weight Decay

Gradient: **Batch:** [1..N]

$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Noisy ('Stochastic') Gradient: **Minibatch:** B elements b(1), b(2),…, b(B): sampled from [1,N]

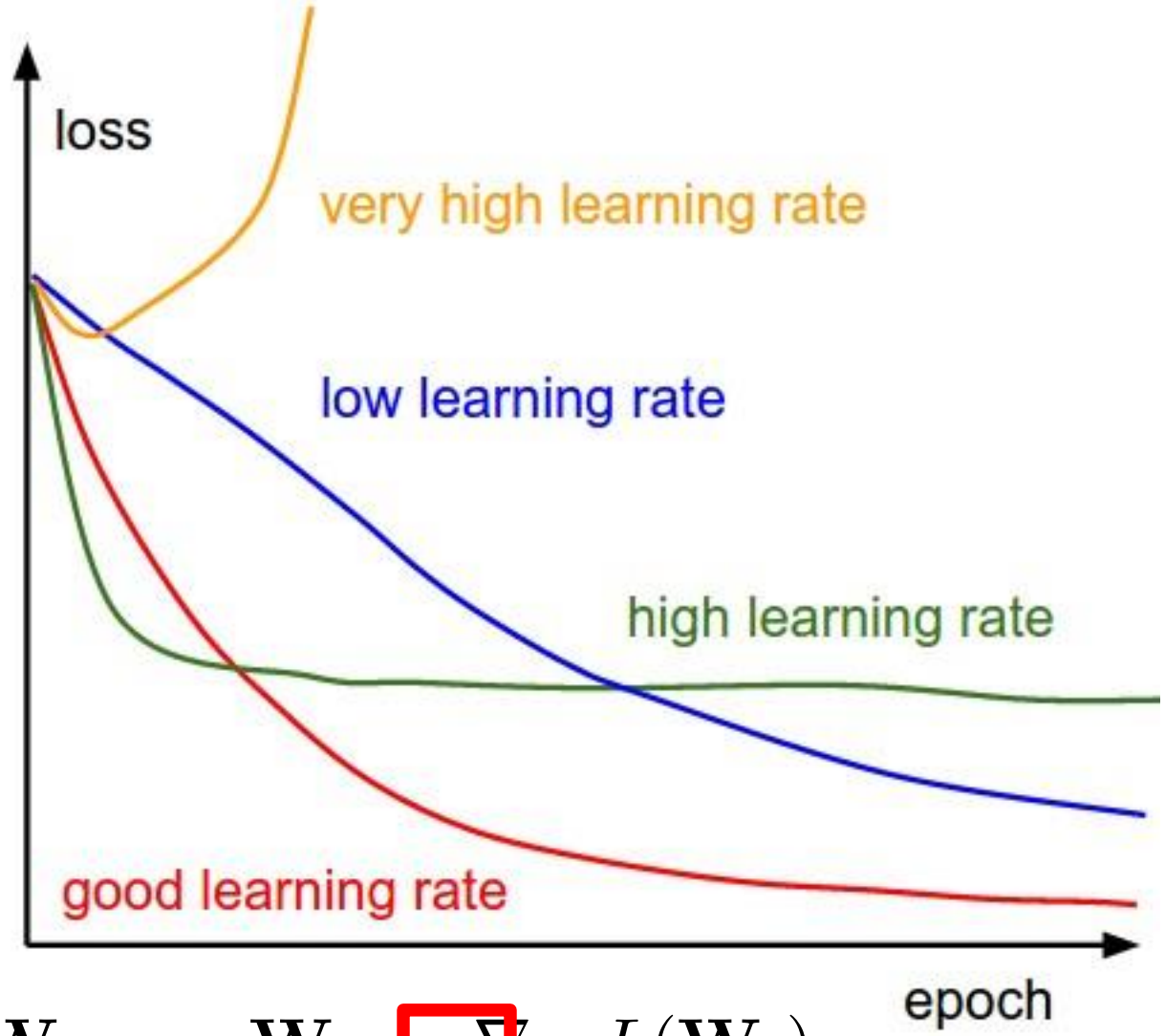
$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} \simeq \frac{1}{B}\sum_{i=1}^{B}\frac{\partial l(\mathbf{y}^{b(i)}, \hat{\mathbf{y}}^{b(i)})}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Back-prop on minibatch

"Weight decay"

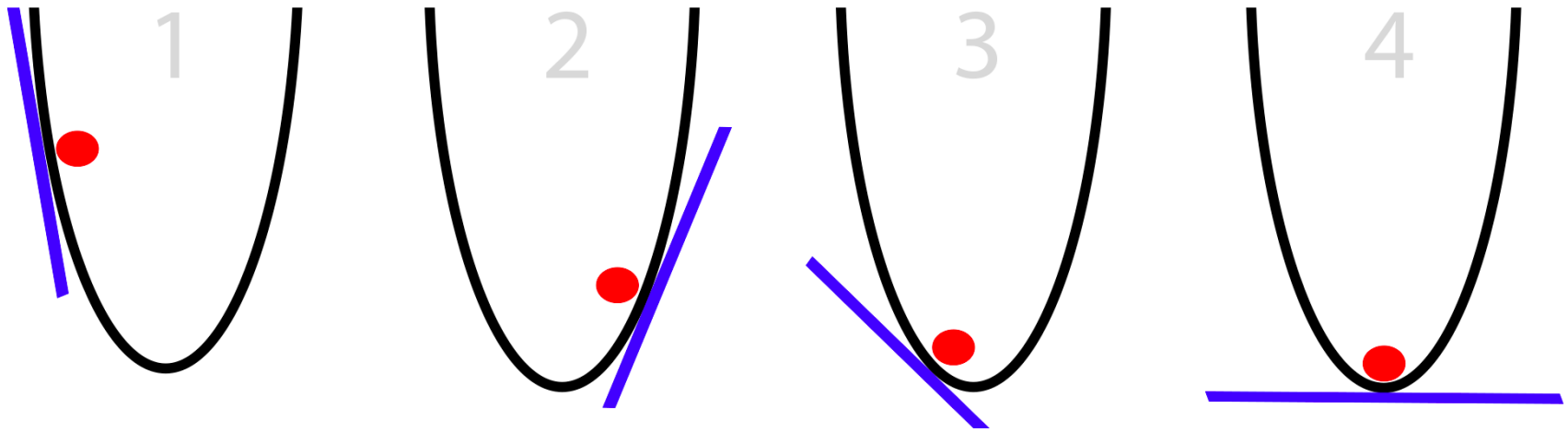
Epoch: N samples, N/B batches

Learning rate

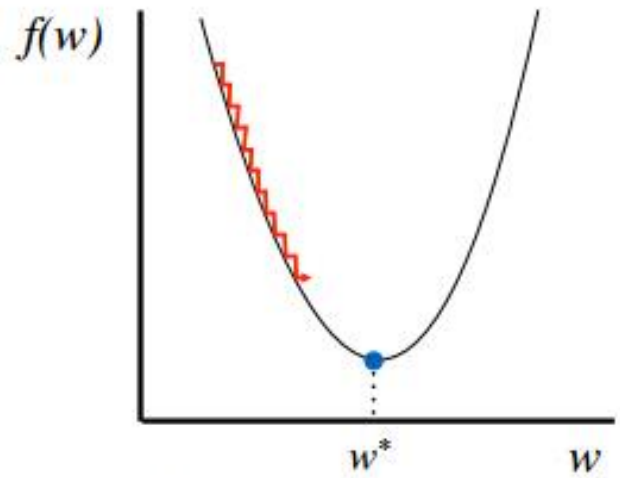


$$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

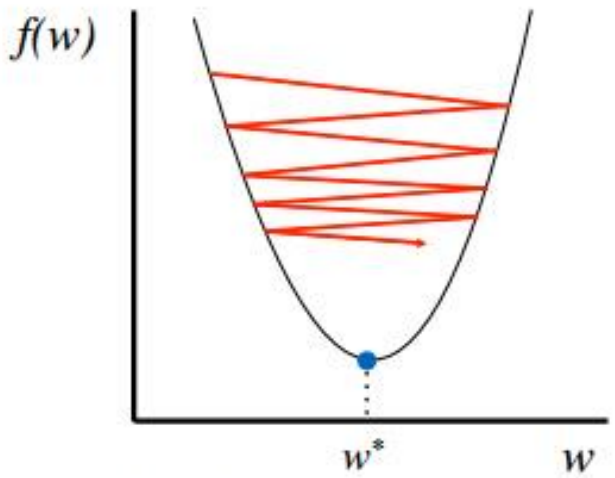
Gradient Descent



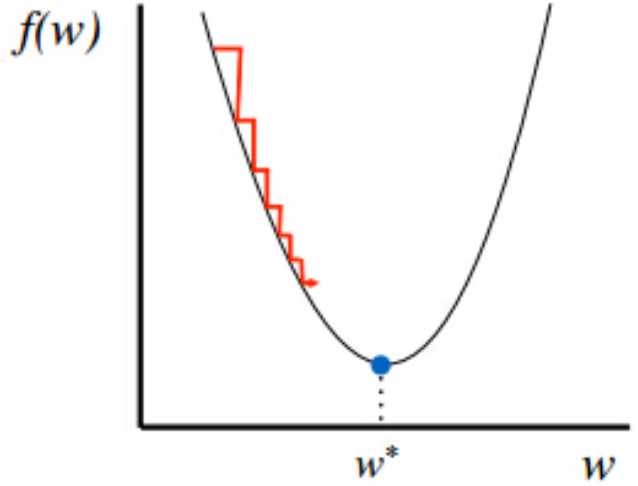
(S)GD with adaptable stepsize



Too small: converge very slowly



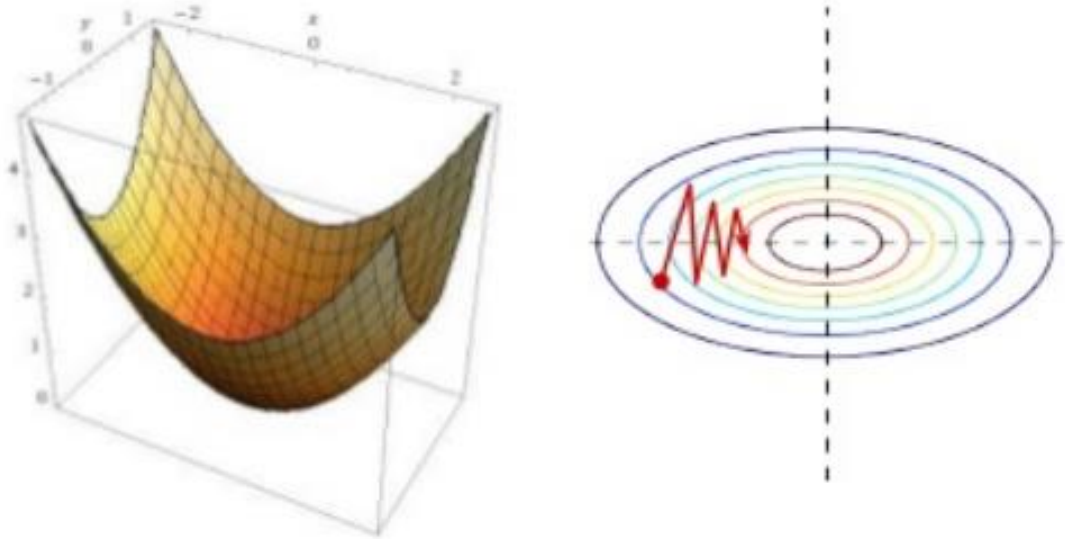
Too big: overshoot and even diverge



Reduce size over time

e.g. $\epsilon_t = \frac{c}{t}$

(S)GD with momentum



Main idea: retain long-term trend of updates, drop oscillations

(S)GD $$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \nabla_{\mathbf{W}} L(\mathbf{W})$$

(S)GD + momentum

$$\mathbf{V}_{t+1} = \mu \mathbf{V}_t + (1-\mu)\nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

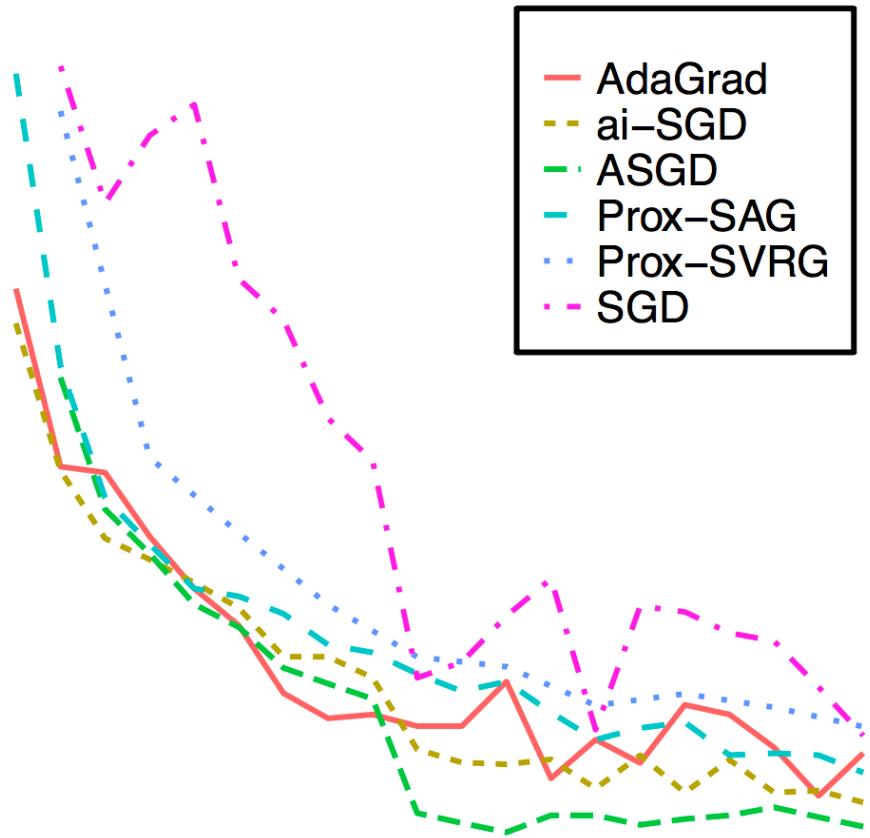
$$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \mathbf{V}_{t+1}$$

Code example

Multi-layer perceptron classification

Step-size Selection & Optimizers: research problem

- Nesterov's Accelerated Gradient (NAG)
- R-prop
- AdaGrad
- RMSProp
- AdaDelta
- Adam
- …



Neural Network Training: Old & New Tricks

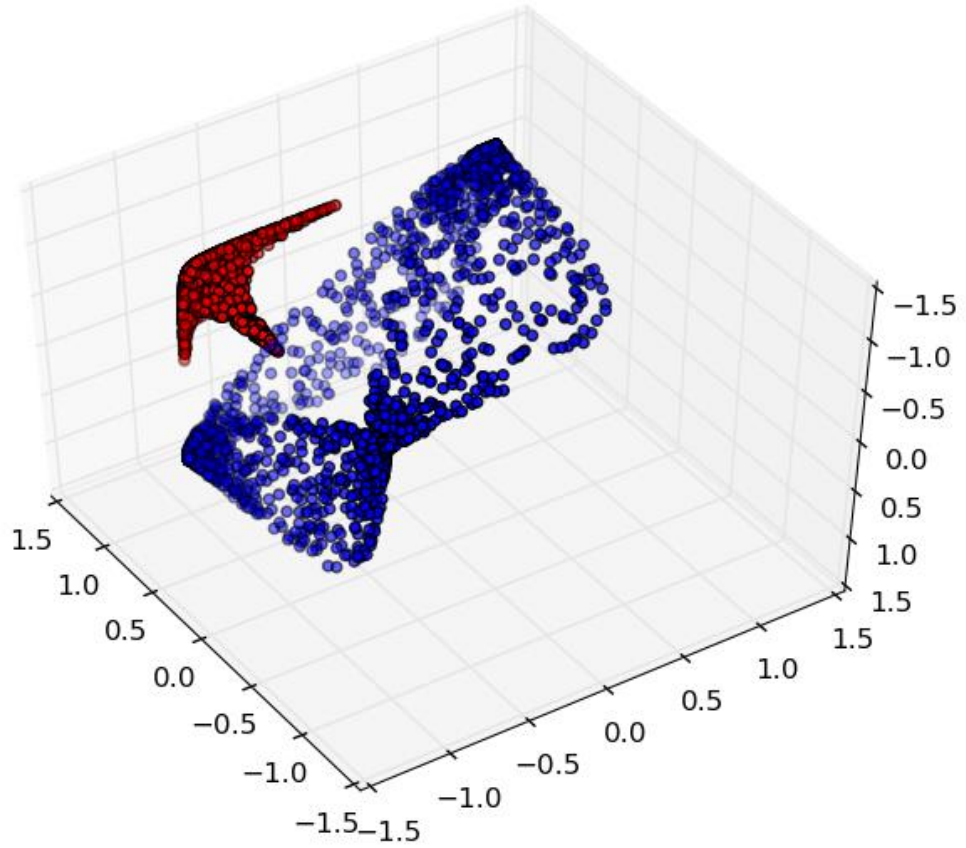
Old: (80's)

- **Stochastic Gradient Descent, Momentum, "weight decay"**

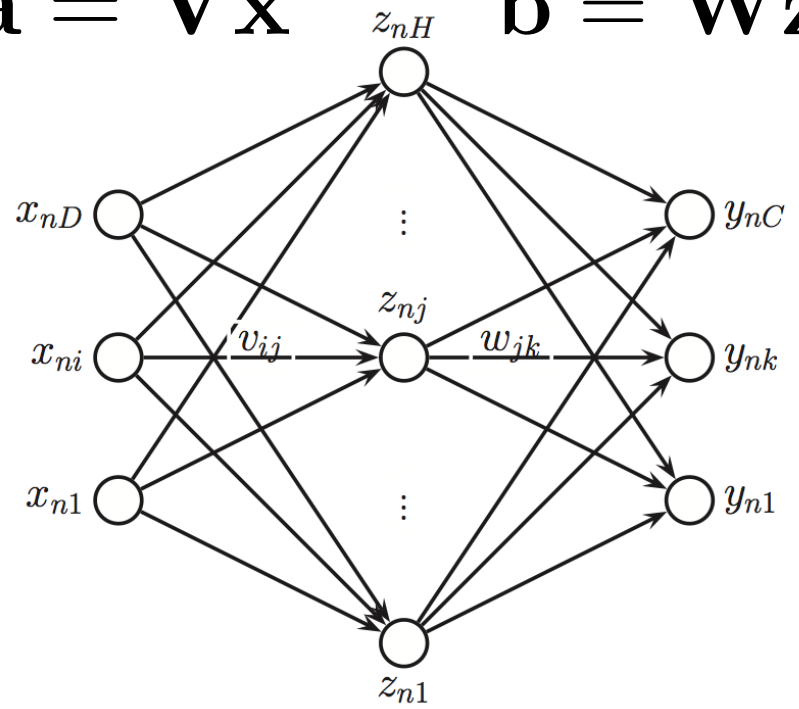
New: (last 5-6 years)

- **Dropout**
- **ReLUs**
- **Batch Normalization**

Linearization: may need higher dimensions



$$\mathbf{a} = \mathbf{V}\mathbf{x} \qquad \mathbf{b} = \mathbf{W}\mathbf{z}$$



http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

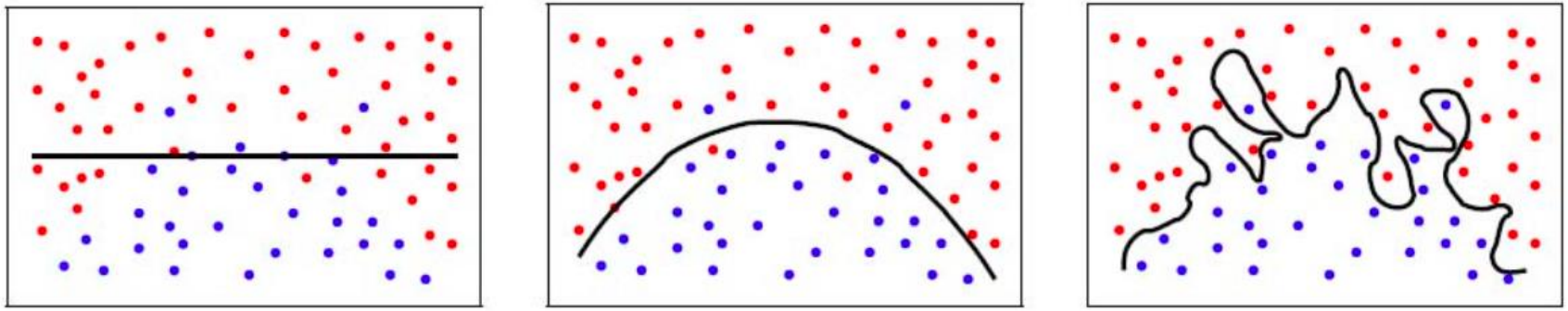
Reminder: Overfitting, in images

Classification

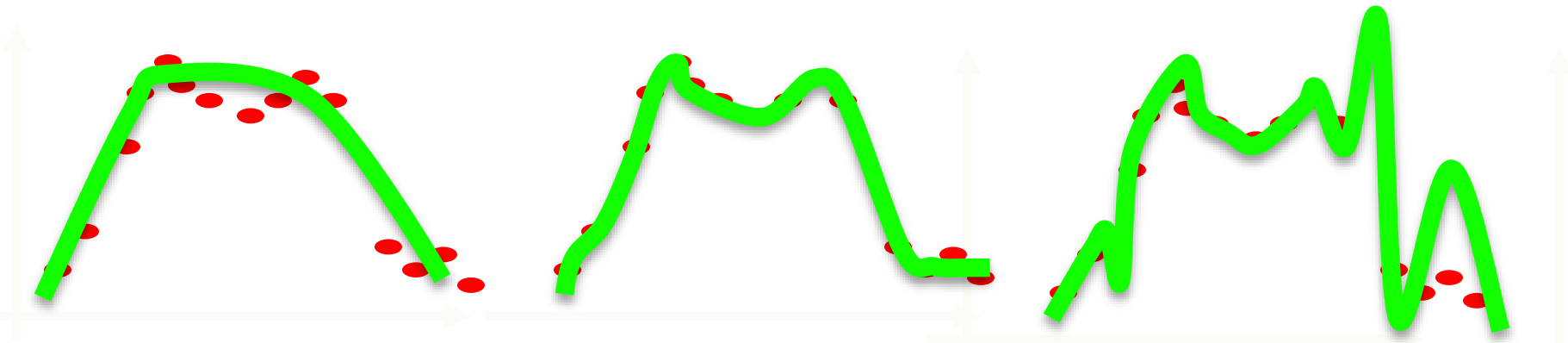
Underfitting

just right

Overfitting



Regression



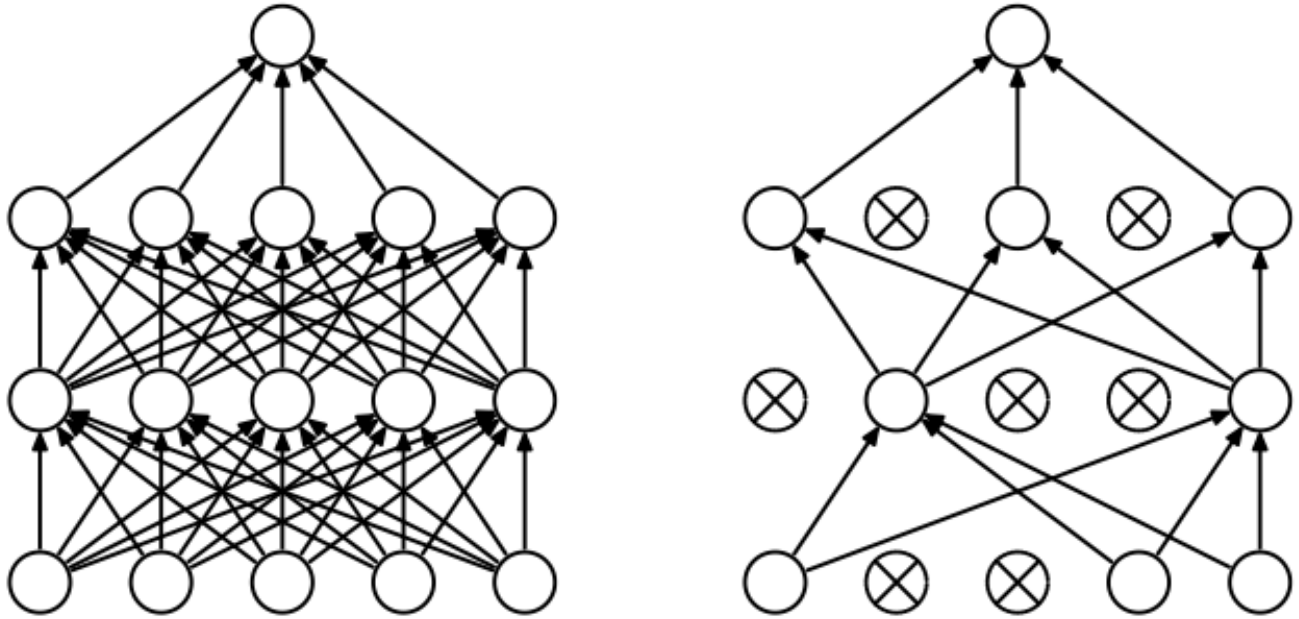
Previously: l2 Regularization

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}_{k,m}^l)^2$$

Per-sample loss

Per-layer regularization

Dropout



(a) Standard Neural Net

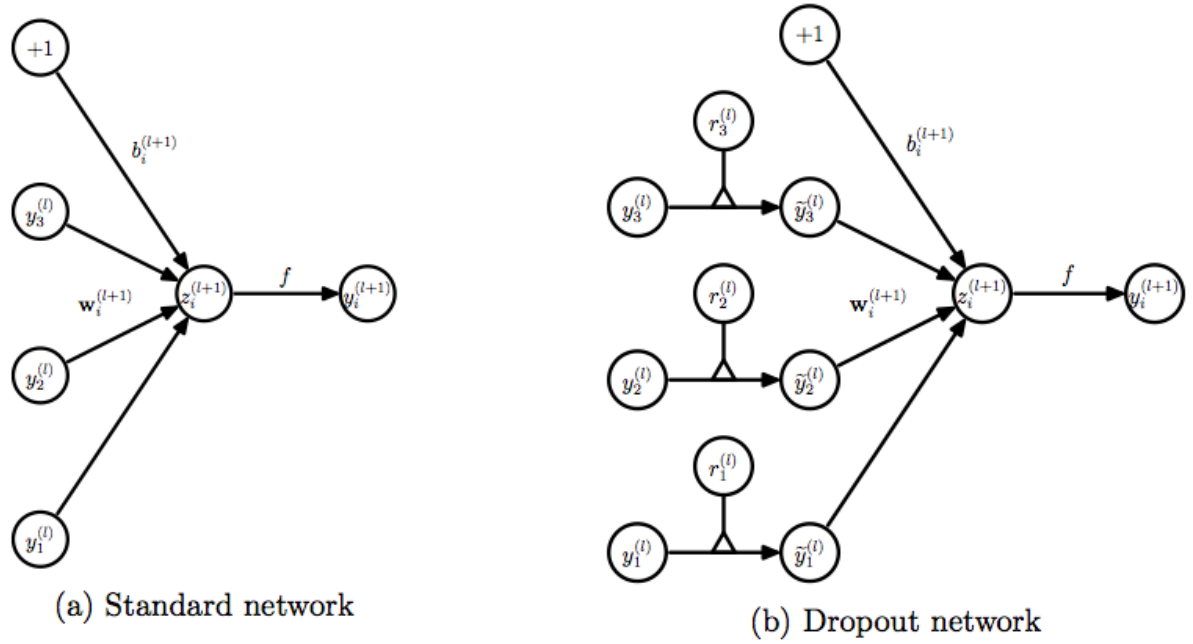
(b) After applying dropout.

Each sample is processed by a ‘decimated’ neural net

Decimated nets: distinct classifiers

But: they should all do the same job

Dropout block



(a) Standard network (b) Dropout network

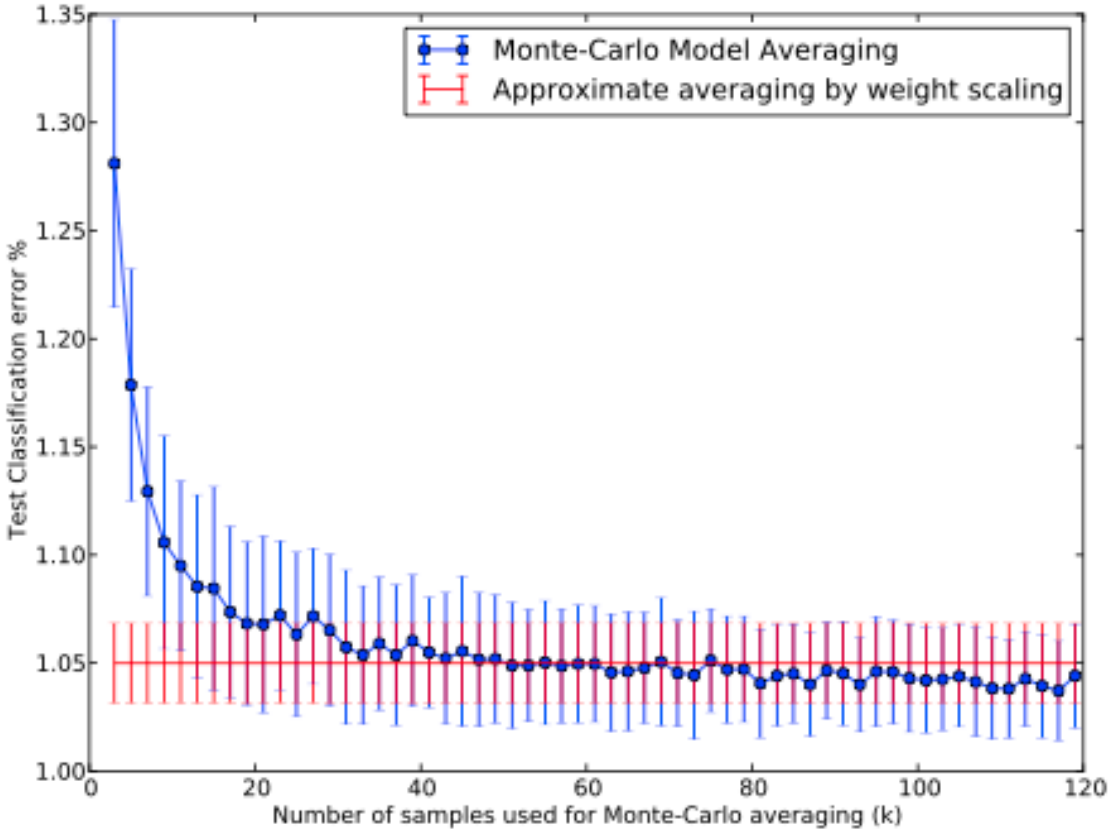
Figure 3: Comparison of the basic operations of a standard and dropout network.

