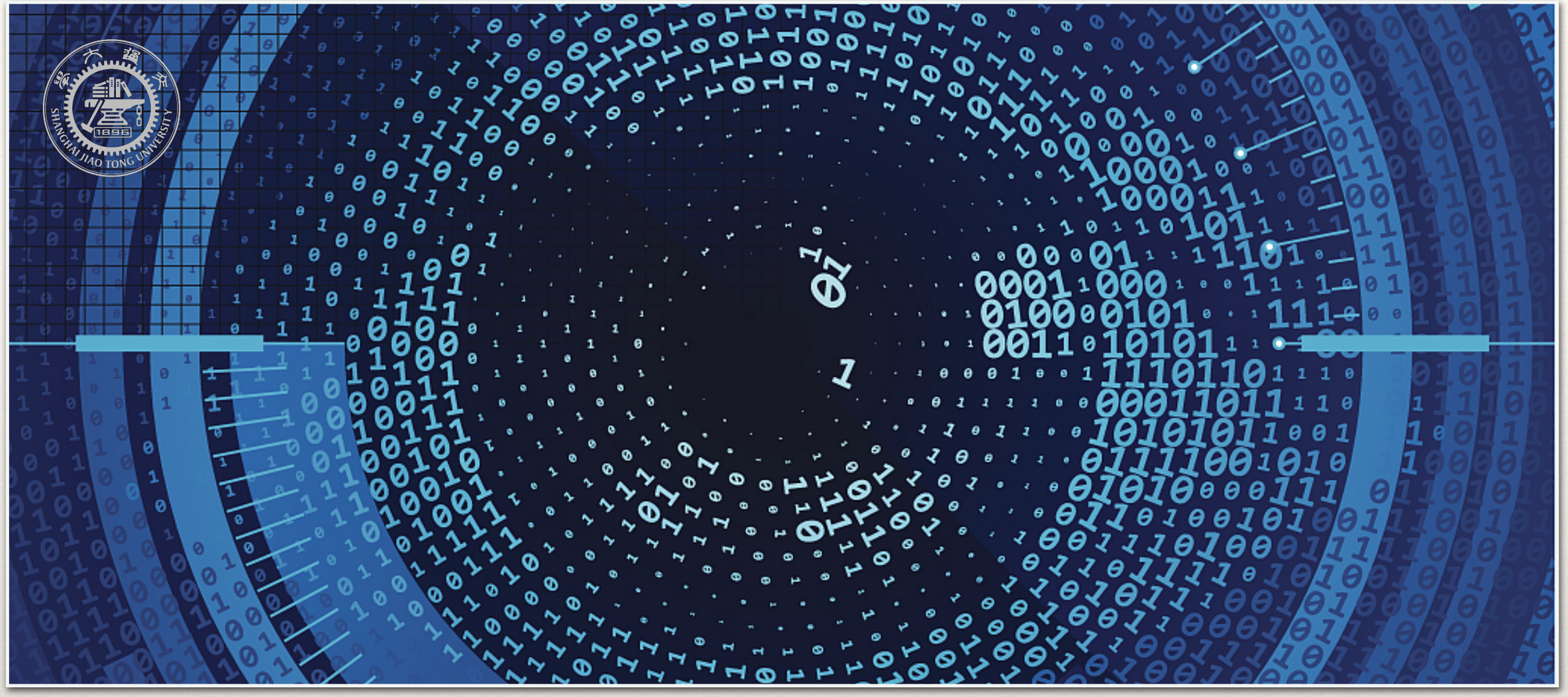


Shanghai Jiao Tong University

Computer Vision

Instructor: Xu Zhao

Class No.: C032703 F032528

Spring 2020

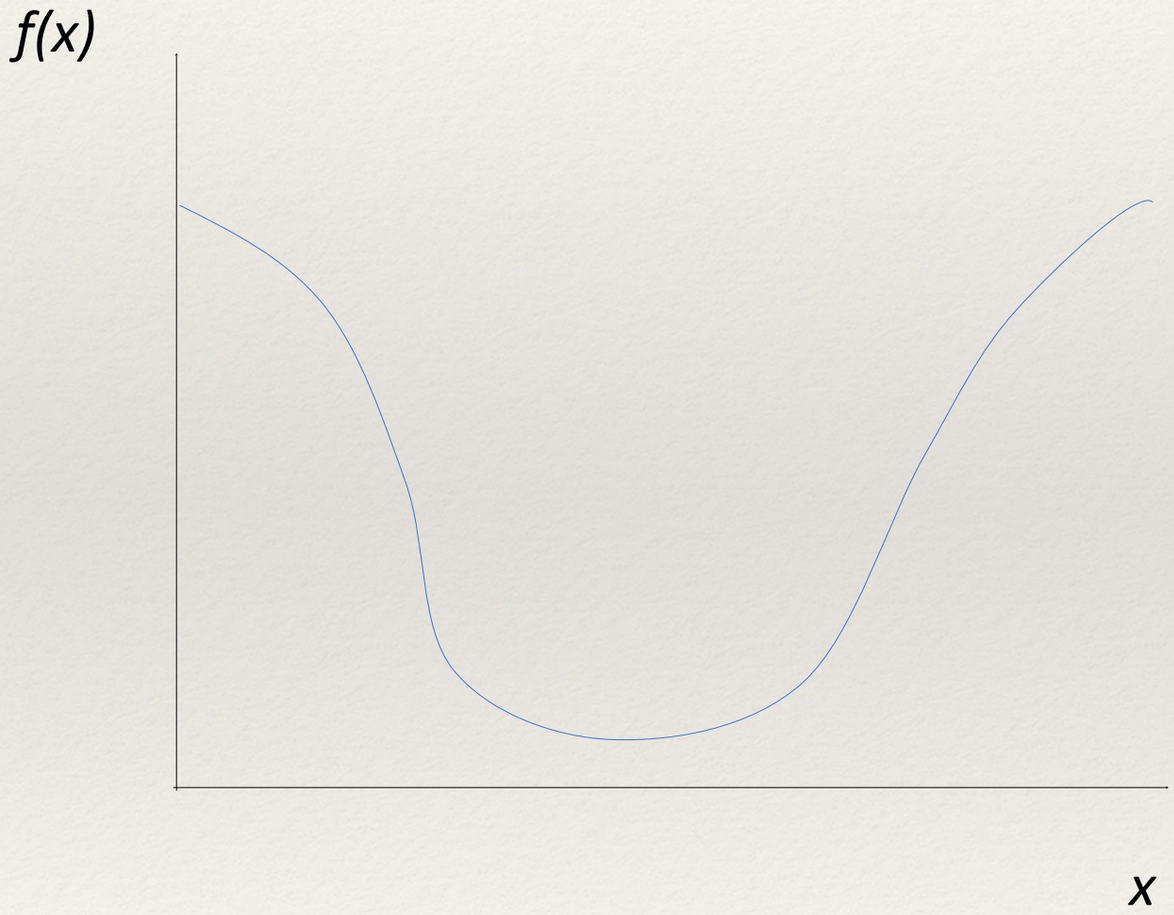
Xu Zhao @ Shanghai Jiao Tong university

Lecture 12: Deep Neural Networks - Part 2

Contents

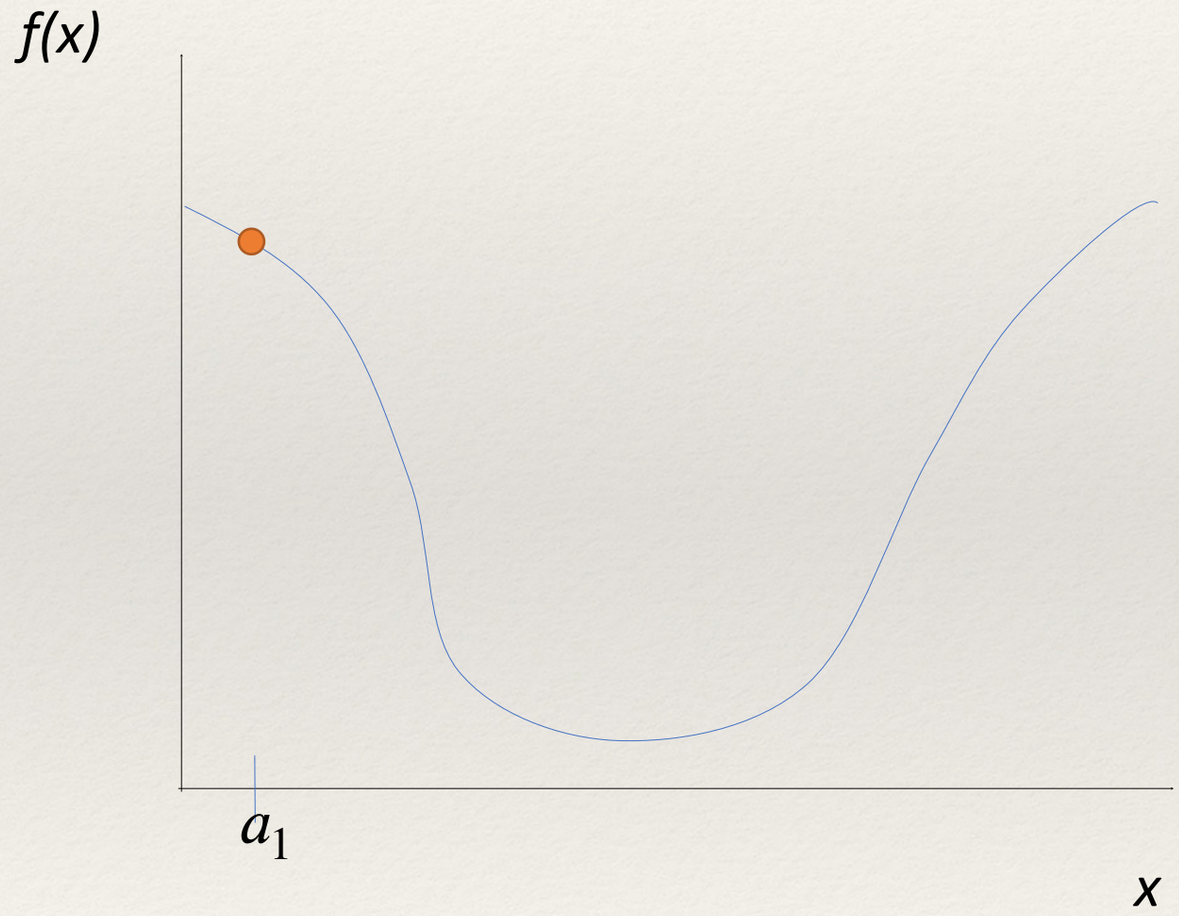
- Training Neural Networks
- Specific Architectures for Diverse Applications

Gradient descent



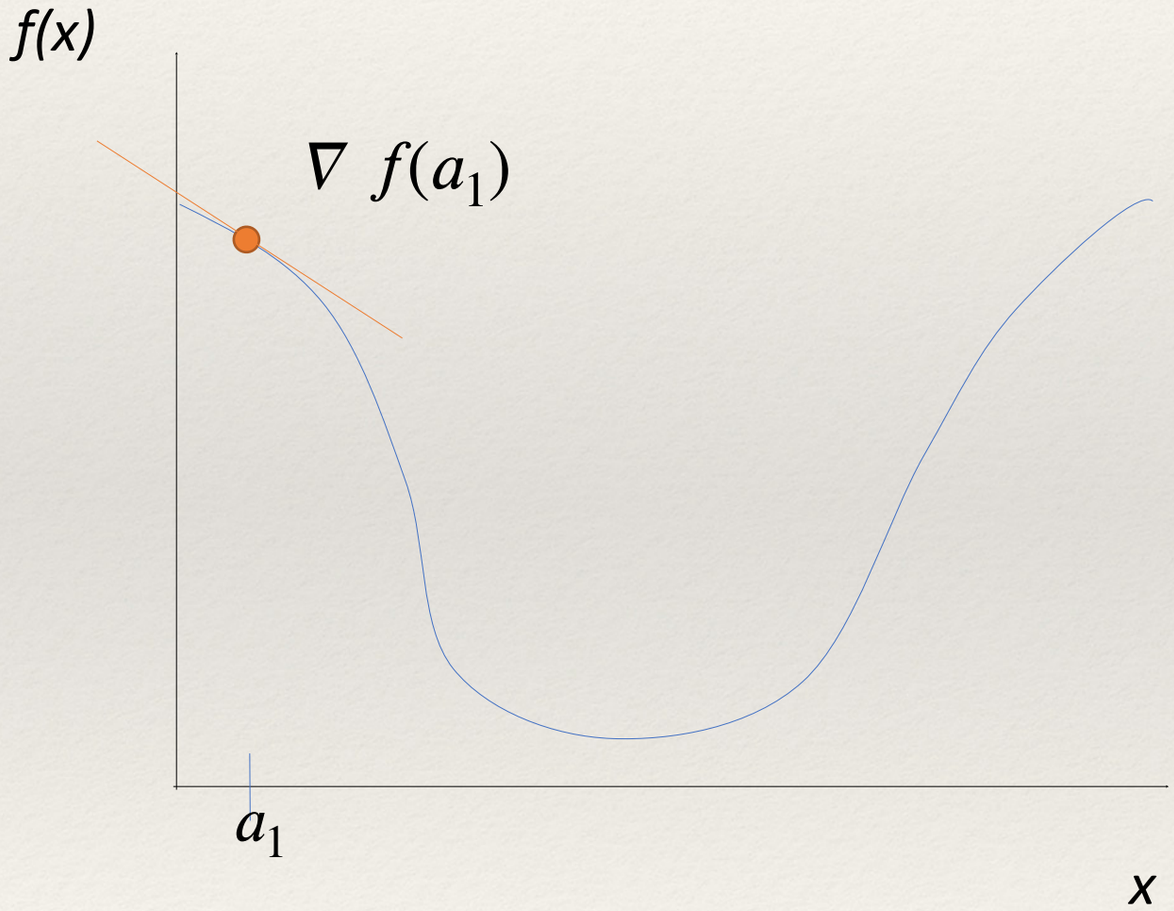
General approach

Pick random starting point.



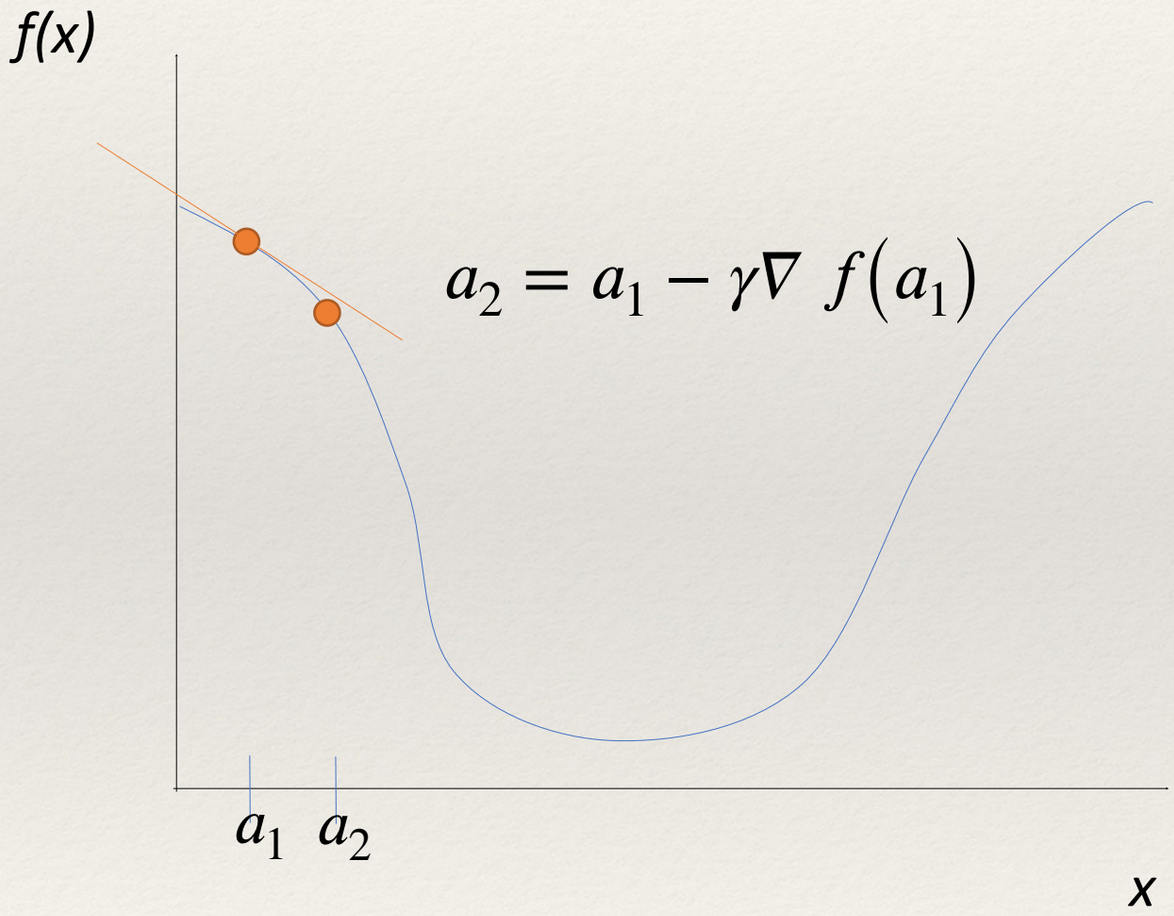
General approach

Compute gradient at point (analytically or by finite differences)



General approach

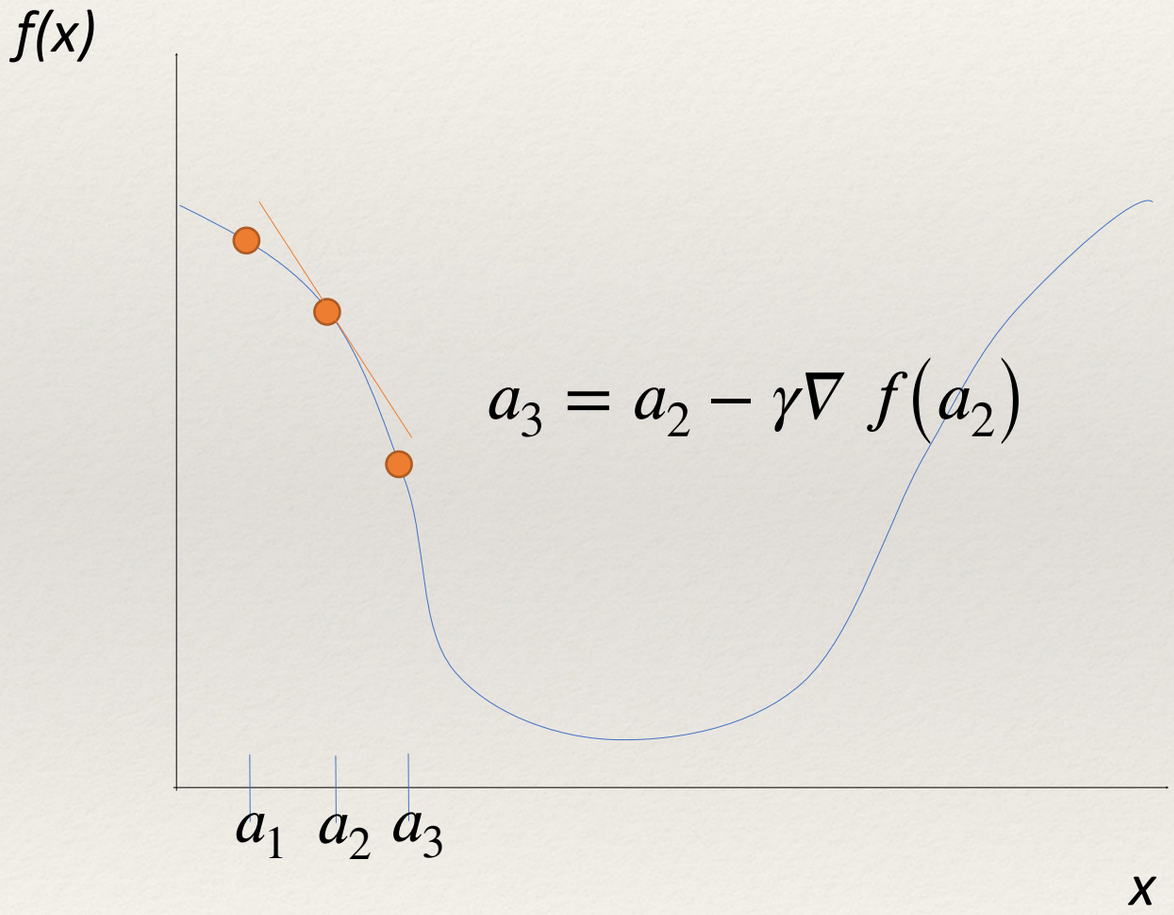
Move along parameter space in direction of negative gradient



γ = amount to move
= *learning rate*

General approach

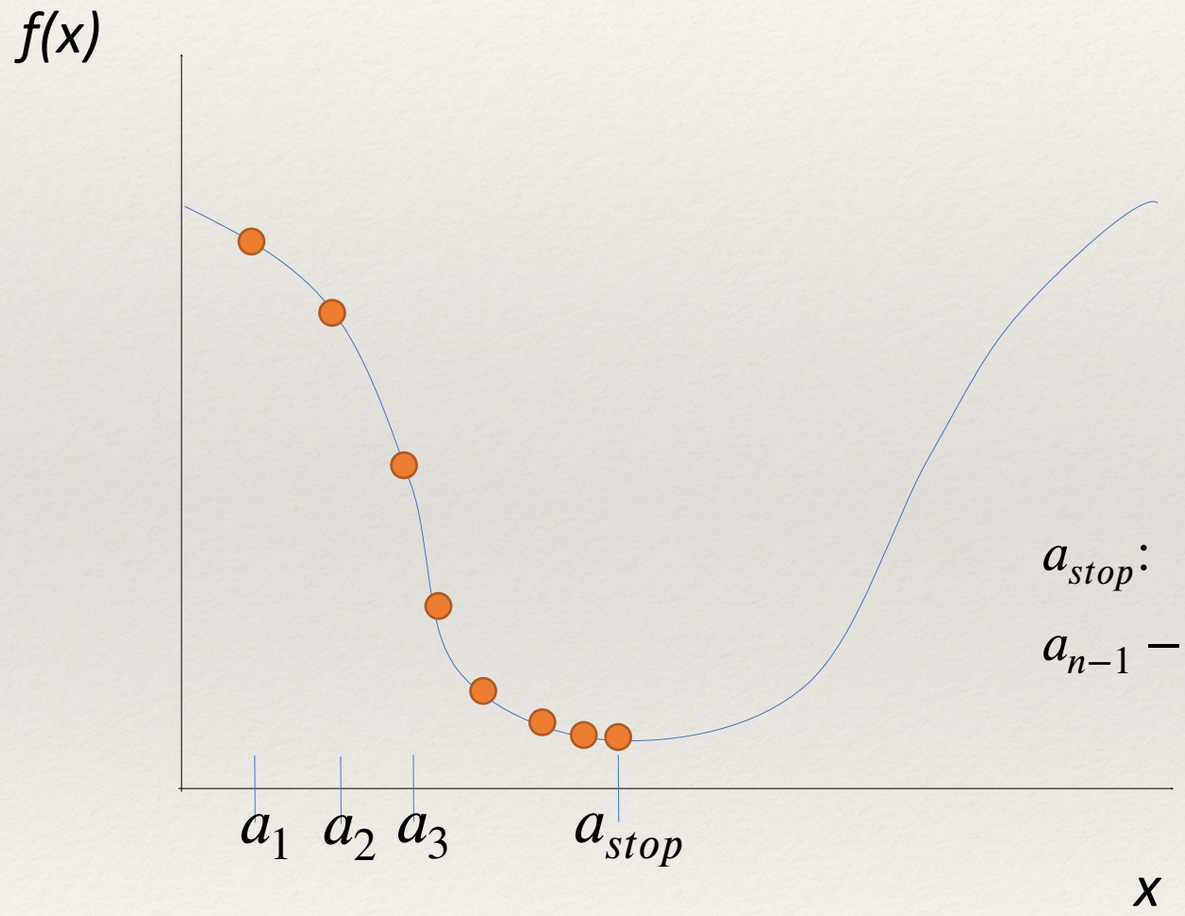
Move along parameter space in direction of negative gradient.