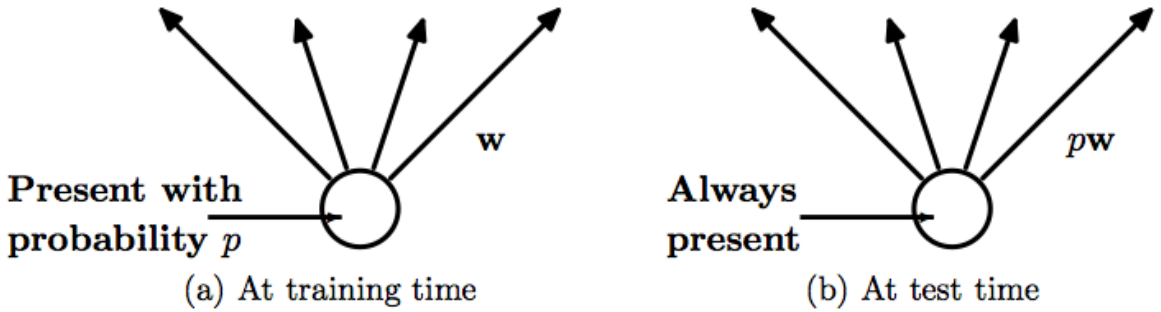
$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}),
\end{aligned}
$$

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\widetilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \widetilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$

‘Feature noising’

Test time: Deterministic Approximation





(a) At training time (b) At test time

Dropout Performance

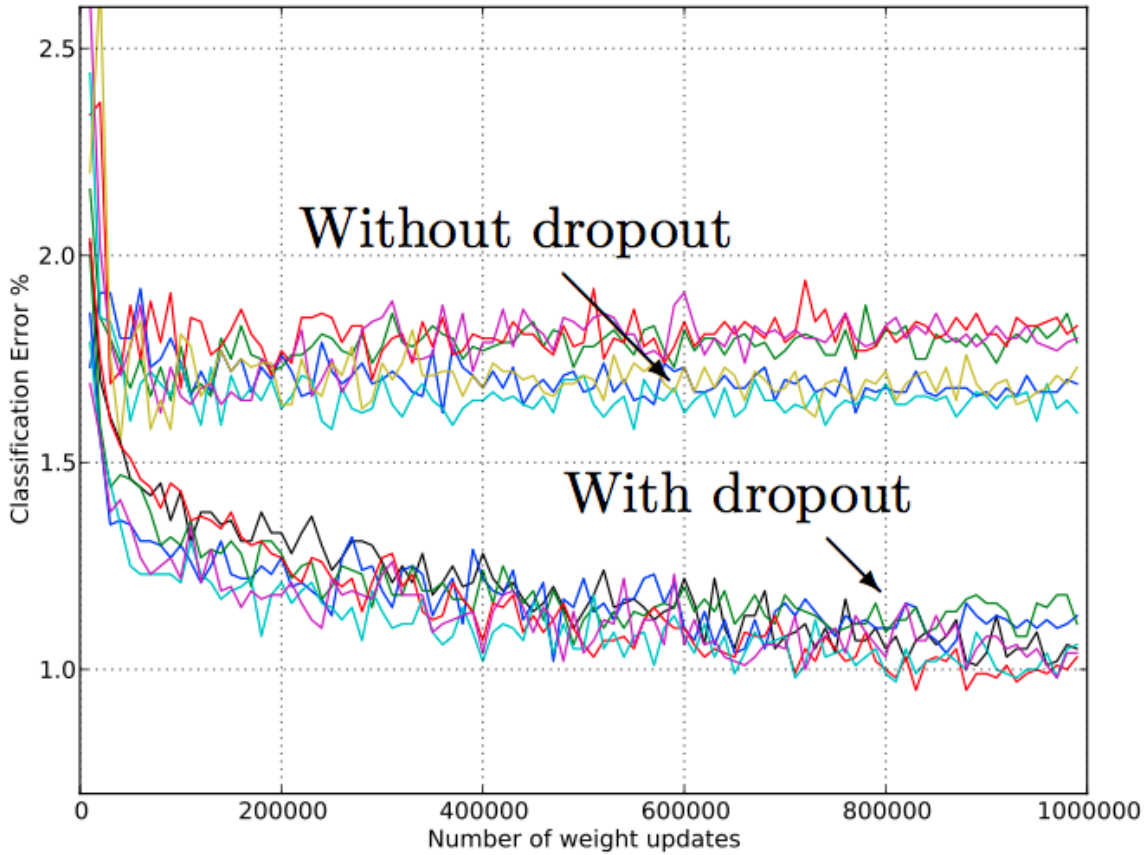


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Neural Network Training: Old & New Tricks

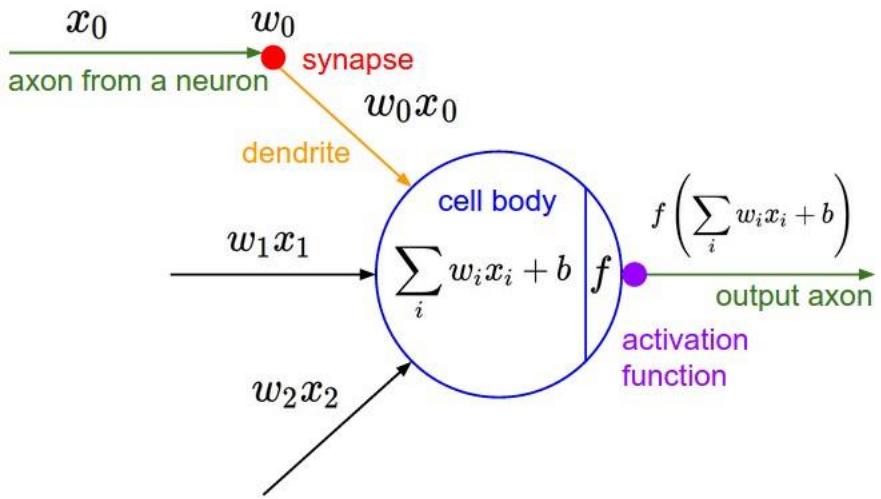
Old: (80's)

- **Stochastic Gradient Descent, Momentum, "weight decay"**

New: (last 5-6 years)

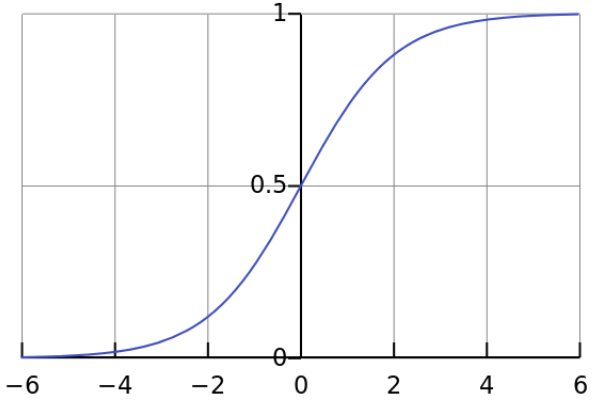
- **Dropout**
- **ReLUs**
- **Batch Normalization**

‘Neuron’: Cascade of Linear and Nonlinear Function



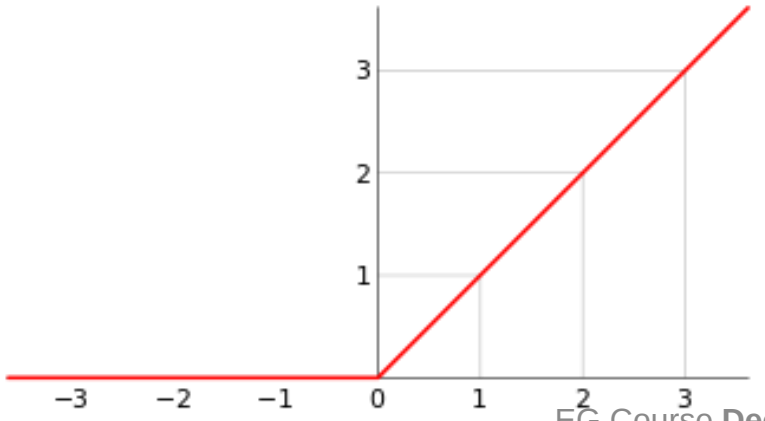
Sigmoidal (“logistic”)

$$g(a) = \frac{1}{1 + \exp(-a)}$$



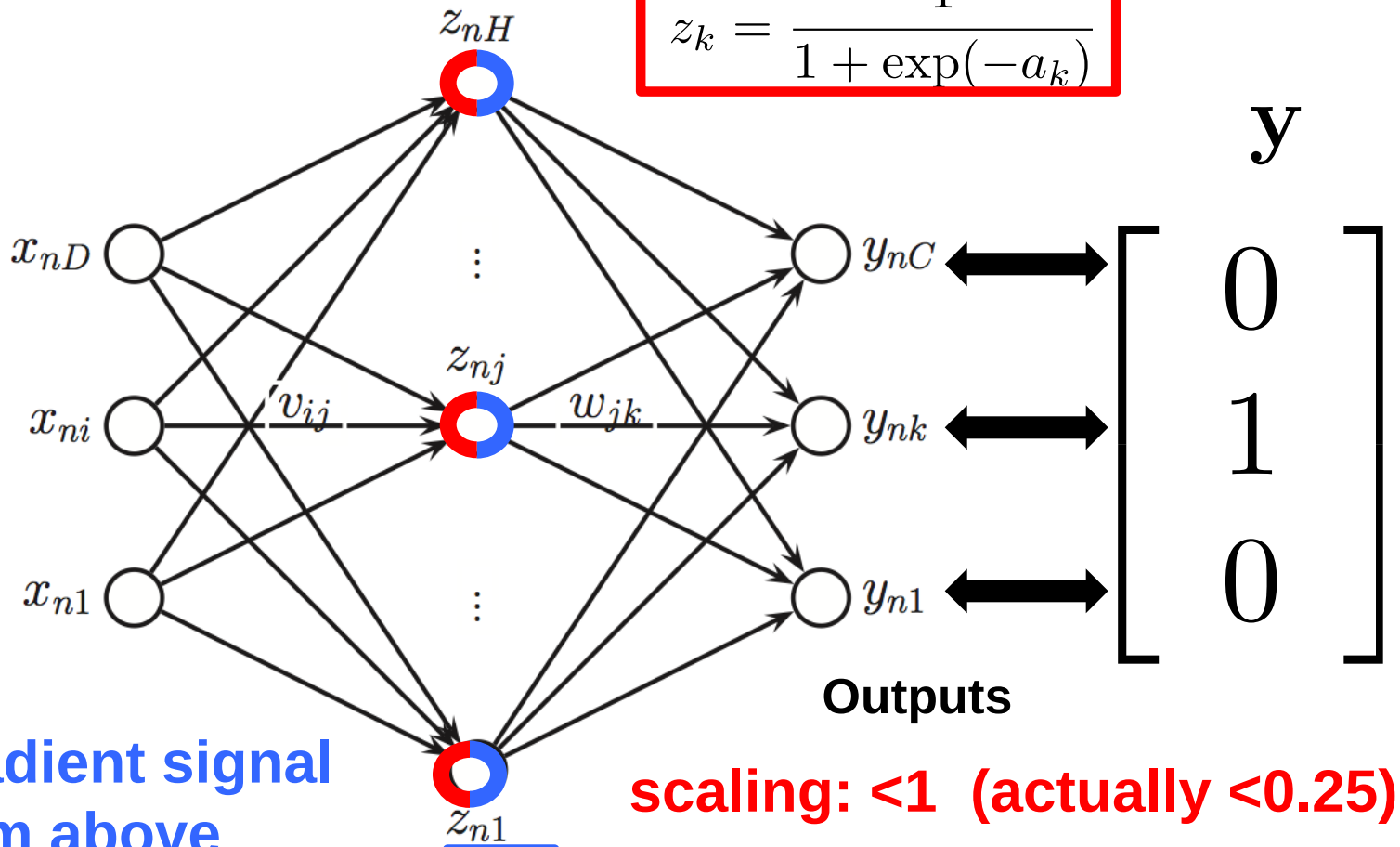
Rectified Linear Unit (RELU)

$$g(a) = \max(0, a)$$



Reminder: a network in backward mode

$$z_k = \frac{1}{1 + \exp(-a_k)}$$



Gradient signal from above

scaling: <1 (actually <0.25)

$$\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m}\frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_k} g'(a_k) = \frac{\partial l}{\partial z_k} g(a_k)(1 - g(a_k))$$

Vanishing Gradients Problem

Gradient signal from above

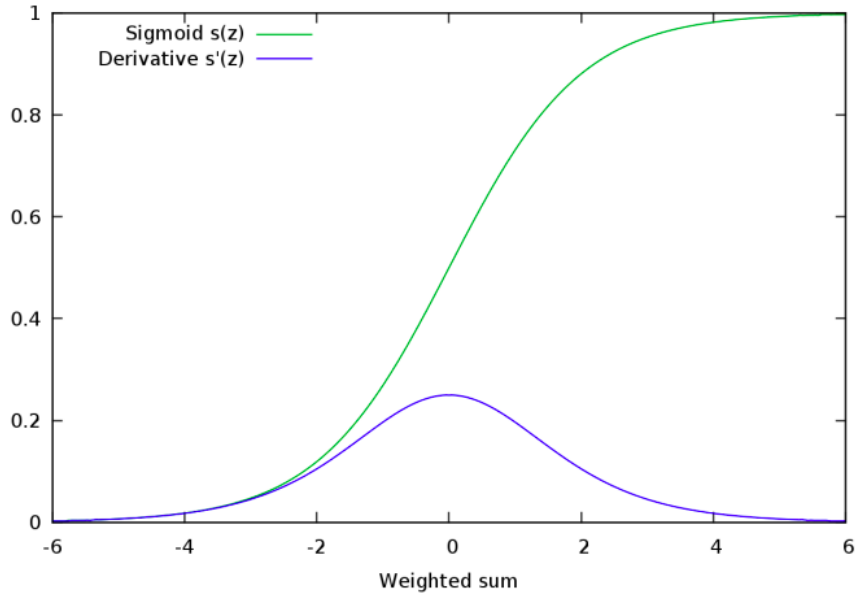
$$\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m}\frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_k} g'(a_k) = \frac{\partial l}{\partial z_k} g(a_k)(1 - g(a_k))$$

scaling: <1 (actually <0.25)

Do this 10 times: updates in the first layers get minimal

Top layer knows what to do, lower layers "don't get it"

Sigmoidal Unit: Signal is not getting through!



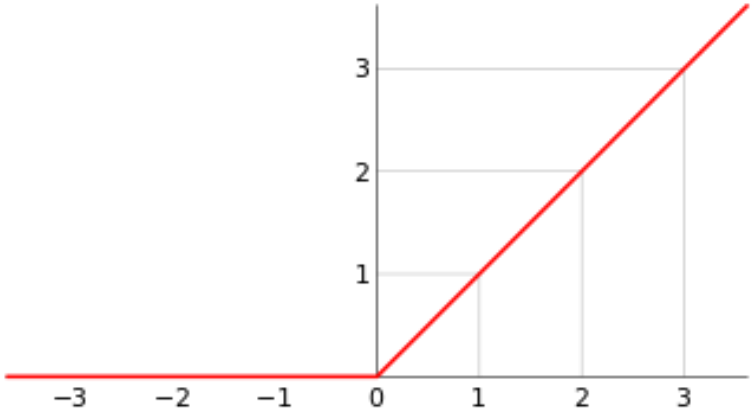
Vanishing Gradients Problem: ReLU Solves It

Gradient signal from above

Scaling: {0,1}

$$\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m}\frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_k} g'(a_k)$$

$$g(a) = \max(0, a)$$



$$g'(a) = \begin{cases} 1 & a > 0 \\ 0 & a < 0 \end{cases}$$

Neural Network Training: Old & New Tricks

Old: (80's)

- **Stochastic Gradient Descent, Momentum, "weight decay"**

New: (last 5-6 years)

- **Dropout**
- **ReLUs**
- **Batch Normalization**

External Covariate Shift: your input changes

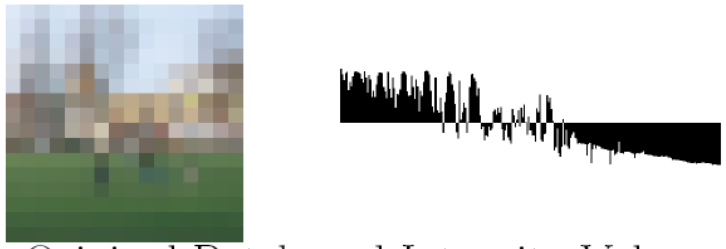
10 am

2pm

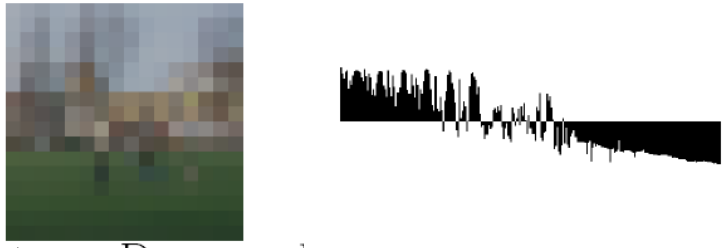
7pm

“Whitening”: Set Mean = 0, Variance = 1

Photometric transformation: $I \rightarrow a\,I + b$



Original Patch and Intensity Values



Brightness Decreased

Contrast increased,

- Make each patch have zero mean:

$$\mu = \frac{1}{N}\sum_{x,y} I(x,y)$$

$$Z(x,y) = I(x,y) - \mu$$

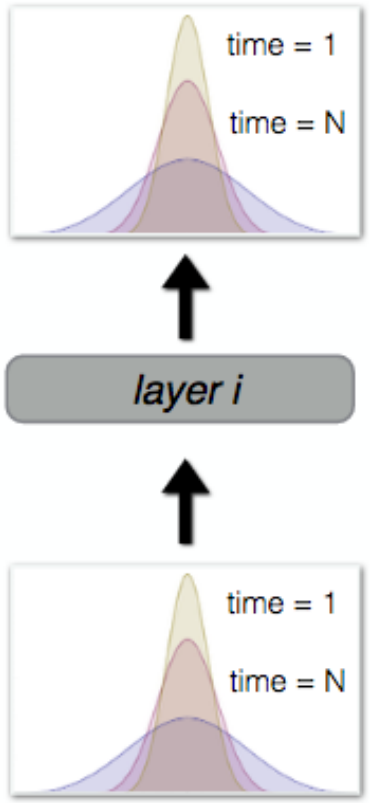
- Then make it have unit variance:

$$\sigma^2 = \frac{1}{N}\sum_{x,y} Z(x,y)^2$$

$$ZN(x,y) = \frac{Z(x,y)}{\sigma}$$

Internal Covariate Shift

Neural network activations during training: moving target



Batch Normalization

Whiten-as-you-go:

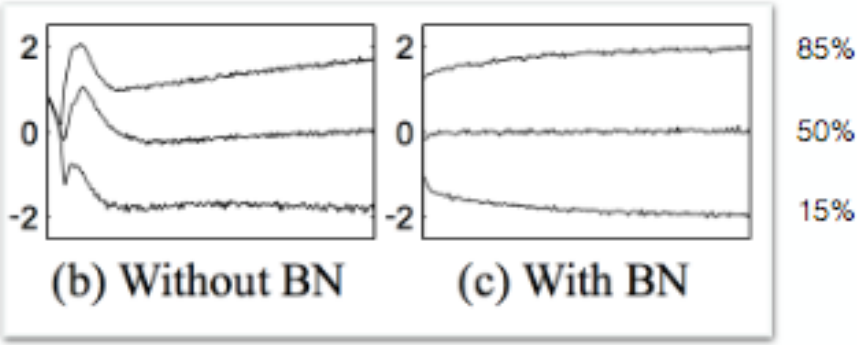
- Normalize the activations in each layer within a mini-batch.
- Learn the mean and variance (γ, β) of each layer as parameters

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

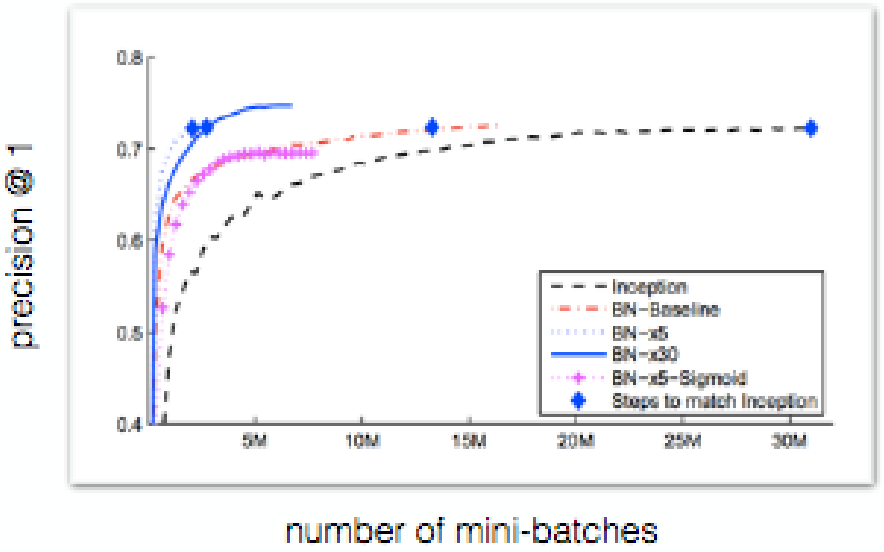
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
S Ioffe and C Szegedy (2015)

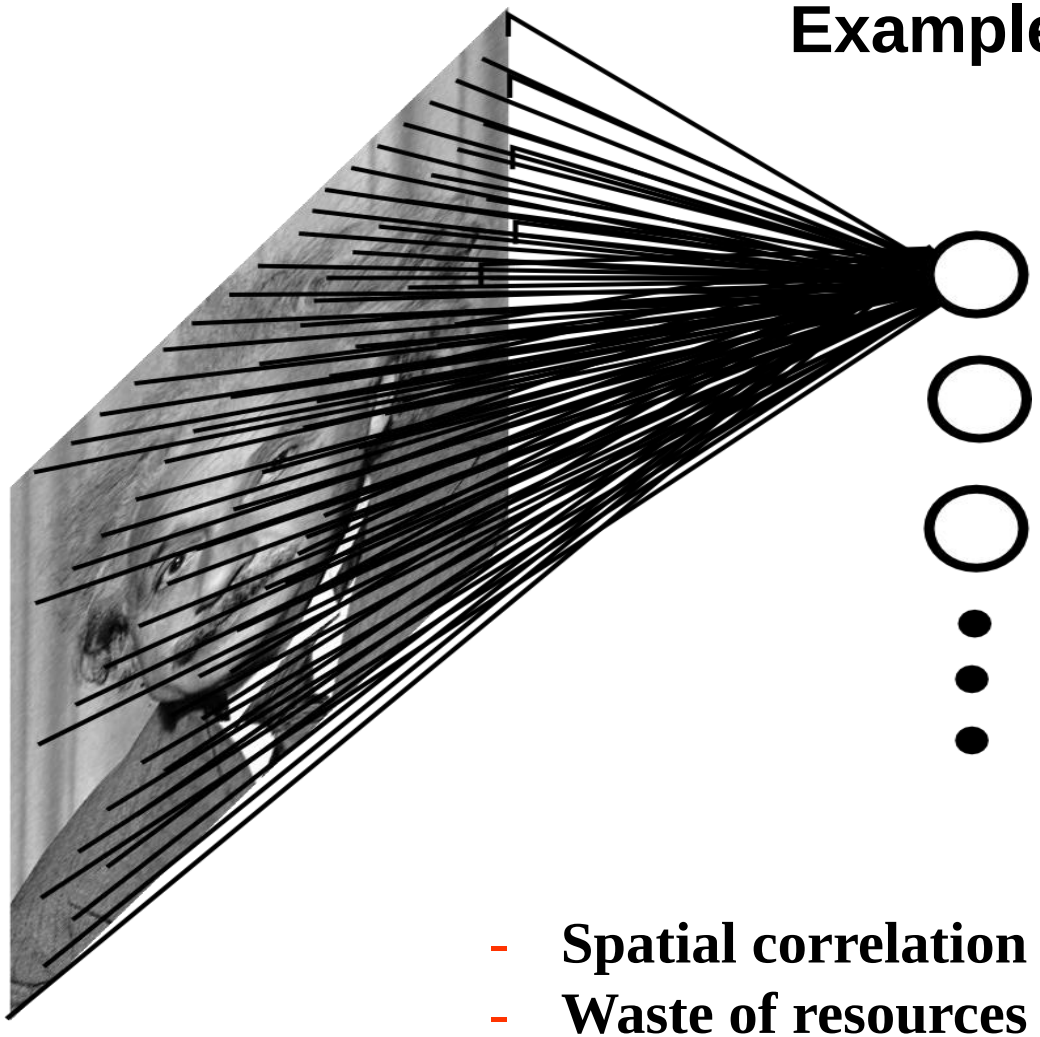
Batch Normalization: used in all current systems

- Multi-layer CNN's train faster with fewer data samples (15x).
- Employ faster learning rates and less network regularizations.
- Achieves state of the art results on ImageNet.



Convolutional Neural Networks

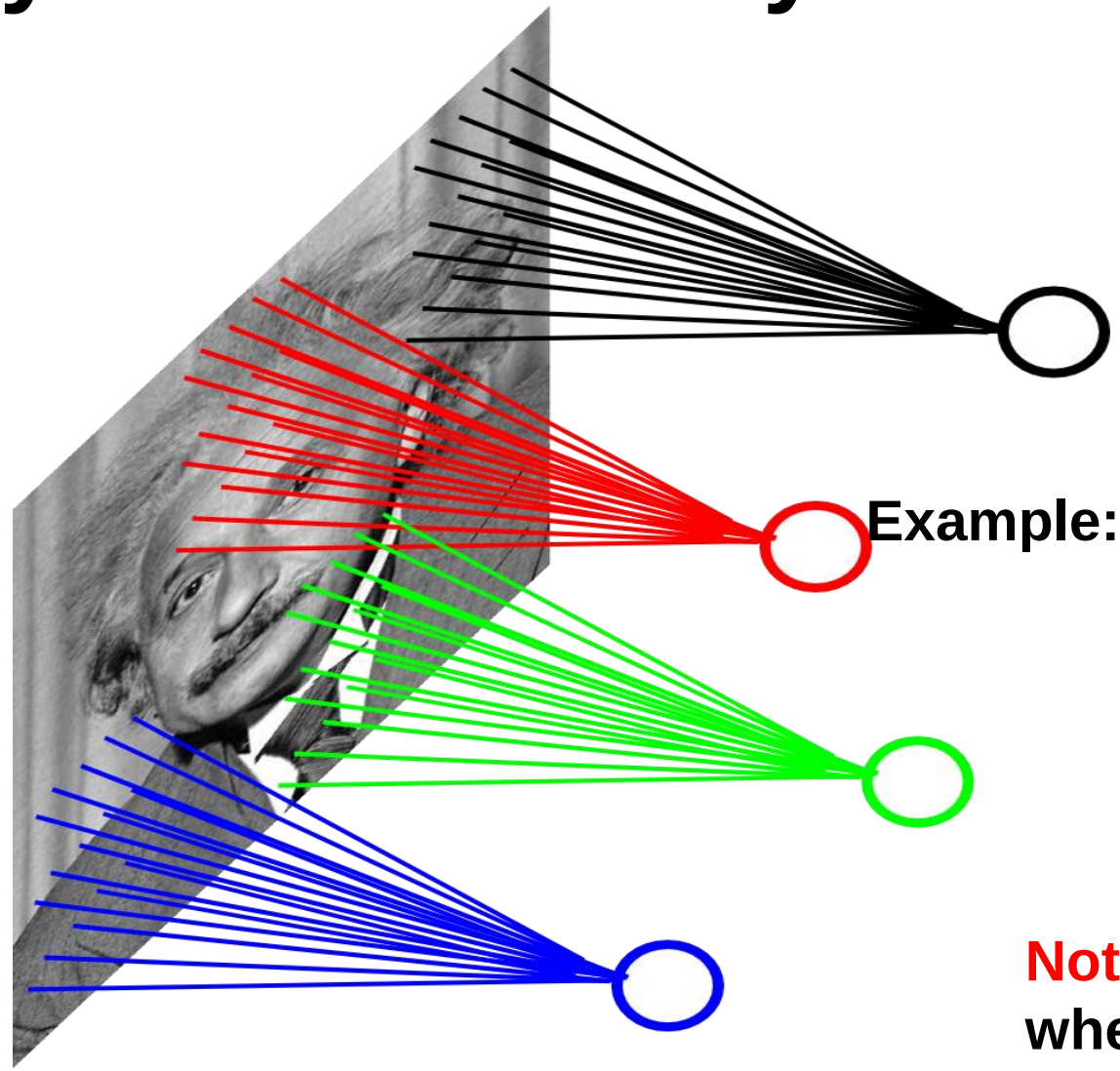
Fully-connected Layer



Example: 200x200 image
40K hidden units
~2B parameters!!!

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

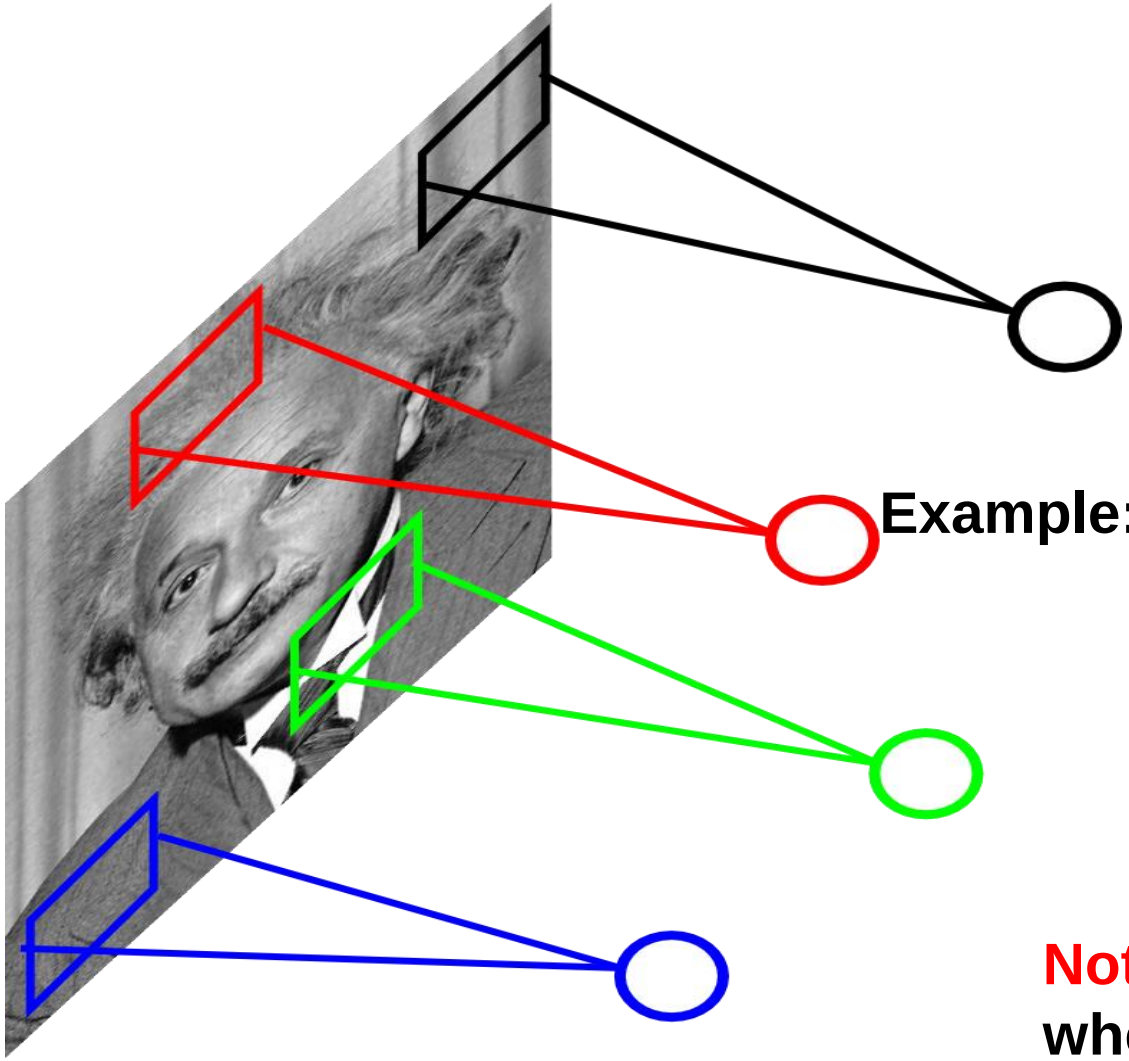
Locally-connected Layer



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

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

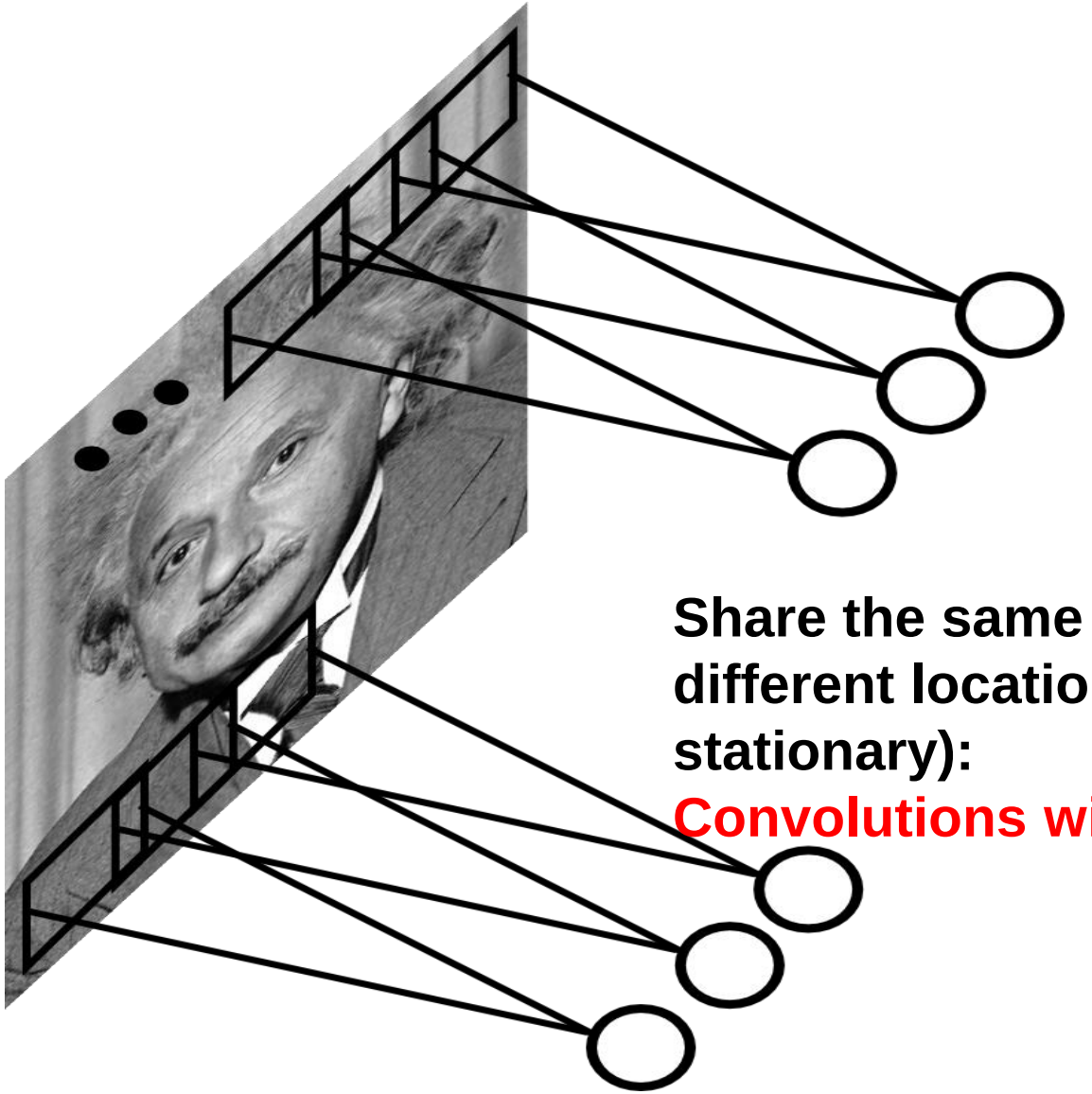
Locally-connected Layer



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

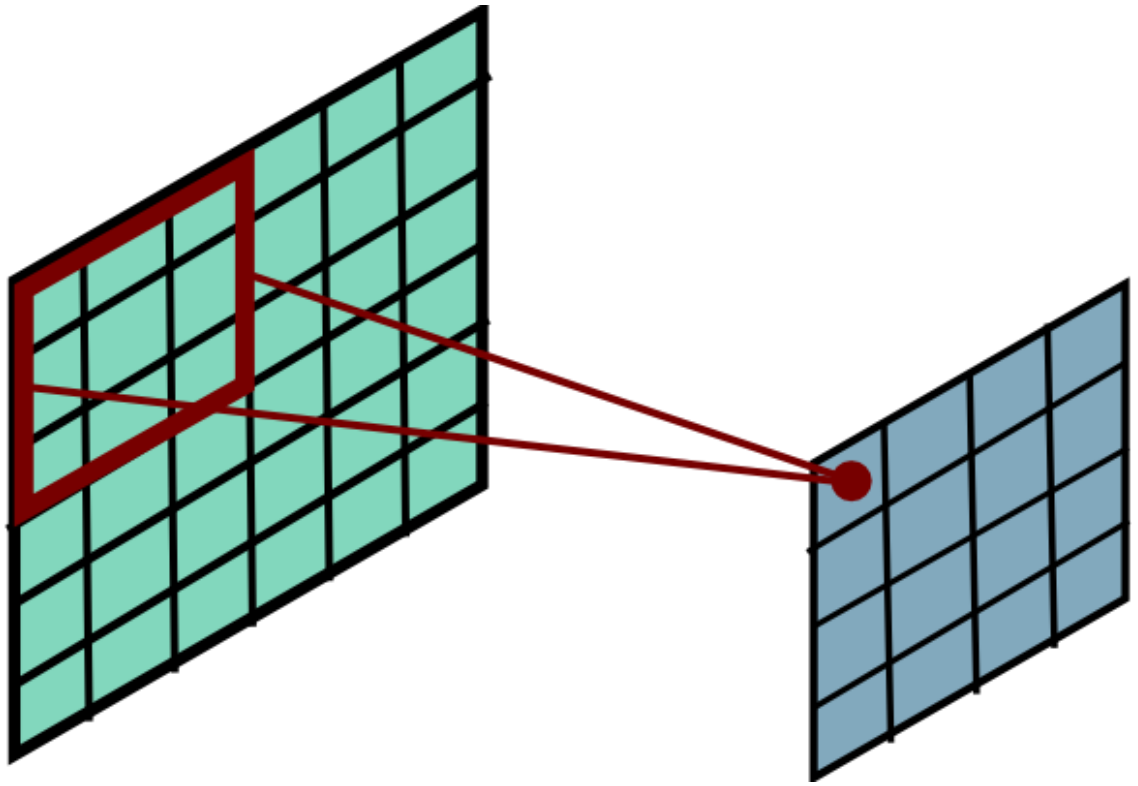
Note: This parameterization is good when input image is registered (e.g., face recognition).

Convolutional Layer

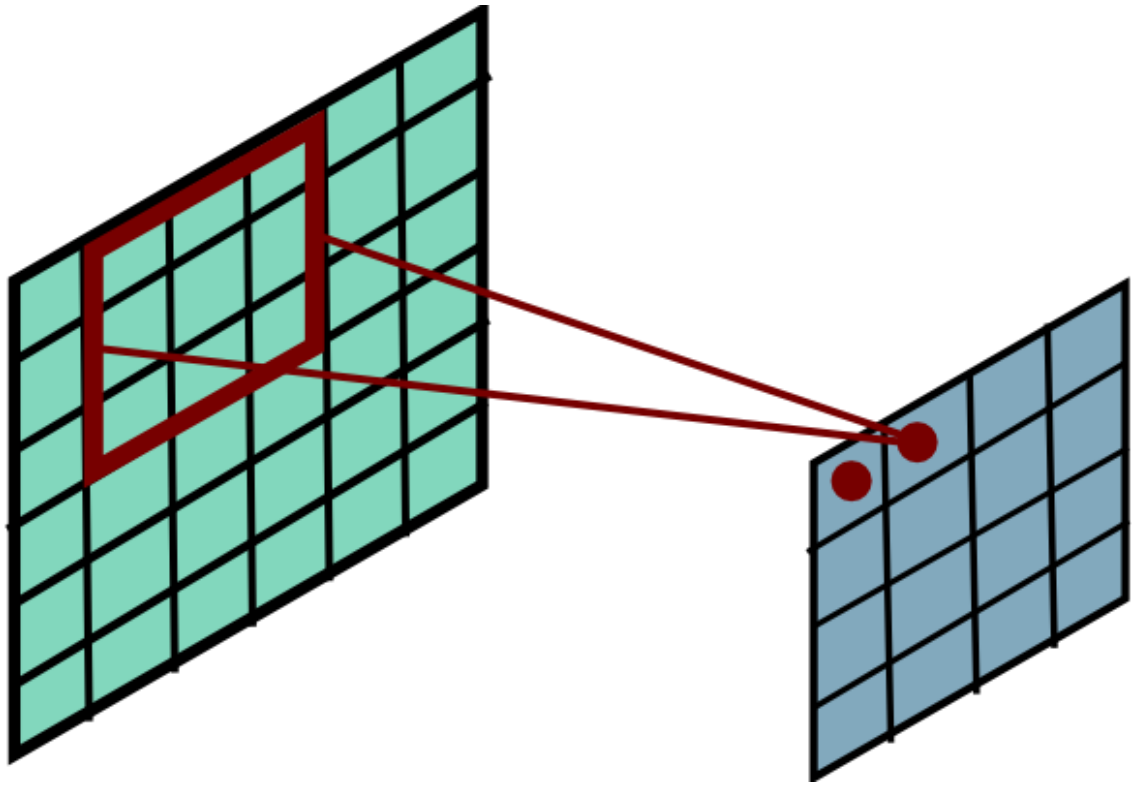


Share the same parameters across different locations (assuming input is stationary):
Convolutions with learned kernels