γ = amount to move
= *learning rate*

General approach

Stop when we don't move any more.

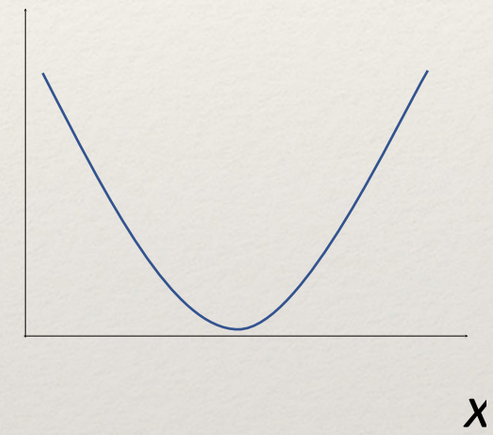


a_{stop}:

$$a_{n-1} - \gamma \nabla f\left(a_{n-1}\right) = 0$$

Gradient descent

- Optimizer for functions.
- Guaranteed to find optimum for convex functions.
  - Non-convex = find *local* optimum.
  - Most vision problems aren't convex.



- Works for multi-variate functions.
  - Need to compute matrix of *partial derivatives ("Jacobian")*

Why would I use this over Least Squares?

If my function is convex,
why can't I just use linear least squares?

$$A\mathbf{x} - \mathbf{b} = 0$$

$$F(\mathbf{x}) = \|A\mathbf{x} - \mathbf{b}\|^2.$$

$$\nabla F(\mathbf{x}) = 2A^T(A\mathbf{x} - \mathbf{b}).$$

Analytic solution = normal equations

$$x = \left(A^T A\right)^{-1} A^T b$$

You can, yes.

Why would I use this over Least Squares?

But now imagine that I have 1,000,000 data points.

Matrices are _huge_.

Even for convex functions, gradient descent allows me to iteratively solve the solution without requiring very large matrices.

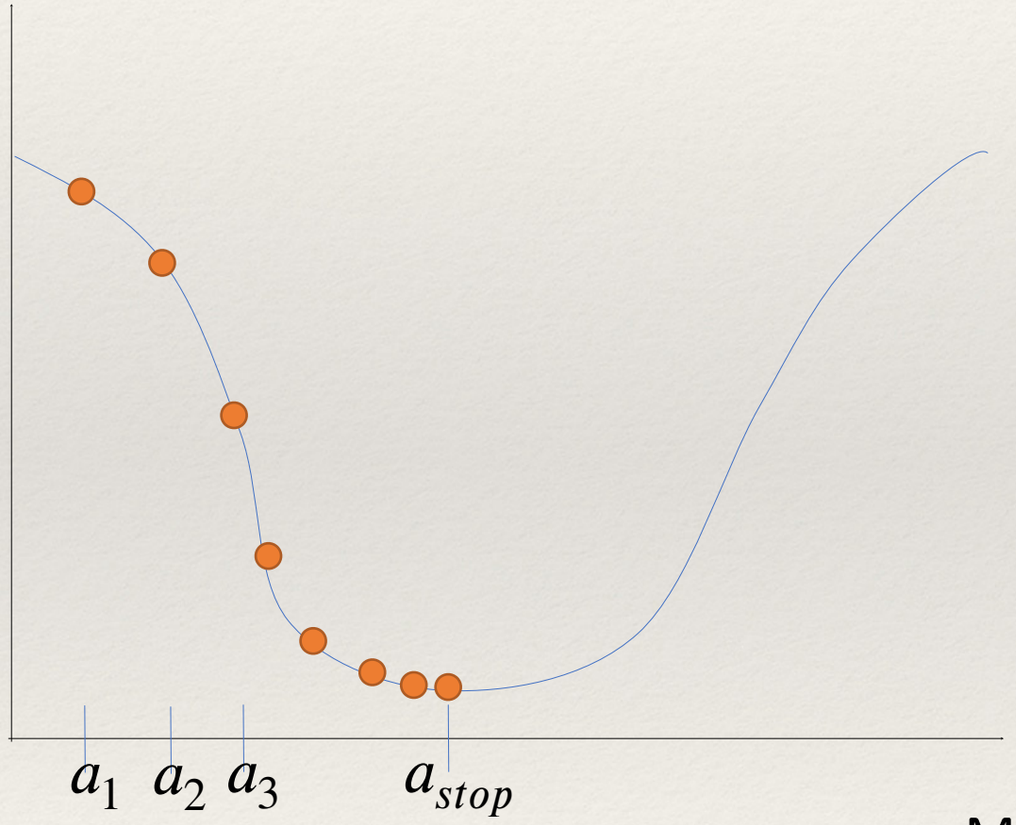
We'll see how.

Train NN with Gradient Descent

- x^i, y^i = *n* training examples
- $f(x)$ = feed forward neural network
- L(**x**, y; **θ**) = some *loss function*
- *Loss function* measures how 'good' our network is at classifying the training examples wrt. the parameters of the model (the perceptron weights).

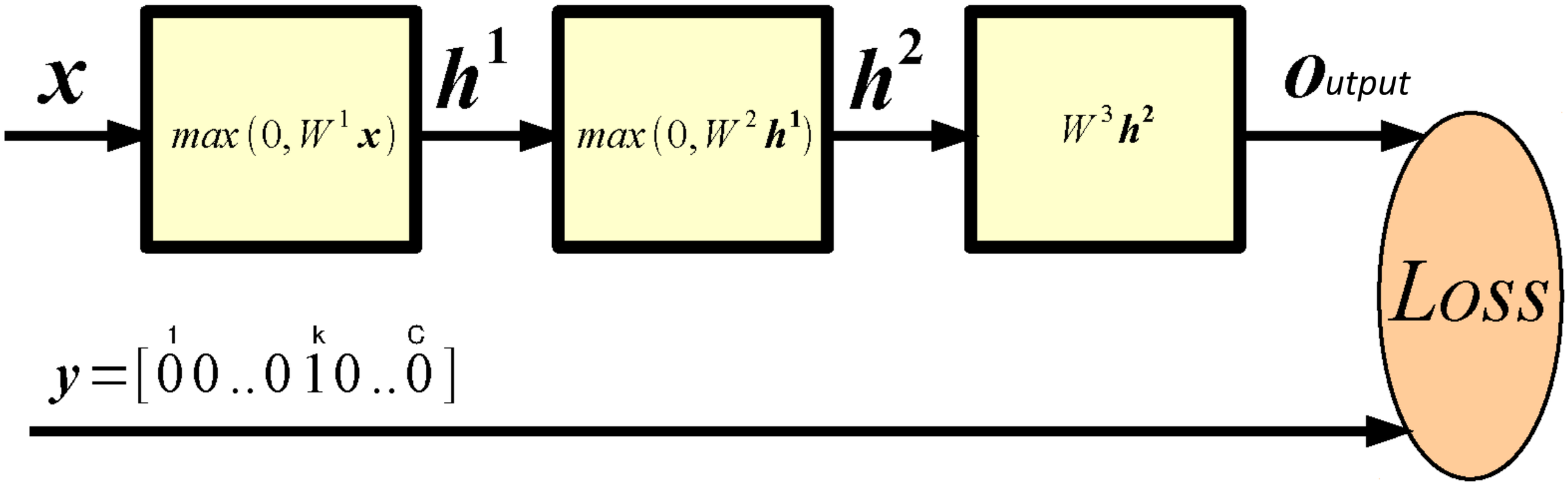
Train NN with Gradient Descent

Loss function
(Evaluate NN
on training data)



Model parameters
(perceptron weights)

How Good is a Network?



What is an appropriate loss?

- Define some output threshold on detection
- Classification: compare training class to output class
- Zero-one loss L (per class)

$y = true\ label \hat{y} = predicted\ label$

$$L(\hat{y}, y) = I(\hat{y} \neq y),$$

- Is it good?
  - Nope – it's a step function.
  - I need to compute the *gradient* of the loss.
  - This loss is not differentiable, and 'flips' easily.

Classification as probability

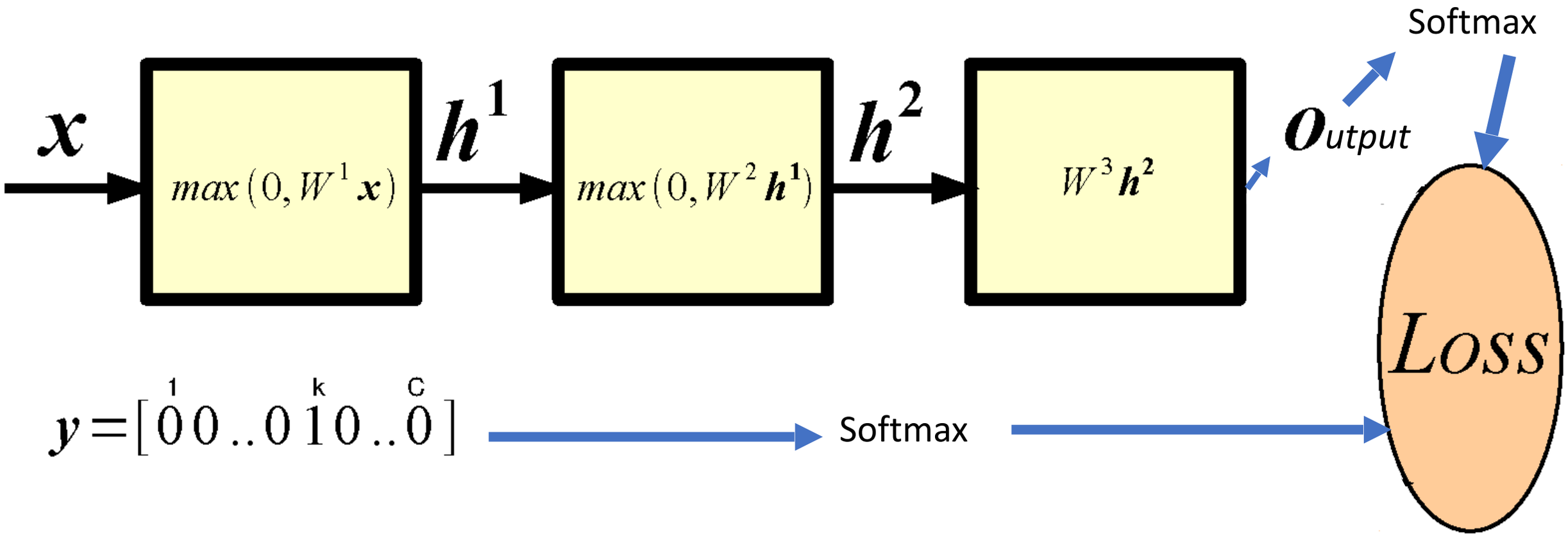
Special function on last layer - 'S*oftmax*':

"Squashes" a C-dimensional vector $\boldsymbol{o}$ of arbitrary real values to a C-dimensional vector $\sigma(\boldsymbol{o})$ of real values in the range (0, 1) that add up to 1.

Turns the output into a probability distribution on classes.

$$p(c_k = 1 | \boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

How Good is a Network?



Probability of class k given input (softmax):

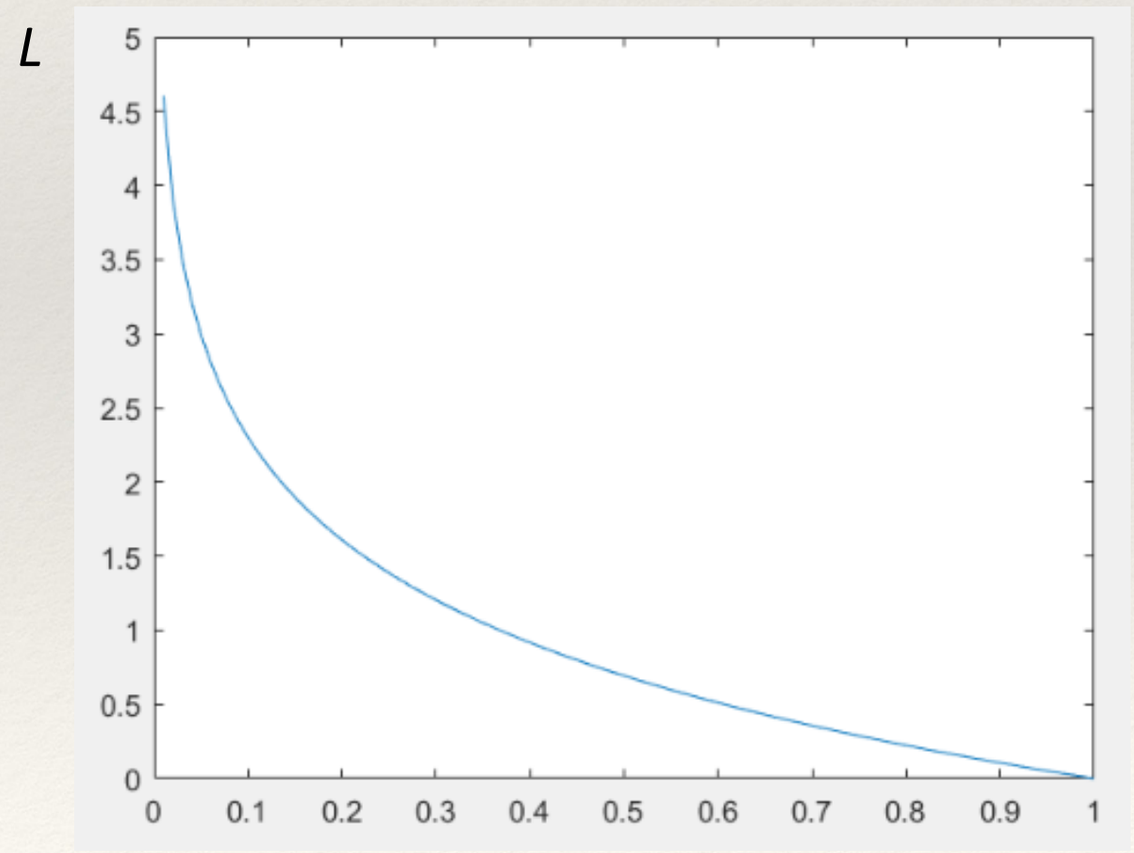
$$p(c_k = 1 | \boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

Cross-entropy loss function

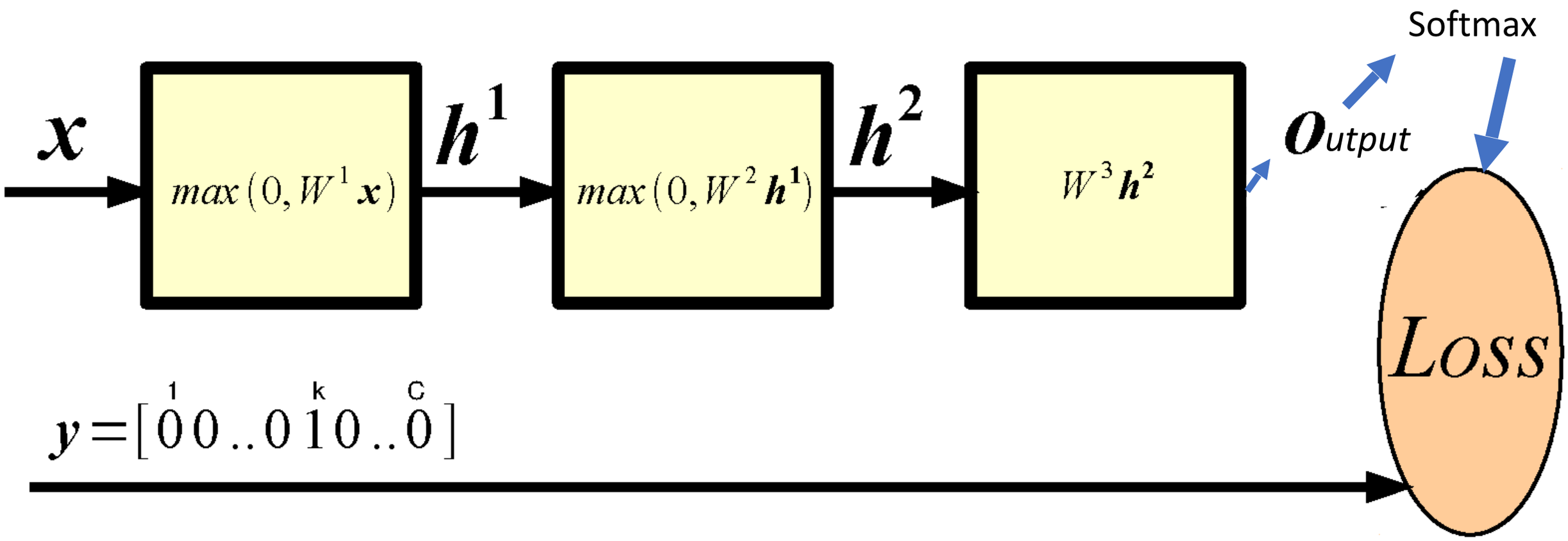
- Negative log-likelihood

$$L\left(\boldsymbol{x}, y; \boldsymbol{\theta}\right) = -\sum_{j} y_j \log p(c_j \mid \boldsymbol{x})$$

- Minimizing this is equivalent to minimizing KL-divergence on predicted and target probability distributions
- Is it a good loss?
  - Differentiable
  - Cost decreases as probability increases



How Good is a Network?



Probability of class k given input (softmax):

$$p(c_k = 1|\boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

(Per-sample) **Loss**; e.g., negative log-likelihood (good for classification of small number of classes):

$$L(\boldsymbol{x}, y; \boldsymbol{\theta}) = -\sum_j y_j \log p(c_j|\boldsymbol{x})$$

Training

Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set.

$$\boldsymbol{\theta}^* = arg\,min_{\boldsymbol{\theta}} \sum_{n=1}^{P} L\left(\boldsymbol{x}^n, y^n; \boldsymbol{\theta}\right)$$

Training

Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set.

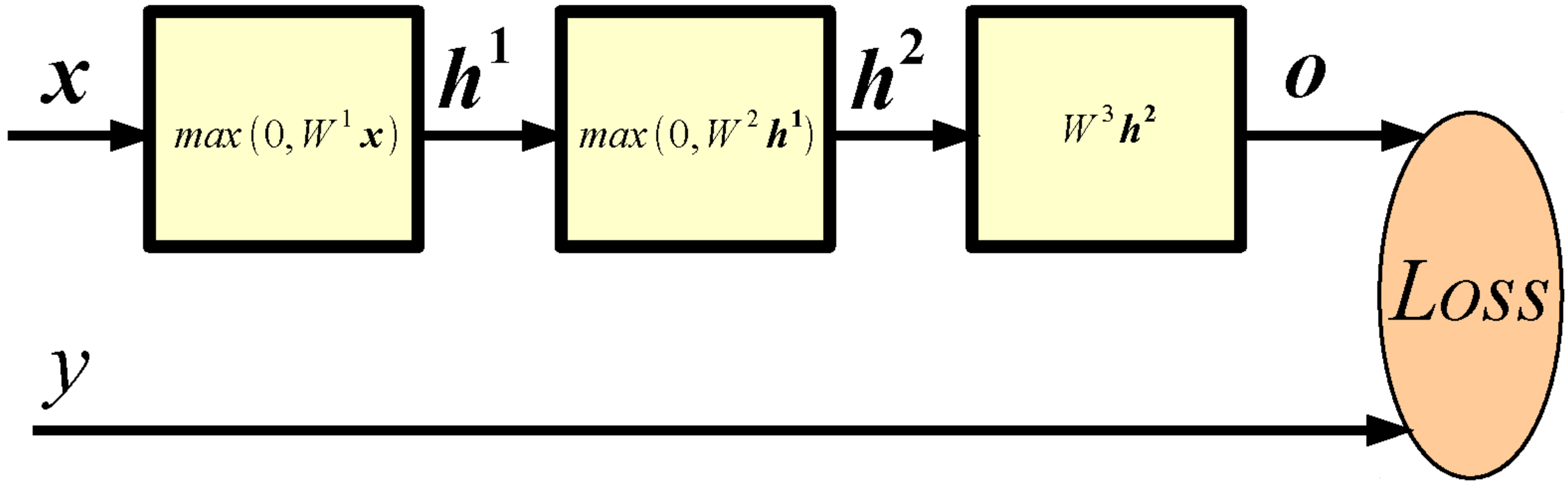
$$\boldsymbol{\theta}^* = arg\,min_{\boldsymbol{\theta}} \sum_{n=1}^{P} L\left(\boldsymbol{x}^n, y^n; \boldsymbol{\theta}\right)$$

Question: How to minimize a complicated function of the parameters?