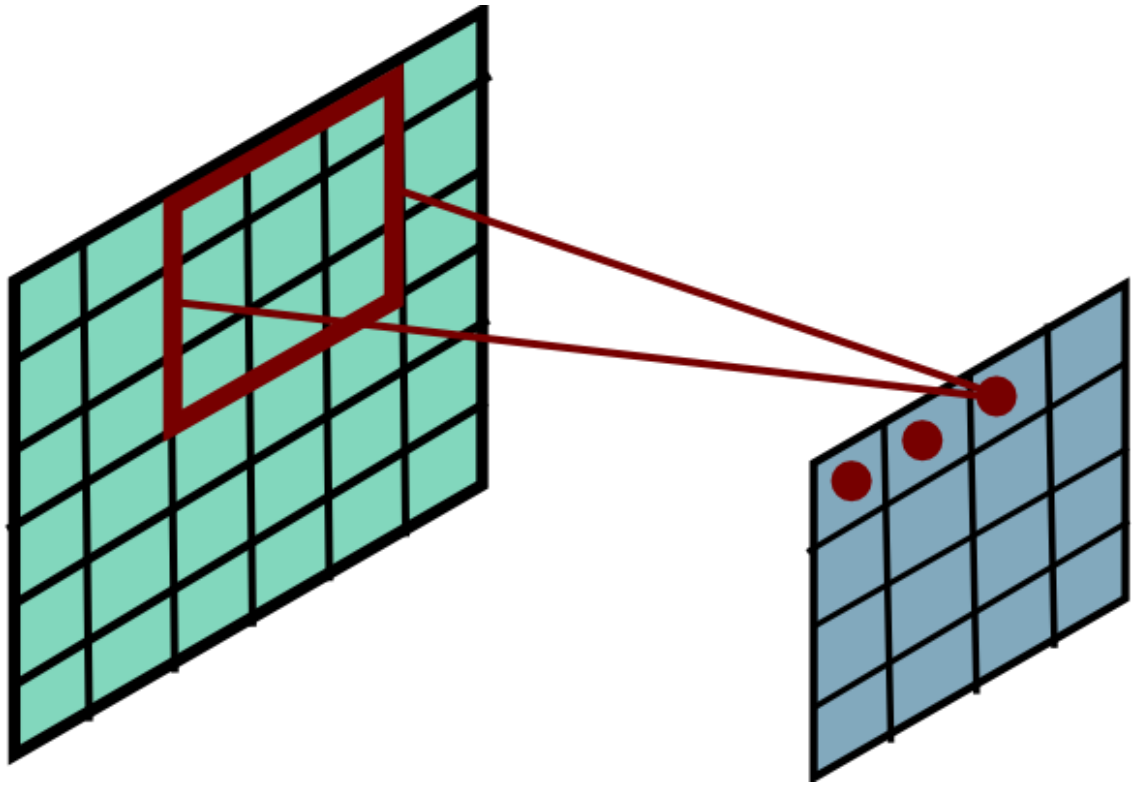
Convolutional Layer



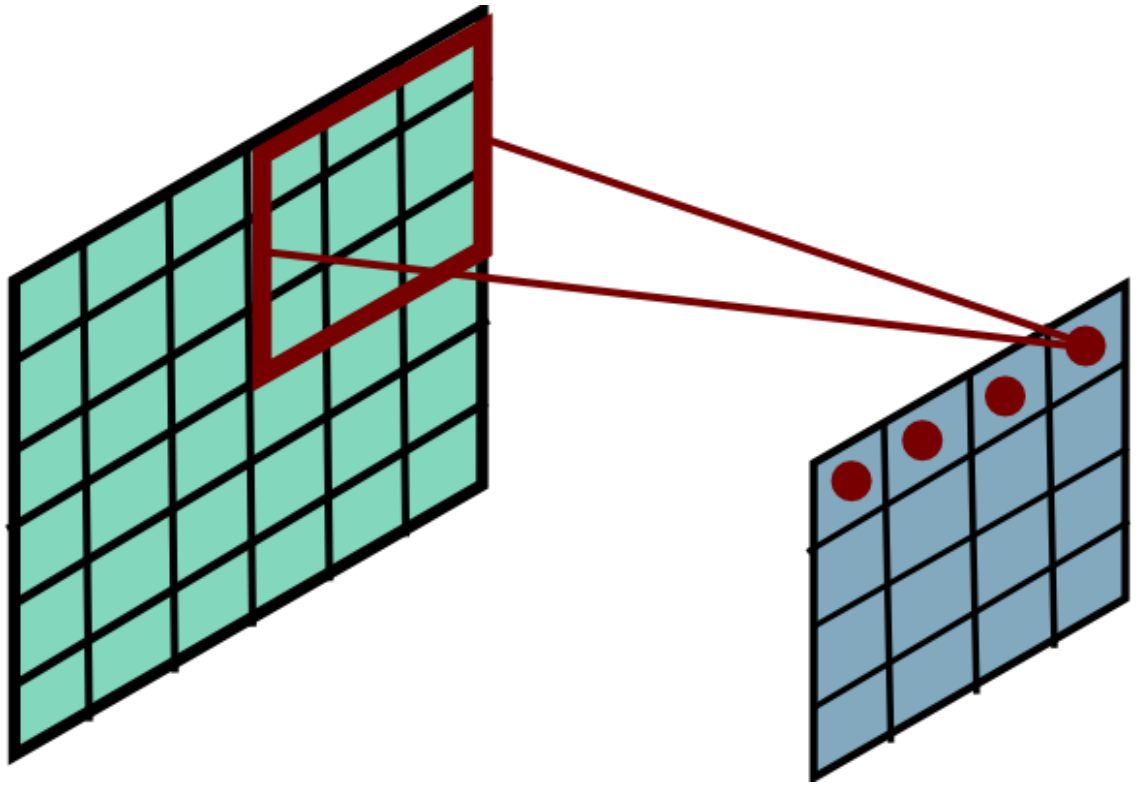
Convolutional Layer



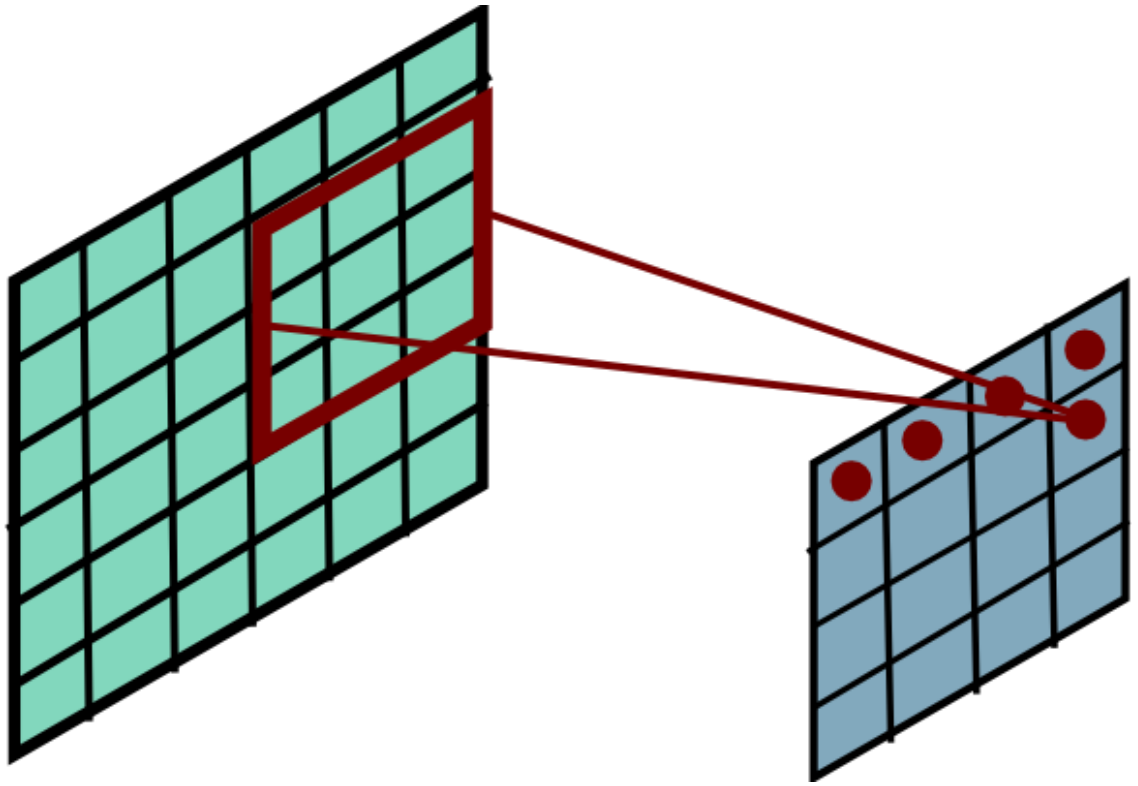
Convolutional Layer



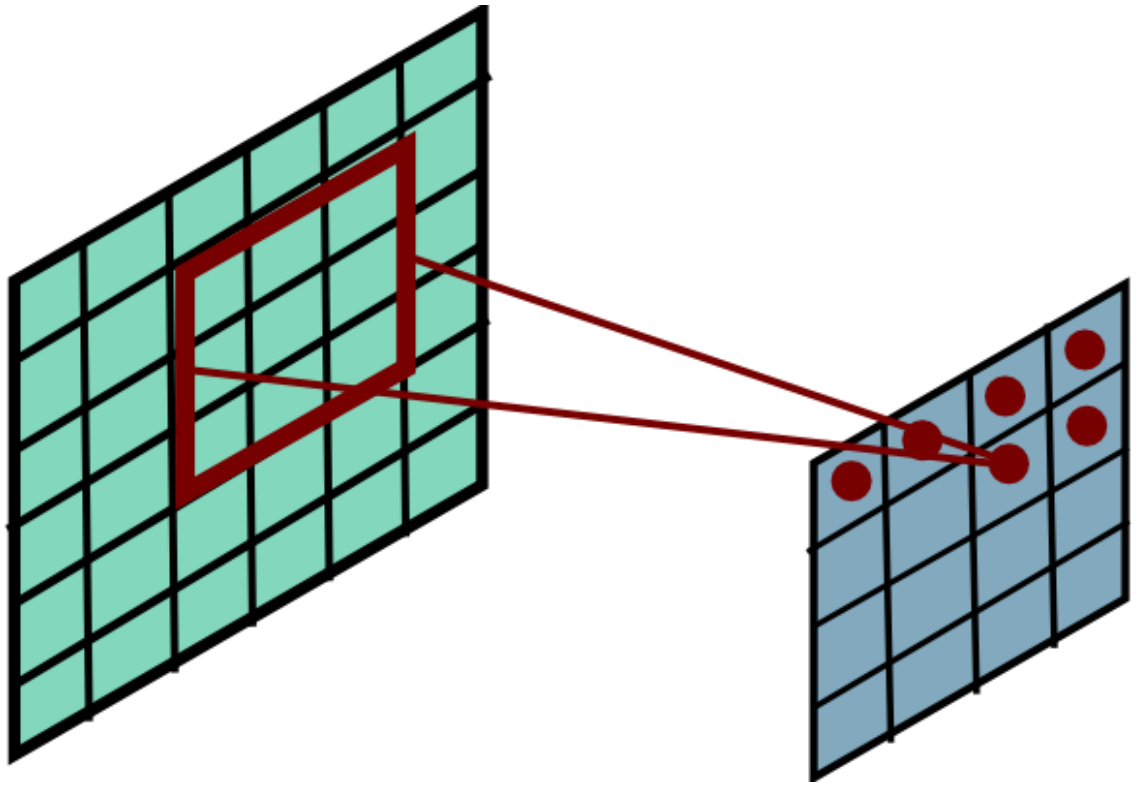
Convolutional Layer



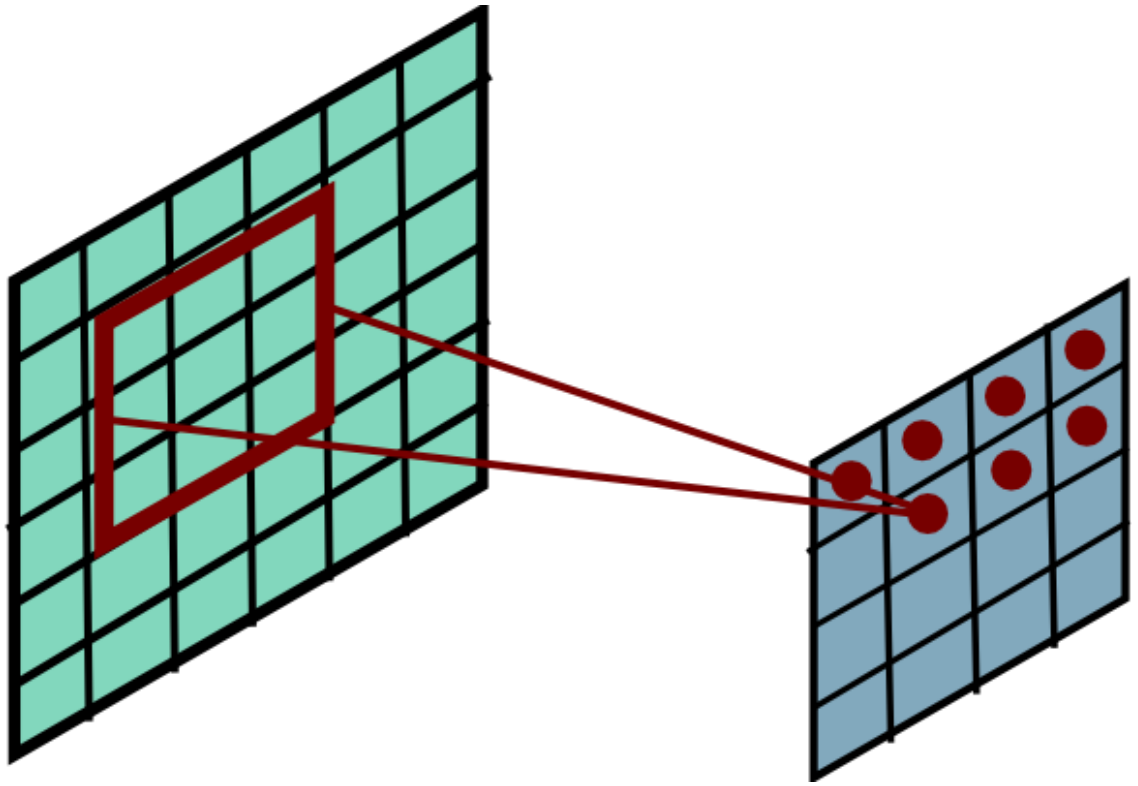
Convolutional Layer



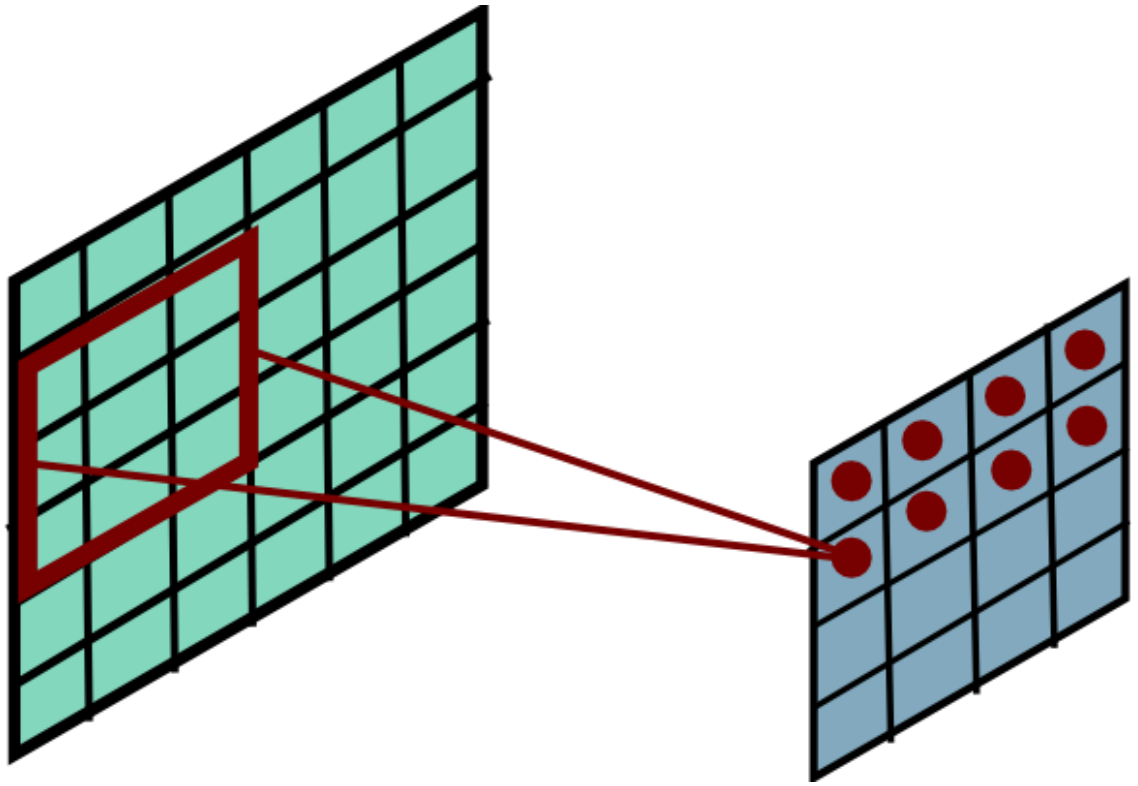
Convolutional Layer



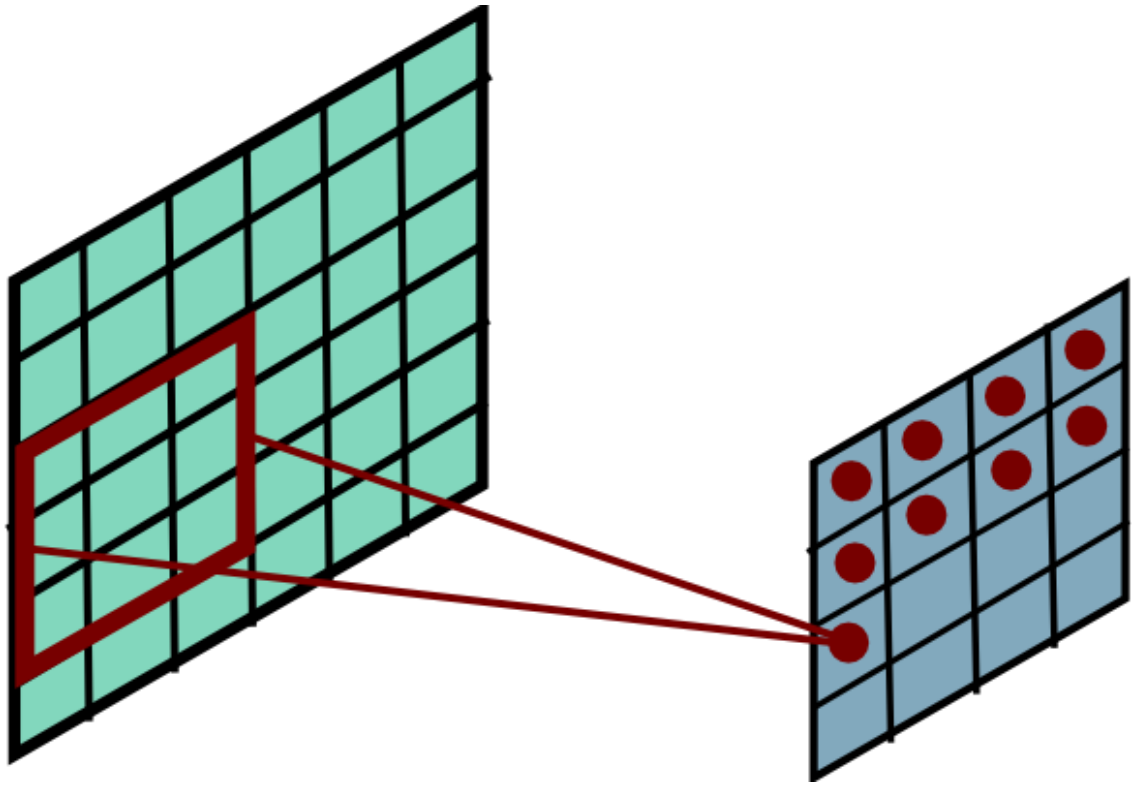
Convolutional Layer



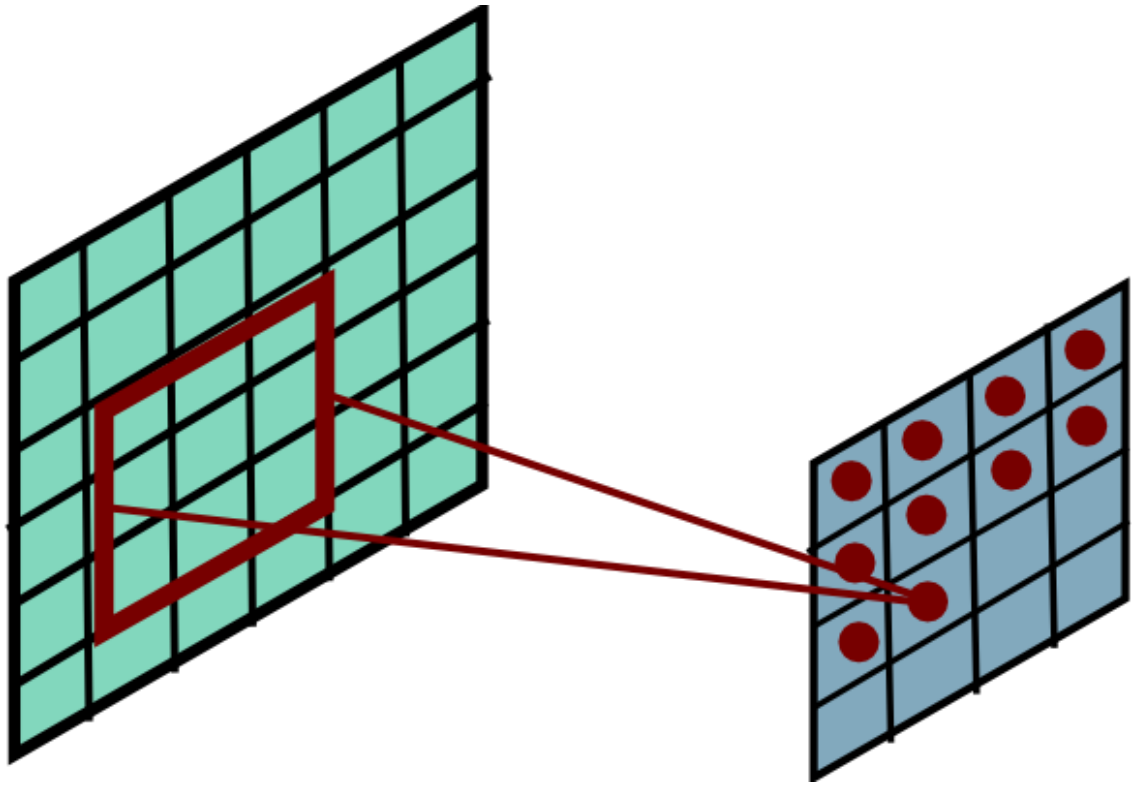
Convolutional Layer



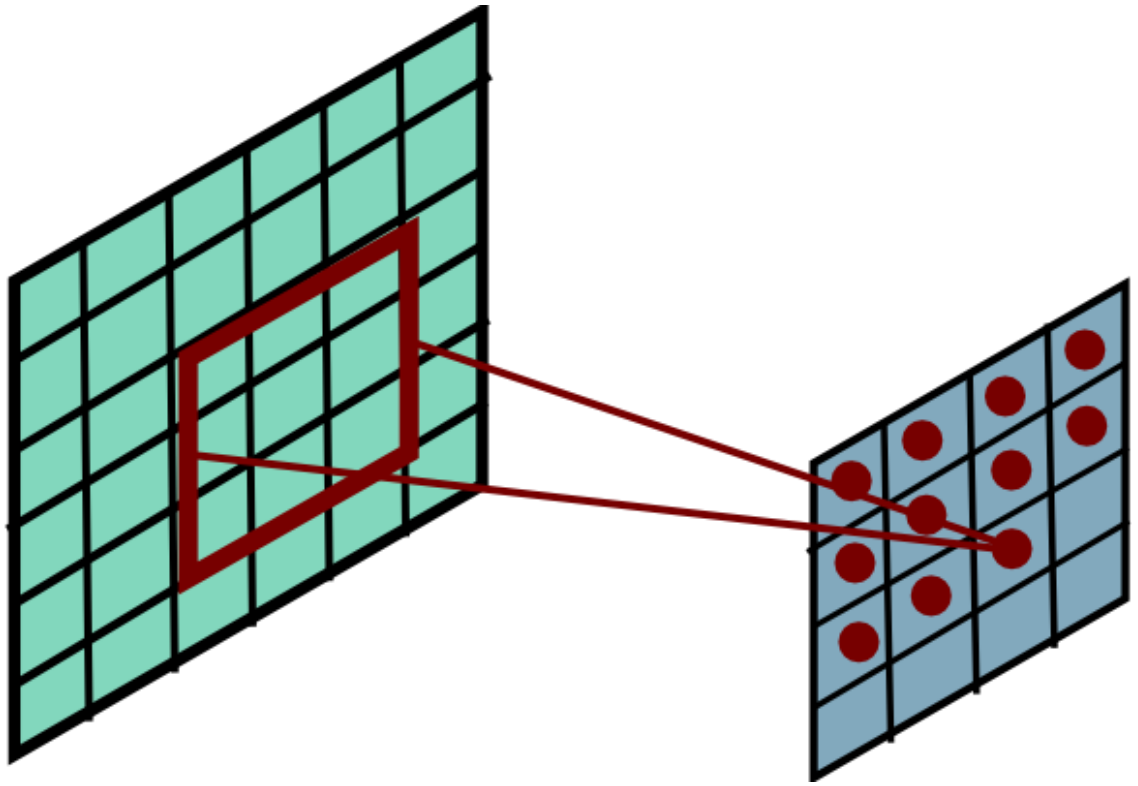
Convolutional Layer



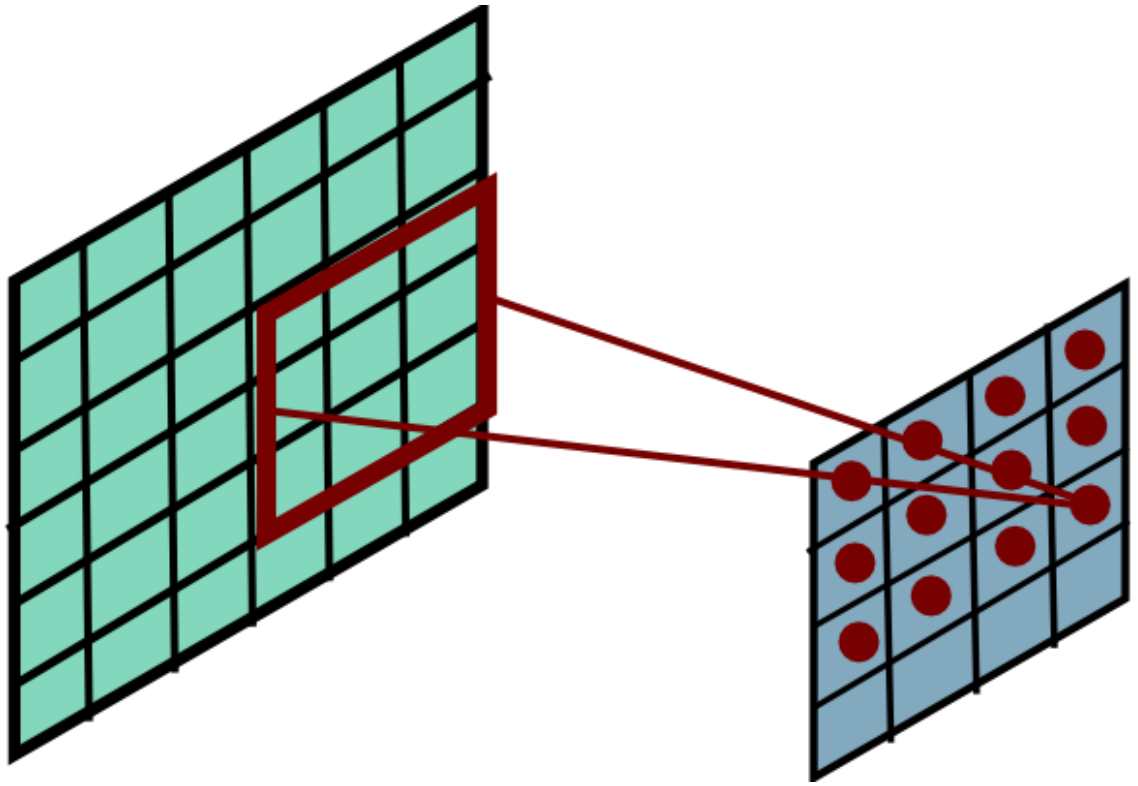
Convolutional Layer



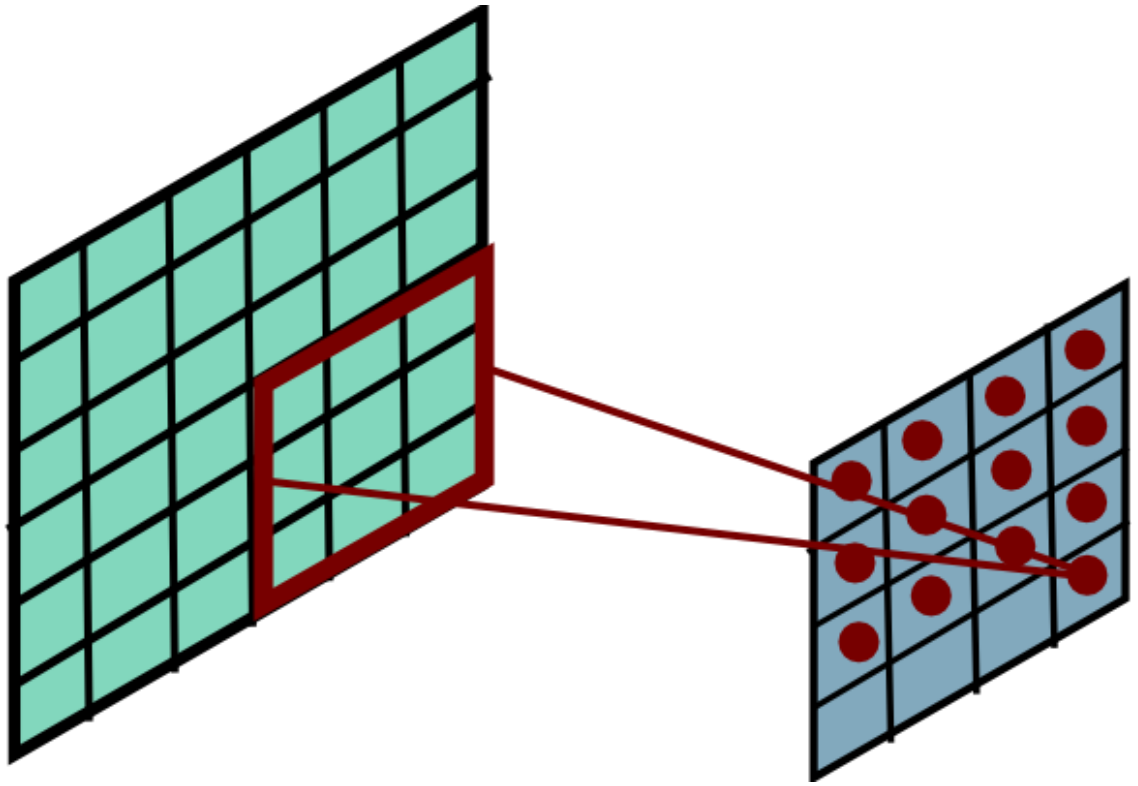
Convolutional Layer



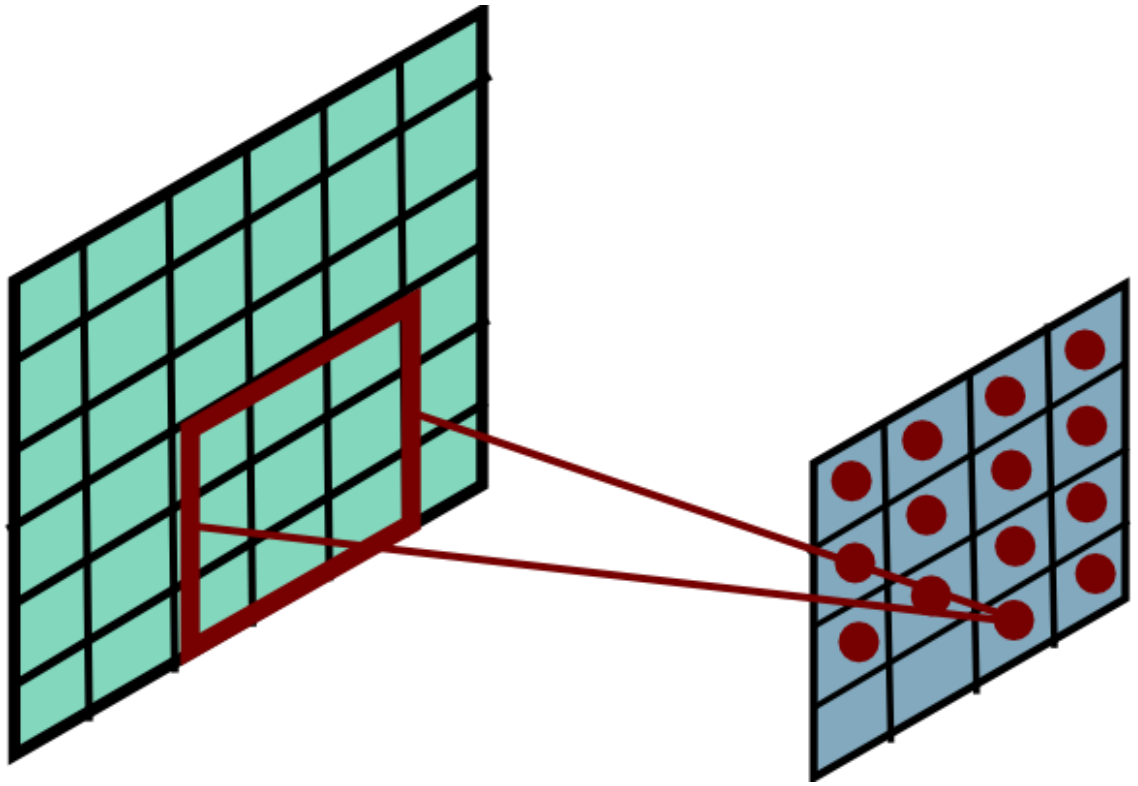
Convolutional Layer



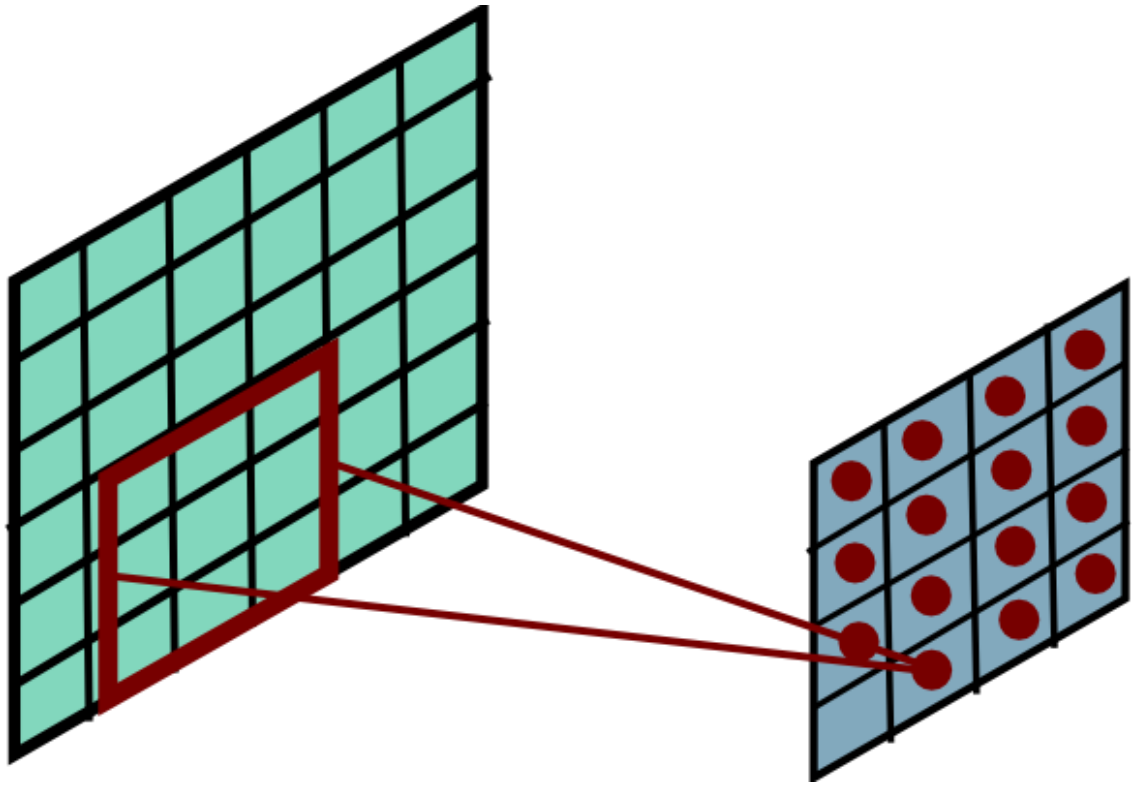
Convolutional Layer



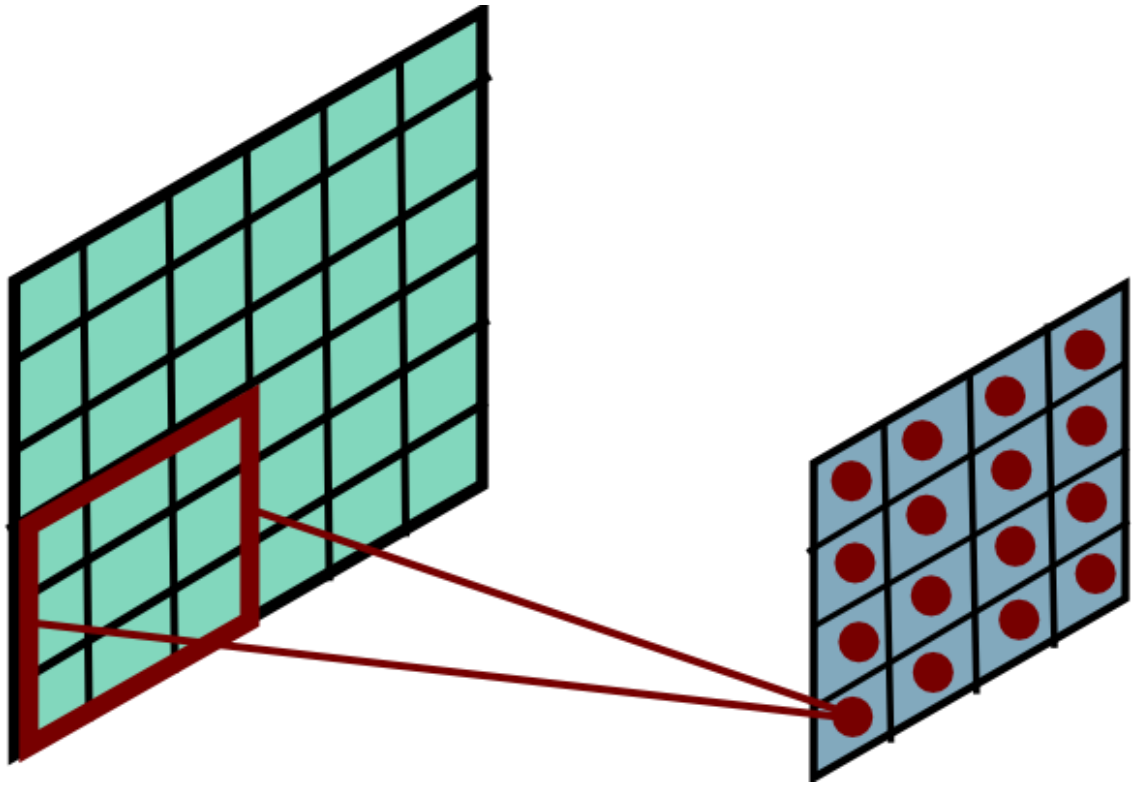
Convolutional Layer



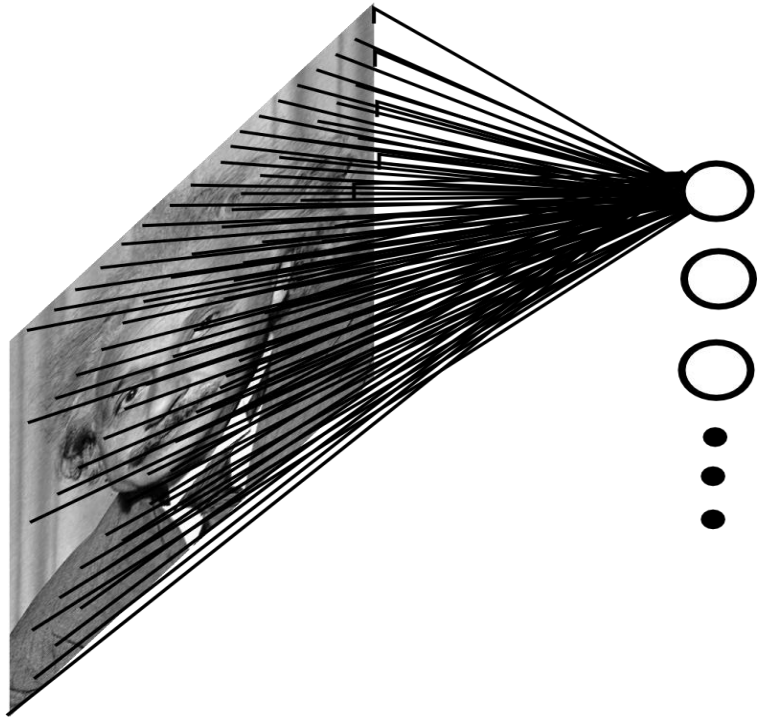
Convolutional Layer



Convolutional Layer



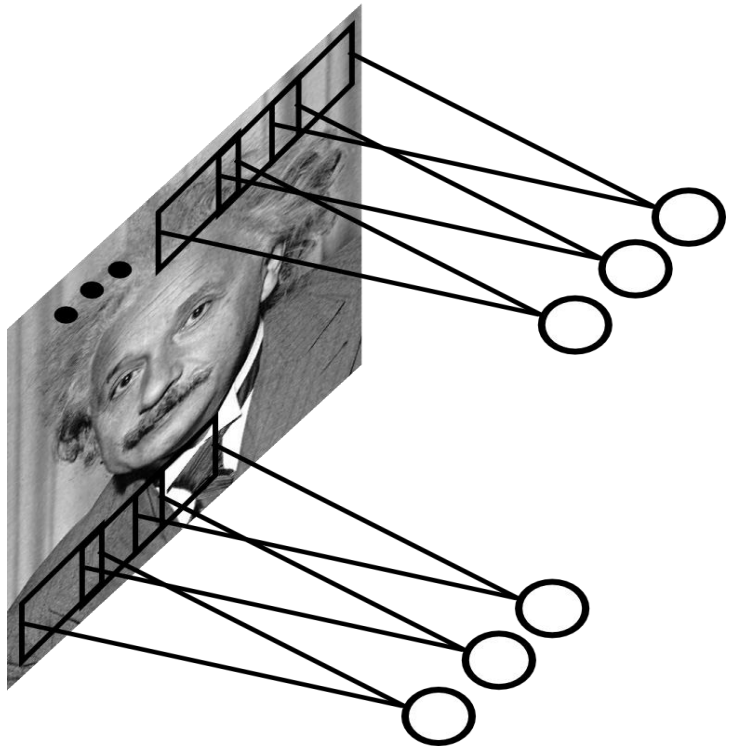
Fully-connected layer



#of parameters: K^2

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_K \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \dots & w_{1,K} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \dots & w_{2,K} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \dots & w_{3,K} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \dots & w_{4,K} \\ & & \vdots & & & \\ w_{K,1} & w_{K,2} & w_{K,3} & w_{K,4} & \dots & w_{K,K} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_K \end{bmatrix}$$

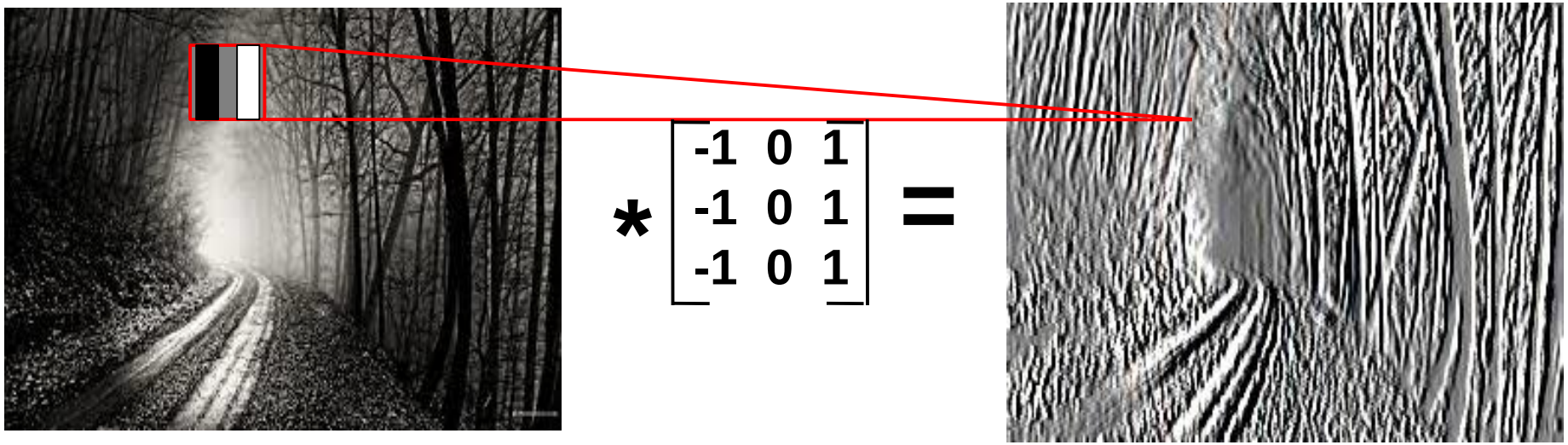
Convolutional layer



#of parameters: size of window

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_K \end{bmatrix} = \begin{bmatrix} w_0 & w_1 & w_2 & 0 & \dots & 0 \\ 0 & w_0 & w_1 & w_2 & \dots & 0 \\ 0 & 0 & w_0 & w_1 & \dots & 0 \\ 0 & 0 & 0 & w_0 & \dots & 0 \\ & & \vdots & & & \\ 0 & 0 & 0 & 0 & \dots & w_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_K \end{bmatrix}$$

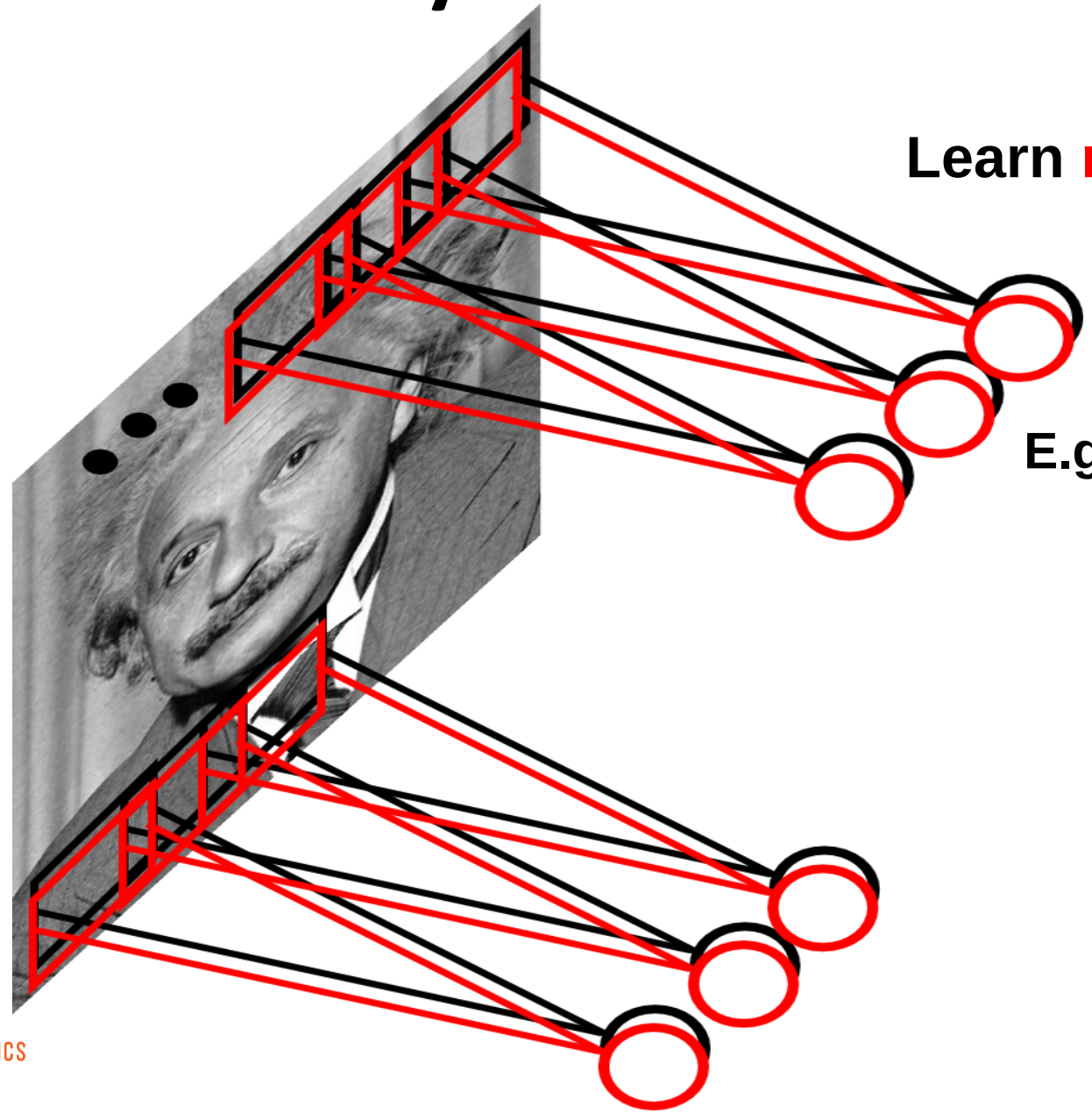
Convolutional layer



Code example

Learning an edge filter

Convolutional layer



Learn multiple filters.

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

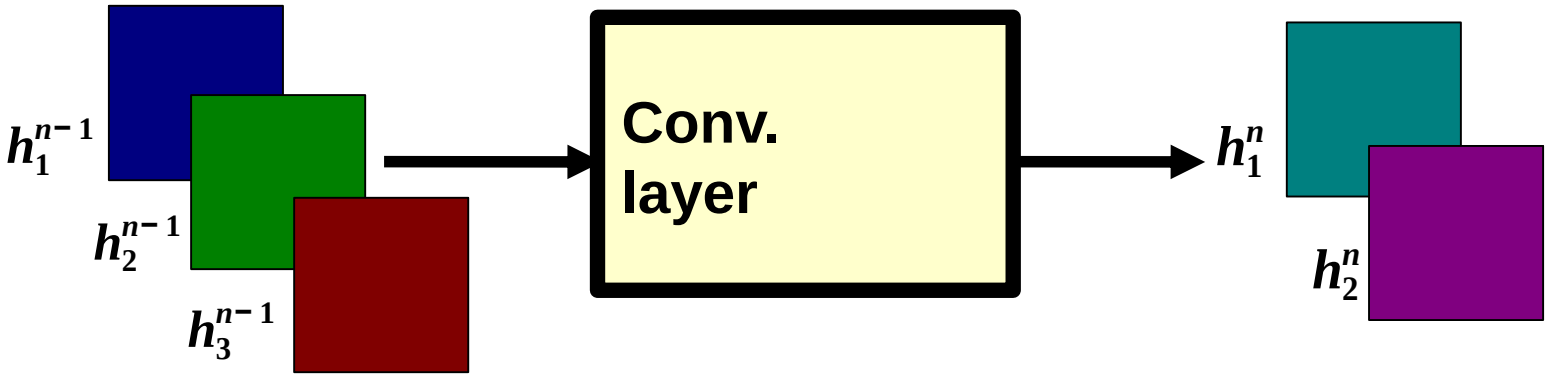
Convolutional layer

$$h_i^n = \max\left\{0, \sum_{j=1}^{\#\text{input channels}} h_j^{n-1} * w_{ij}^n\right\}$$

output feature map

input feature map

kernel



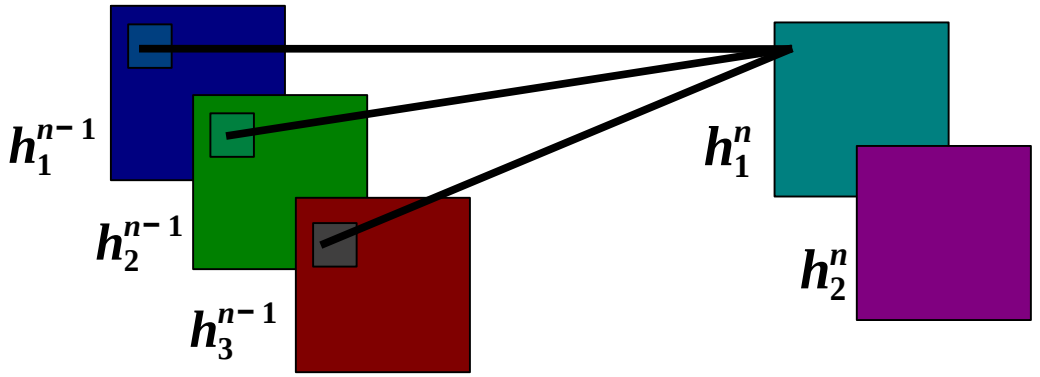
Convolutional layer

$$h_i^n = \max\left\{0, \sum_{j=1}^{\#\text{input channels}} h_j^{n-1} * w_{ij}^n\right\}$$

output feature map

input feature map

kernel



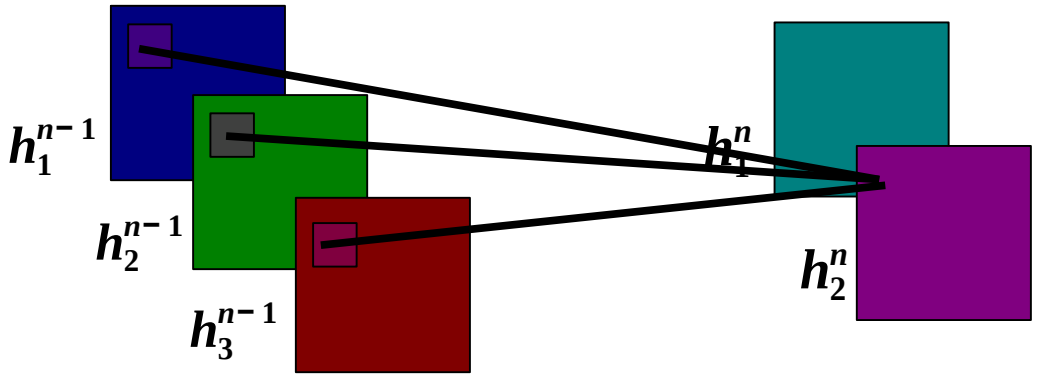
Convolutional layer

$$h_i^n = \max\left\{0, \sum_{j=1}^{\#\text{input channels}} h_j^{n-1} * w_{ij}^n\right\}$$