Answer: Chain rule, a.k.a. **Backpropagation**! That is the procedure to compute gradients of the loss w.r.t. parameters in a multi-layer neural network.

Rumelhart et al. “Learning internal representations by back-propagating..” Nature 1986

Key Idea: Wiggle To Decrease Loss

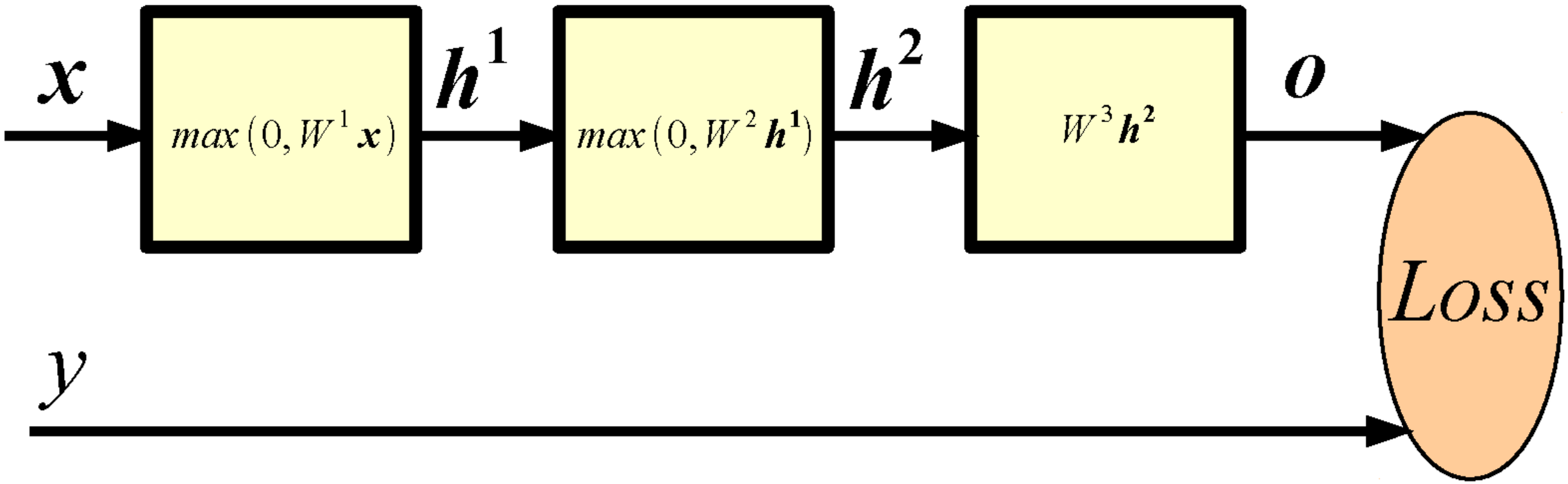


Let's say we want to decrease the loss by adjusting $W^1_{i,j}$.
We could consider a very small $\epsilon = 1\text{e-}6$ and compute:

$$L(\boldsymbol{x}, y; \boldsymbol{\theta})$$

$$L(\boldsymbol{x}, y; \boldsymbol{\theta} \setminus W^1_{i,j}, W^1_{i,j} + \epsilon)$$

Key Idea: Wiggle To Decrease Loss



Let's say we want to decrease the loss by adjusting $W^1_{i,j}$.
We could consider a very small $\epsilon = 1\text{e-}6$ and compute:

$$L(\boldsymbol{x}, y; \boldsymbol{\theta})$$

$$L(\boldsymbol{x}, y; \boldsymbol{\theta} \setminus W^1_{i,j}, W^1_{i,j} + \epsilon)$$

Then, update:

$$W^1_{i,j} \leftarrow W^1_{i,j} + \epsilon \, sgn\left(L(\boldsymbol{x}, y; \boldsymbol{\theta}) - L(\boldsymbol{x}, y; \boldsymbol{\theta} \setminus W^1_{i,j}, W^1_{i,j} + \epsilon) \right)$$

Derivative w.r.t. Input of Softmax

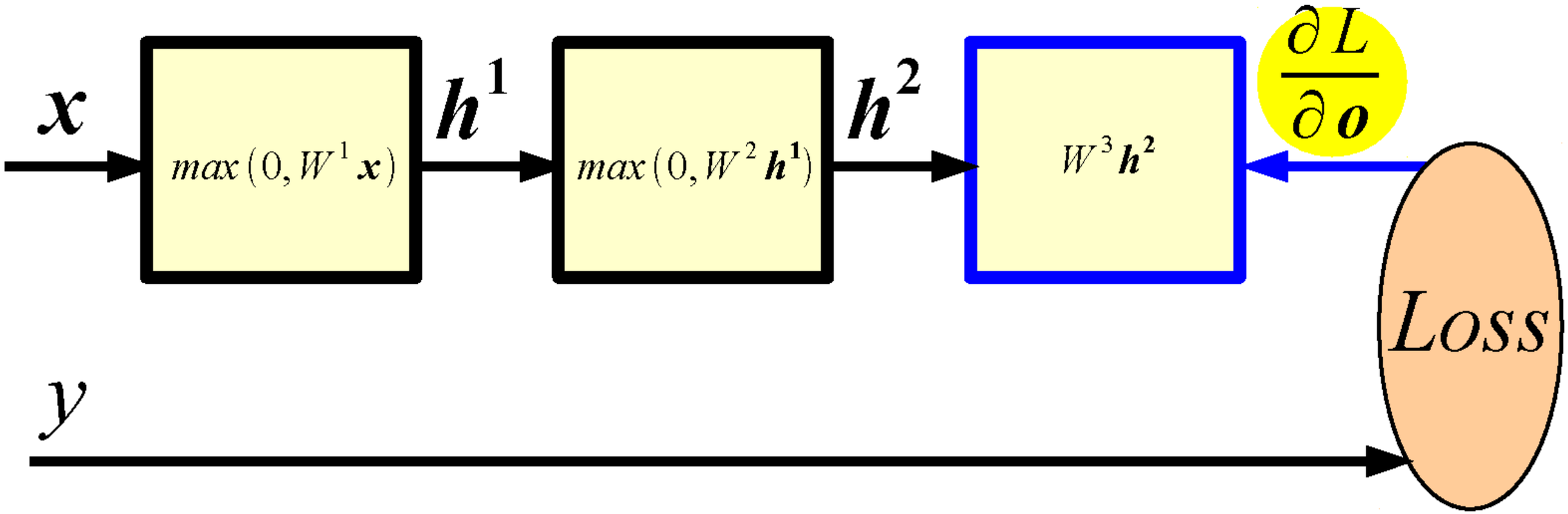
$$p(c_k=1|\boldsymbol{x})=\frac{e^{o_k}}{\sum_j e^{o_j}}$$

$$L(\boldsymbol{x},y;\boldsymbol{\theta})=-\sum_j y_j \log p(c_j|\boldsymbol{x}) \qquad \boldsymbol{y}=[\overset{1}{0}0..0\overset{k}{1}0..\overset{C}{0}]$$

By substituting the fist formula in the second, and taking the derivative w.r.t. $\boldsymbol{o}$ we get:

$$\frac{\partial L}{\partial o}=p(c|\boldsymbol{x})-\boldsymbol{y}$$

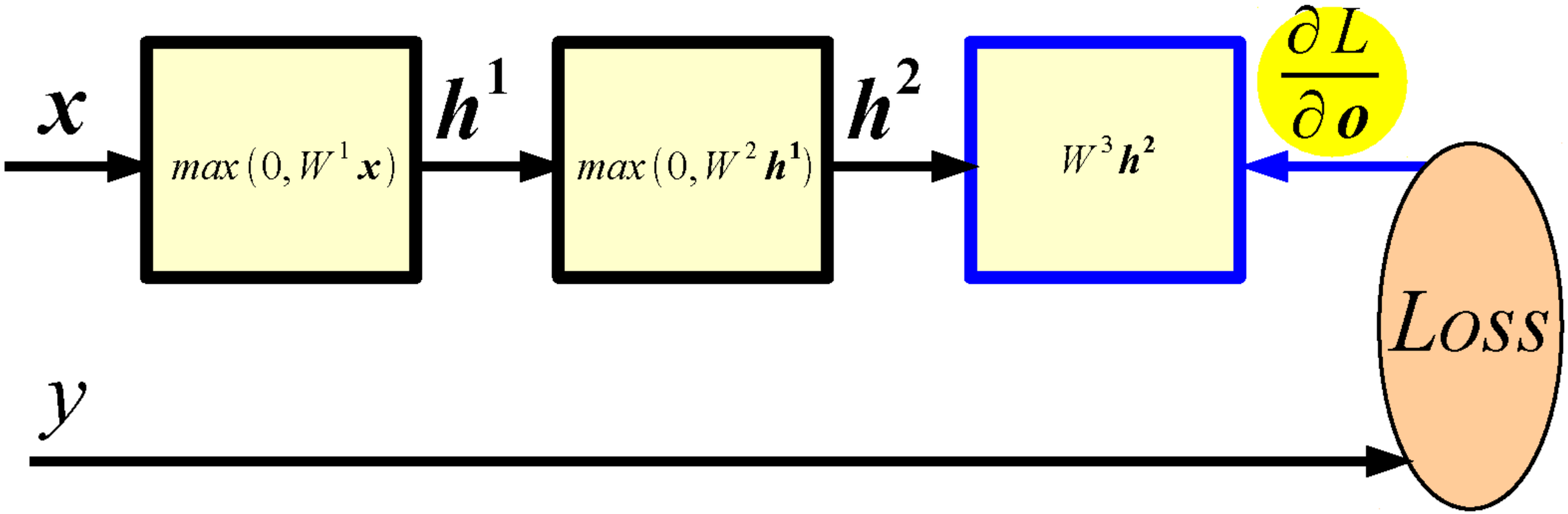
Backward Propagation



Given $\partial L / \partial \boldsymbol{o}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3}$$

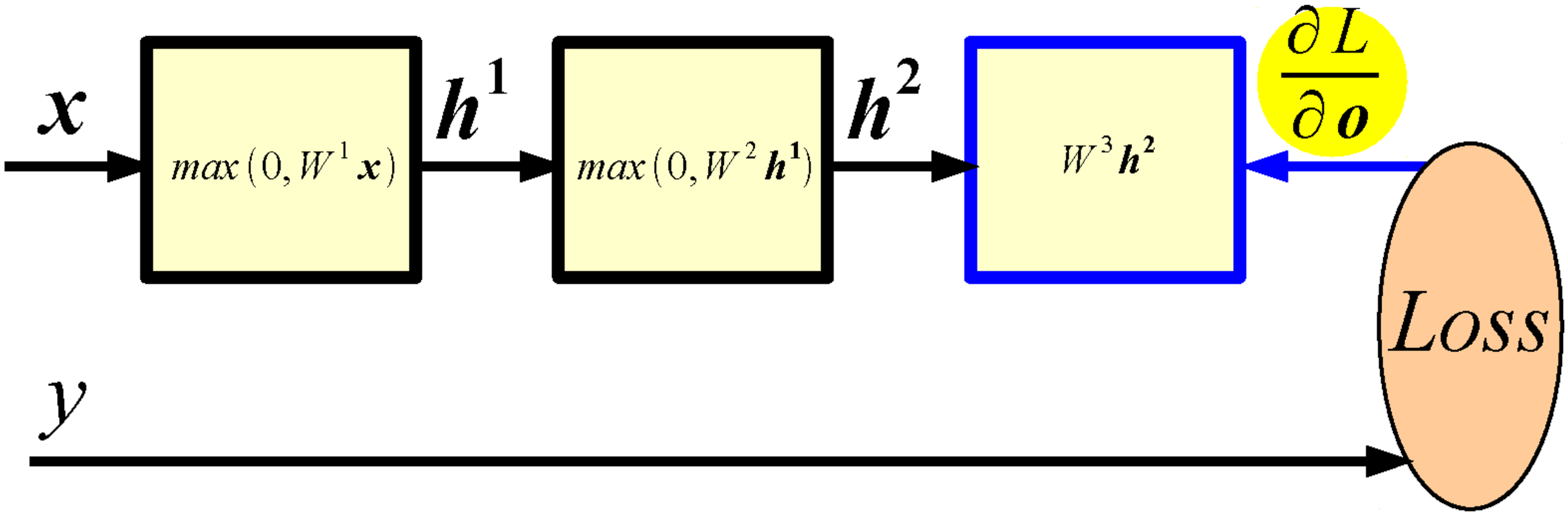
Backward Propagation



Given $\partial L / \partial \boldsymbol{o}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^2}$$

Backward Propagation

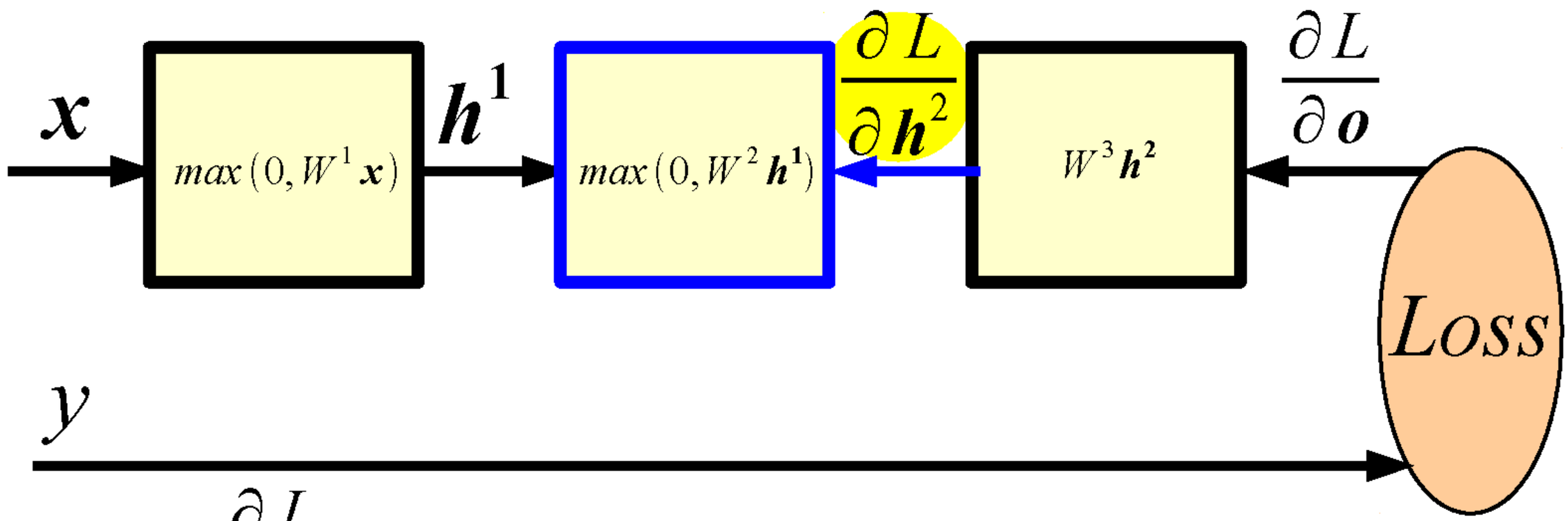


Given $\partial L / \partial \boldsymbol{o}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^2}$$

$$\frac{\partial L}{\partial W^3} = (p(c|\boldsymbol{x}) - \boldsymbol{y})\ \boldsymbol{h}^{2\,T} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = W^{3\,T}(p(c|\boldsymbol{x}) - \boldsymbol{y})$$

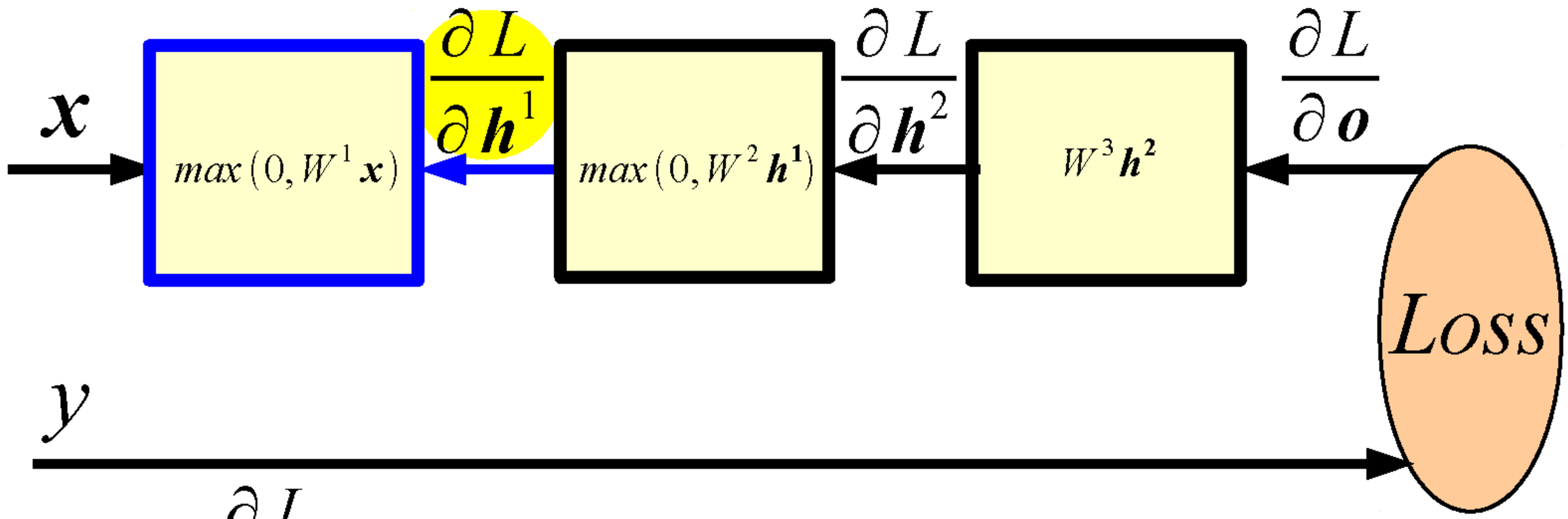
Backward Propagation



Given $\frac{\partial L}{\partial h^2}$ we can compute now:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial h^2} \frac{\partial h^2}{\partial W^2} \qquad \frac{\partial L}{\partial h^1} = \frac{\partial L}{\partial h^2} \frac{\partial h^2}{\partial h^1}$$

Backward Propagation



Given $\frac{\partial L}{\partial h^1}$ we can compute now:

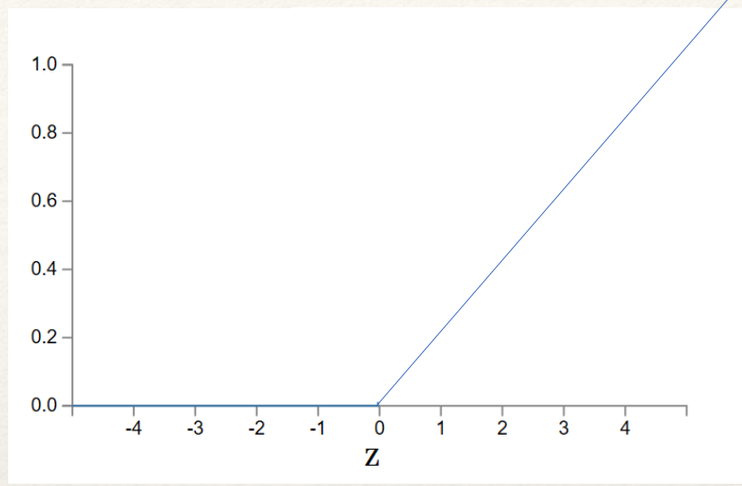
$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial h^1} \frac{\partial h^1}{\partial W^1}$$

Backward Propagation

Question: Does BPROP work with ReLU layers only?

Answer: Nope, any a.e. differentiable transformation works.

But the ReLU is not differentiable at 0!



- ‘0’ is the best place for this to occur, because we don’t care about the result (it is no activation).
- ‘Dead’ perceptrons
- ReLU has unbounded positive response:
  - Potential faster convergence / overstep

Backward Propagation

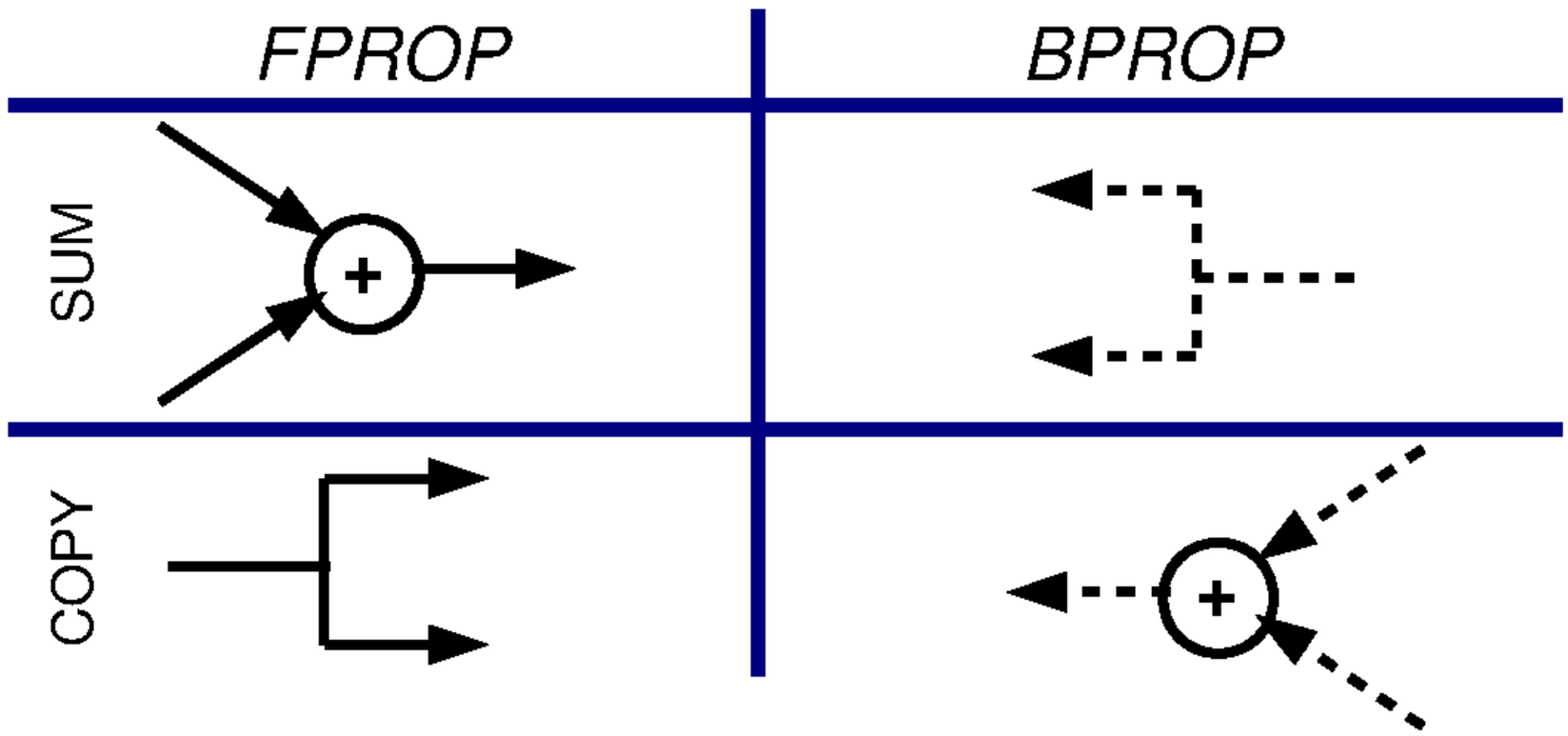
Question: Does BPROP work with ReLU layers only?

Answer: Nope, any a.e. differentiable transformation works.

Question: What's the computational cost of BPROP?

Answer: About twice FPROP (need to compute gradients w.r.t. input and parameters at every layer).

Note: FPROP and BPROP are dual of each other. E.g.,:



Toy Code (Matlab): Neural Net Trainer

```
% F-PROP
for i = 1 : nr_layers - 1
  [h{i}  jac{i}]  =  nonlinearity(W{i} * h{i-1} +  b{i});
end
h{nr_layers-1}  =  W{nr_layers-1} * h{nr_layers-2}  +  b{nr_layers-1};
prediction  =  softmax(h{l-1});


% CROSS ENTROPY LOSS
loss  =  -  sum(sum(log(prediction)  .*  target)) / batch_size;
```

```
% B-PROP
dh{l-1}  =  prediction  -  target;
for i = nr_layers - 1 : -1 : 1
  Wgrad{i}  =  dh{i} * h{i-1}';
  bgrad{i}  =  sum(dh{i}, 2);
  dh{i-1}  =  (W{i}' * dh{i})  .*  jac{i-1};
end


% UPDATE
for i = 1 : nr_layers - 1
  W{i}  =  W{i}  -  (lr / batch_size)  *  Wgrad{i};
  b{i}  =  b{i}  -  (lr / batch_size)  *  bgrad{i};
end
```

Stochastic Gradient Descent

- Dataset can be too large to strictly apply gradient descent.
- Instead, randomly sample a data point, perform gradient descent per point, and iterate.
  - True gradient is approximated only
  - Picking a subset of points: *"mini-batch"*

Pick starting W and learning rate γ

While not at minimum:

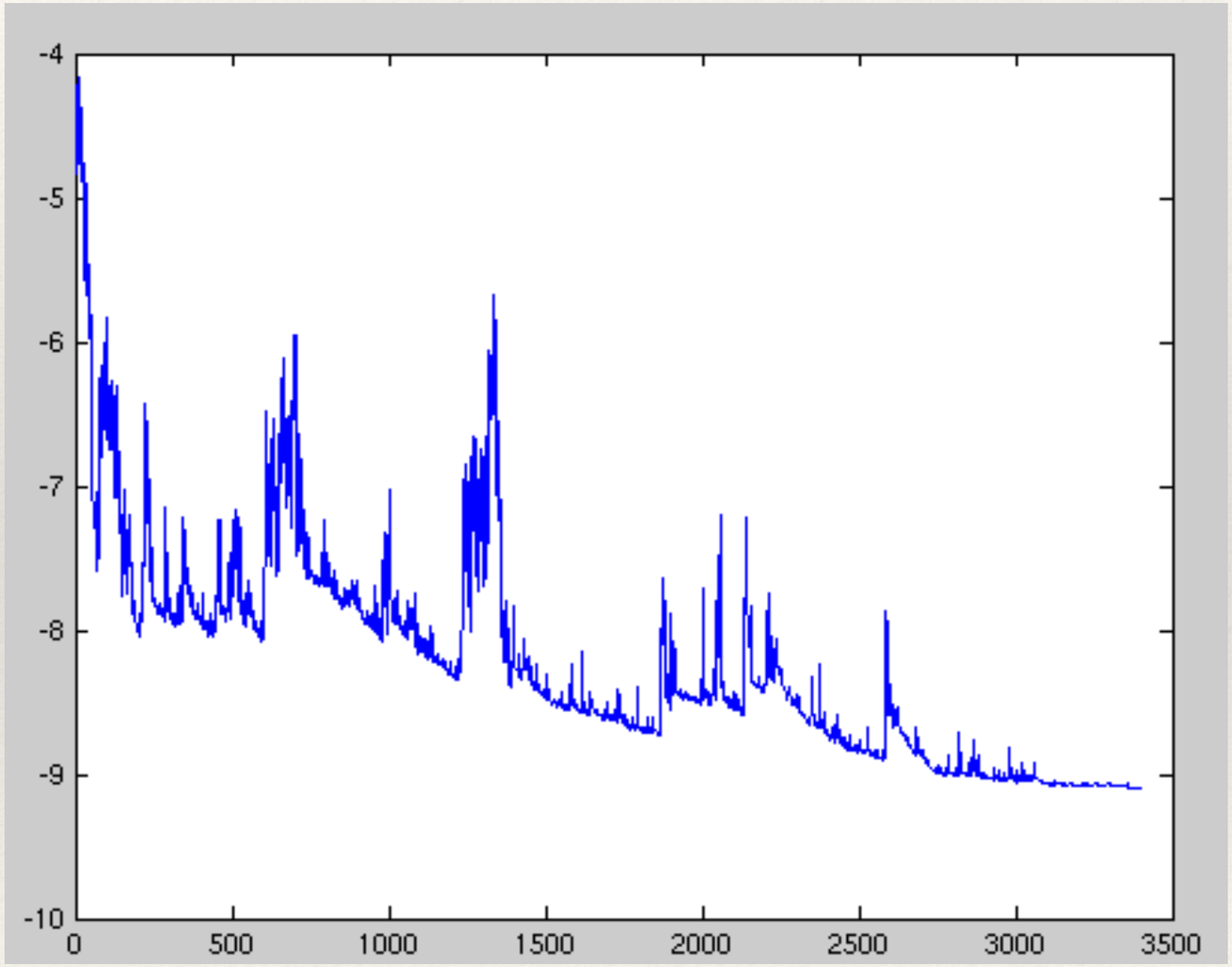
- Shuffle training set
- For each data point *i=1…n (maybe as mini-batch)*
  - *Gradient descent*

} *"Epoch"*

Stochastic Gradient Descent

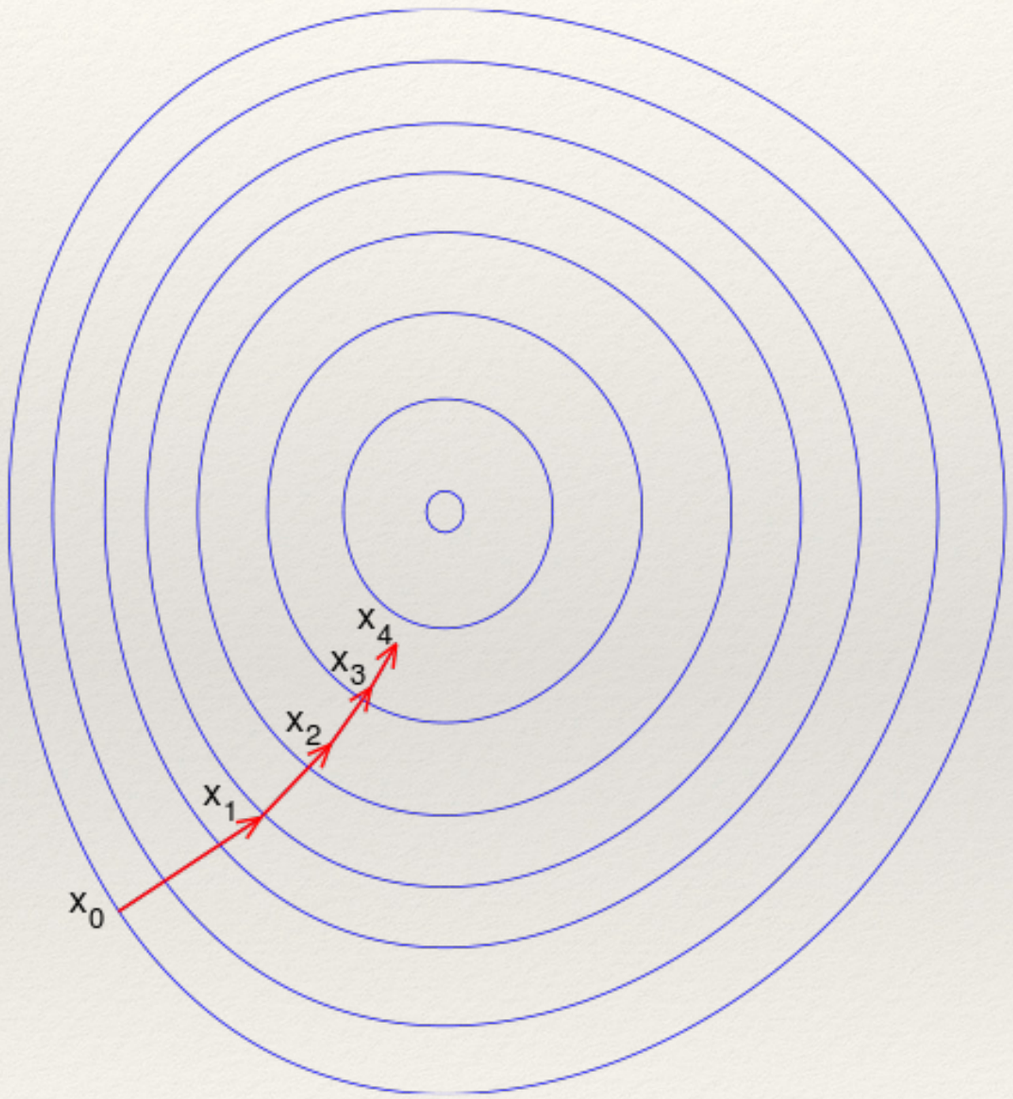
Loss will not always decrease (locally) as training data point is random.

Still converges over time.



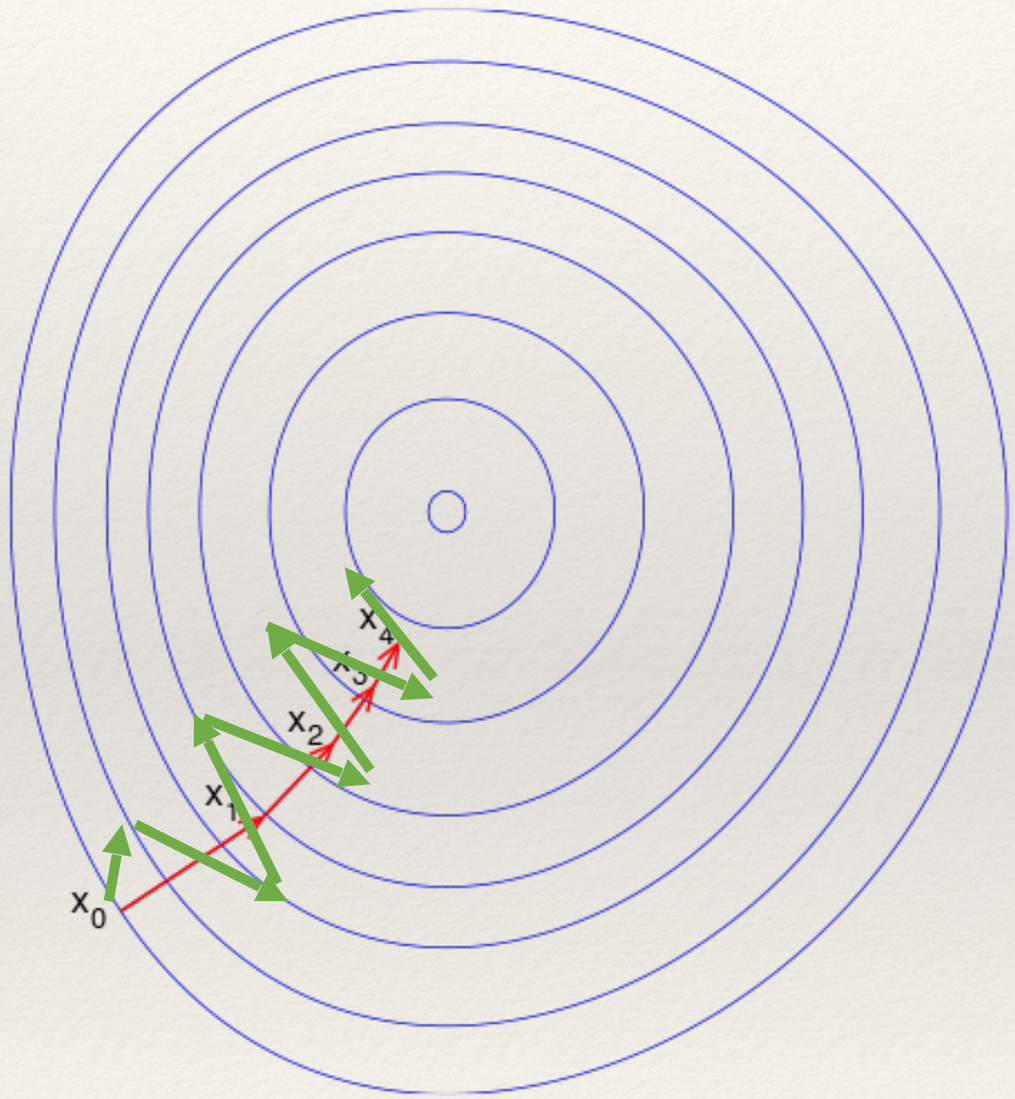
Wikipedia

Gradient descent oscillations



Wikipedia

Gradient descent oscillations



Slow to converge to the (local) optimum

Wikipedia

Momentum

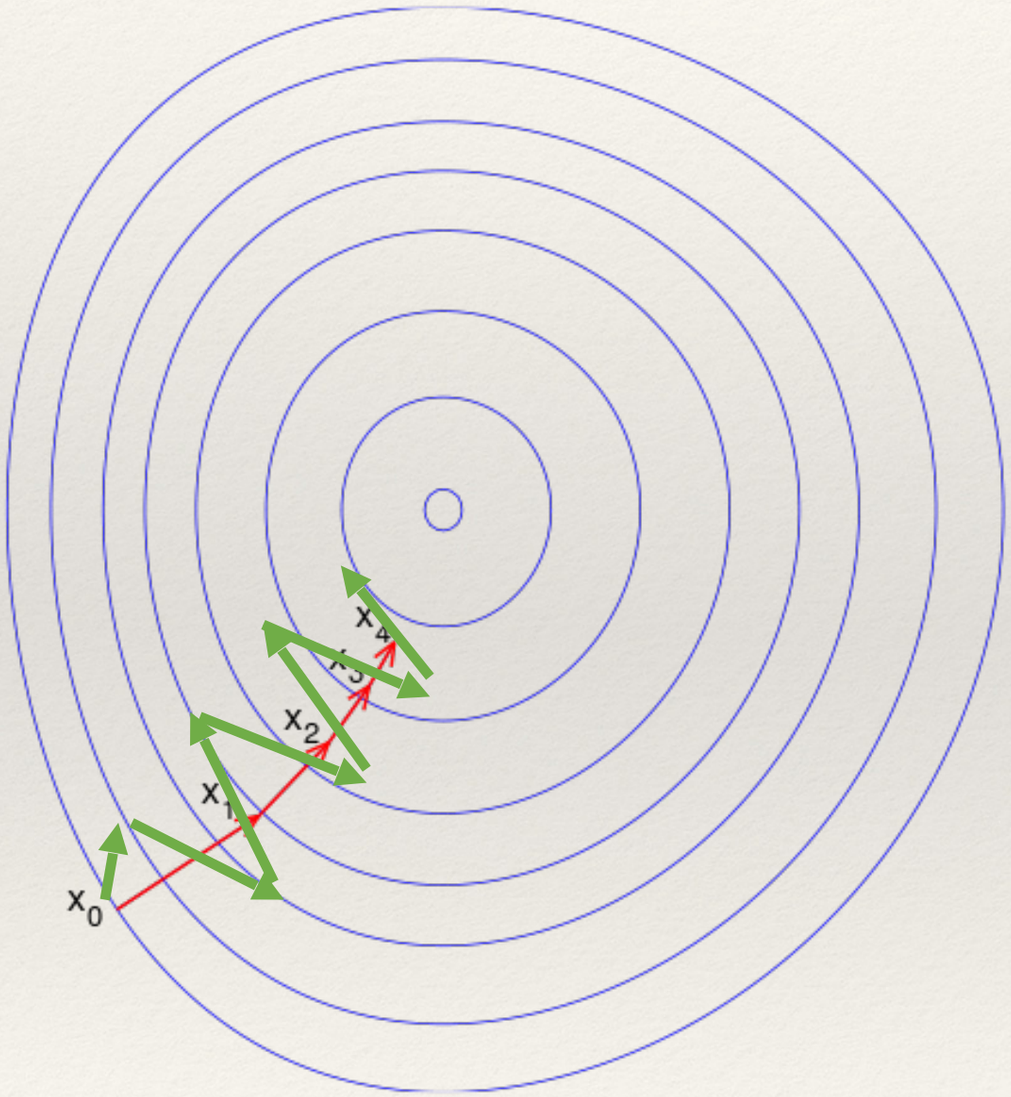
- Adjust the gradient by a weighted sum of the previous amount plus the current amount.

- Without momentum: $\theta_{t+1} = \theta_t - \gamma \frac{\partial L}{\partial \theta}$

- With momentum (new α parameter):

$$\theta_{t+1} = \theta_t - \gamma \left(\alpha \left[\frac{\partial L}{\partial \theta} \right]_{t-1} + \left[\frac{\partial L}{\partial \theta} \right]_t \right)$$