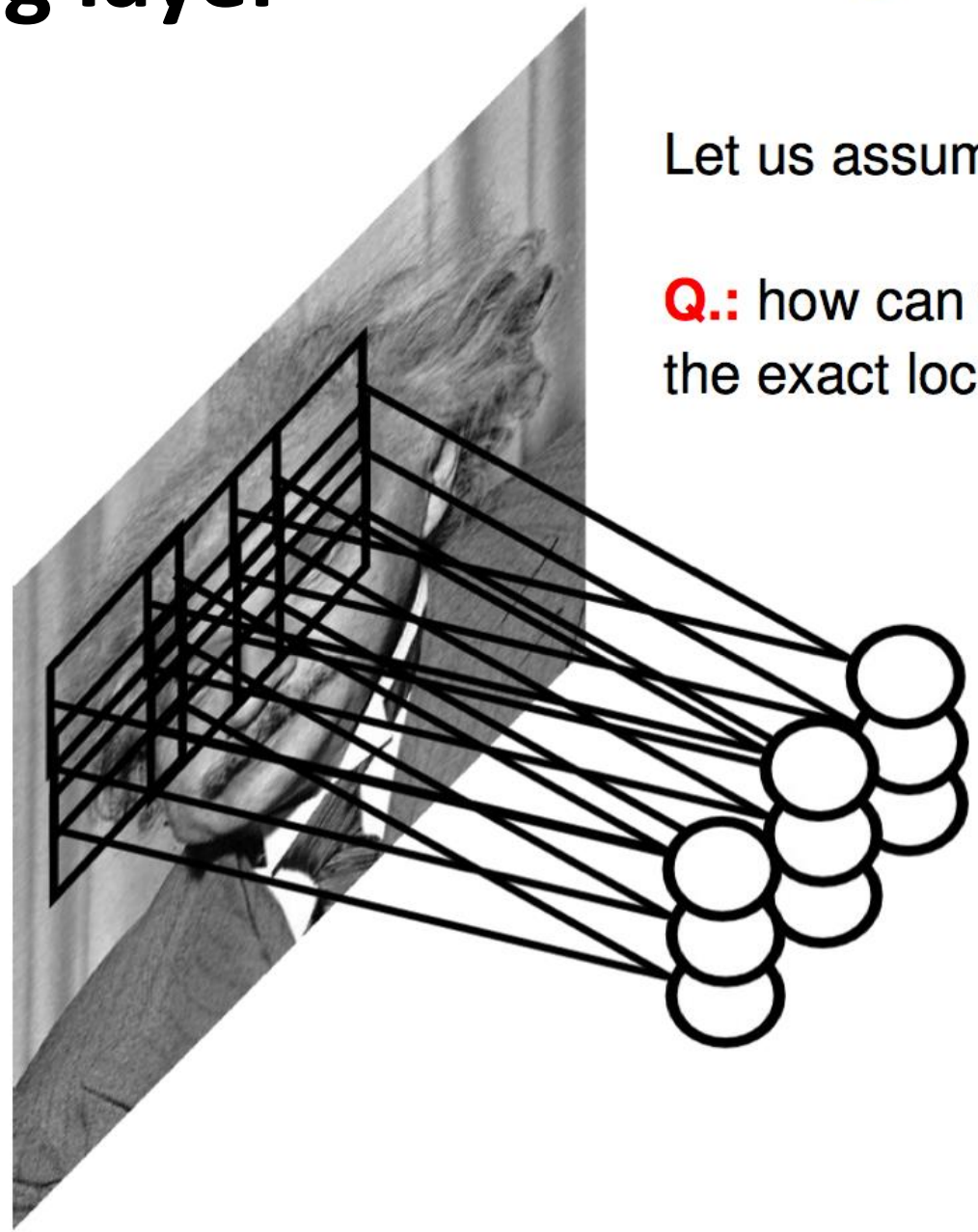
output feature map

input feature map

kernel



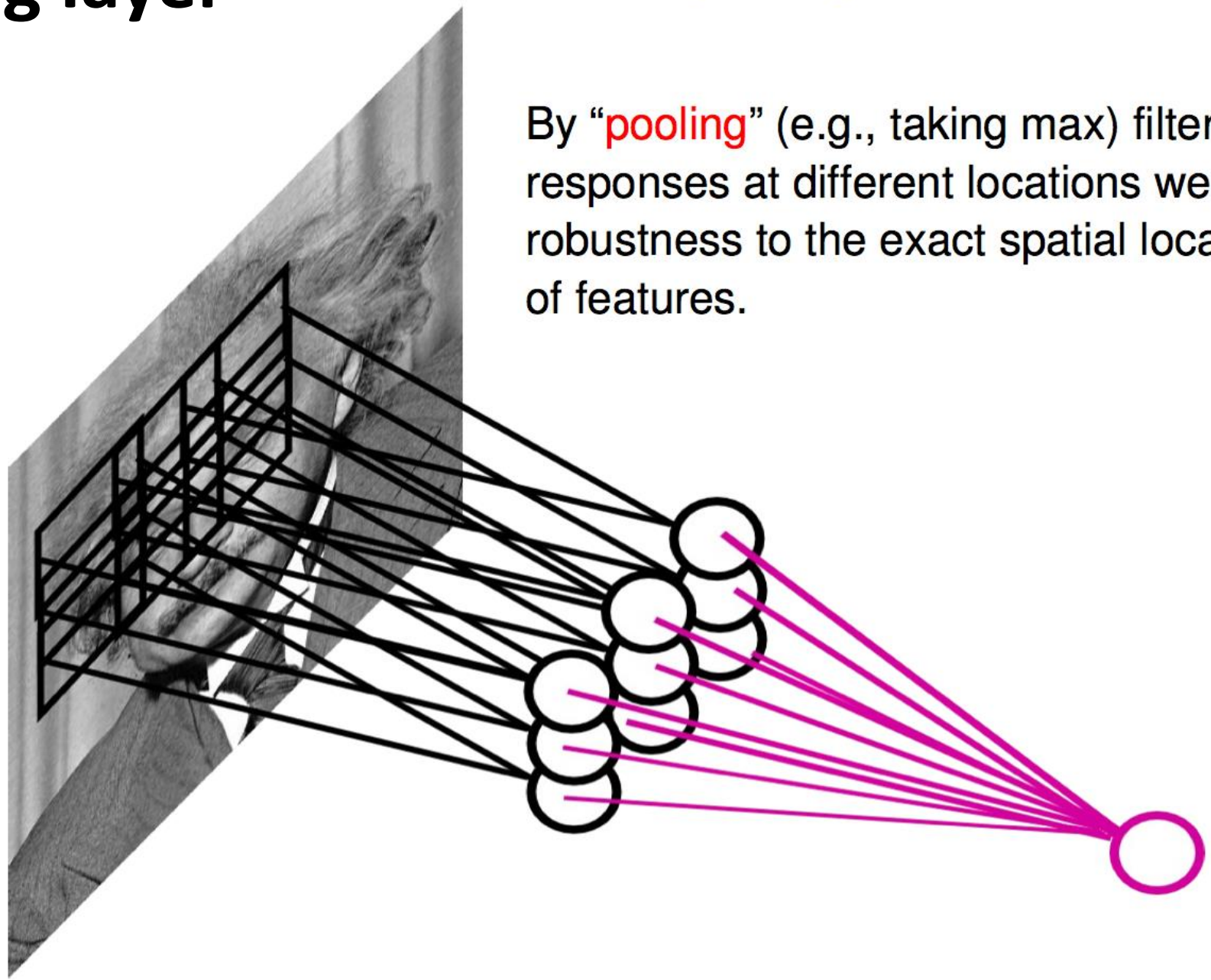
Pooling layer



Let us assume filter is an “eye” detector.

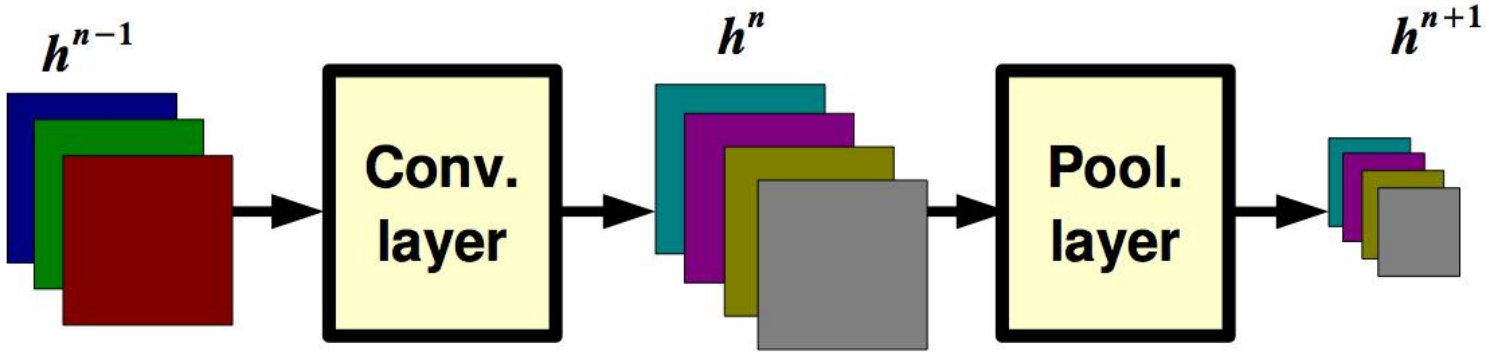
Q.: how can we make the detection robust to the exact location of the eye?

Pooling layer

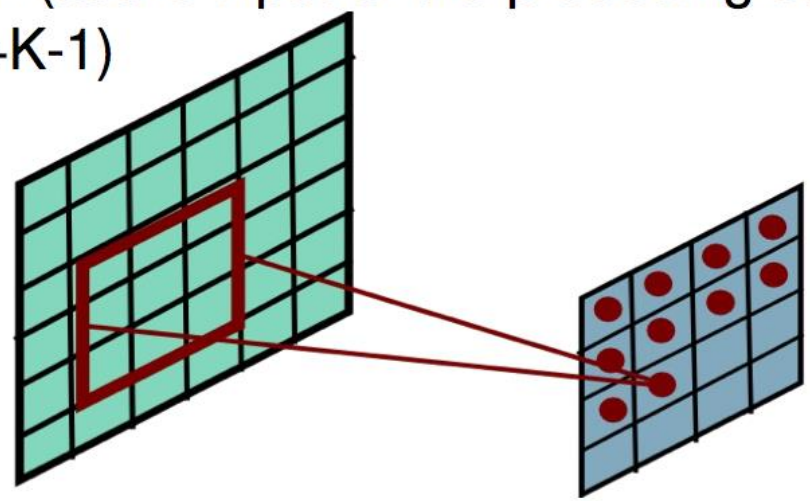


By “pooling” (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

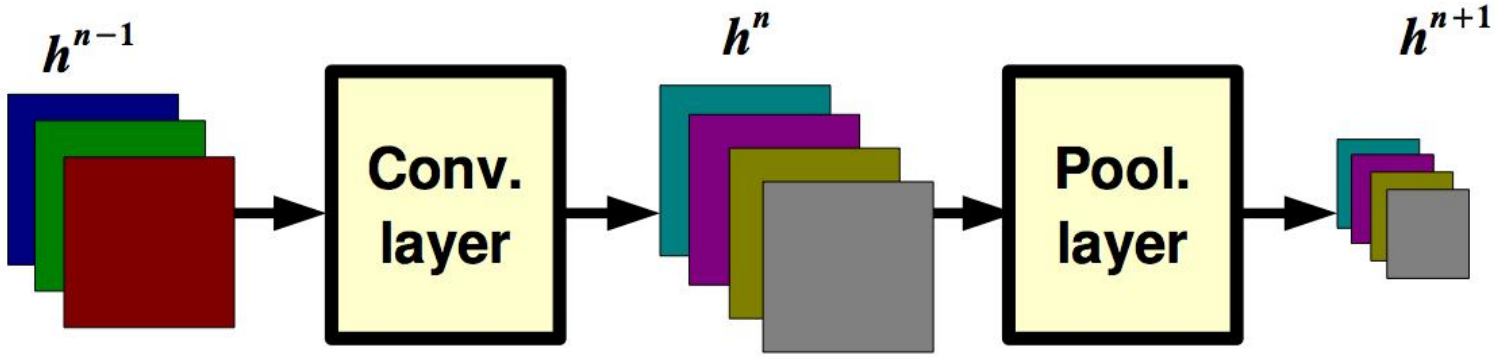
Pooling layer: receptive field size



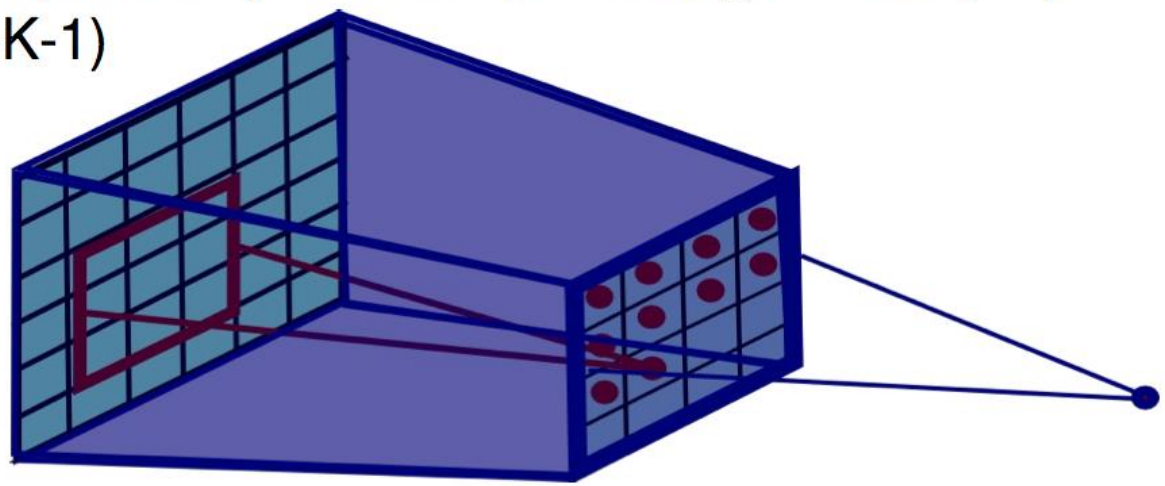
If convolutional filters have size KxK and stride 1, and pooling layer has pools of size PxP, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: (P+K-1)x(P+K-1)



Pooling layer: receptive field size



If convolutional filters have size KxK and stride 1, and pooling layer has pools of size PxP, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: (P+K-1)x(P+K-1)



Receptive field

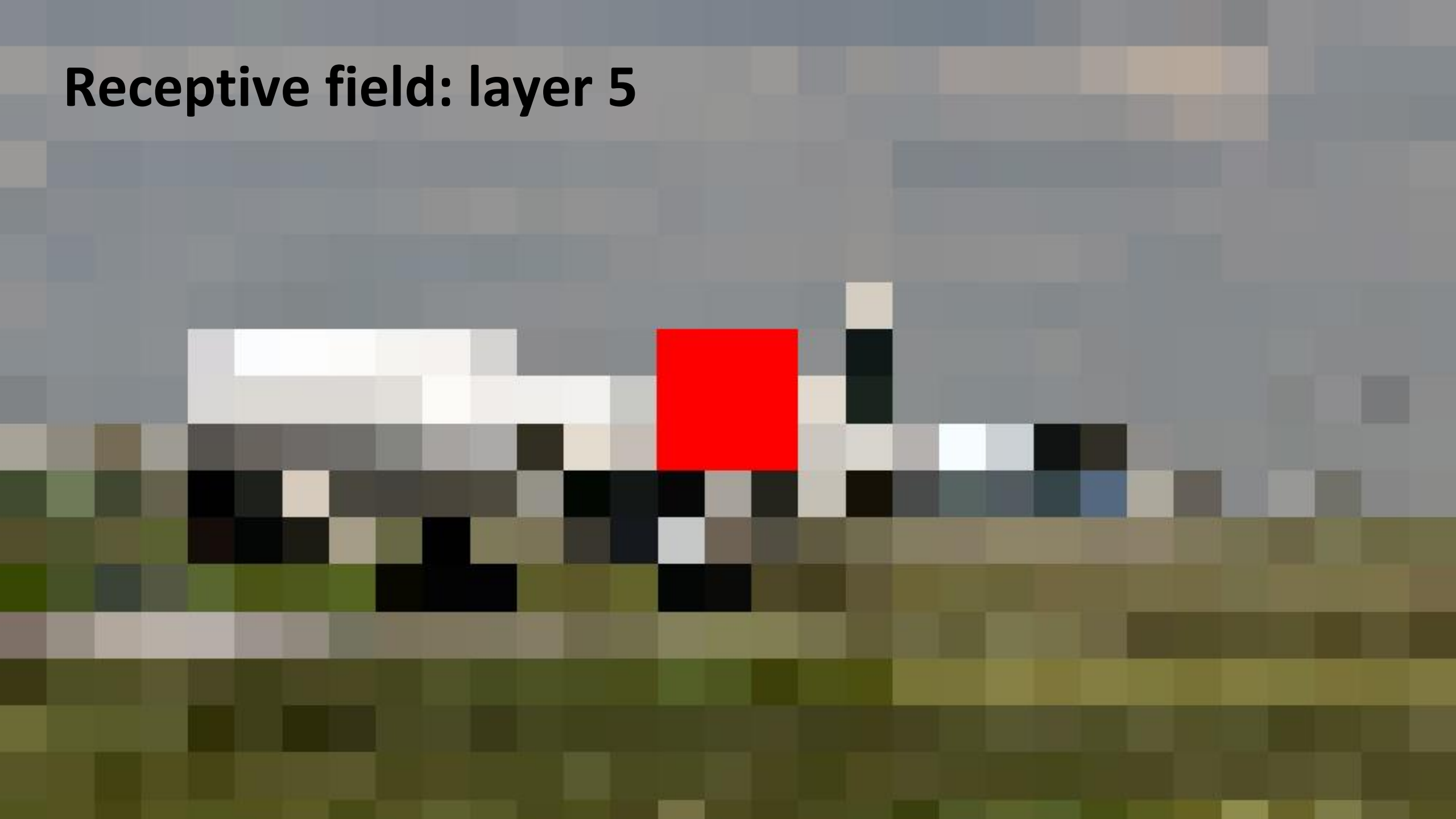
Receptive field: layer 1

Receptive field: layer 2

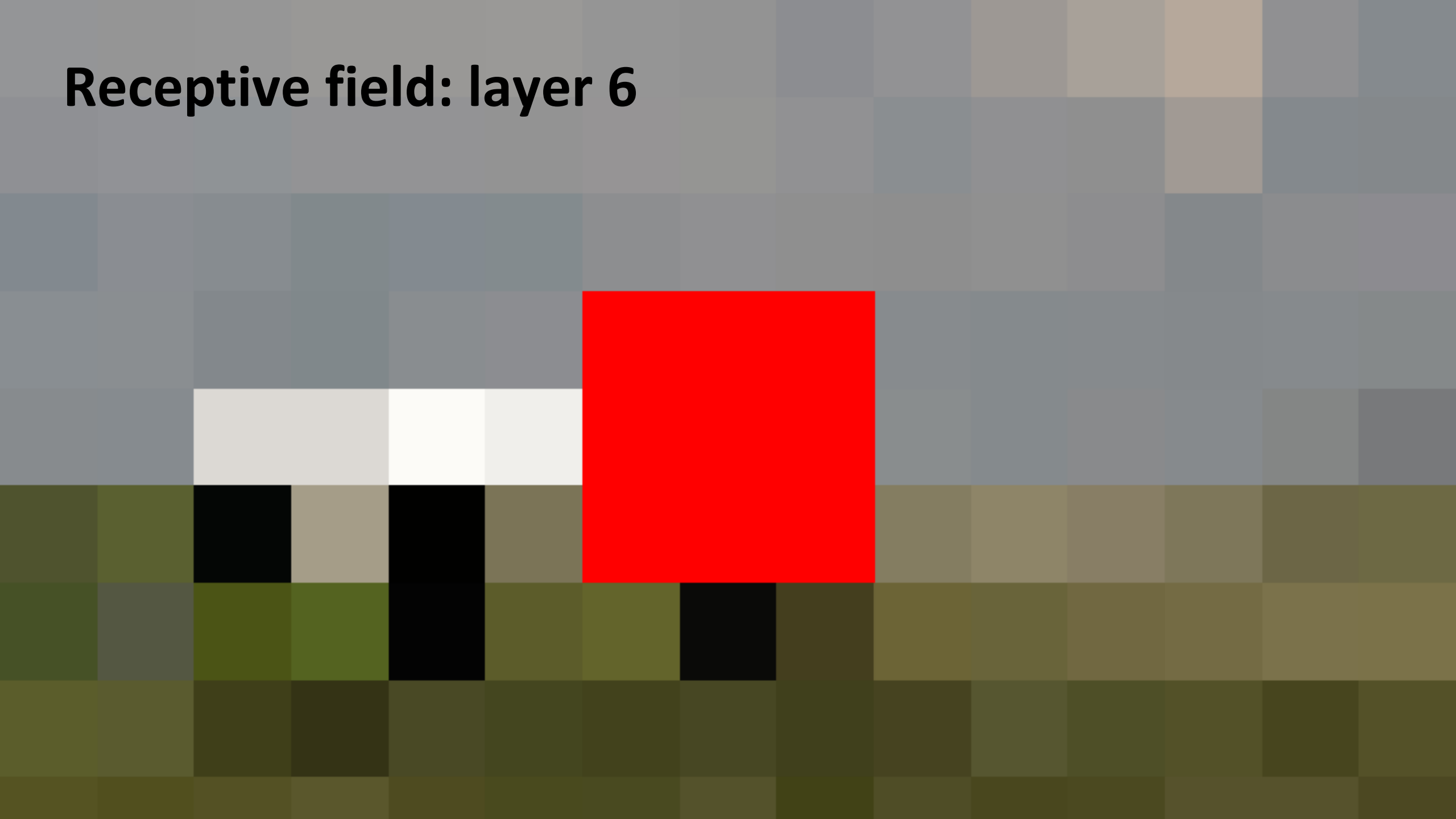
Receptive field: layer 3

Receptive field: layer 4

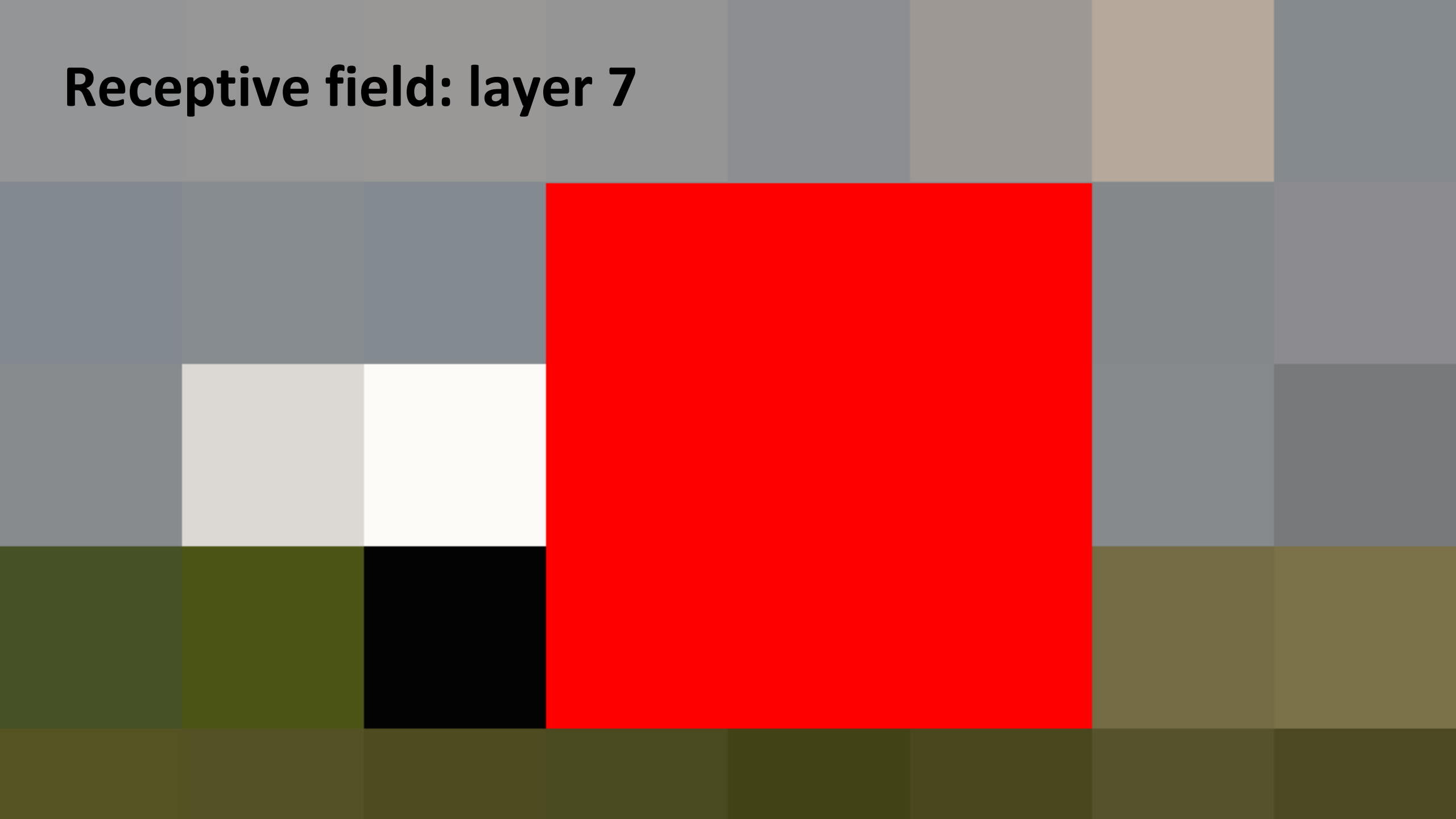
Receptive field: layer 5



Receptive field: layer 6



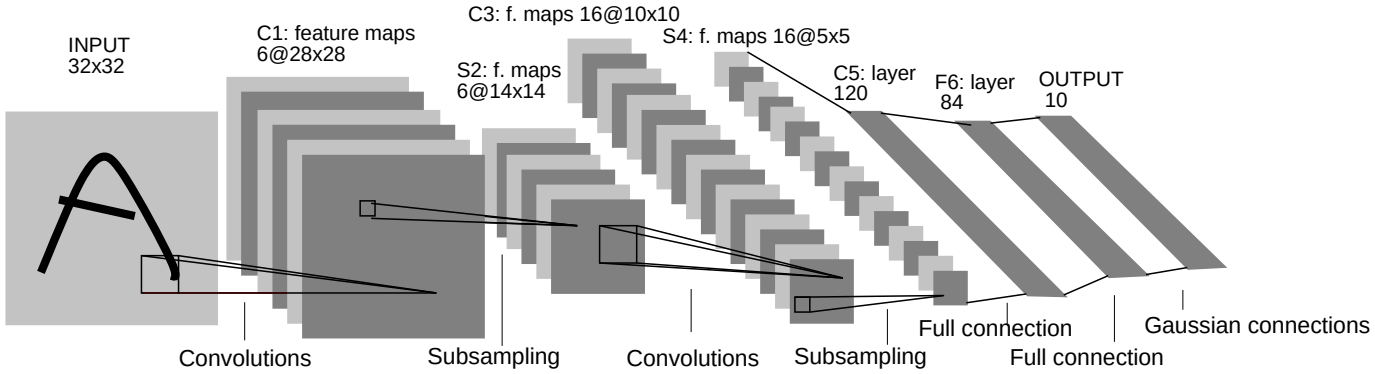
Receptive field: layer 7



Receptive field: layer 8

Modern Architectures

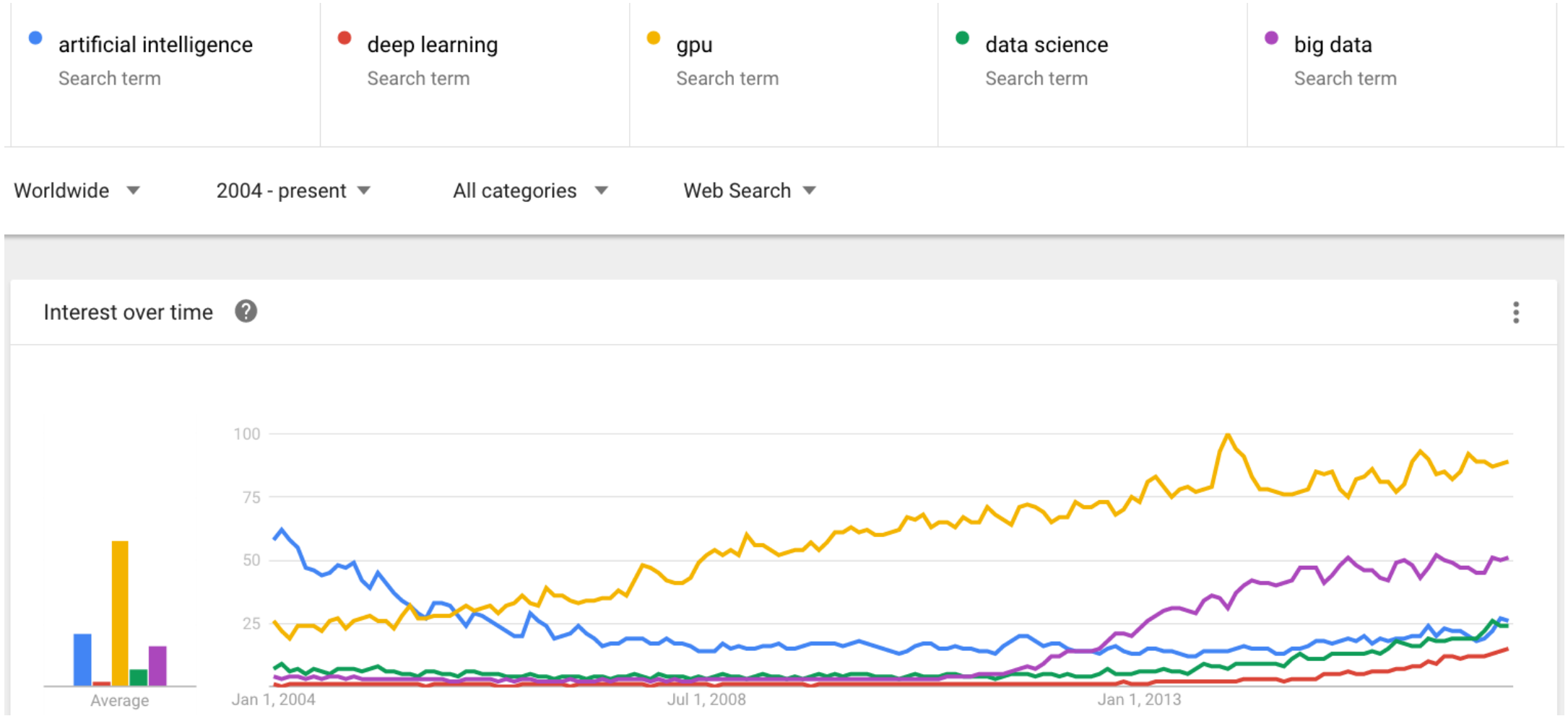
CNNs, late 1980's: LeNet



Gradient-based learning applied to document recognition.
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998

https://www.youtube.com/watch?v=FwFduRA_L6Q

What happened in between?



deep learning = neural networks (+ big data + GPUs) **+ a few more recent tricks!**

Code example

Convolutional Network
and
Filter Visualizations

Parameter Initialization

- All zero initialization: All parameters get the same gradient and same updates

$$w = 0$$

- Random initialization: Is sometimes used in practice, but variance of output depends on number of inputs, which may cause instability early on

$$w \sim \mathcal{N}(\mu = 0, \sigma = 0.01)$$

- Kaiming initialization: divide standard deviation by number of inputs

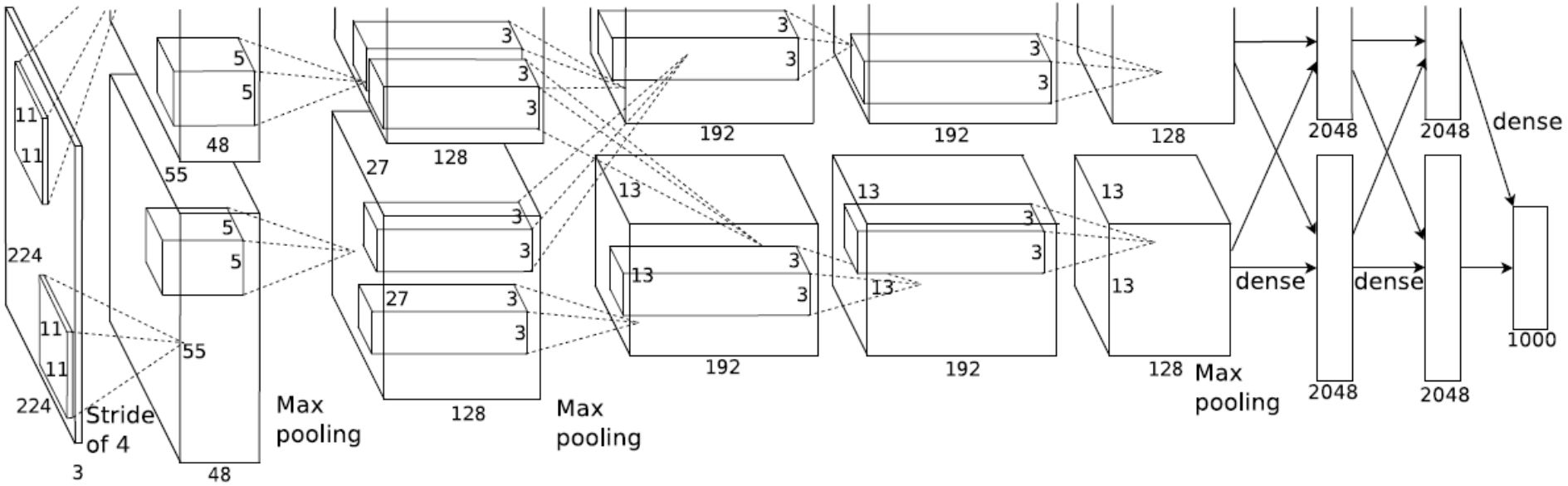
$$w \sim \mathcal{N}(\mu = 0, \sigma = \frac{0.01}{n})$$

n : number of inputs (fan-in)