Lowering the learning rate = smaller steps in SGD

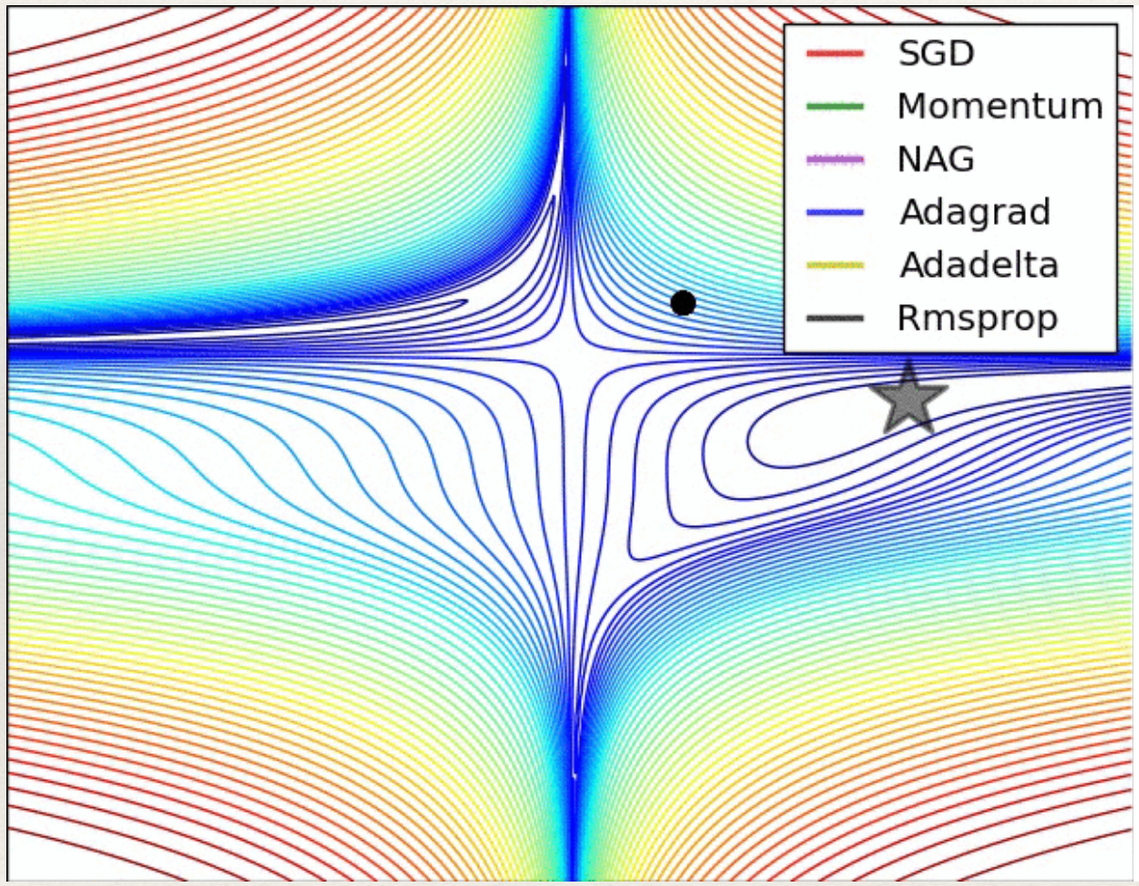


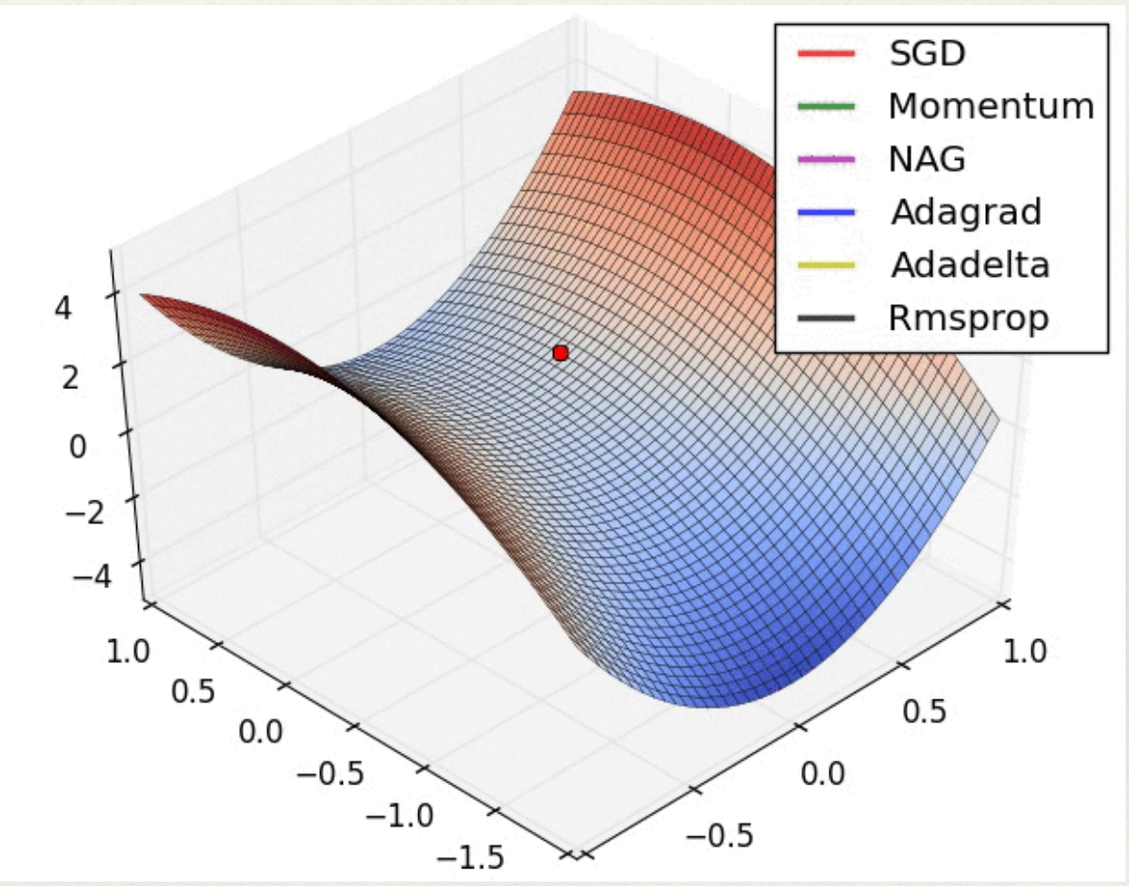
-Less ‘ping pong’

-Takes longer to get to the optimum

Wikipedia

Flat regions in energy landscape





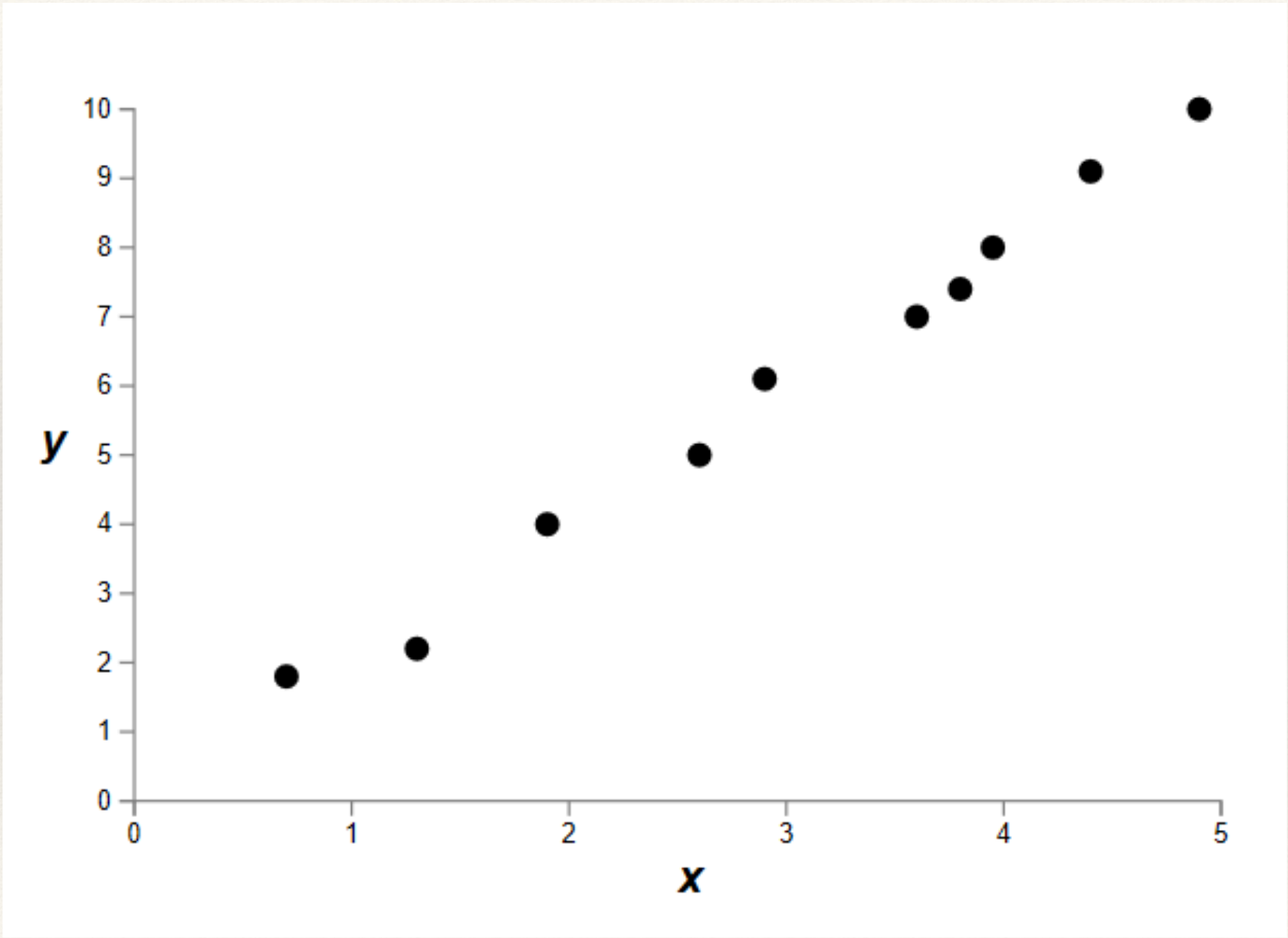
Problem of fitting

- Too many parameters = overfitting
- Not enough parameters = underfitting
- More data = less chance to overfit
- How do we know what is required?

Regularization

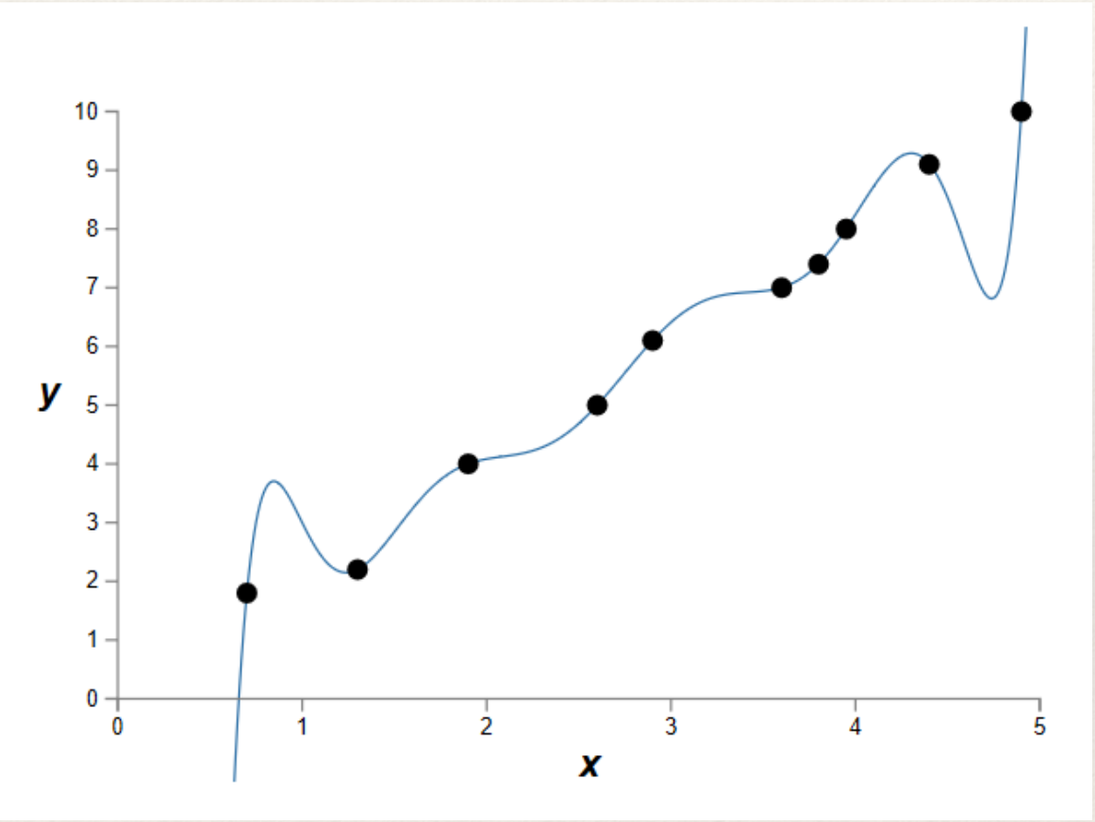
- Attempt to guide solution to *not overfit*
- But still give freedom with many parameters

Data fitting problem

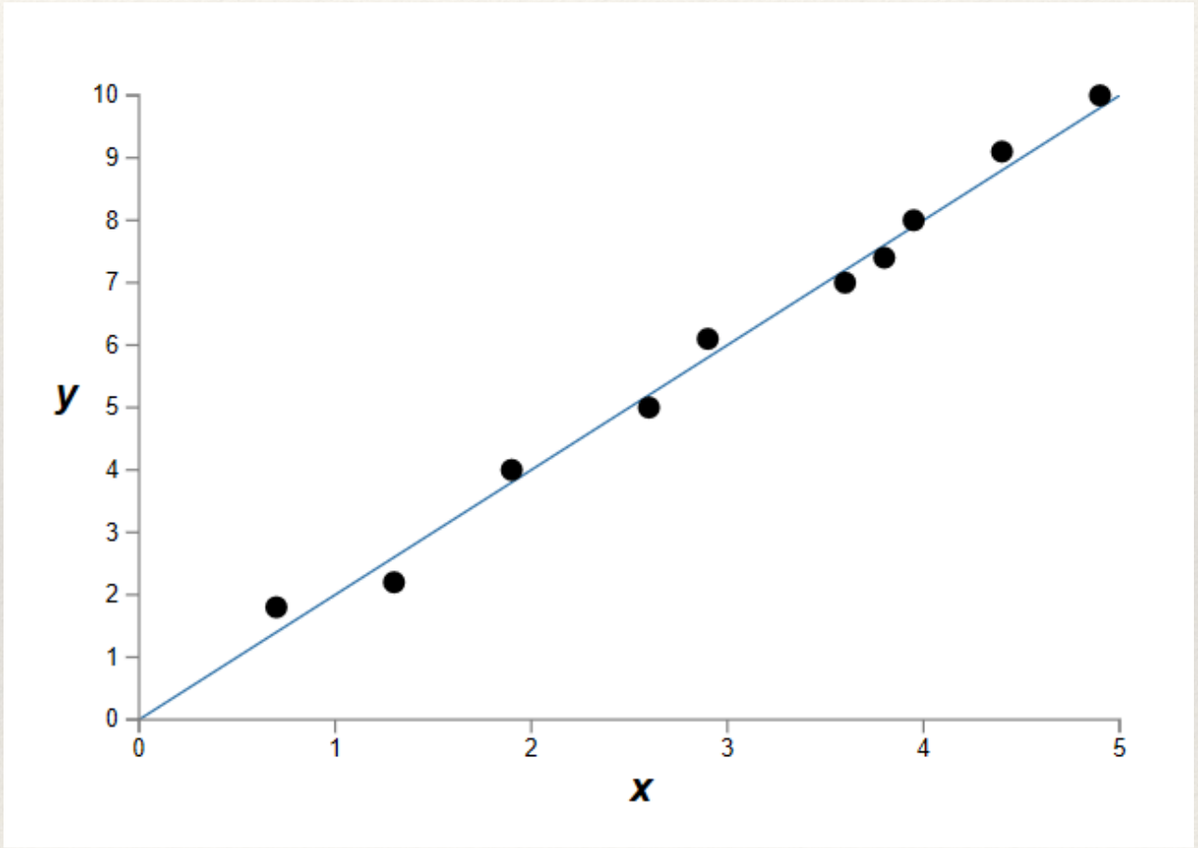


[Nielson]

Which is better?
Which is better a priori?



9th order polynomial



1st order polynomial

[Nielson]

Regularization

- Attempt to guide solution to *not overfit*
- But still give freedom with many parameters

- Idea:

  Penalize the use of parameters to prefer small weights.

Regularization:

- Idea: add a cost to having high weights
- λ = regularization parameter

$$C = C_0 + \lambda \sum_w w^2,$$

[Nielson]

Both can describe the data…

- …but one is simpler.

- Occam's razor:

  "Among competing hypotheses, the one with the fewest assumptions should be selected"

For us:

Large weights cause large changes in behaviour in response to small changes in the input.

Simpler models (or smaller changes) are more robust to noise.

Regularization

- Idea: add a cost to having high weights
- λ = regularization parameter

$$C = C_0 + \lambda \sum_w w^2,$$

$$C = -\frac{1}{n} \sum_{xj} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] + \lambda \sum_w w^2.$$

Normal cross-entropy loss (binary classes)

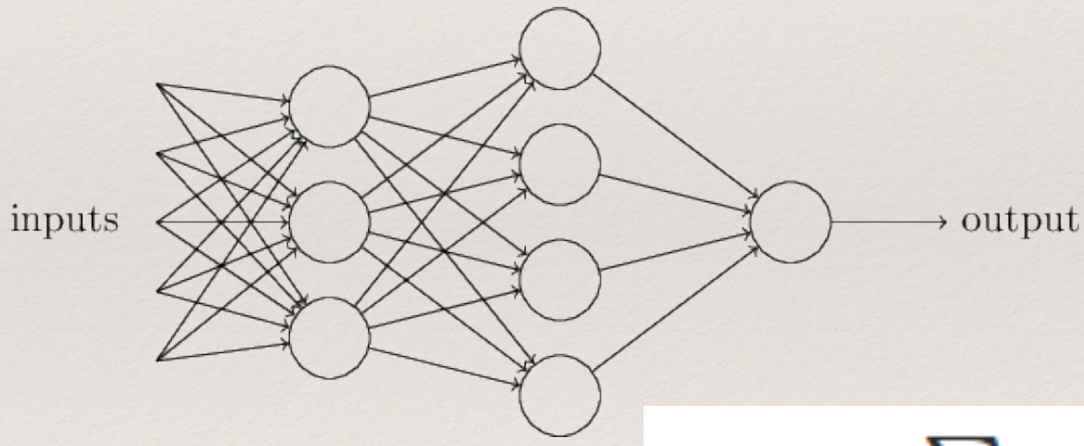
Regularization term

[Nielson]

Regularization: Dropout

- Our networks typically start with random weights.
- Every time we train = slightly different outcome.

- Why random weights?
- If weights are all equal, response across filters will be equivalent.
  - Network doesn't train.



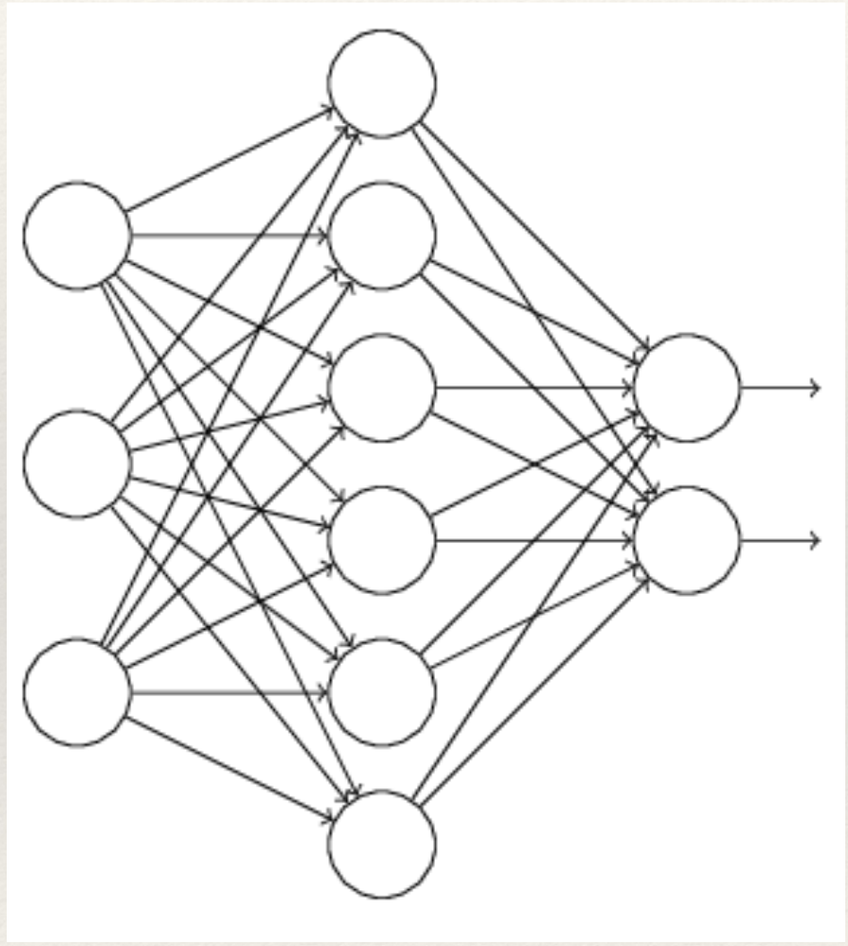
$$w \cdot x \equiv \sum_j w_j x_j$$

[Nielson]

Regularization

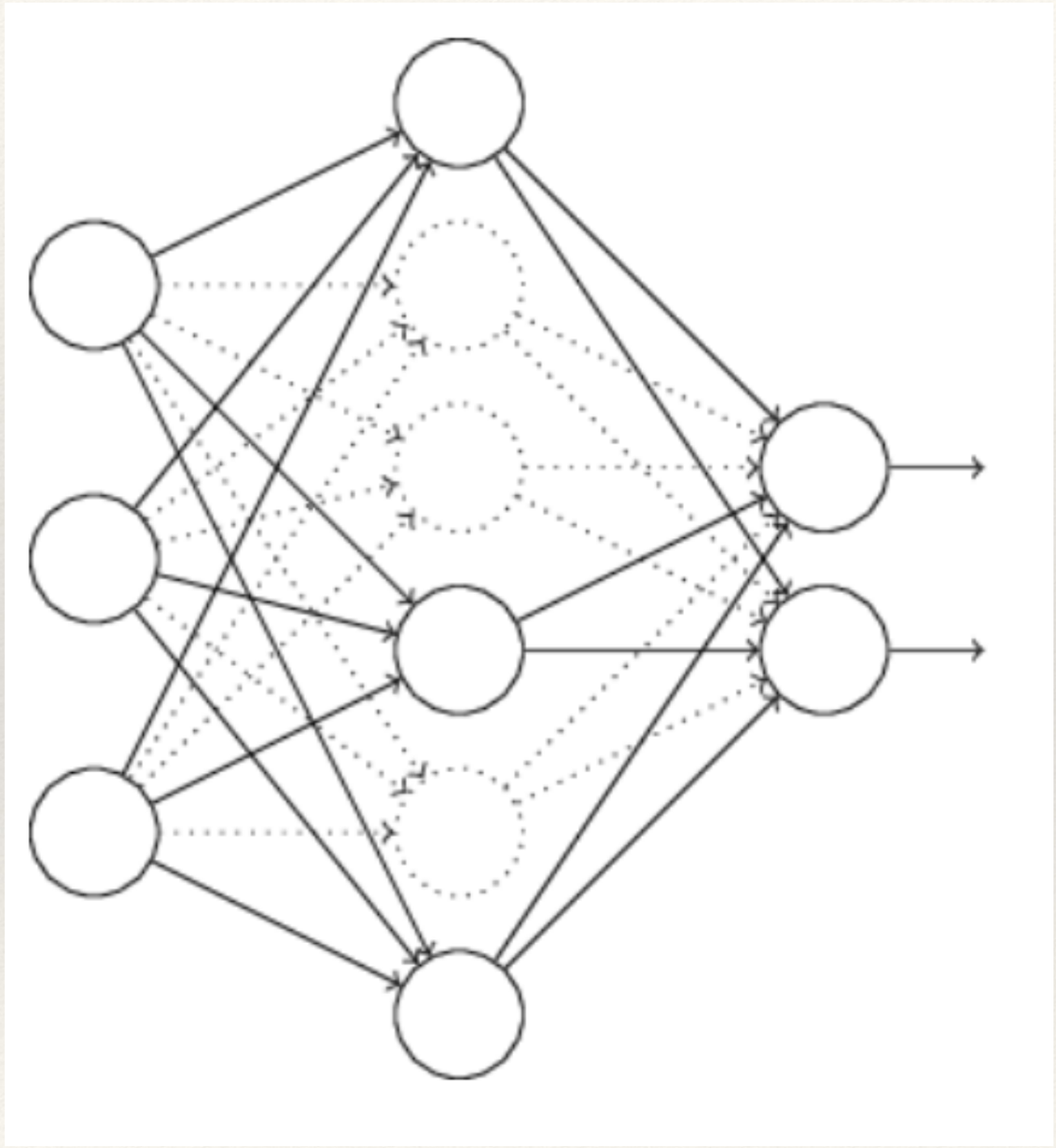
- Our networks typically start with random weights.
- Every time we train = slightly different outcome.
- Why not train 5 different networks with random starts and vote on their outcome?
  - Works fine!
  - Helps generalization because error is averaged.

Regularization: Dropout



[Nielson]

Regularization: Dropout



At each mini-batch:

- Randomly select a subset of neurons.
- Ignore them.

On test: half weights outgoing to compensate for training on half neurons.

Effect:

- Neurons become less dependent on output of connected neurons.
- Forces network to learn more robust features that are useful to more subsets of neurons.
- Like averaging over many different trained networks with different random initializations.
- Except cheaper to train.

[Nielson]

Many forms of ‘regularization’

- Adding more data is a kind of regularization
- Pooling is a kind of regularization
- Data augmentation is a kind of regularization

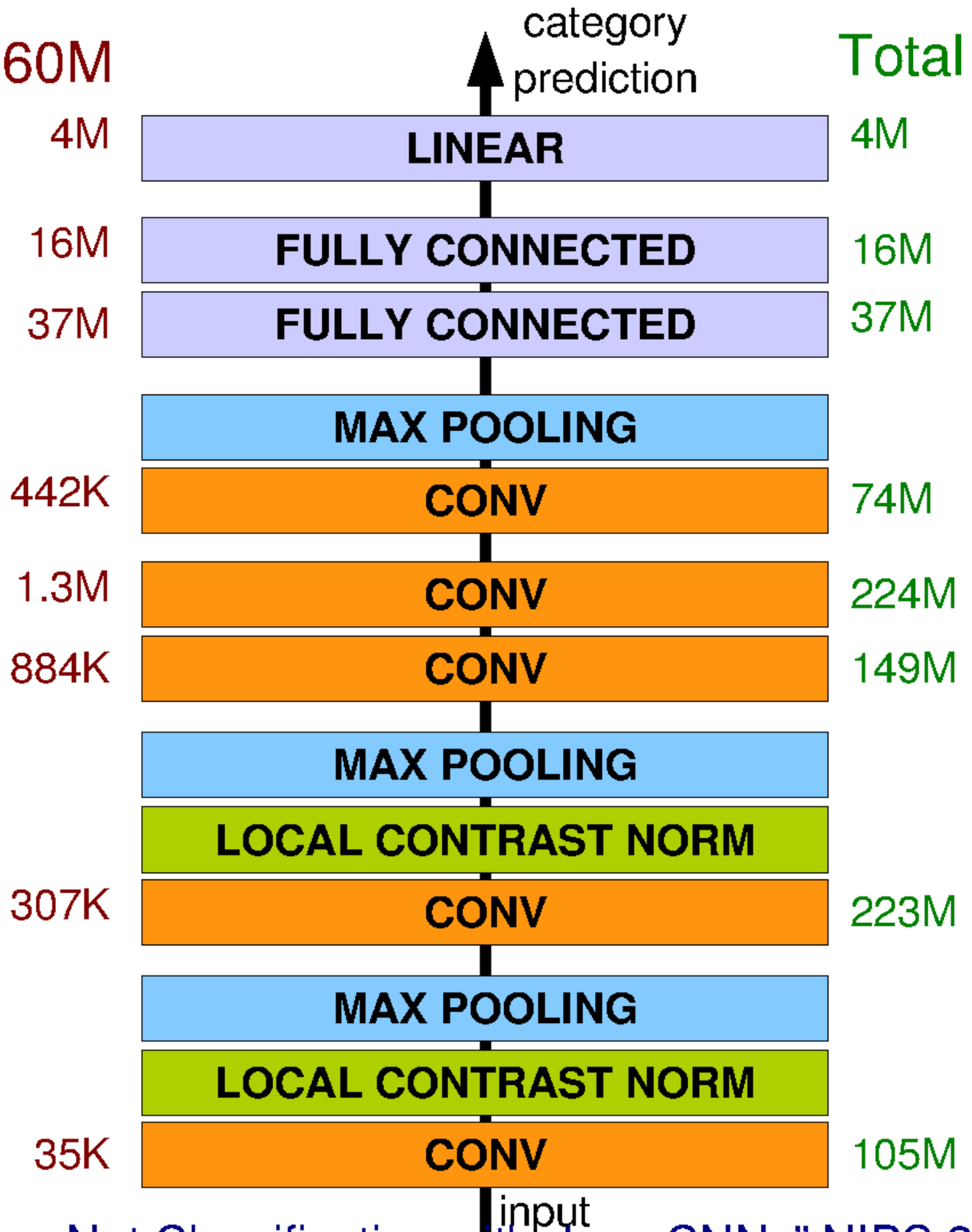
Network Architecture: big space of designs

- But we still don’t even know how many layers we need.

Architecture for Classification

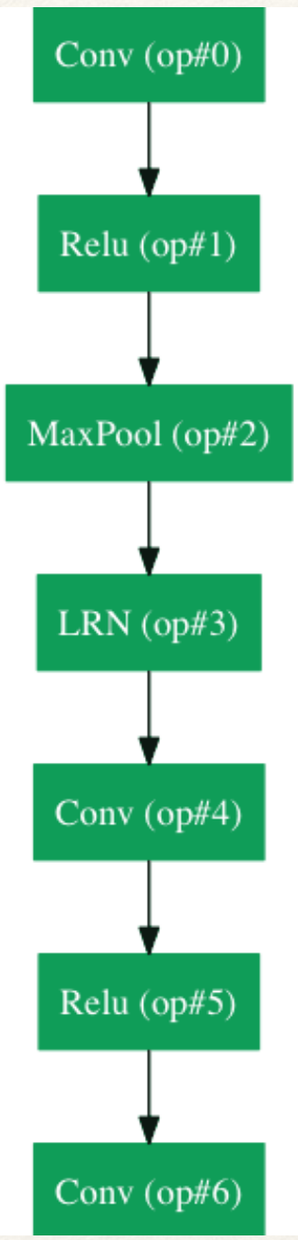
Total nr. params: 60M

Total nr. flops: 832M



Krizhevsky et al. “ImageNet Classification with deep CNNs” NIPS 2012

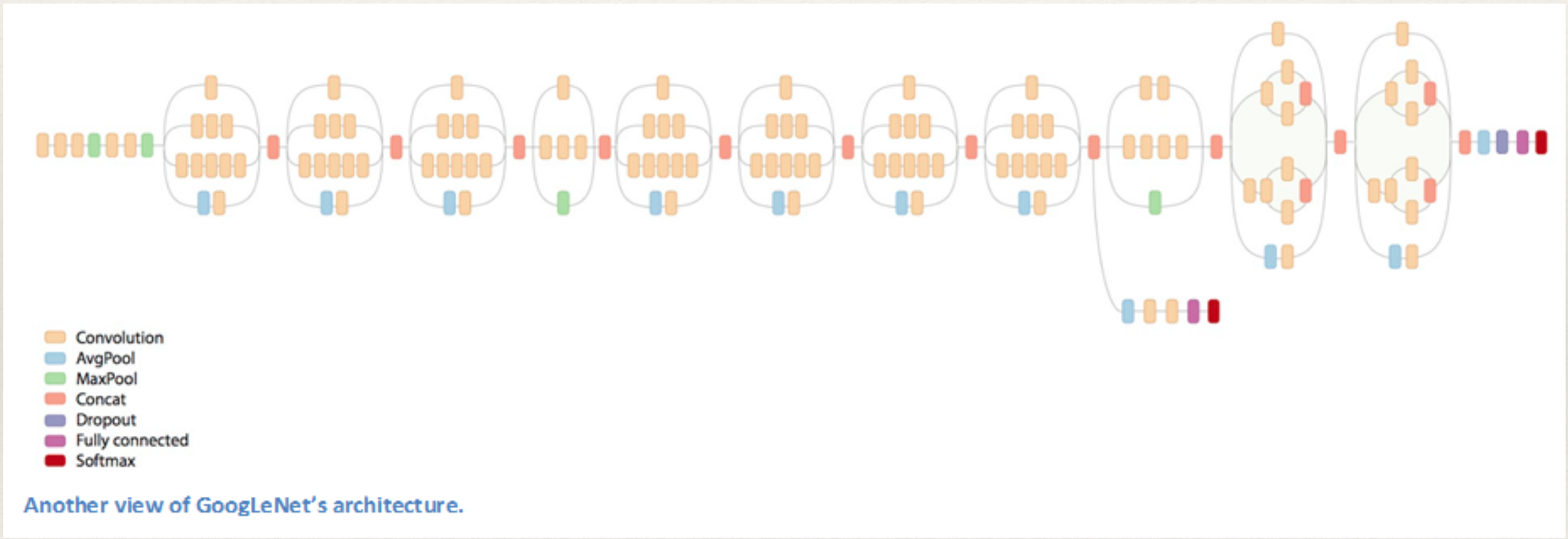
Google LeNet (2014)



22 layers

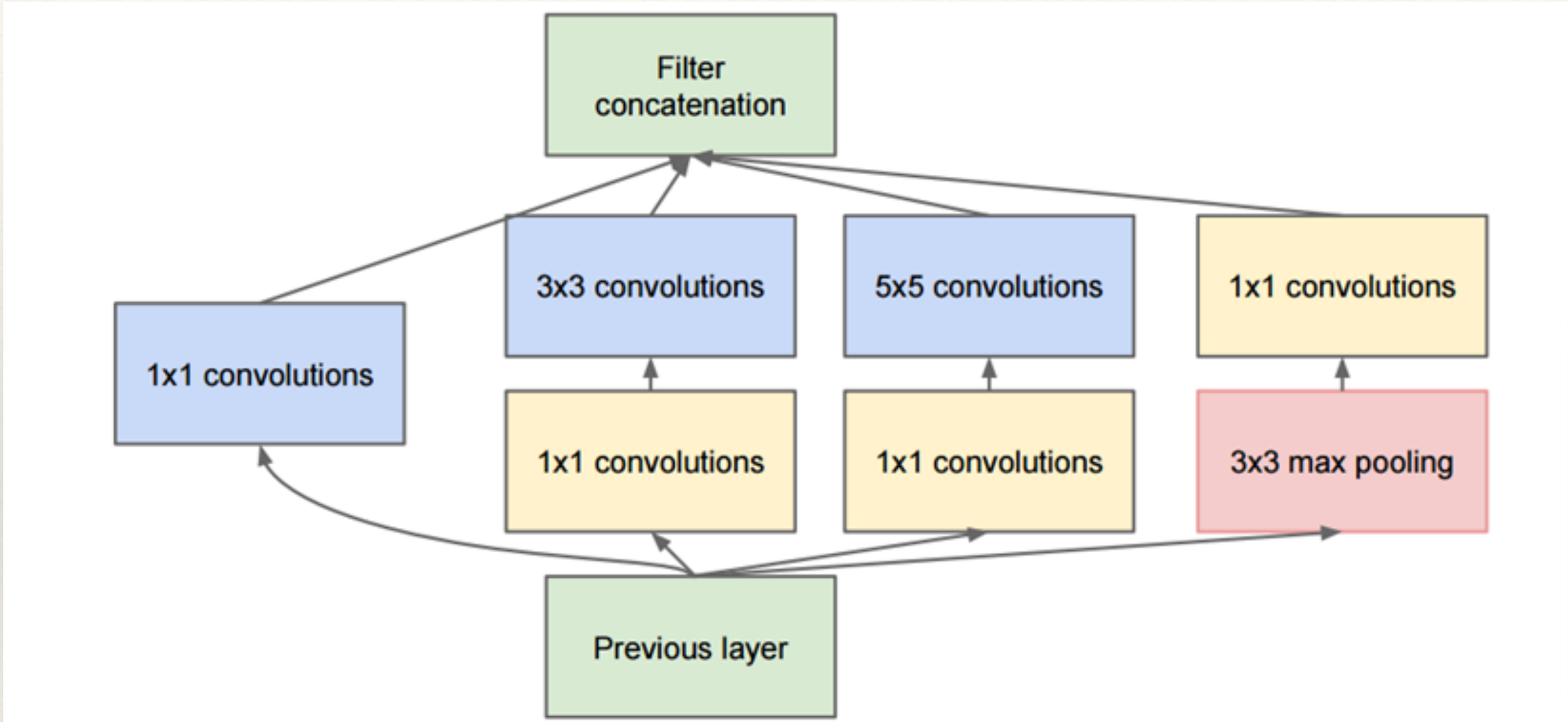
6.67% error
ImageNet top 5

Inception!



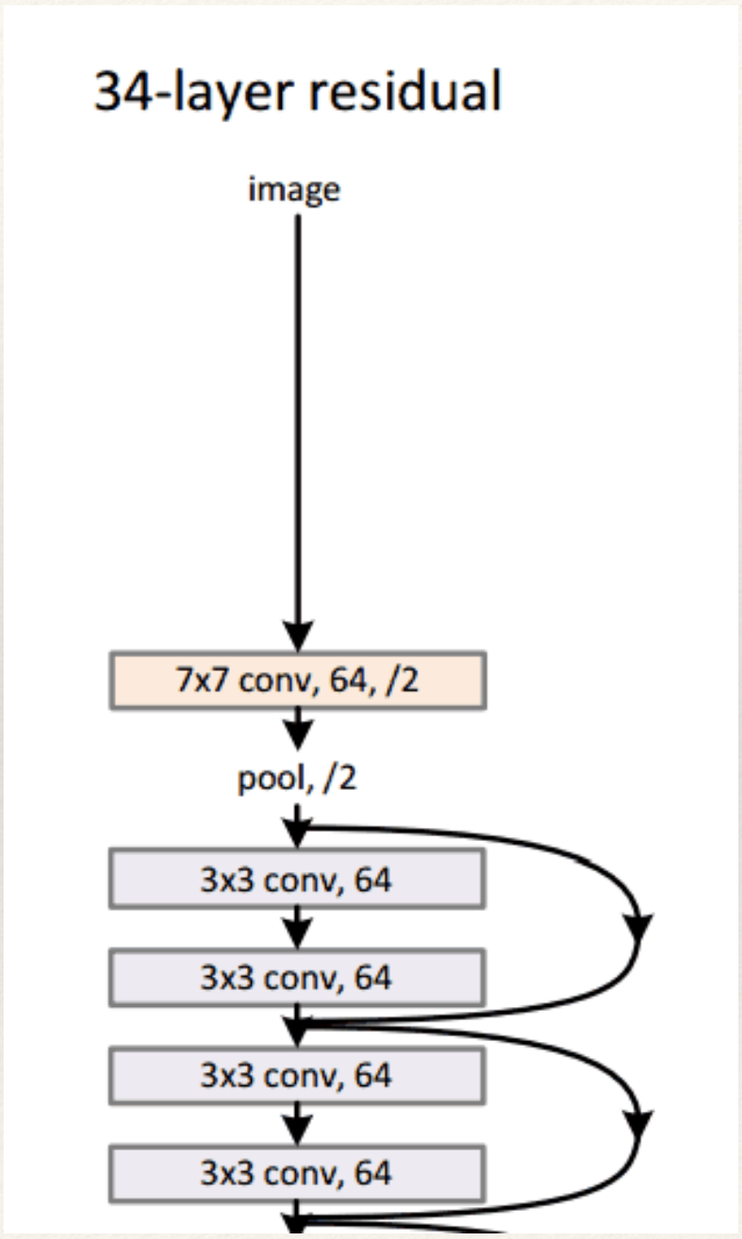
Another view of GoogLeNet's architecture.

Parallel layers



Full Inception module

ResNet (He et al., 2015)

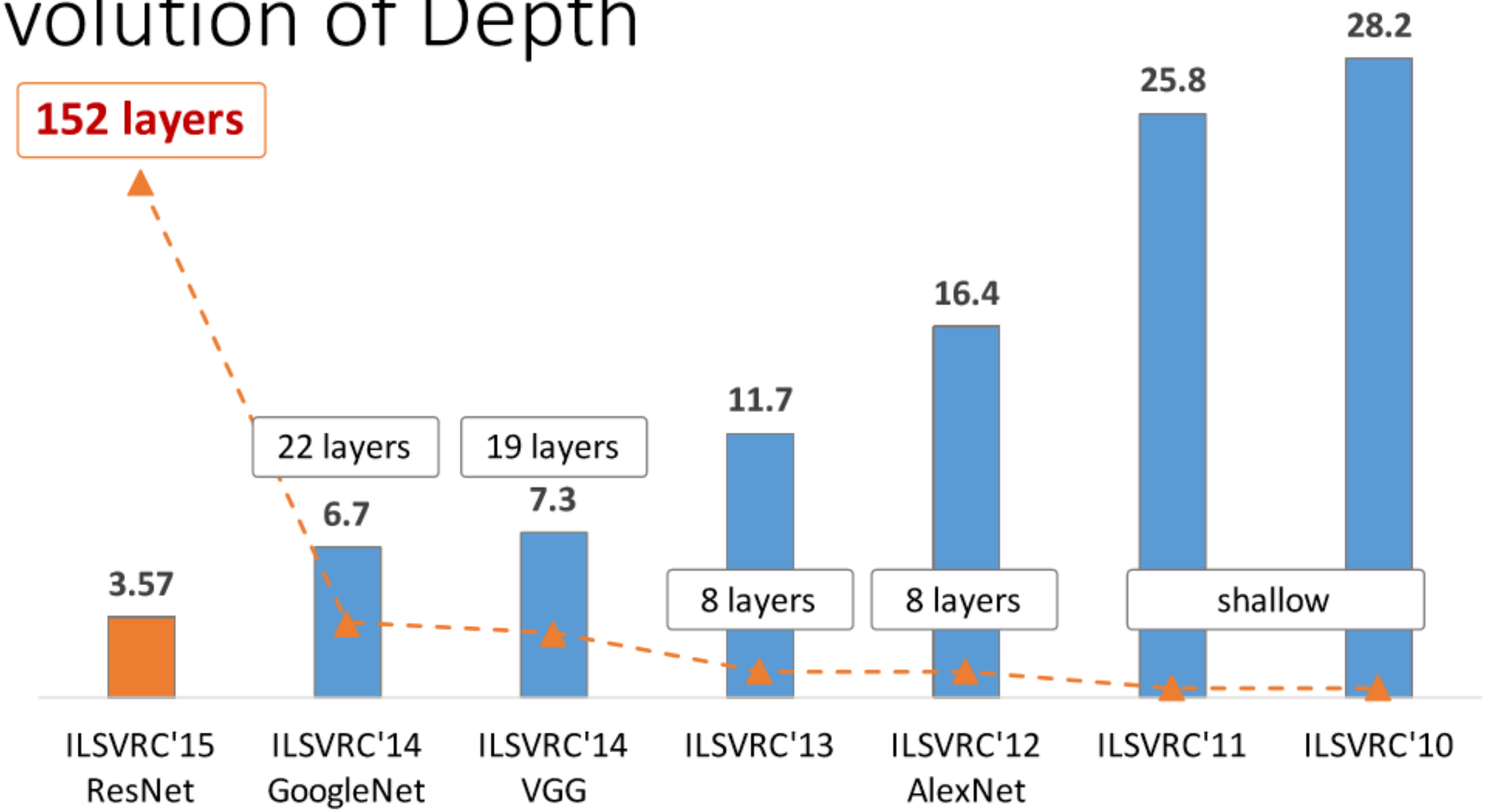


ResNet won ILSVRC 2015 with a top-5 error rate of 3.6%

Depending on their skill and expertise, humans generally hover around a 5-10% error.

~~Superhuman performance!~~
But the task is arguably not well defined.