Code examples

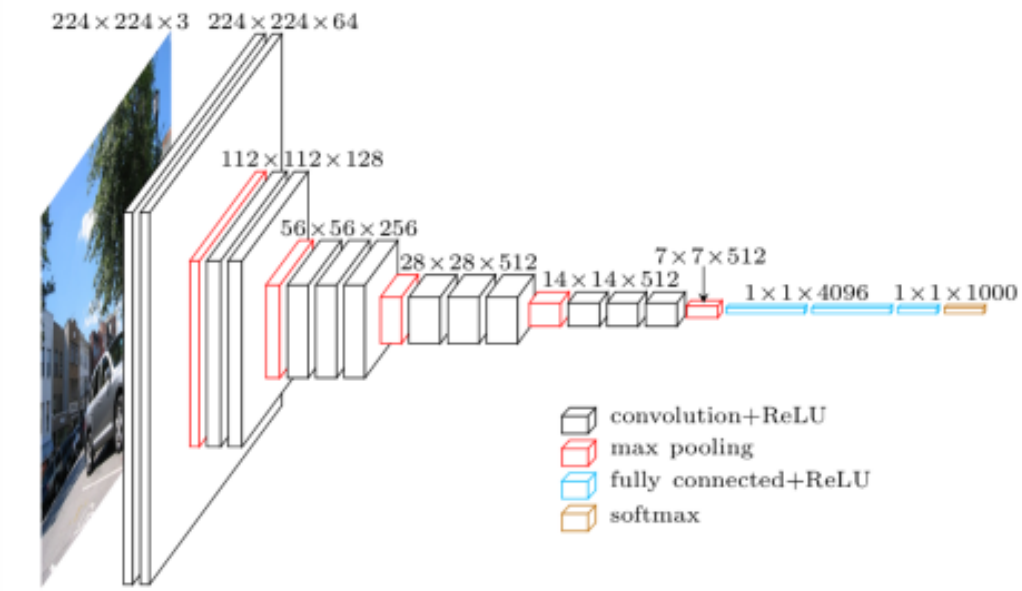
Parameter initialization

CNNs, 2012



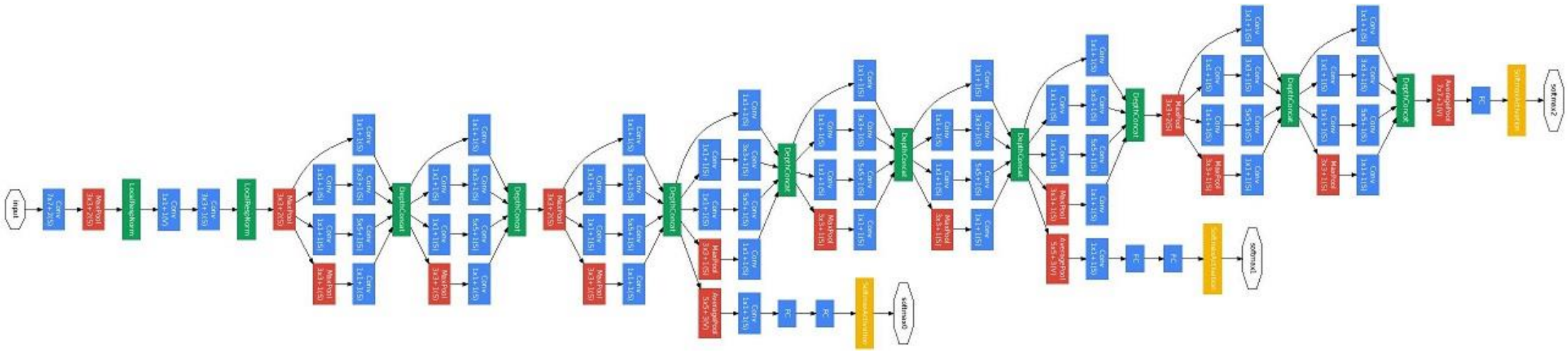
AlexNet
Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton: ImageNet classification with deep convolutional neural networks. Commun. ACM 60(6): 84-90 (2017)

CNNs, 2014: VGG



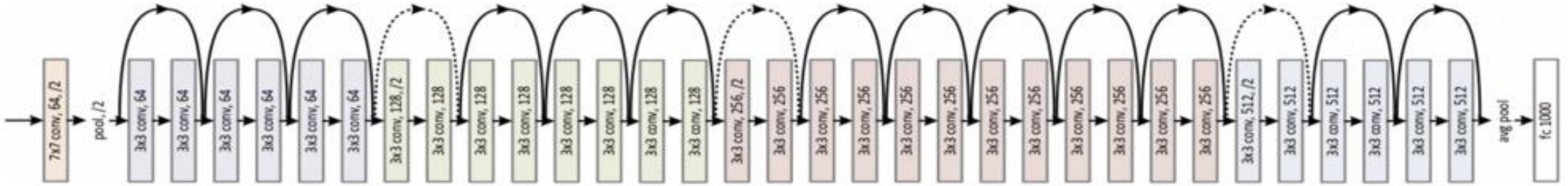
Karen Simonyan, Andrew Zisserman (=Visual Geometry Group)
Very Deep Convolutional Networks for Large-Scale Image Recognition, arxiv, 2014.

CNNs, 2014: GoogLeNet



Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich
Going Deeper with Convolutions, CVPR 2015

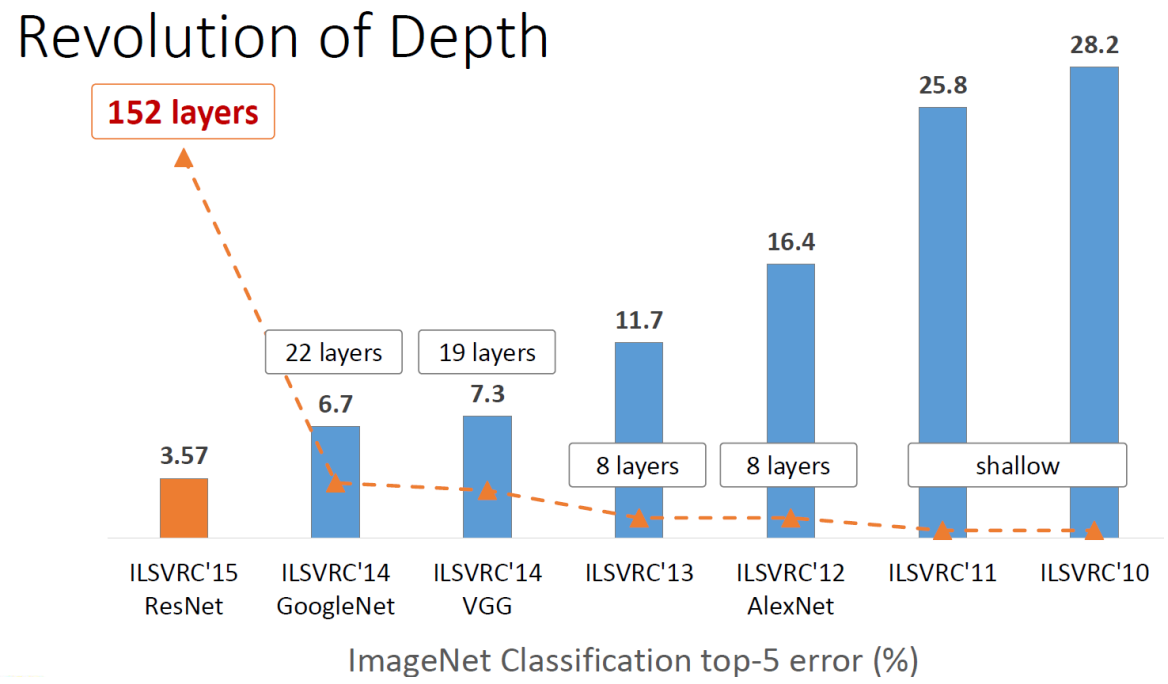
CNNs, 2015: ResNet



ResNet
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun,
Deep Residual Learning for Image Recognition
CVPR 2016

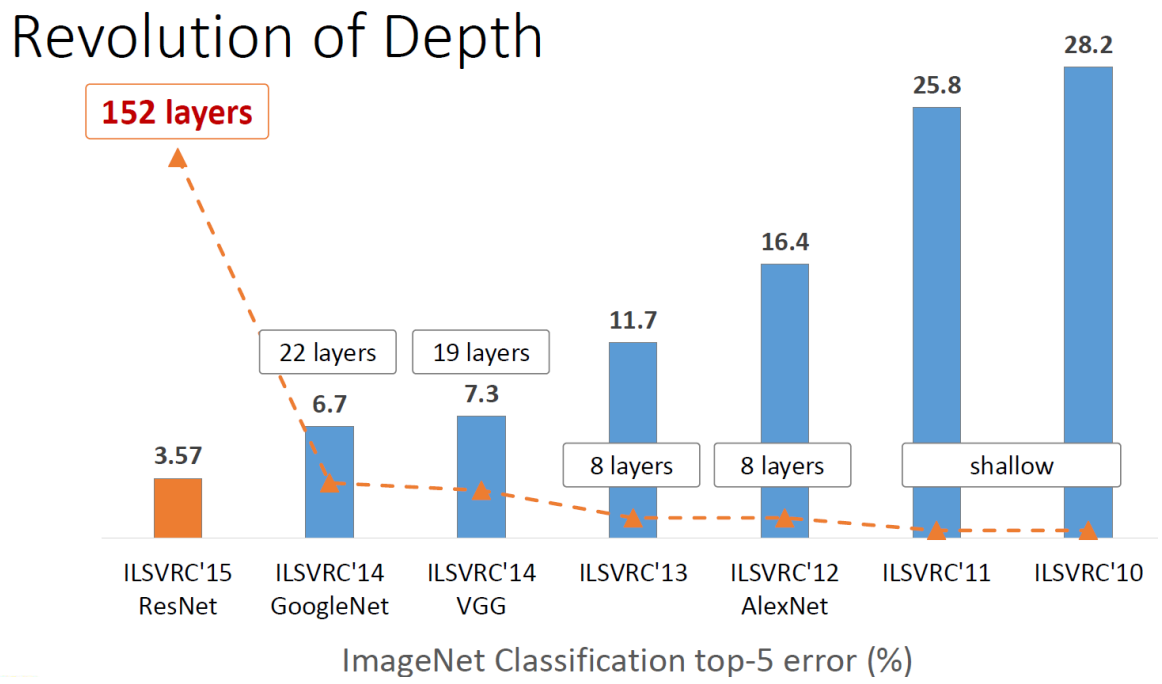
The Deeper, the Better

- Deeper networks can cover more complex problems
  - Increasingly large receptive field size & rich patterns



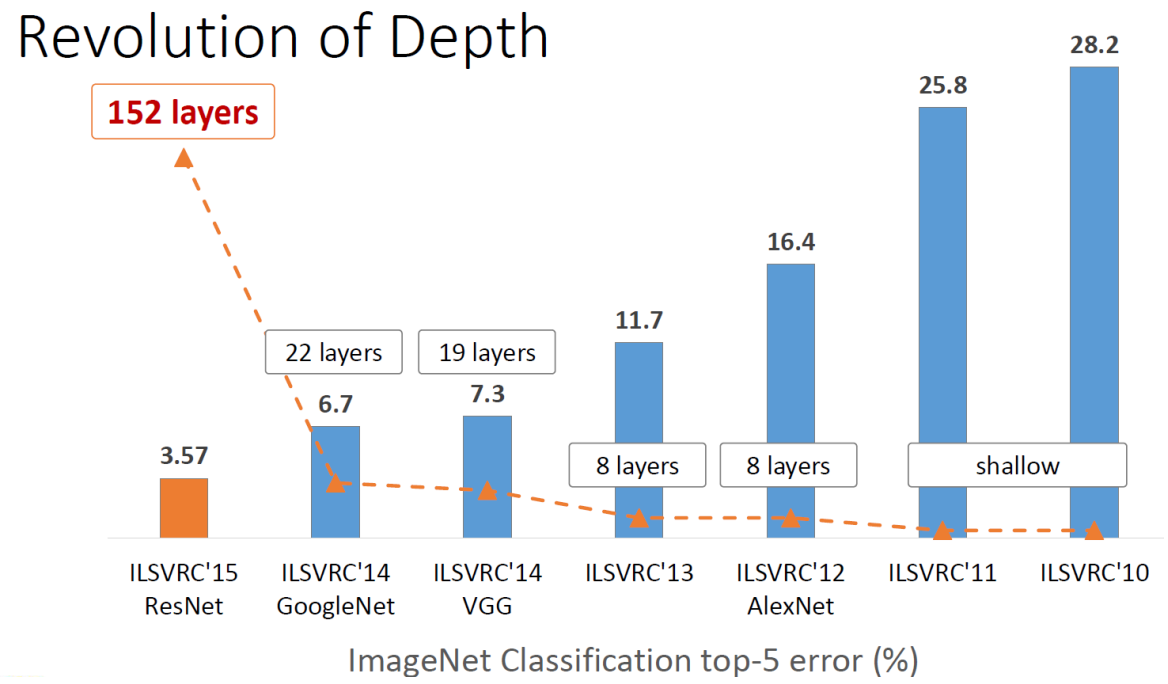
Going Deeper

- From 2 to 10: 2010-2012
  - ReLUs
  - Dropout
  - …



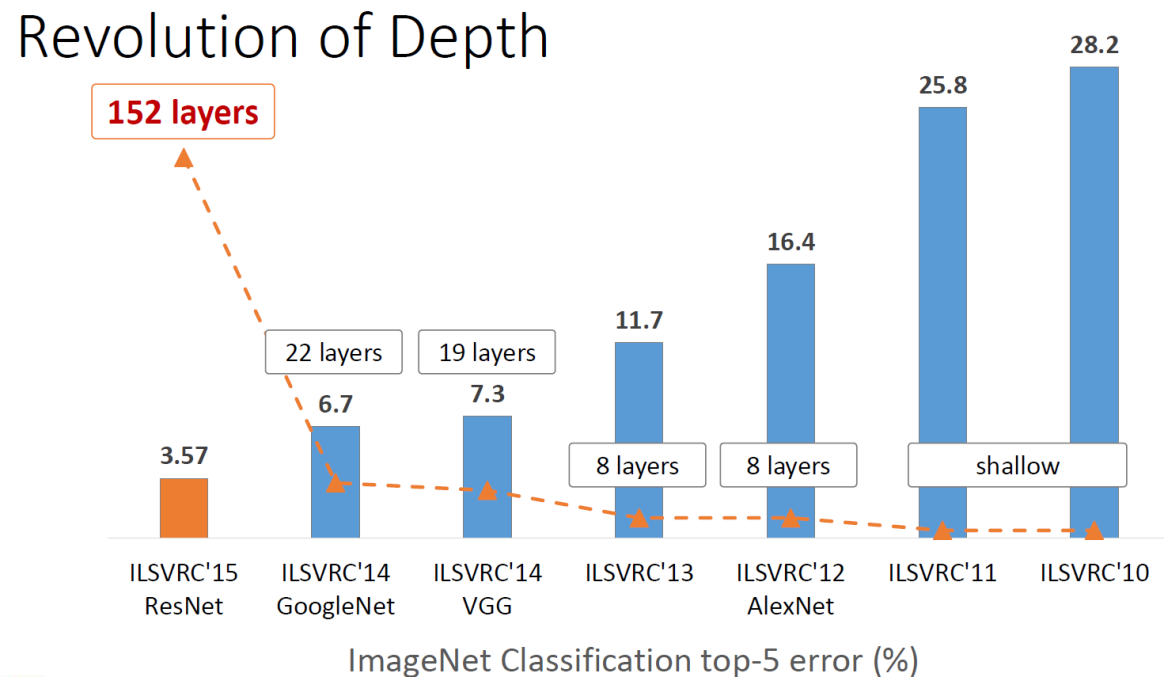
Going Deeper

- From 10 to 20: 2015
  - Batch Normalization



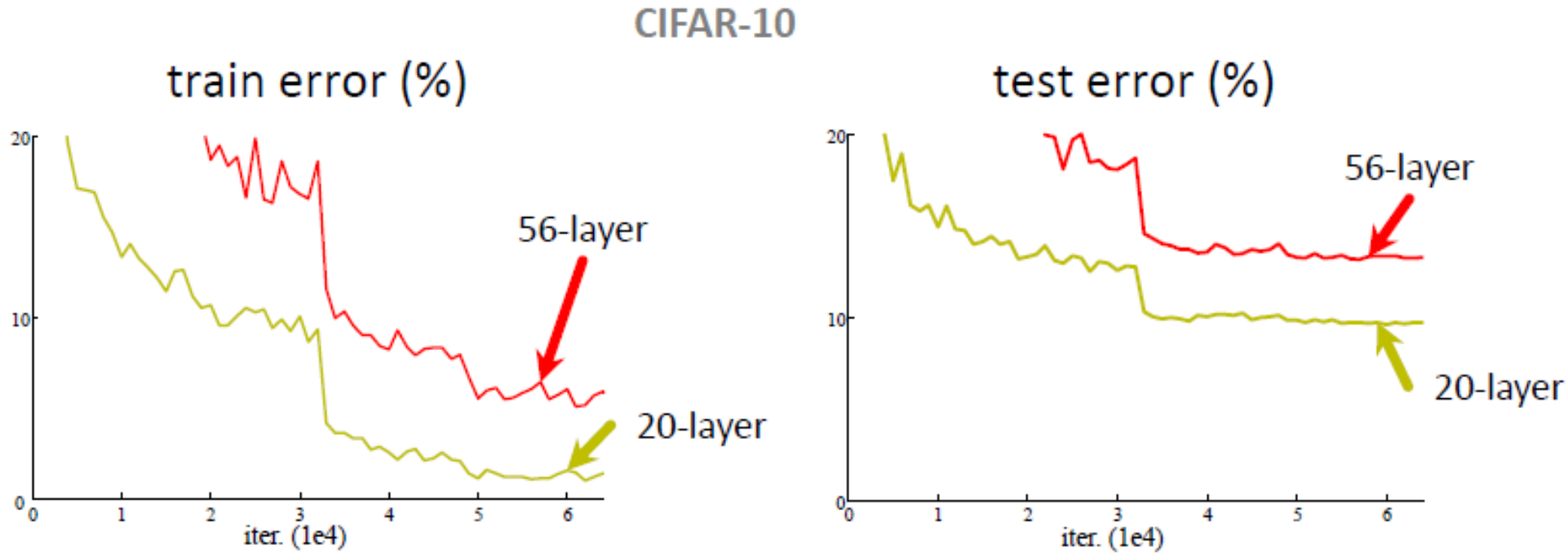
Going Deeper

- From 20 to 100/1000
  - Residual networks



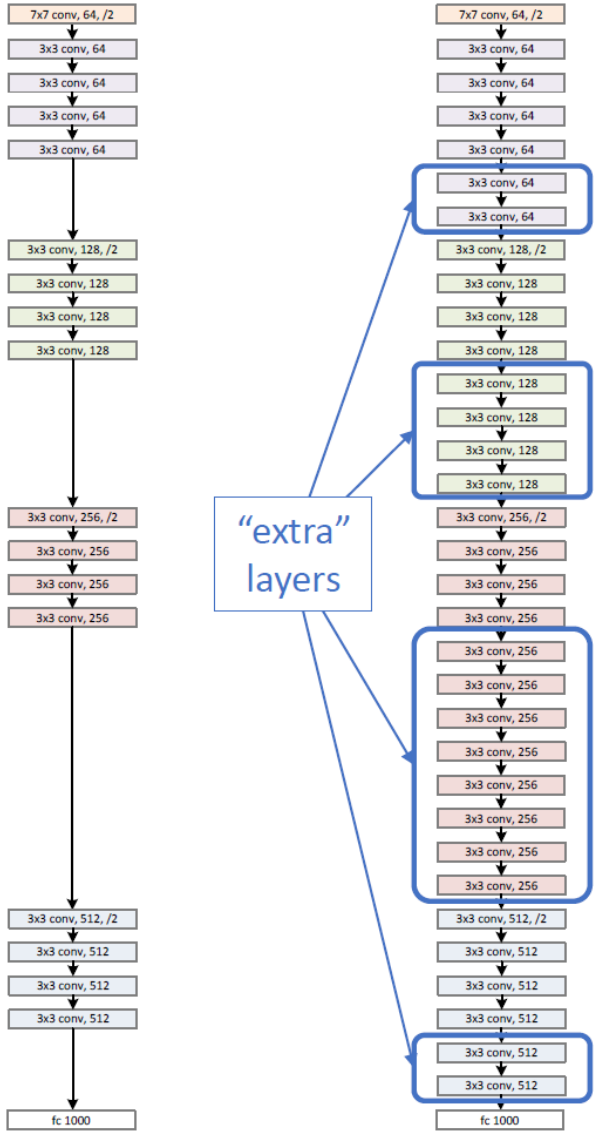
Plain network: deeper is not necessarily better

- Plain nets: stacking 3x3 conv layers
- 56-layer net has higher training error and test error than 20-layer net



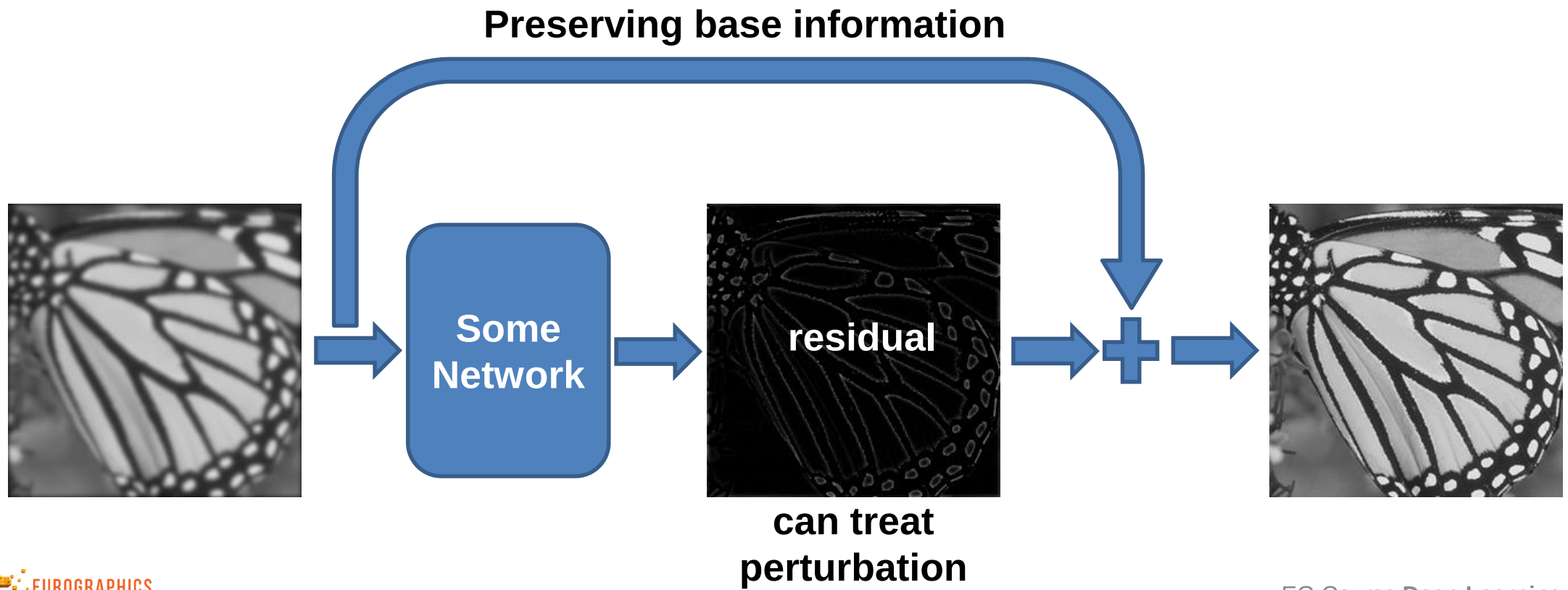
Residual Network

- Naïve solution
  - If extra layers are an identity mapping, then training errors can not increase



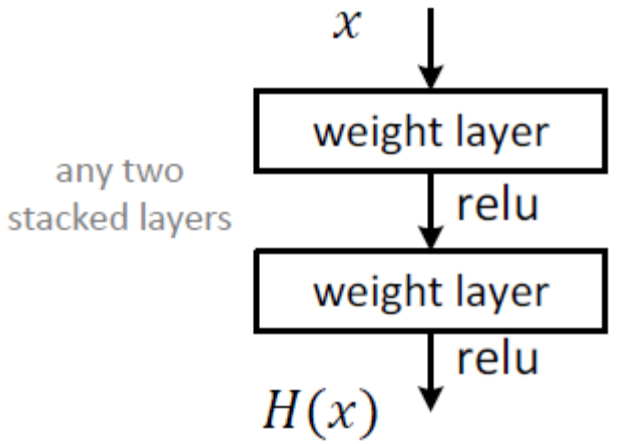
Residual Modelling: Basic Idea in Image Processing

- Goal: estimate update between an original image and a changed image



Residual Network

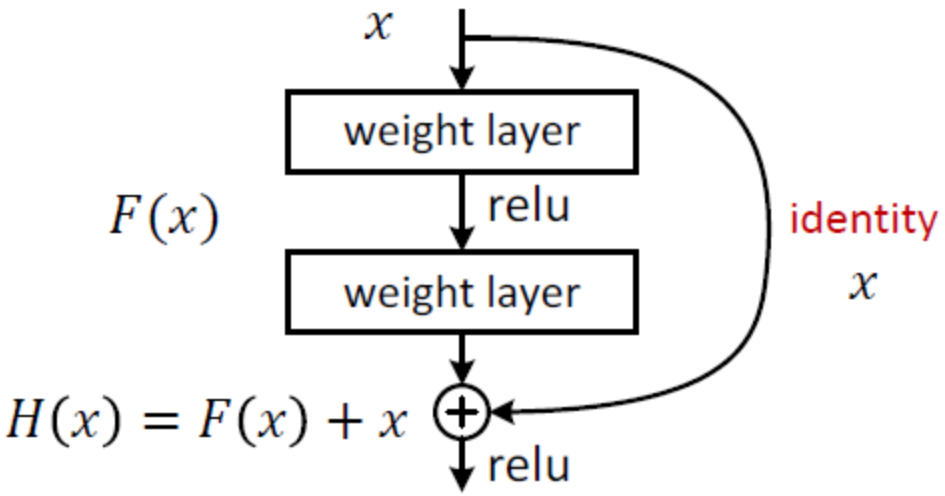
- Plain block
  - Difficult to make identity mapping because of multiple non-linear layers



Residual Network

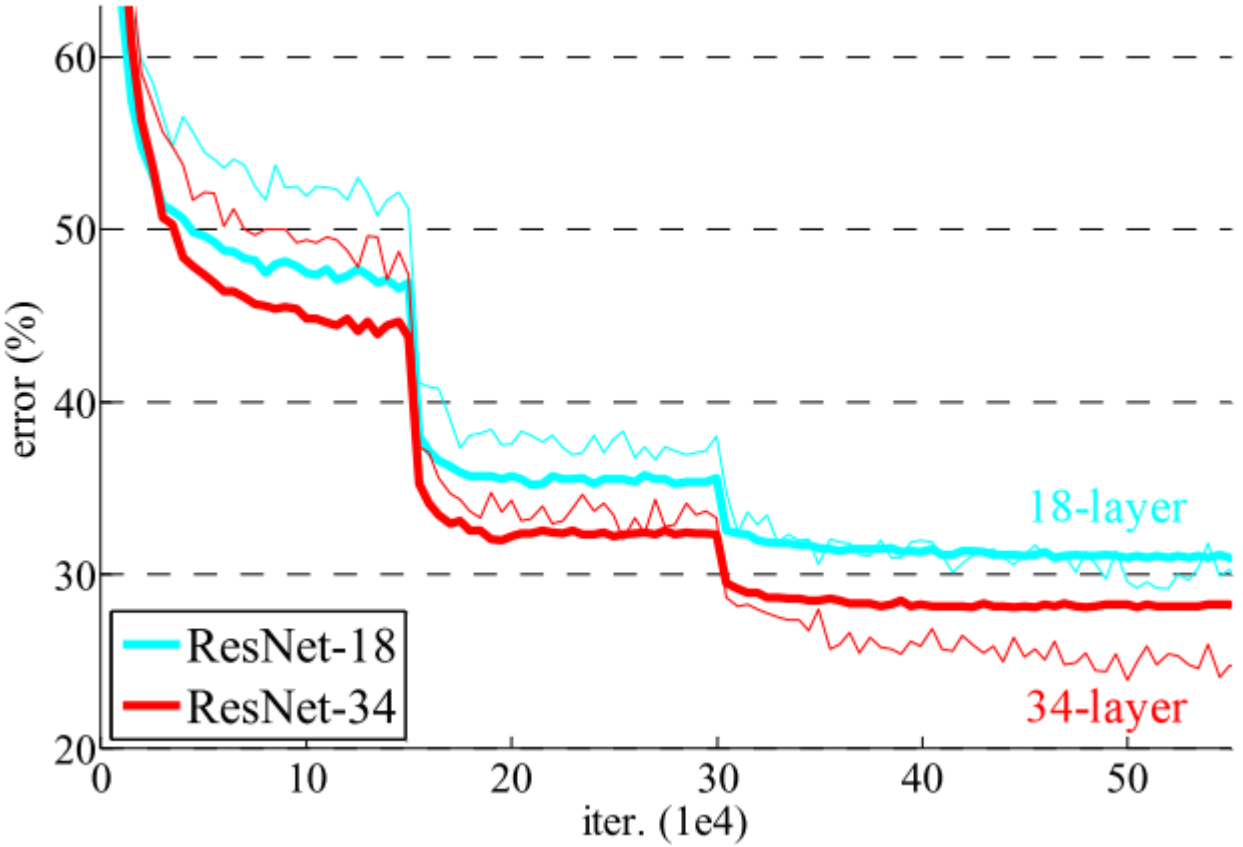
- Residual block
  - If identity were optimal, easy to set weights as 0
  - If optimal mapping is closer to identity, easier to find small fluctuations