Revolution of Depth



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

Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)

VGG, 19 layers
(ILSVRC 2014)

ResNet, 152 layers
(ILSVRC 2015)

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

Vanishing/exploding gradient problem

Backpropagation:

- Compute gradient update for every neuron which was involved in the output across layers

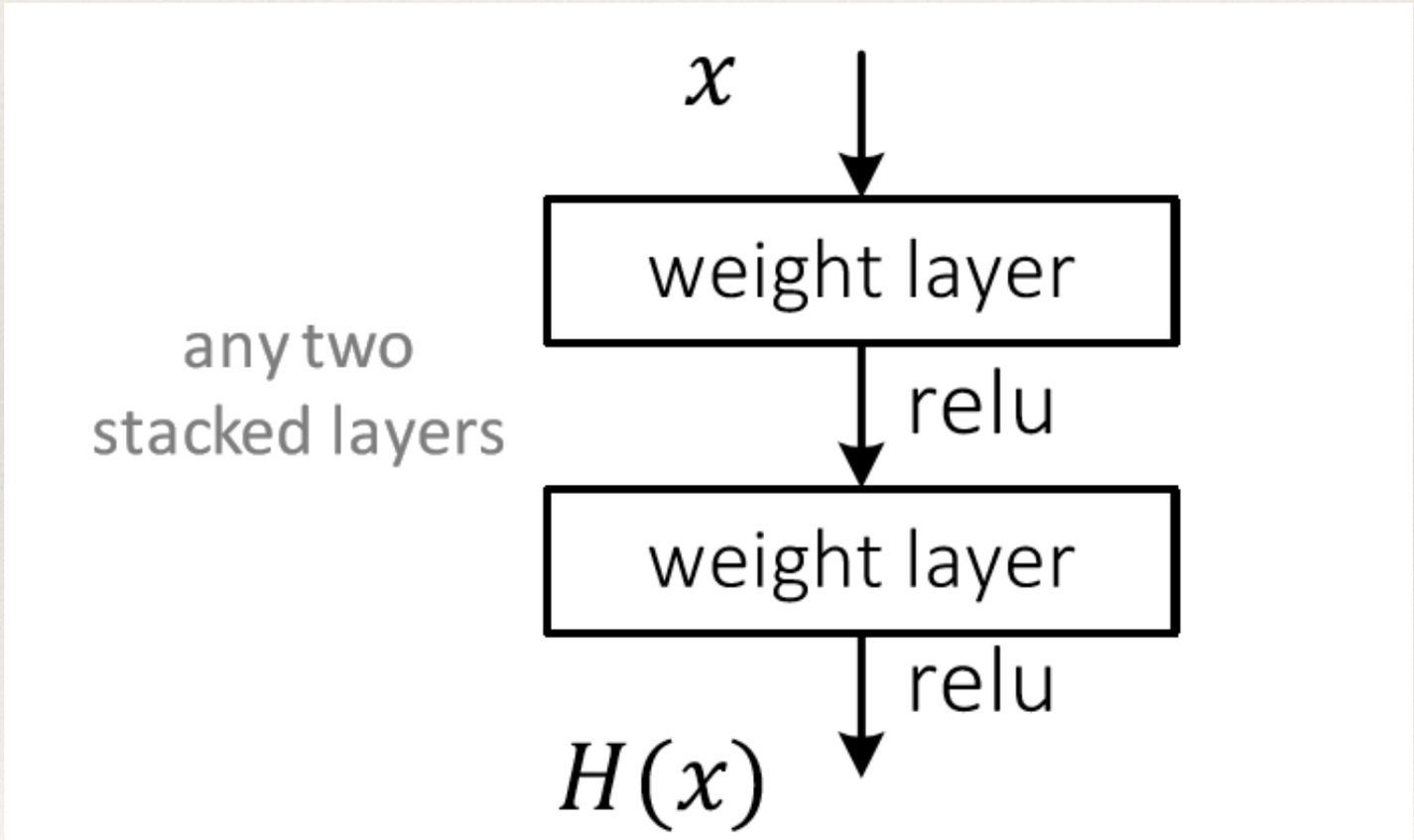
Involves chaining partial derivates over many layers!

- If derivative < 1, gradient gets smaller and smaller as we go deeper and deeper -> *vanishing gradients!*
- If derivative > 1, gradient gets larger and larger as we go deeper and deeper -> *exploding gradients!*

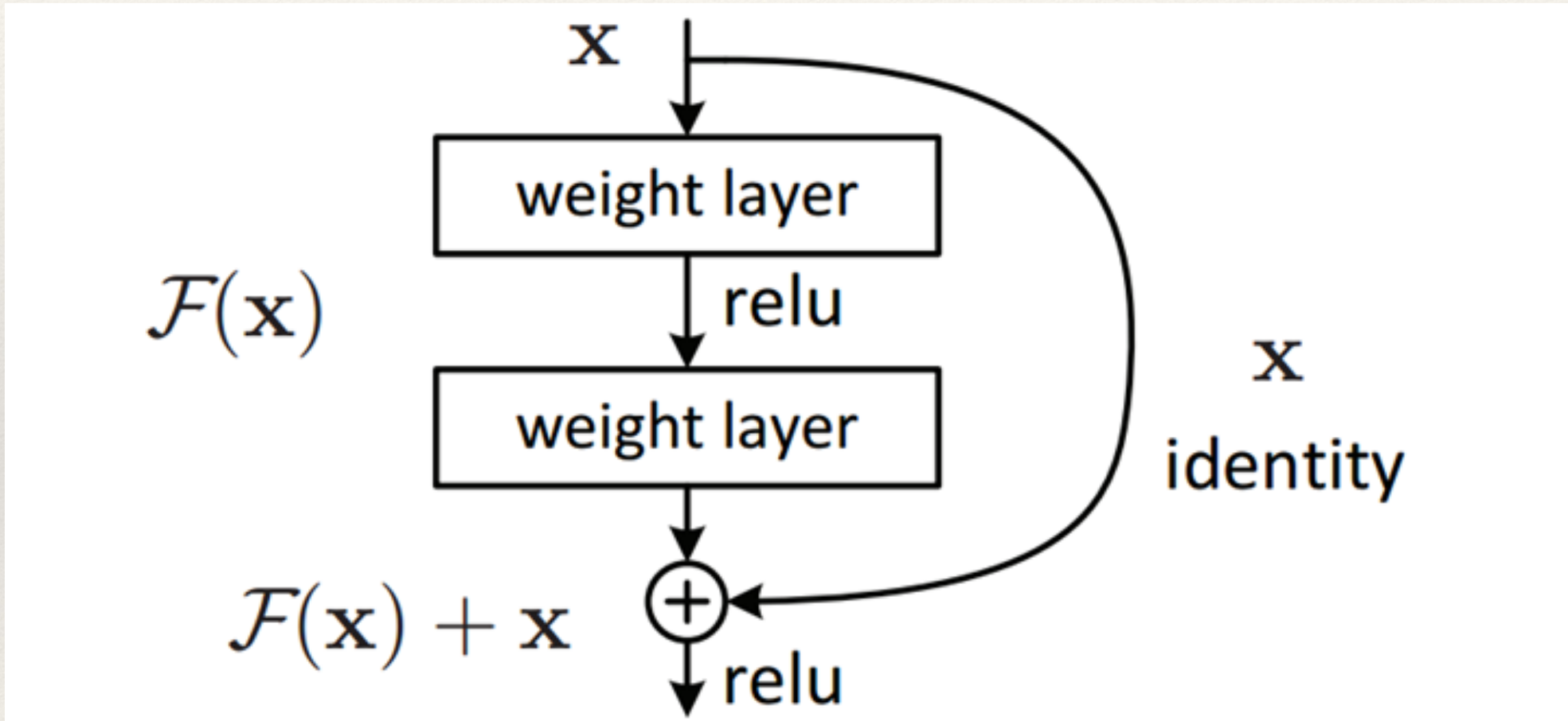
Regular net

$H(x)$ is any desired mapping,

hope the 2 weight layers fit $H(x)$



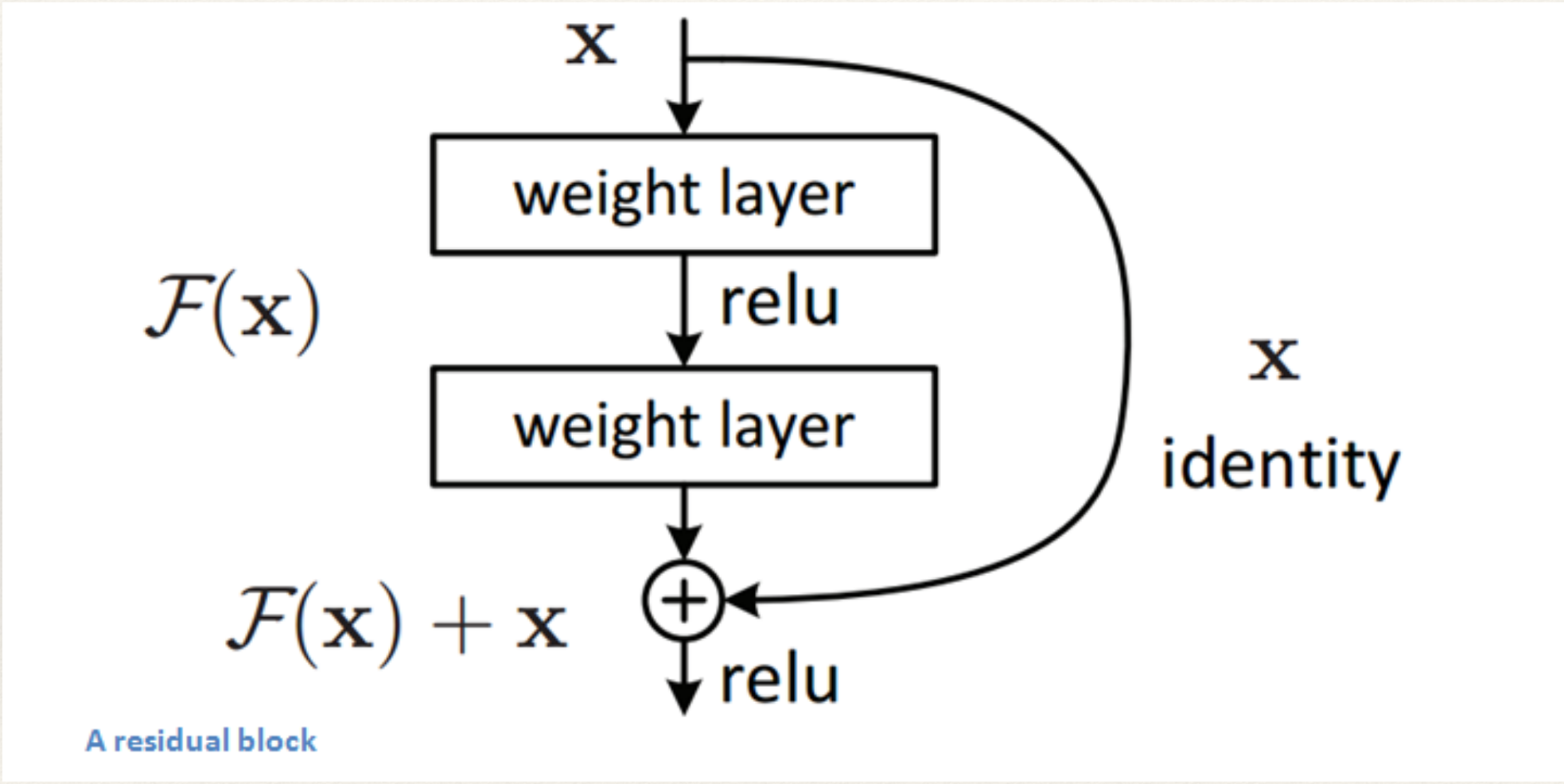
Residual Unit



A residual block

Residual Unit

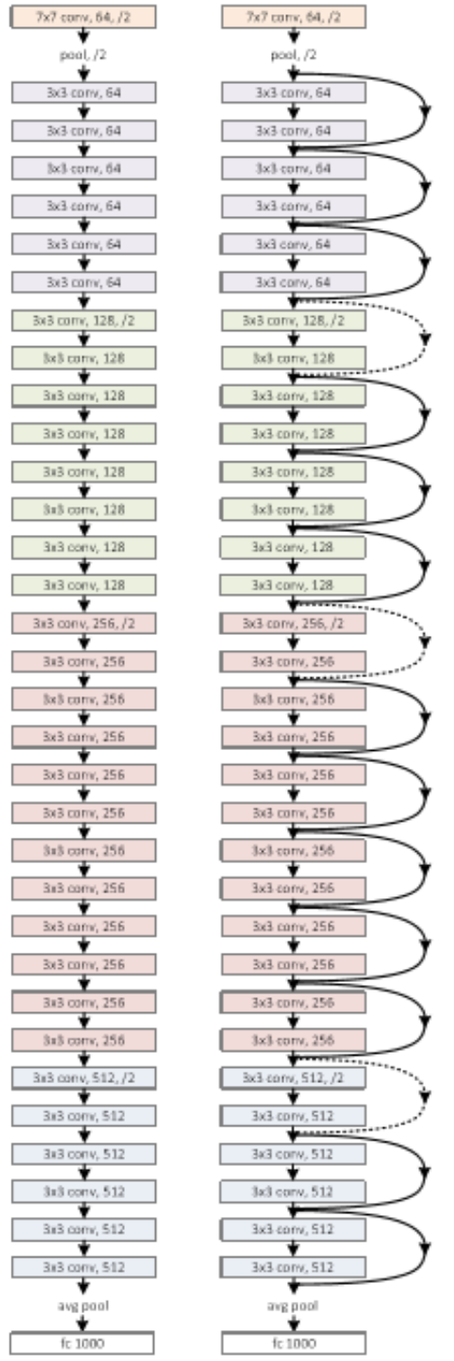
The inputs of a lower layer is made available to a node in a higher layer.



A residual block

Network “Design”

plain net

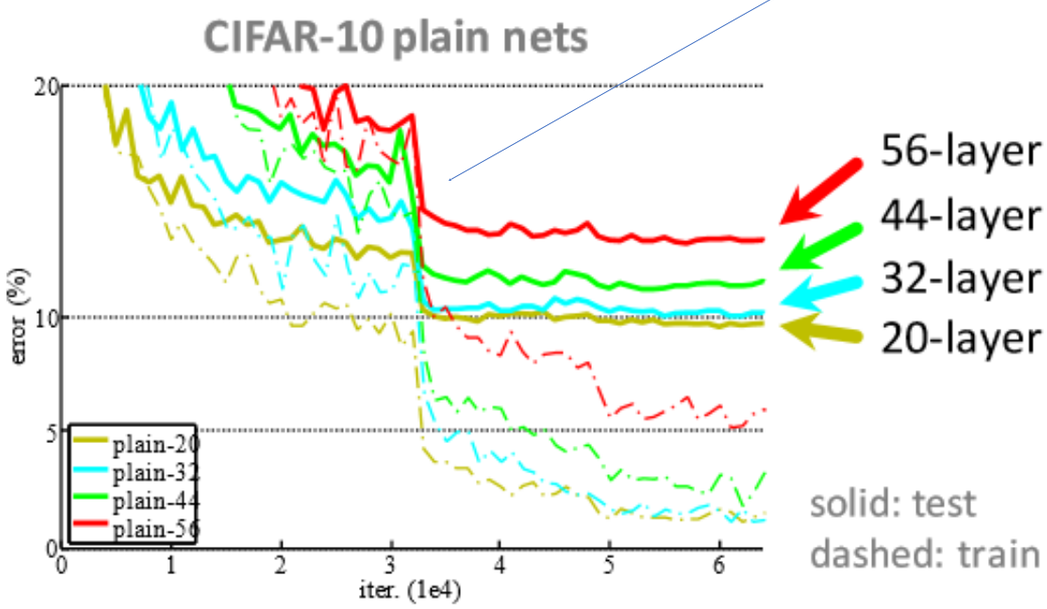


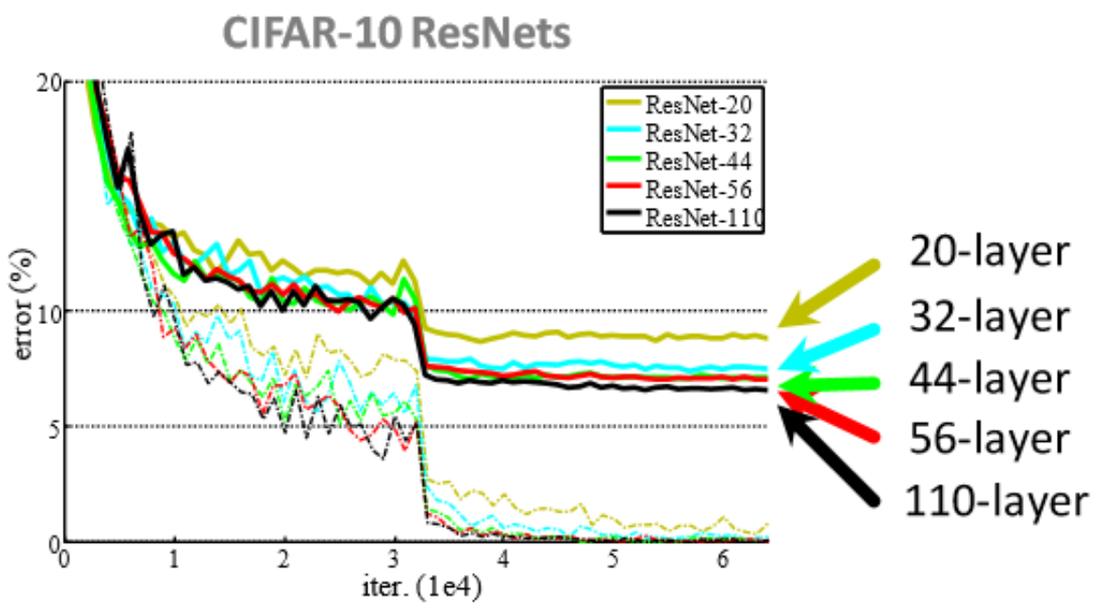
ResNet

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016.

Why so steep?

CIFAR-10 experiments



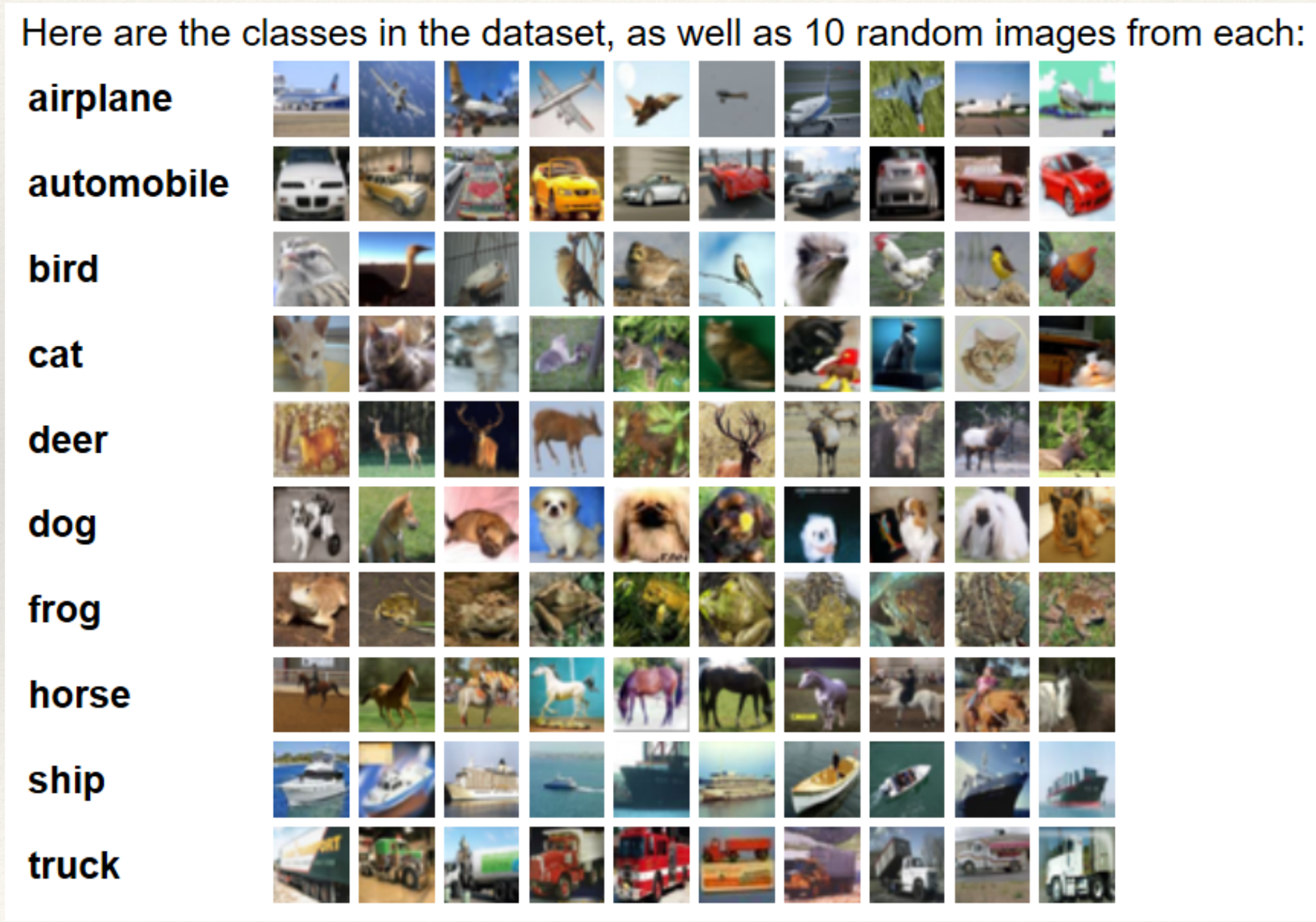


- Deep ResNets can be trained without difficulties
- Deeper ResNets have **lower training error**, and also lower test error