Appropriate for treating perturbation as keeping a base information

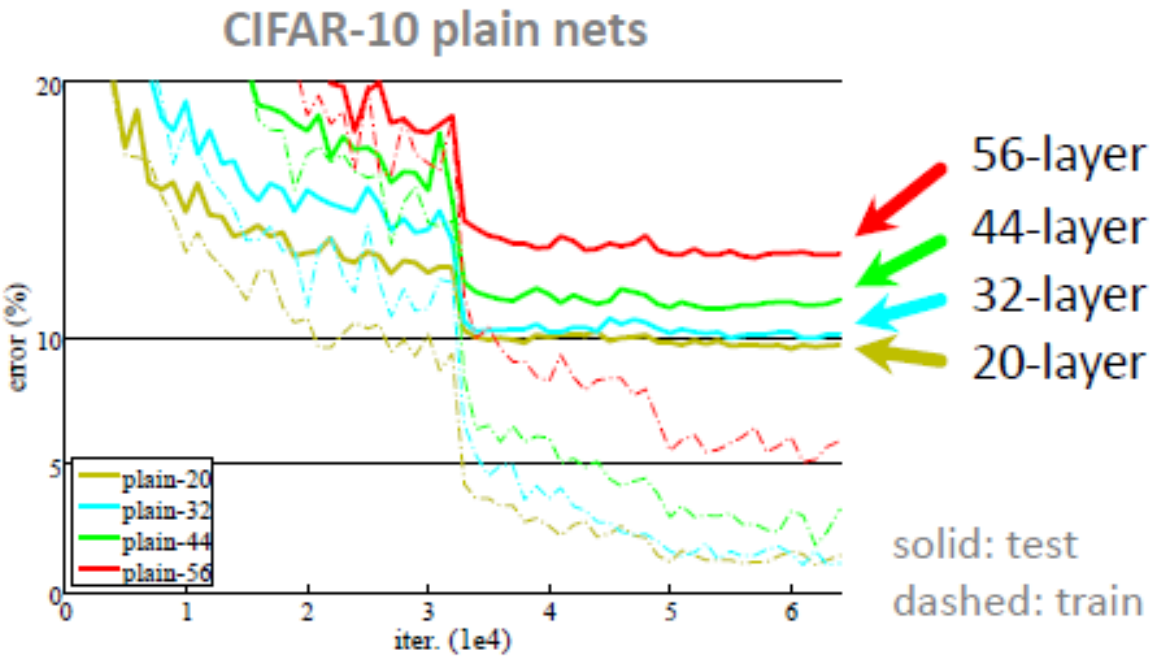


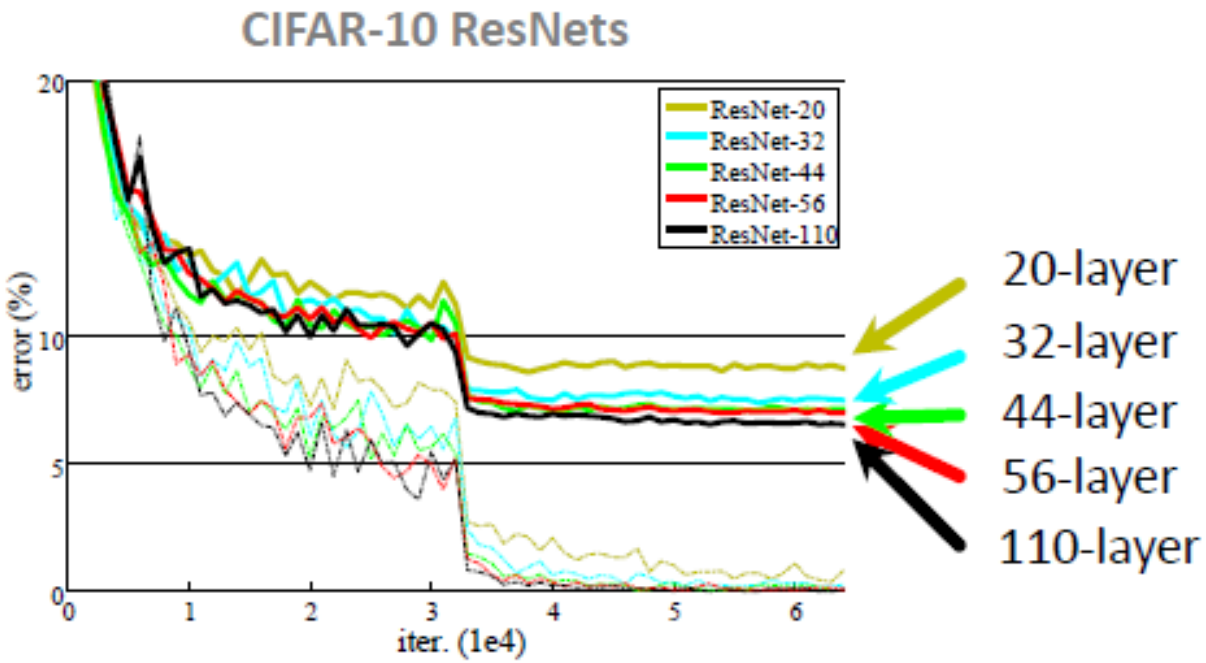
Residual Network: deeper is better

- Deeper ResNets have lower training error



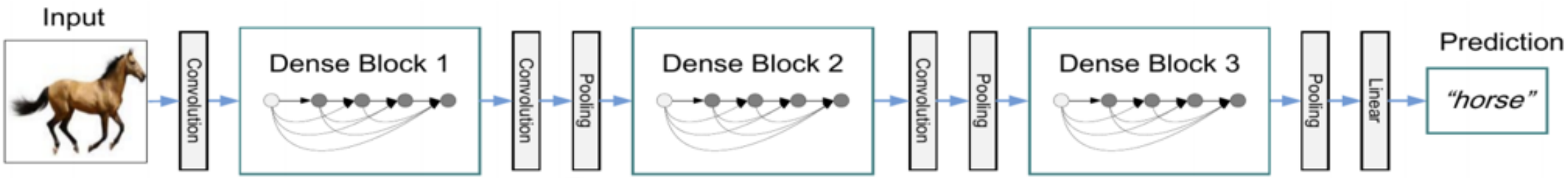
Residual Network: deeper is better



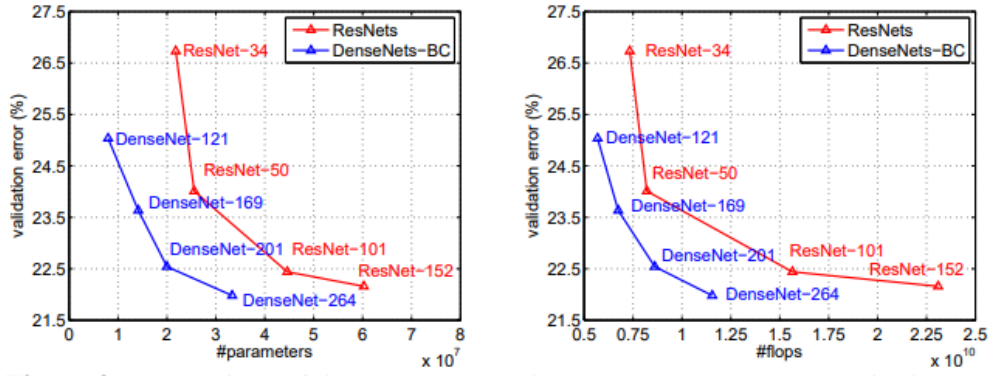


CNNs, 2017: DenseNet

Densely Connected Convolutional Networks, CVPR 2017
Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger



Recently proposed, better performance/parameter ratio



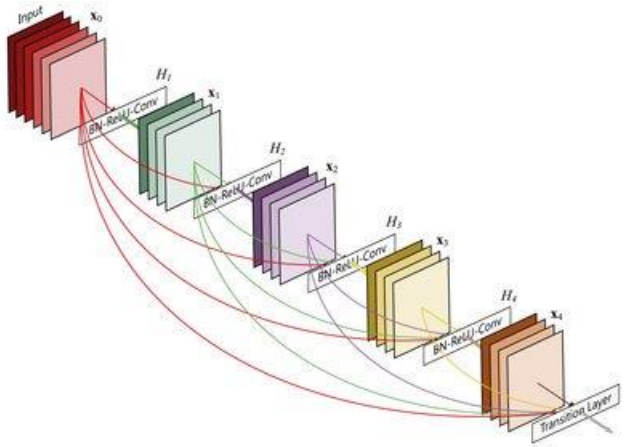
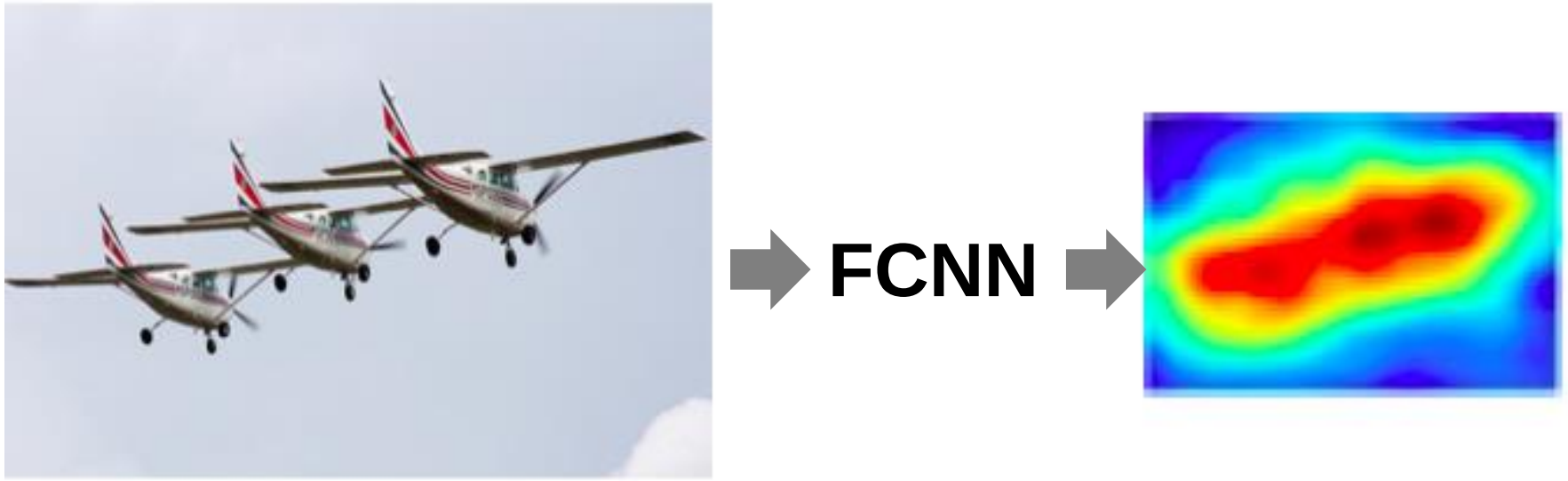


Image-to-Image

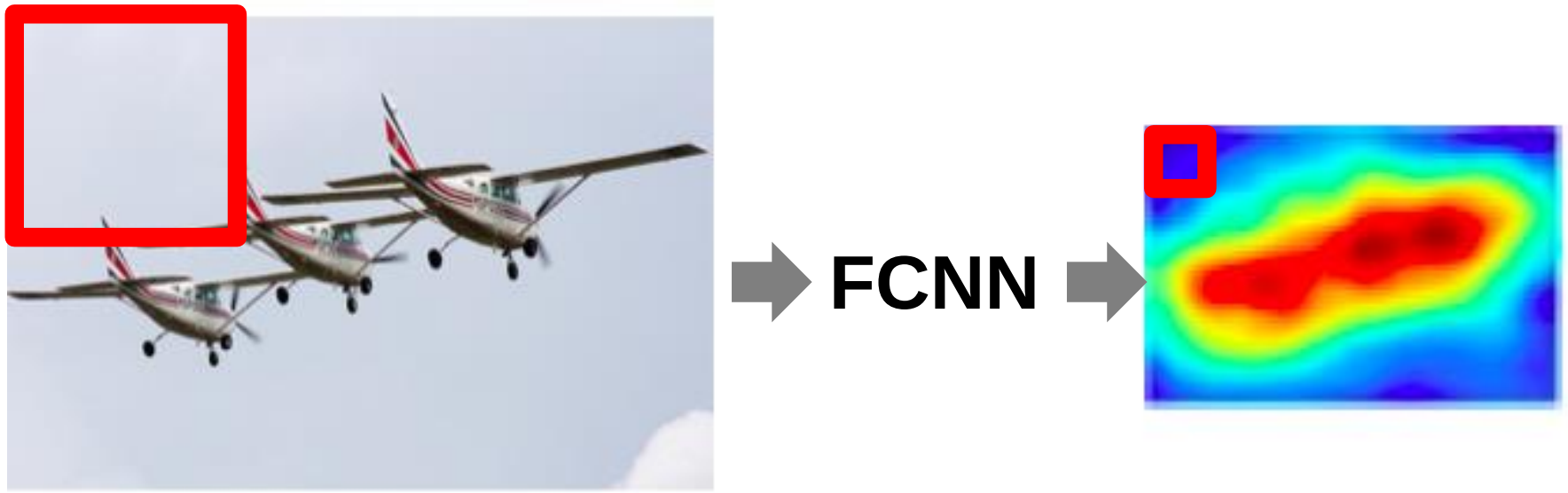
Image-to-image

- So far we mapped an image image to a number or label
- In graphics, output often is “richer”:
  - An image
  - A volume
  - A 3D mesh
  - …
- Architectures
  - Encoder-Decoder
  - Skip connections

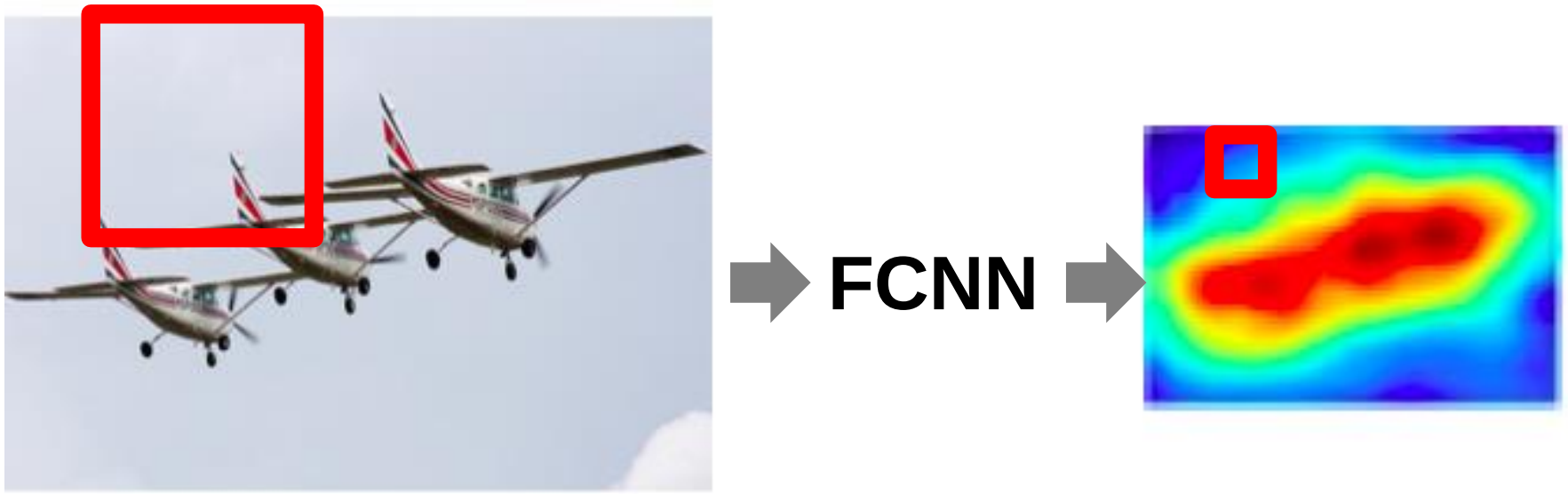
Fully-convolutional Neural Networks



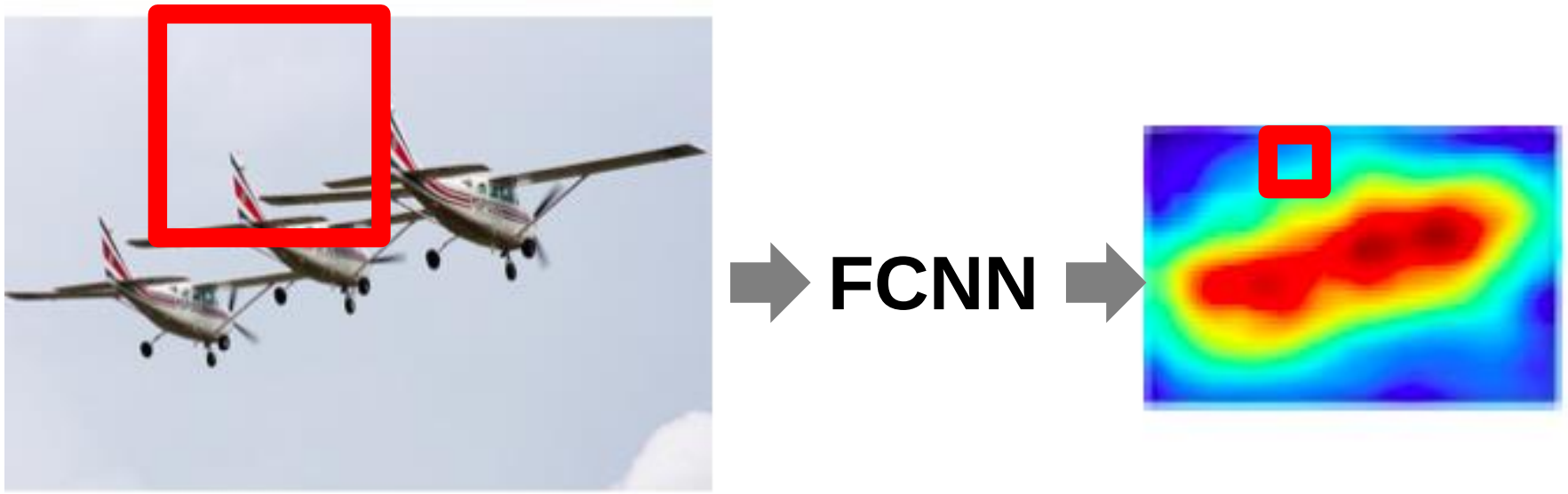
Fully-convolutional Neural Networks



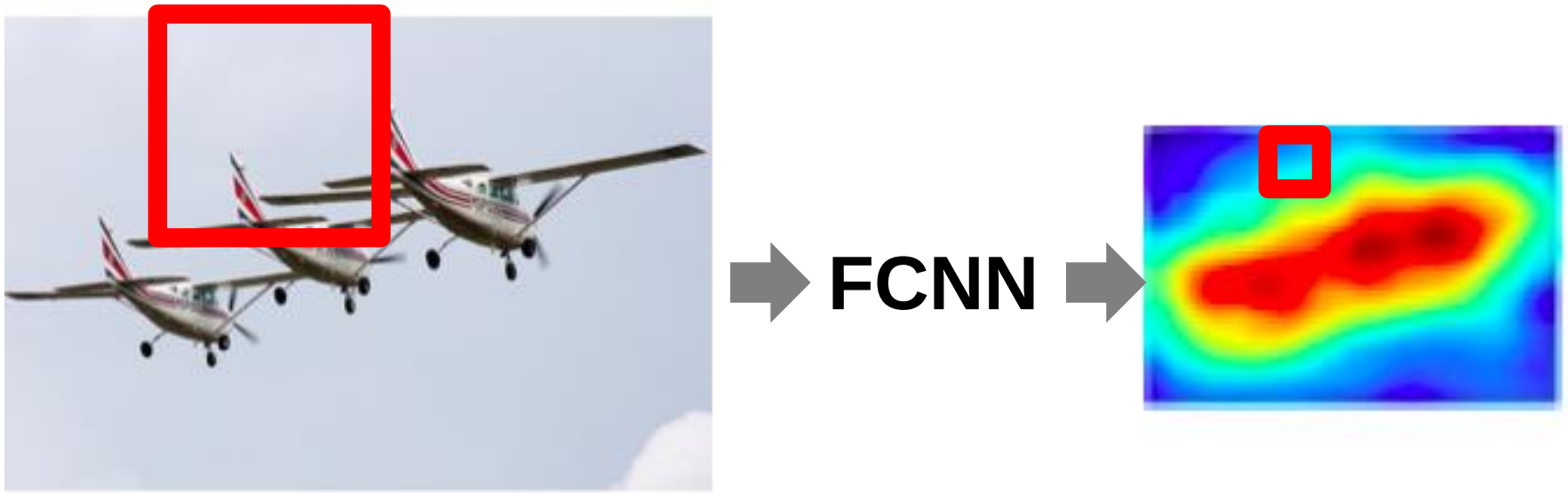
Fully-convolutional Neural Networks



Fully-convolutional Neural Networks



Fully-convolutional Neural Networks

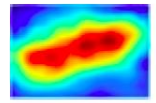


Fast (shared convolutions)
Simple (dense)

Fully Convolutional Neural Networks in Practice

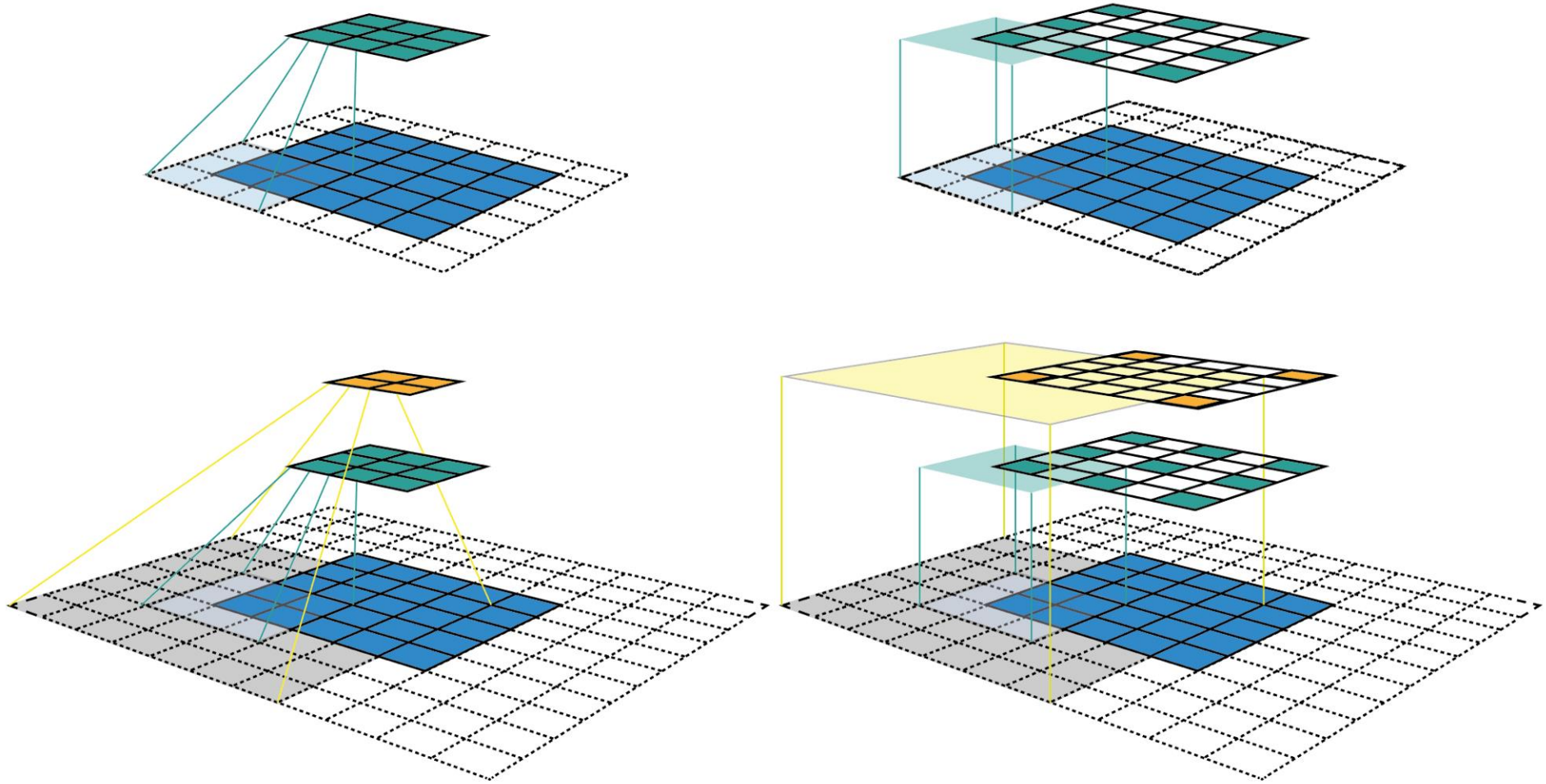
32-fold decimation
224x224 to 7x7

FCNN



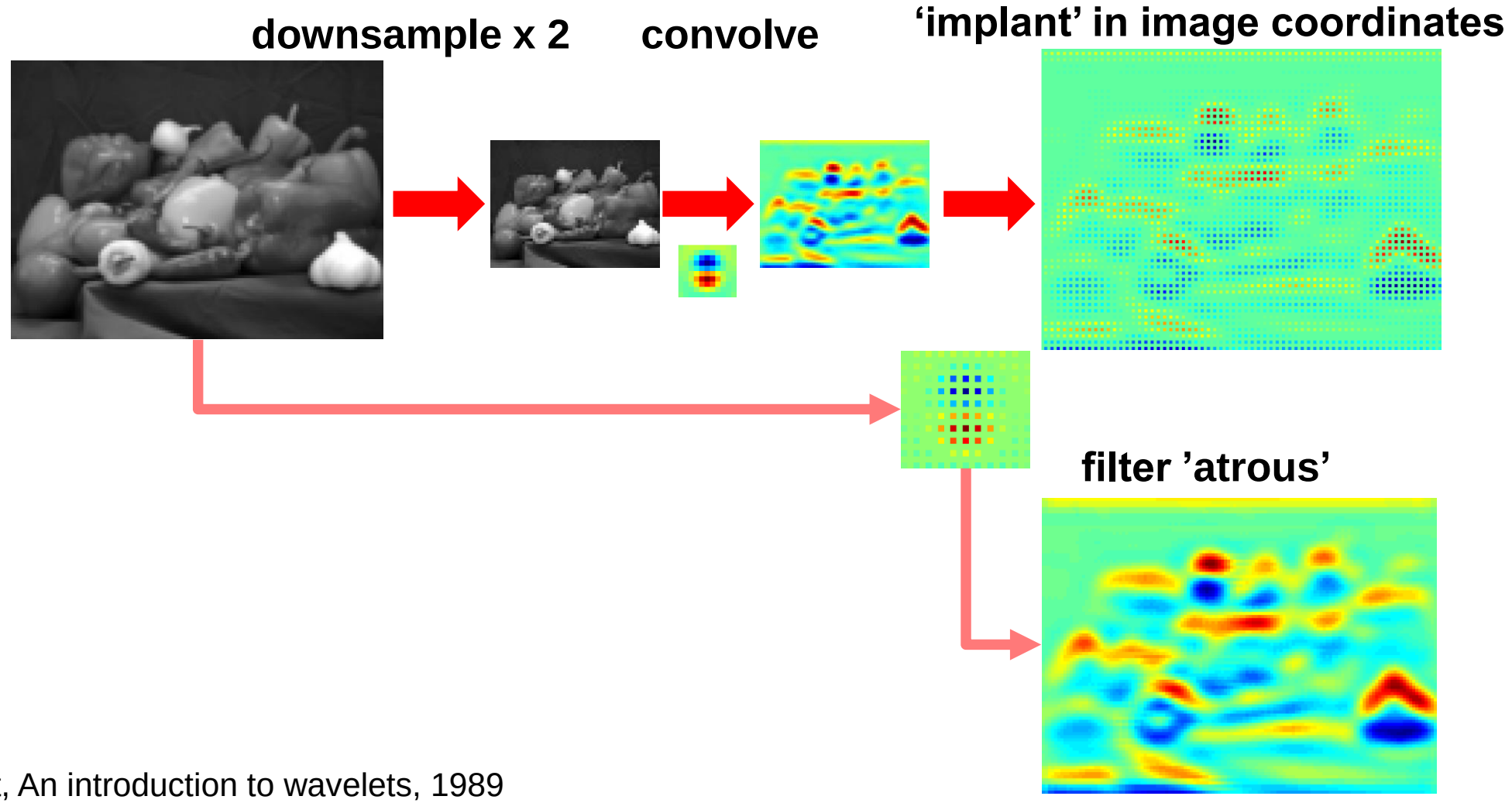
Fast (shared convolutions)
Simple (dense)
Low resolution

Receptive field arithmetic



https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807

Atrous convolution



S. Mallat, An introduction to wavelets, 1989

DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs
Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, Alan L. Yuille

Atrous convolution = Dilated Convolution

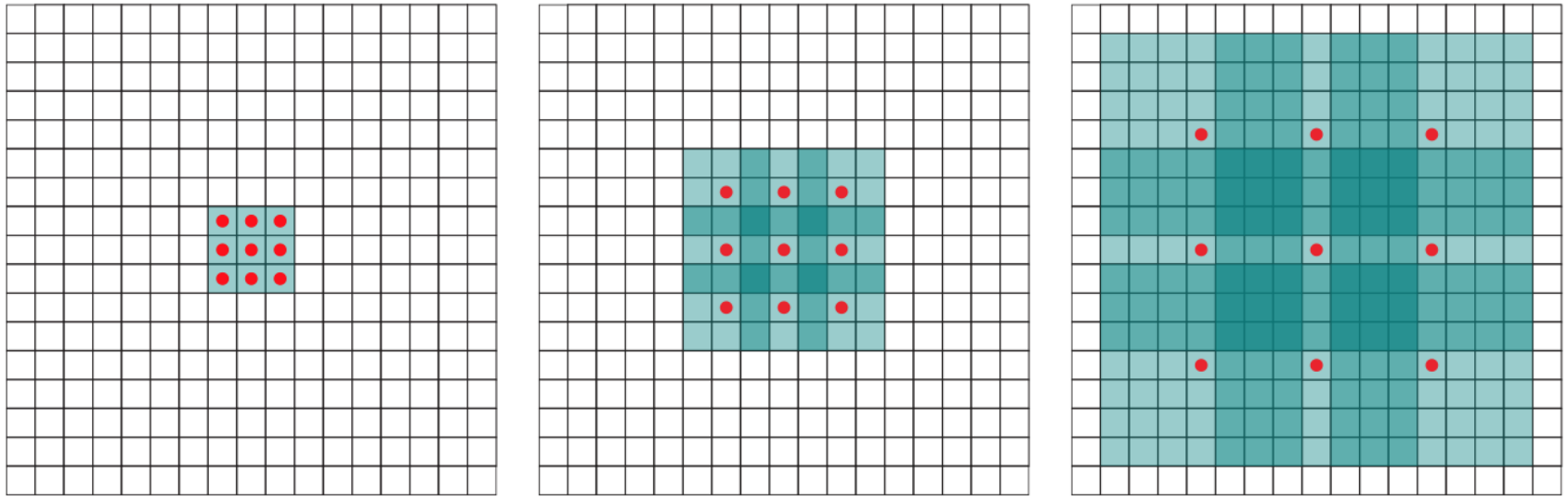
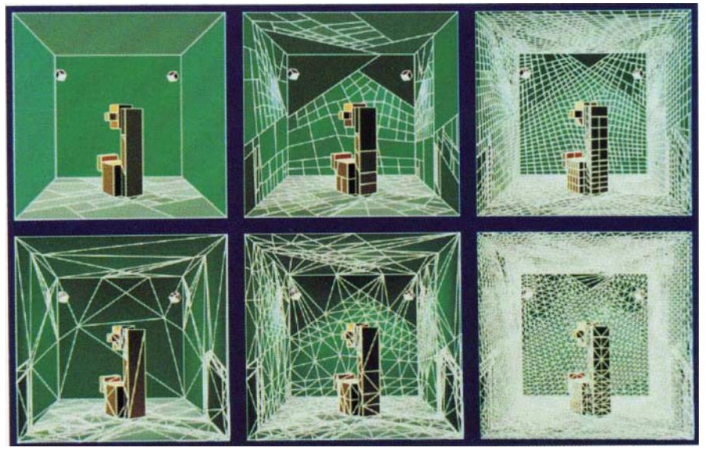
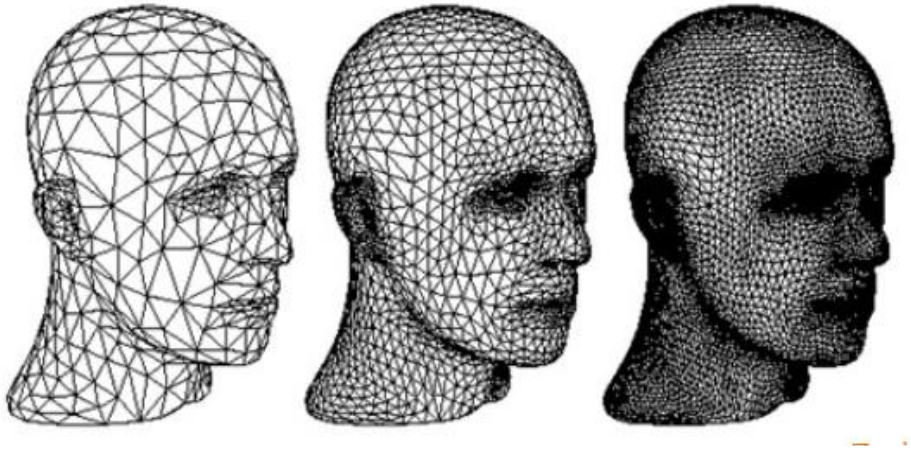


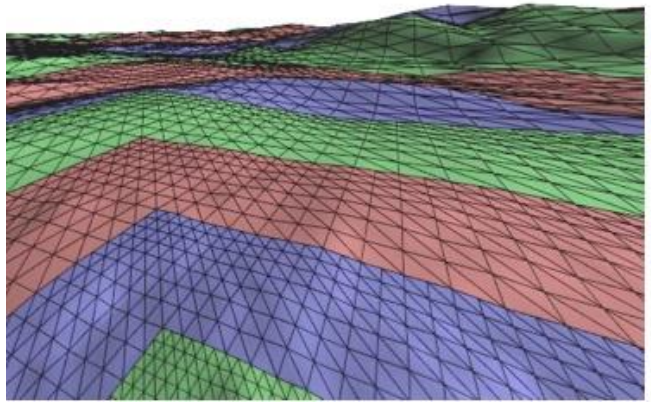
Figure 1: Systematic dilation supports exponential expansion of the receptive field without loss of resolution or coverage. (a) F_1 is produced from F_0 by a 1-dilated convolution; each element in F_1 has a receptive field of 3×3. (b) F_2 is produced from F_1 by a 2-dilated convolution; each element in F_2 has a receptive field of 7×7. (c) F_3 is produced from F_2 by a 4-dilated convolution; each element in F_3 has a receptive field of 15×15. The number of parameters associated with each layer is identical. The receptive field grows exponentially while the number of parameters grows linearly.

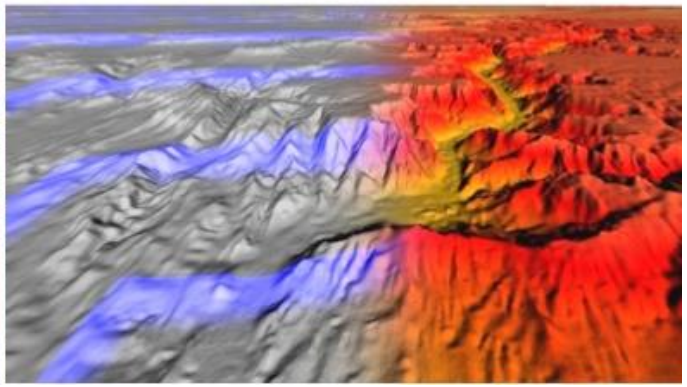
F. Yu, V. Koltun, Multi-Scale Context Aggregation by Dilated Convolutions, ICLR 2016

Graphics: Multiresolution

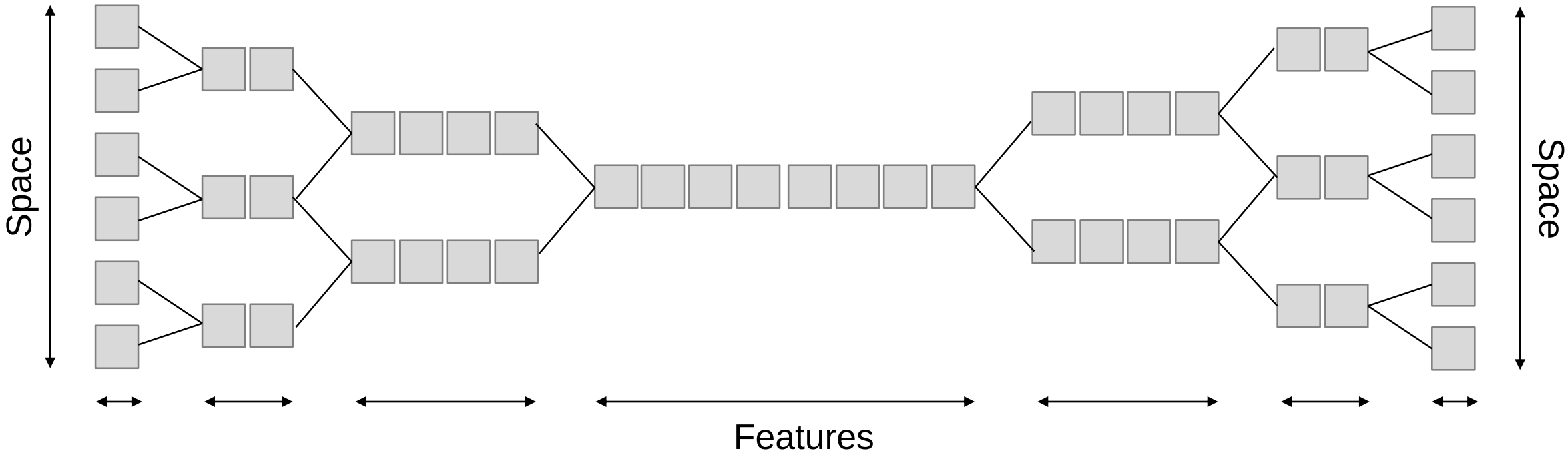








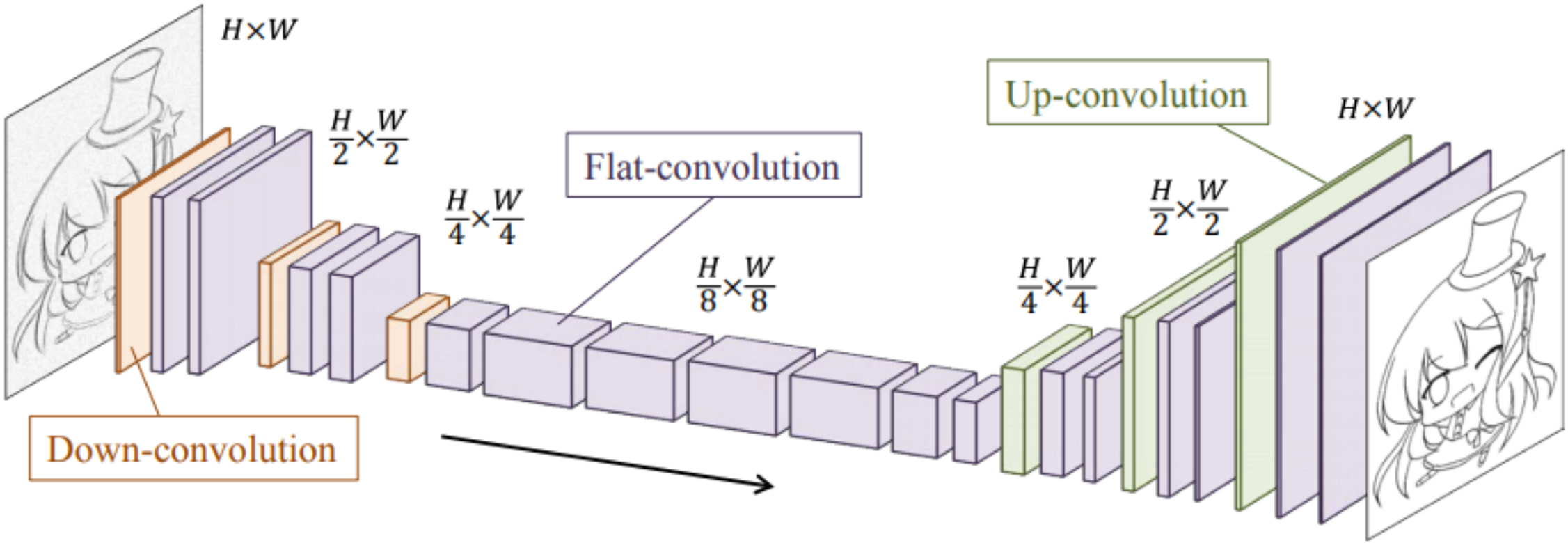
Encoder-decoder



Interpretation

- Turns image into vector
- This vector is a very compact and abstract “code”
- Turns code back into image

Encoder-decoder



Learning to simplify. Simo-Serra et al. 2016

Code example

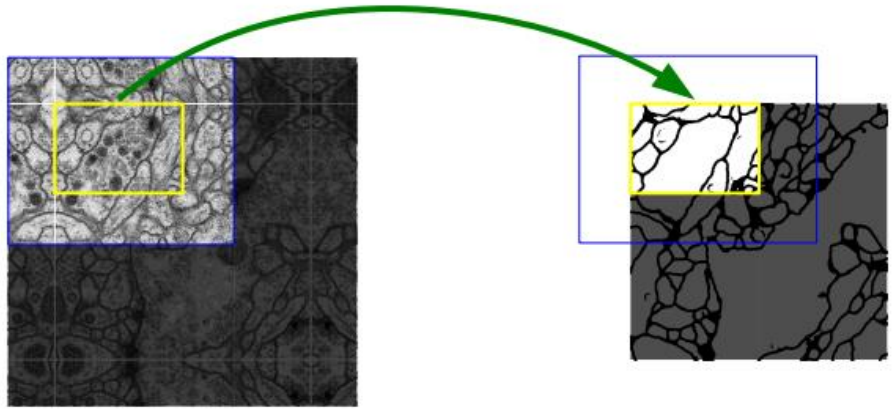
Colorization Network

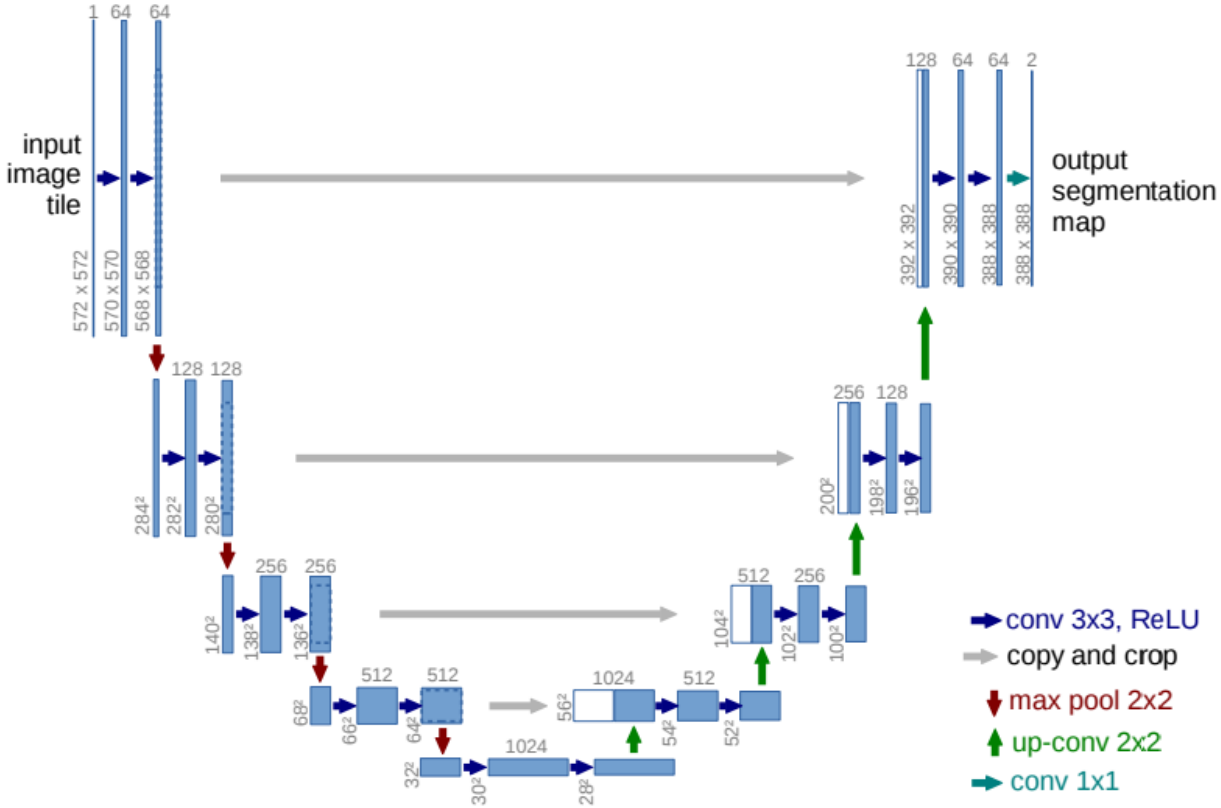
Up-sampling

- We saw
  - … how to keep resolution
  - … how to reduce it with pooling
- But how to increase it again?
- Options
  - Interpolation
  - Padding (insert zeros)
  - Transpose convolutions

Encoder-decoder + Skip connections

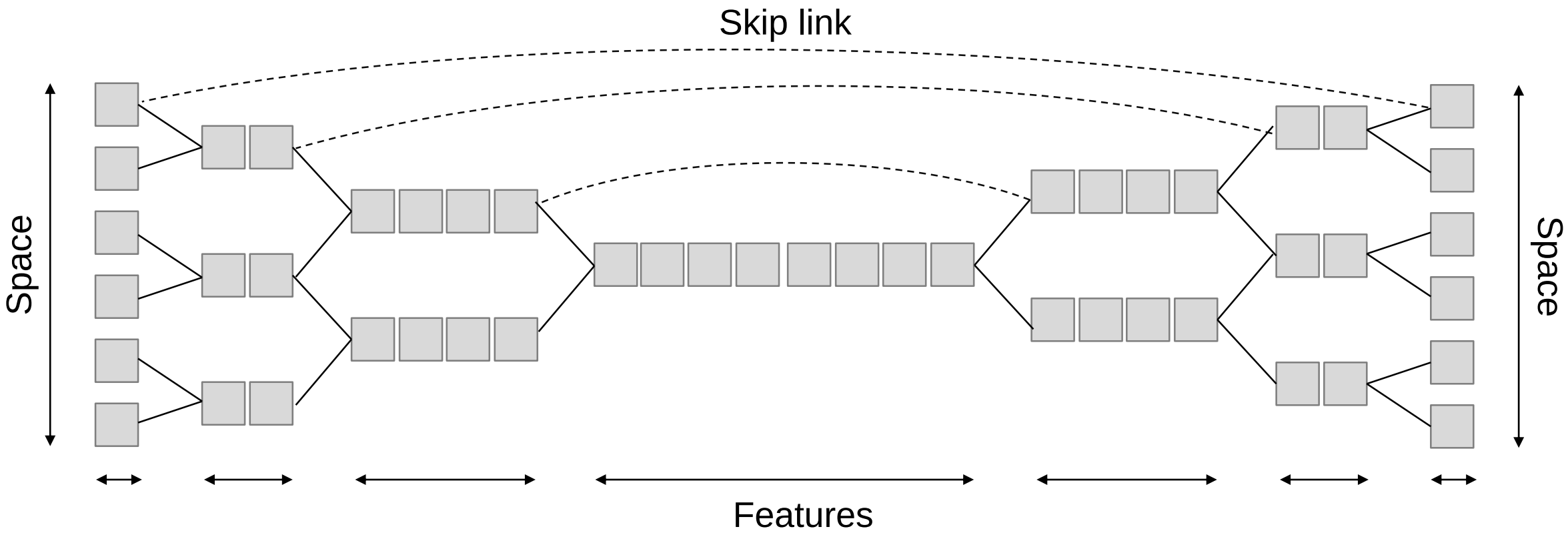
- 1st: Reduce resolutions as before
- 2nd: Increase resolution
- Transposed convolutions





U-Net: Convolutional Networks for Biomedical Image Segmentatio. Ronneberger et al. 2015

Encoder-decoder with skip connections



Interpretation

- Turns image into vector
- Turns vector back into image
- At every step of increasing the resolution, check back with the input to preserve details
- Familiar trick to graphics people
  - (Haar) wavelet
  - Residual coding
  - Pyramidal schemes (Laplacian pyramid, etc.)

Deep Learning Frameworks

Main frameworks

(Python, C++, Java)

(Python, backends support other languages)

(Python)

(C++, Python, Matlab)

Currently less frequently used

(Python)

(Python)

(Python, C++)

(Python, C++, C#)

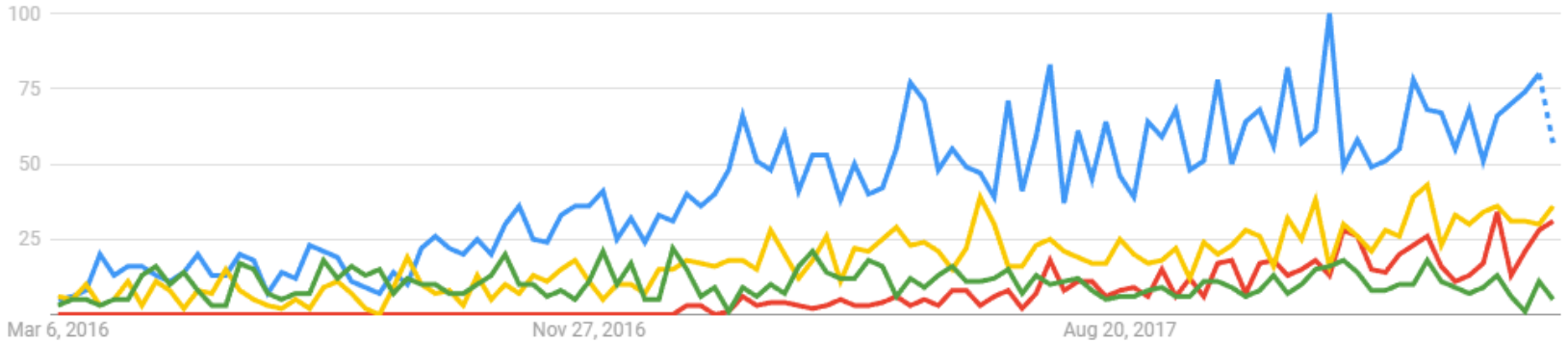
(Matlab)

(Python, Java, Scala)

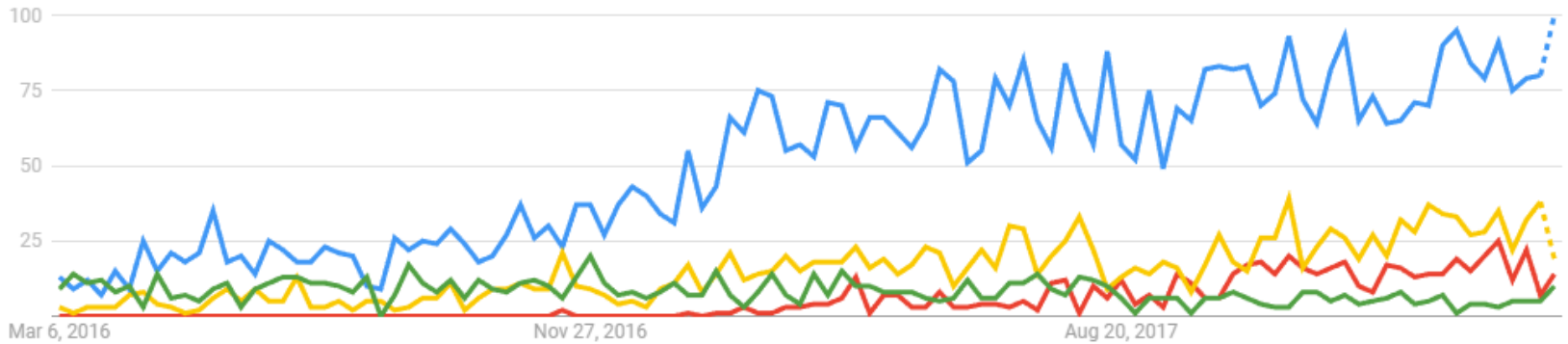
(Python, C++, and others)

Popularity

Google Trends for search terms: “[name] github”



Google Trends for search terms: “[name] tutorial”



TensorFlow™

Keras

PYTORCH

Caffe

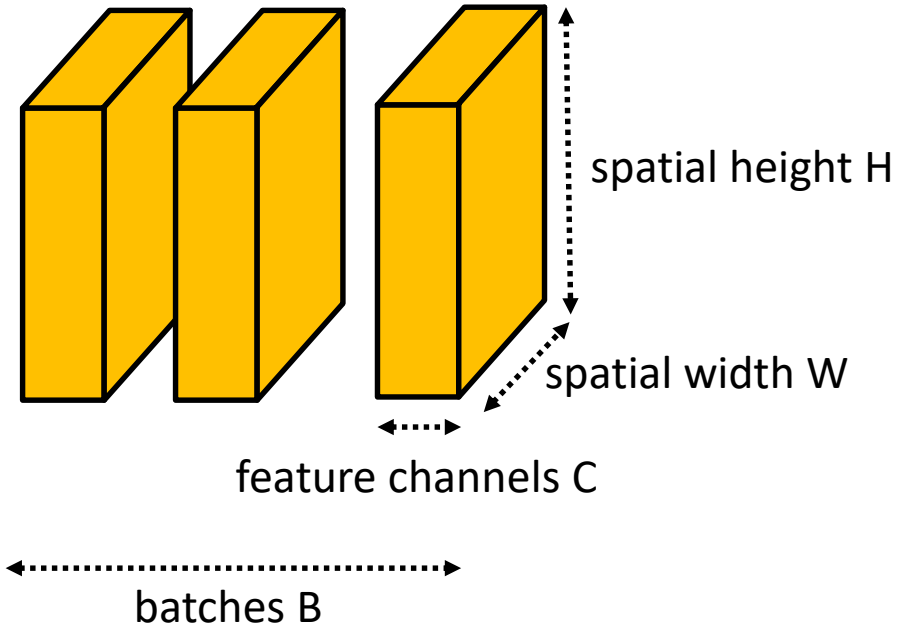
Typical Training Steps

```
for i = 1 .. max_iterations

    input, ground_truth = load_minibatch(data, i)

    output = network_evaluate(input, parameters)

    loss = compute_loss(output, ground_truth)

    # gradients of loss with respect to parameters
    gradients = network_backpropagate(loss, parameters)

    parameters = optimizer_step(parameters, gradients)
```

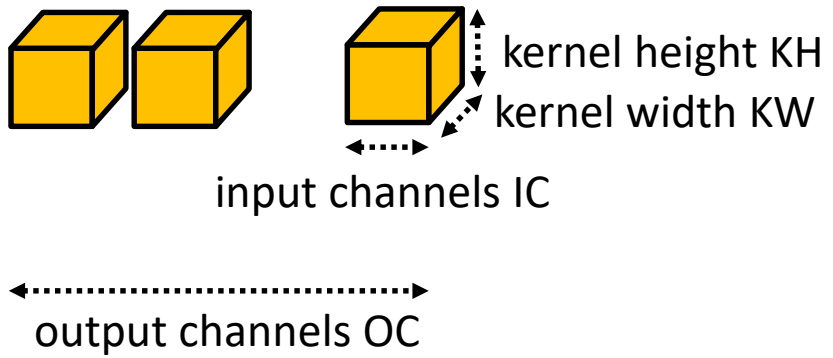
Tensors

- Frameworks typically represent data as tensors
- Examples:

4D input data: B x C x H x W



4D convolution kernel: OC x IC x KH x KW



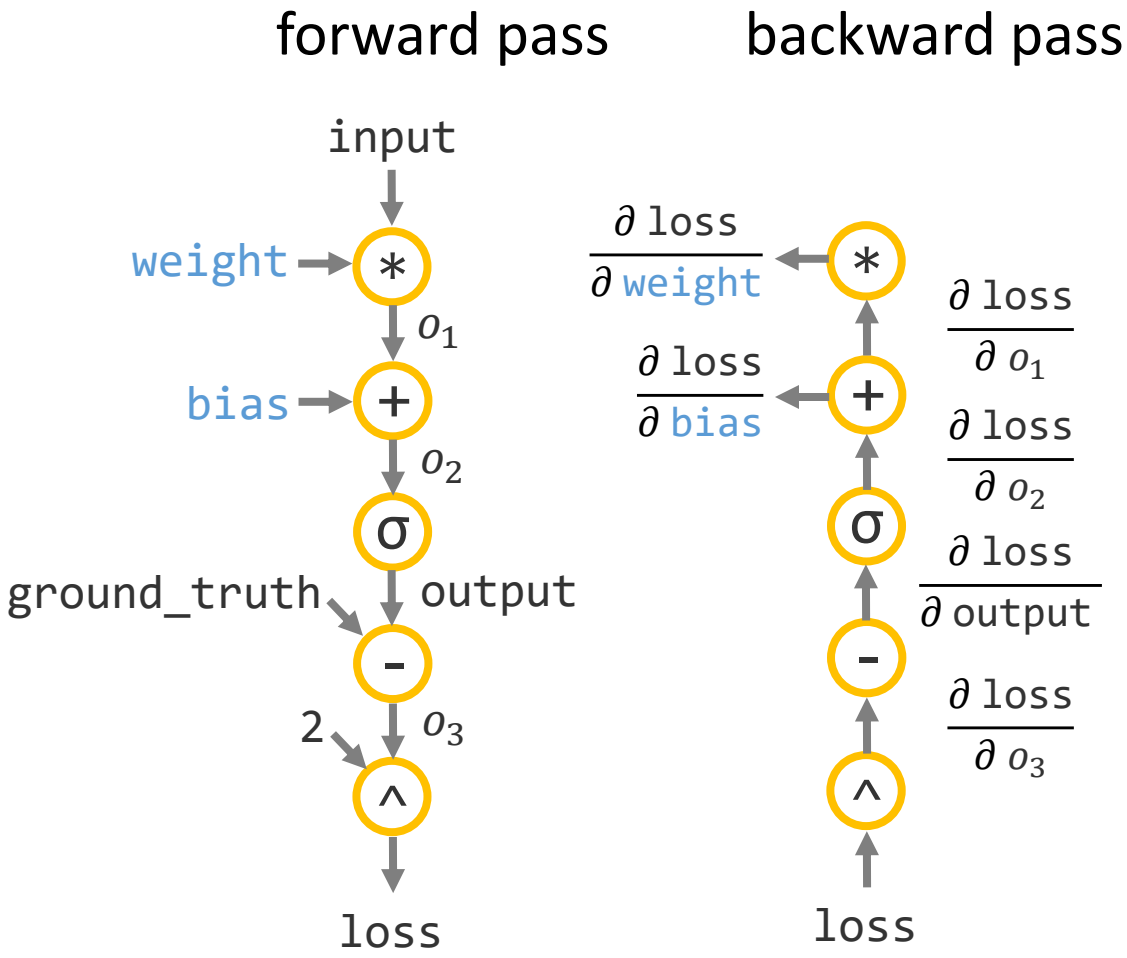
What Does a Deep Learning Framework Do?

- Tensor math
- Common network operations/layers
- Gradients of common operations
- Backpropagation
- Optimizers
- GPU implementations of the above
- usually: data loading, network parameter saving/loading
- sometimes: distributed computing

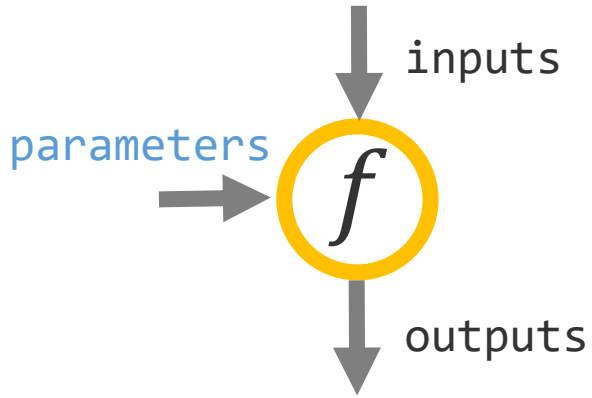
Automatic Differentiation & the Computation Graph

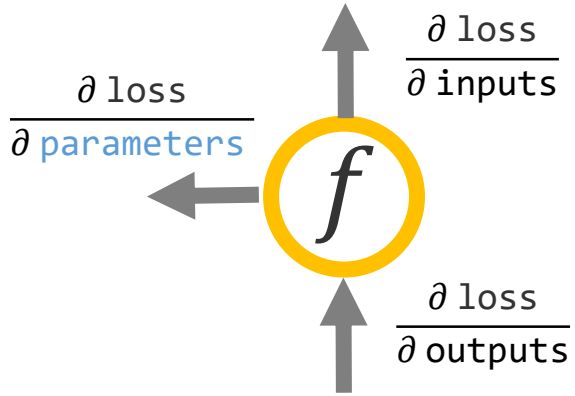
```
parameters = (weight, bias)

output = σ(weight * input + bias)

loss = (output - ground_truth)^2

# gradients of loss with respect to parameters
gradients = backpropagate(loss, parameters)
```

Since loss is a scalar, the gradients
are the same size as the parameters



Automatic Differentiation & the Computation Graph





$$\text{outputs} = \text{forward}(\text{inputs}, \text{parameters})$$

$$\frac{\partial\, \text{loss}}{\partial\, \text{inputs}}\ ,\ \frac{\partial\, \text{loss}}{\partial\, \text{parameters}} = \text{backward}\left(\frac{\partial\, \text{loss}}{\partial\, \text{outputs}}\right)$$

Static vs Dynamic Computation Graphs

- Static analysis allows optimizations and distributing workload
- Dynamic graphs make data-driven control flow easier
- In static graphs, the graph is usually defined in a separate ‘language’
- Static graphs have less support for debugging

define once,
evaluate during training

Static

```
x = Variable()
loss = if_node(x < parameter[0],
    x + parameter[0],
    x - parameter[1])

for i = 1 .. max_iterations
    x = data()
    run(loss)
    backpropagate(loss, parameters)
```

define implicitly by running operations,
a new graph is created in each evaluation

Dynamic

```
for i = 1 .. max_iterations
    x = data()
    if x < parameter[0]
        loss = x + parameter[0]
    else
        loss = x – parameter[1]
    backpropagate(loss, parameters)
```

Tensorflow

- Currently the largest community
- Static graphs (dynamic graphs are in development: Eager Execution)
- Good support for deployment
- Good support for distributed computing
- Typically slower than the other three main frameworks on a single GPU

PyTorch

- Fast growing community
- Dynamic graphs
- Distributed computing is in development (some support is already available)
- Intuitive code, easy to debug and good for experimenting with less traditional architectures due to dynamic graphs
- Very Fast

Keras

- A high-level interface for various backends (Tensorflow, CNTK, Theano)
- Intuitive high-level code
- Focus on optimizing time from idea to code
- Static graphs

Caffe

Caffe

- Created earlier than Tensorflow, PyTorch or Keras
- Less flexible and less general than the other three frameworks
- Static graphs
- Legacy - to be replaced by Caffe2: focus is on performance and deployment
  - Facebook's platform for Detectron (Mask-RCNN, DensePose, …)

Converting Between Frameworks

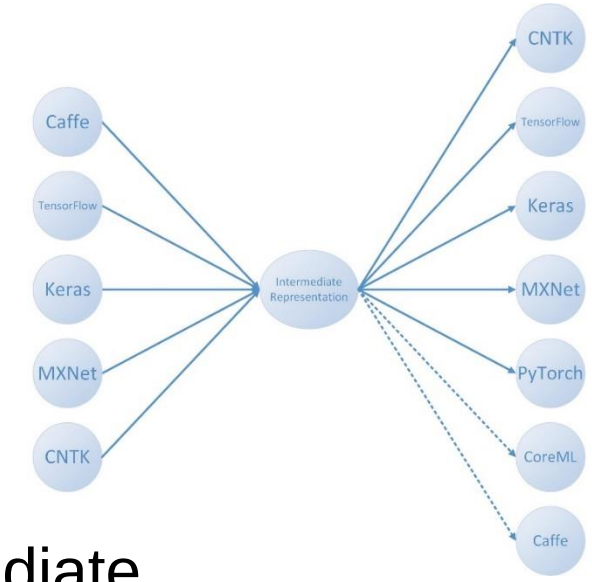
- Example: develop in one framework, deploy in another
- Currently: a large range of converters, but no clear standard
- Standardized model formats are in development

from https://github.com/ysh329/deep-learning-model-convertor

convertor	tensorflow	pytorch	keras	caffe	caffe2	CNTK	chainer	mxnet
tensorflow	-	pytorch-tf/ MMdnn	model-converters/ nn_toolsconvert-to-tensorflow/MMdnn	MMdnn/ nn_tools	None	crosstalk/MMdnn	None	MMdnn
pytorch	pytorch2keras (over Keras)	-	Pytorch2keras/ nn-transfer	Pytorch2caffe/pytorch-caffe-darknet-convert	onnx-caffe2	ONNX	None	None
keras	nn_tools /convert-to-tensorflow/keras_to_tensorflow/keras_to_tensorflow/MMdnn	MMdnn/ nn-transfer	-	MMdnnnn_tools	None	MMdnn	None	MMdnn
caffe	MMdnn/nn_tools/caffe-tensorflow	MMdnn/ pytorch-caffe-darknet-convert/ pytorch-resnet	caffe_weight_converter/ caffe2keras/nn_tools/ kerascaffe2keras/ Deep_Learning_Model_ Converter/MMdnn	-	CaffeToCaffe2	crosstalkcaffe/CaffeConverterMMdnn	None	mxnet/tools/caffe_converter/ResNet_caffe2mxnet/MMdnn
caffe2	None	ONNX	None	None	-	ONNX	None	None
CNTK	MMdnn	ONNX MMdnn	MMdnn	MMdnn	ONNX	-	None	MMdnn
chainer	None	chainer2pytorch	None	None	None	None	-	None
mxnet	MMdnn	MMdnn	MMdnn	MMdnn/MXNet2Caffe/ Mxnet2Caffe	None	MMdnn	None	-

- Standard format for models
- Native support in development for Pytorch, Caffe2, Chainer, CNTK, and MxNet
- Converter in development for Tensorflow

MMdnn



- Converters available for several frameworks
- Common intermediate representation, but no clear standard

The end