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

CIFAR-10

- 60,000 32x32 color images, 10 classes



R-CNN: Region-based CNN

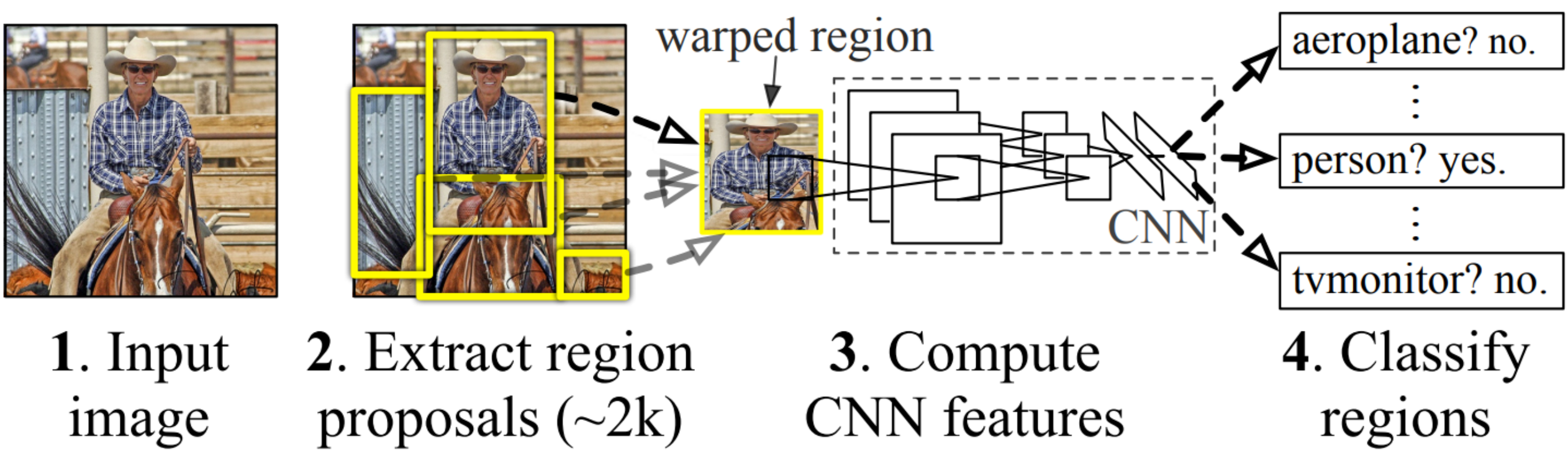
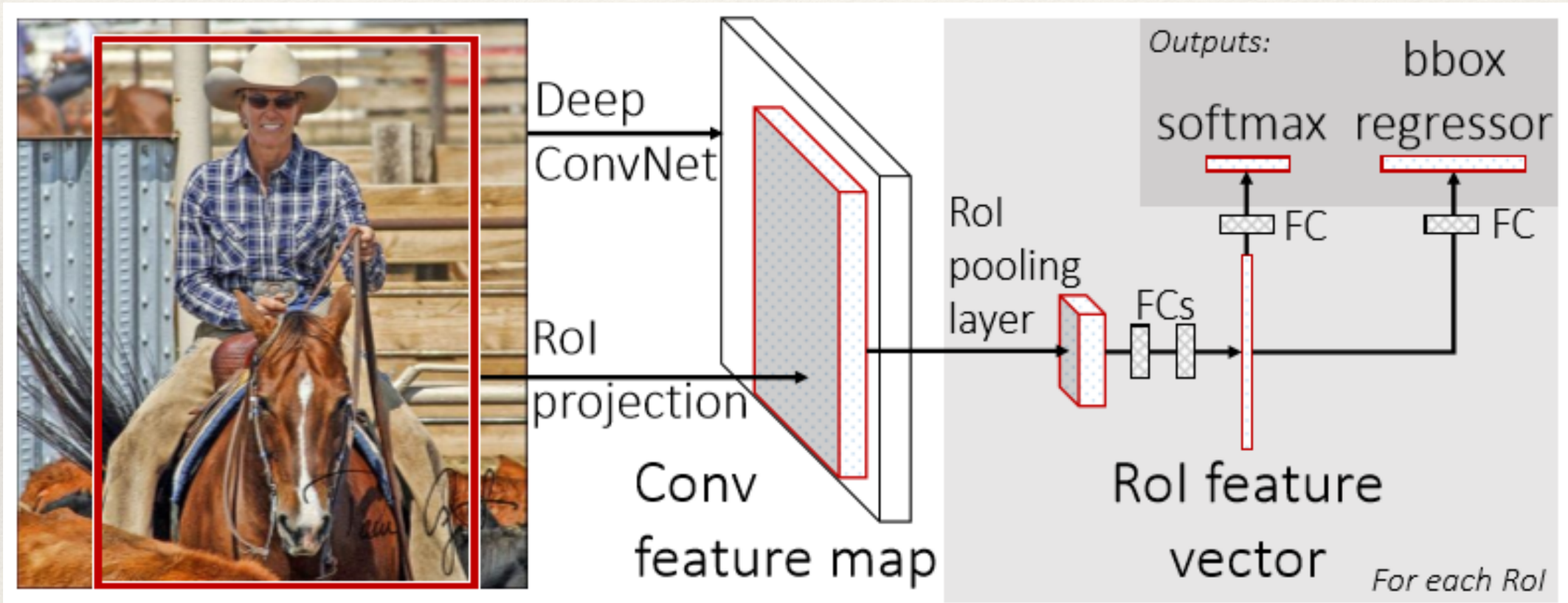


Figure: Girshick et al.

10,000 proposals with recall 0.991 is better….
but still takes 17 seconds per image to generate them.
Then I have to test each one!

Fast R-CNN

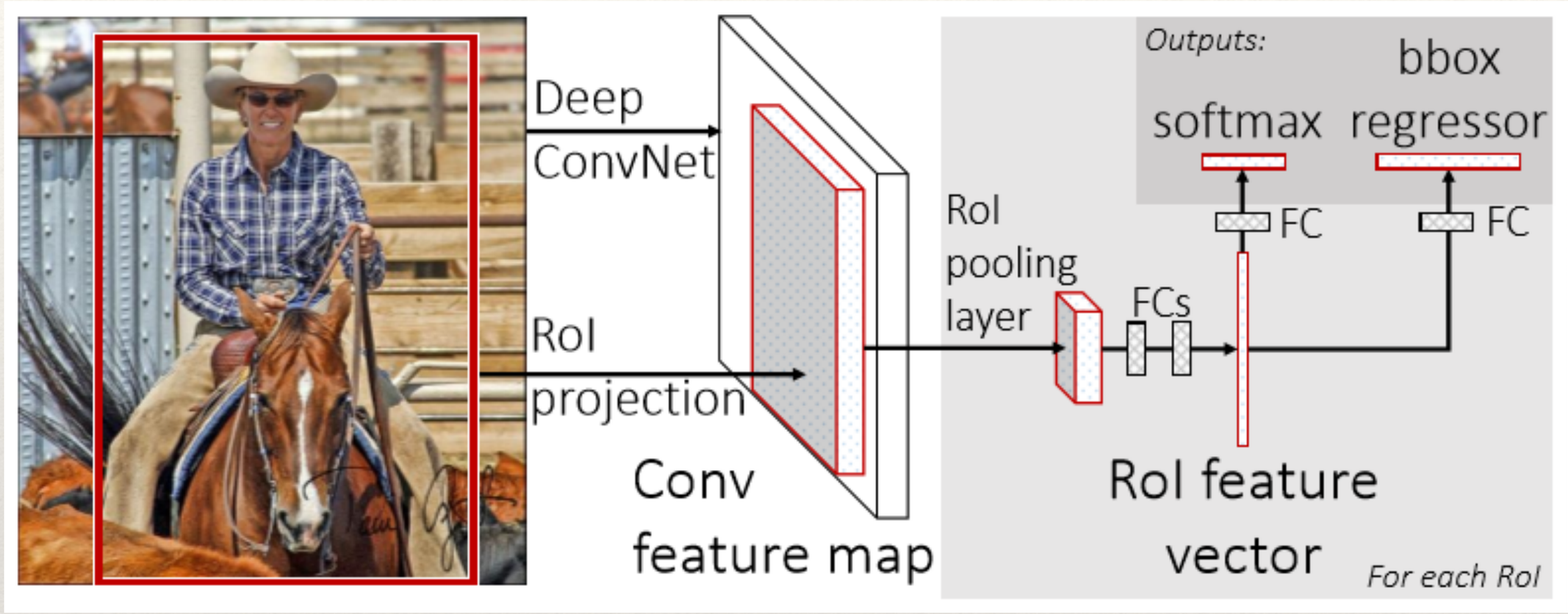
Multi-task loss



RoI = Region of Interest

Figure: Girshick et al.

Fast R-CNN

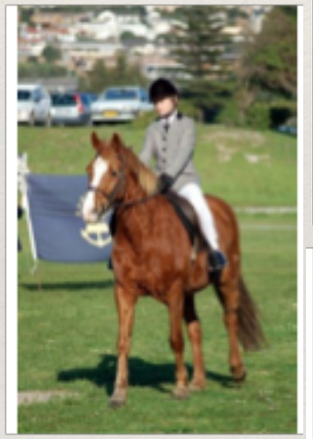


- Convolve whole image into feature map (many layers; abstracted)
- For each candidate RoI:
  - Squash feature map weights into fixed-size ‘RoI pool’ – adaptive subsampling!
    - Divide RoI into H x W subwindows, e.g., 7 x 7, and max pool
  - Learn classification on RoI pool with own fully connected layers (FCs)
  - Output classification (softmax) + bounds (regressor)

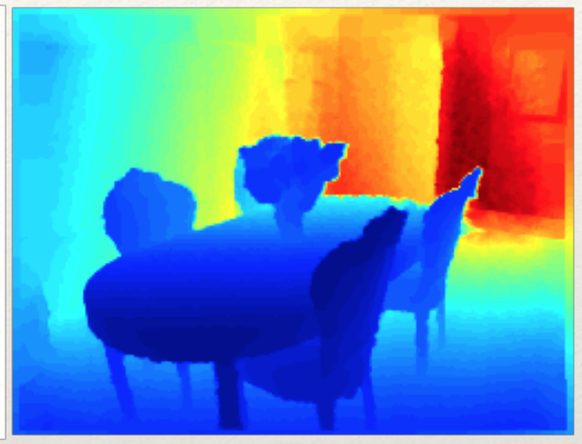
Figure: Girshick et al.

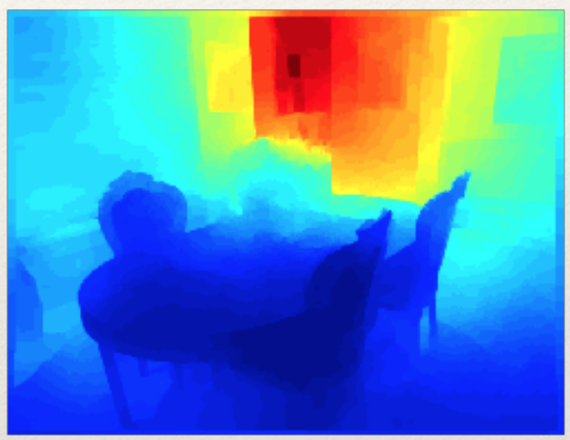
What if we want pixels out?

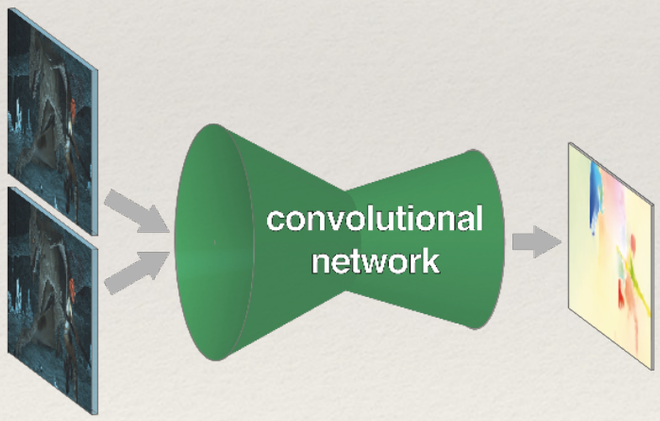
semantic segmentation



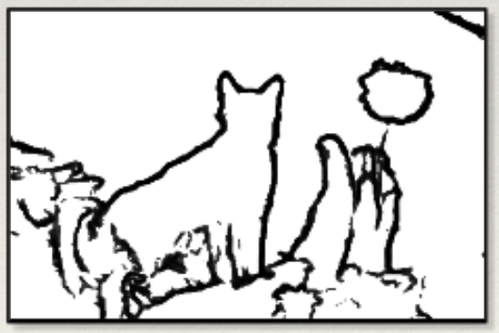
monocular depth estimation Eigen & Fergus 2015







optical flow Fischer et al. 2015



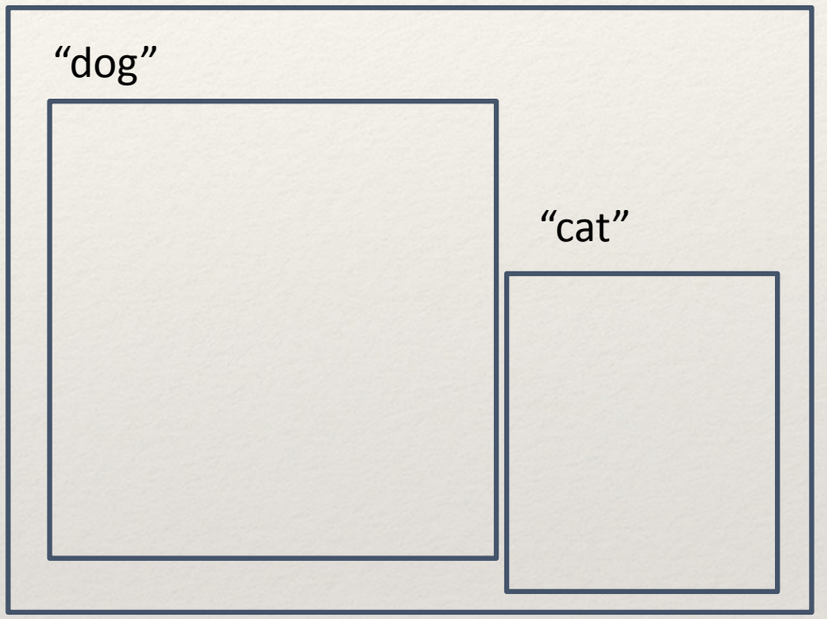
boundary prediction Xie & Tu 2015

[Long et al.]

R-CNN does detection

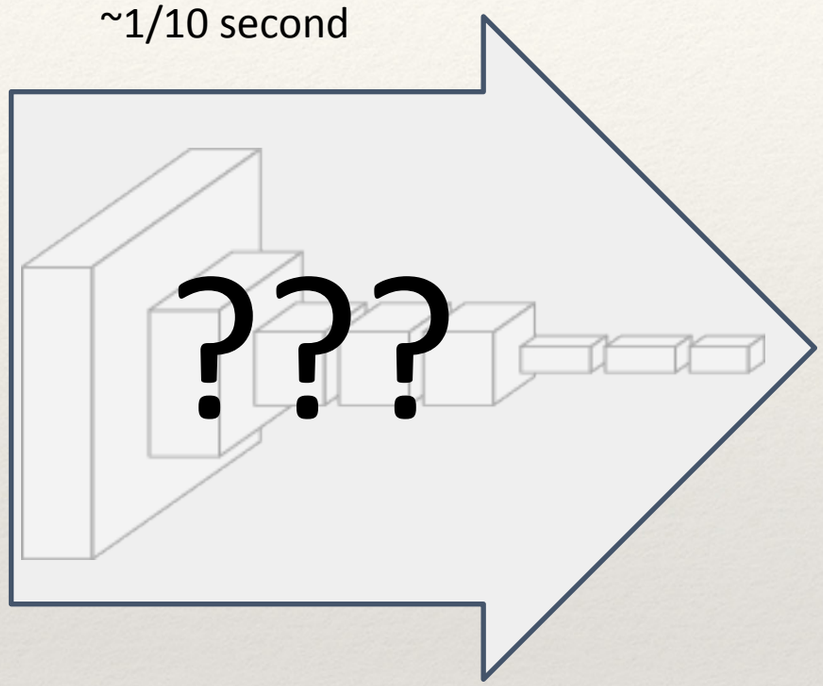
many seconds

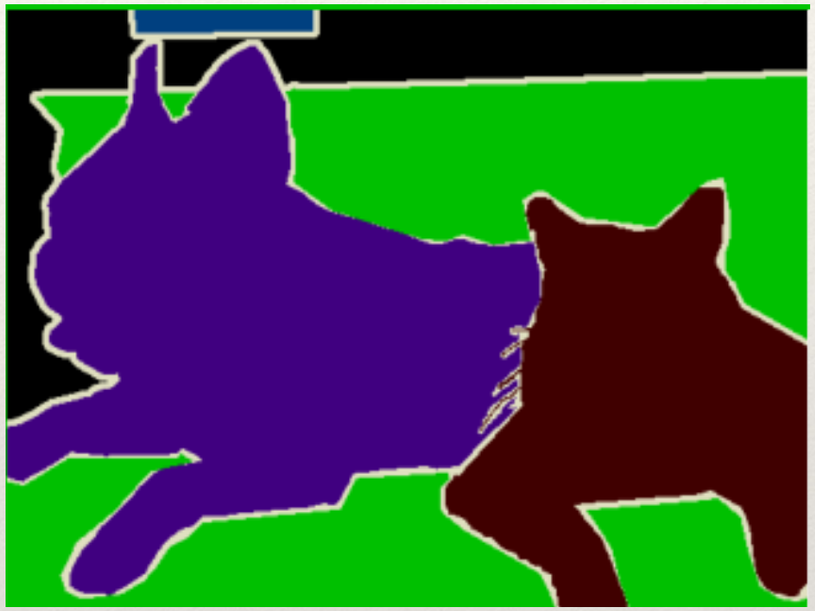
R-CNN



[Long et al.]

~1/10 second



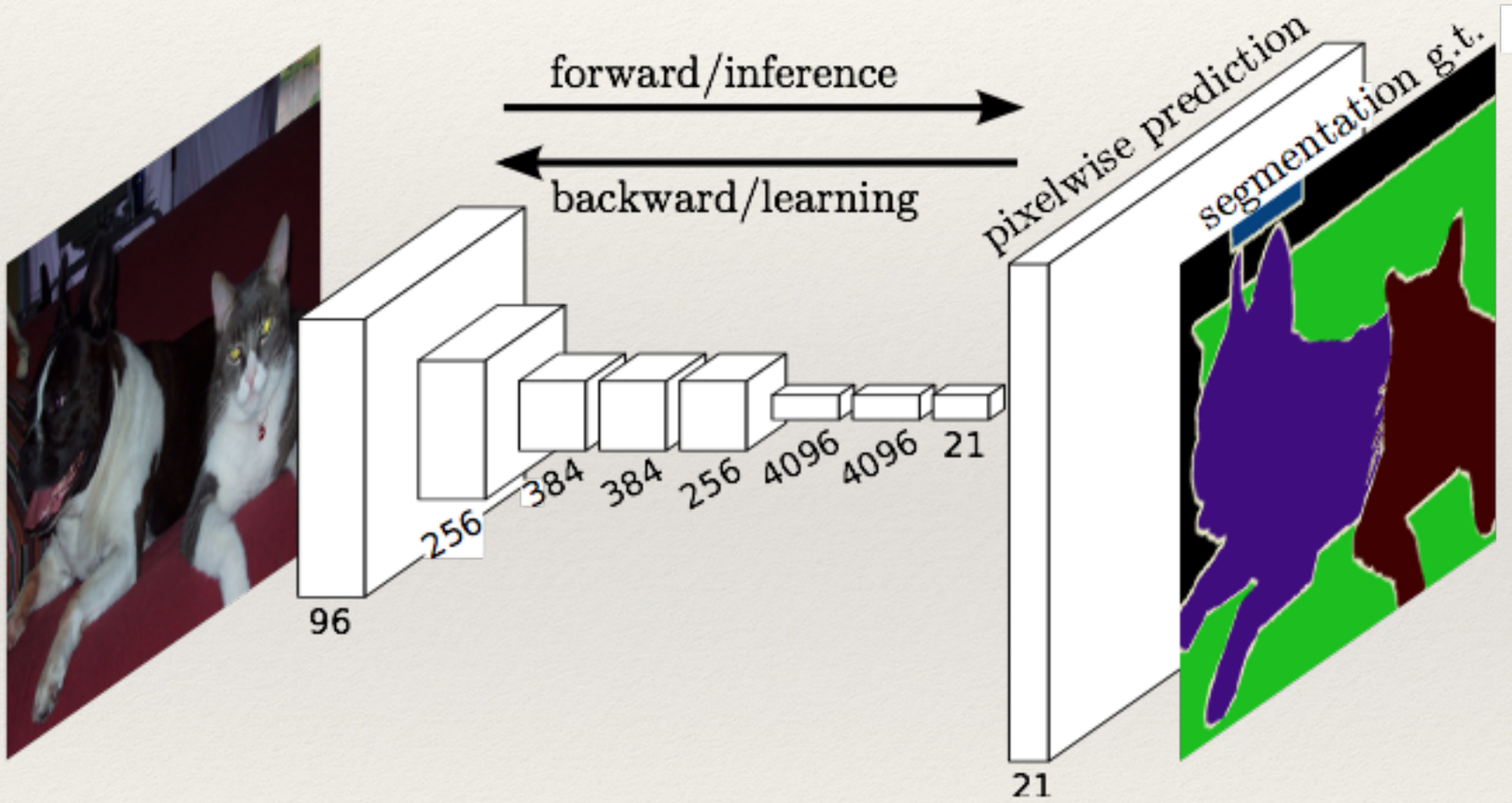


end-to-end learning

[Long et al.]

Fully Convolutional Networks for Semantic Segmentation

- UC Berkeley

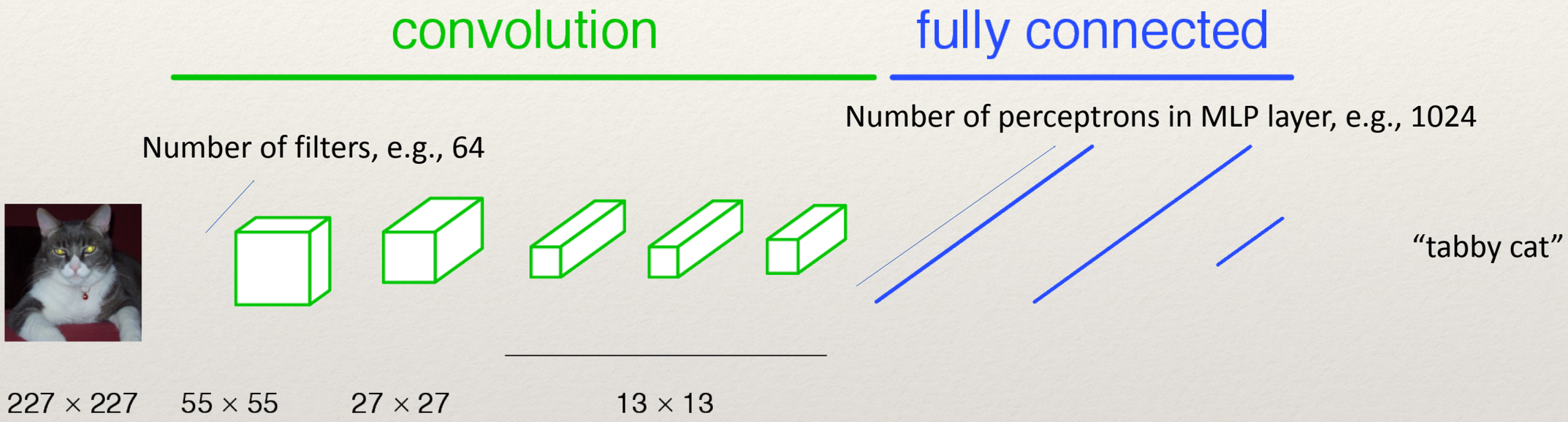


Jonathan Long* Evan Shelhamer* Trevor Darrell

[CVPR 2015]

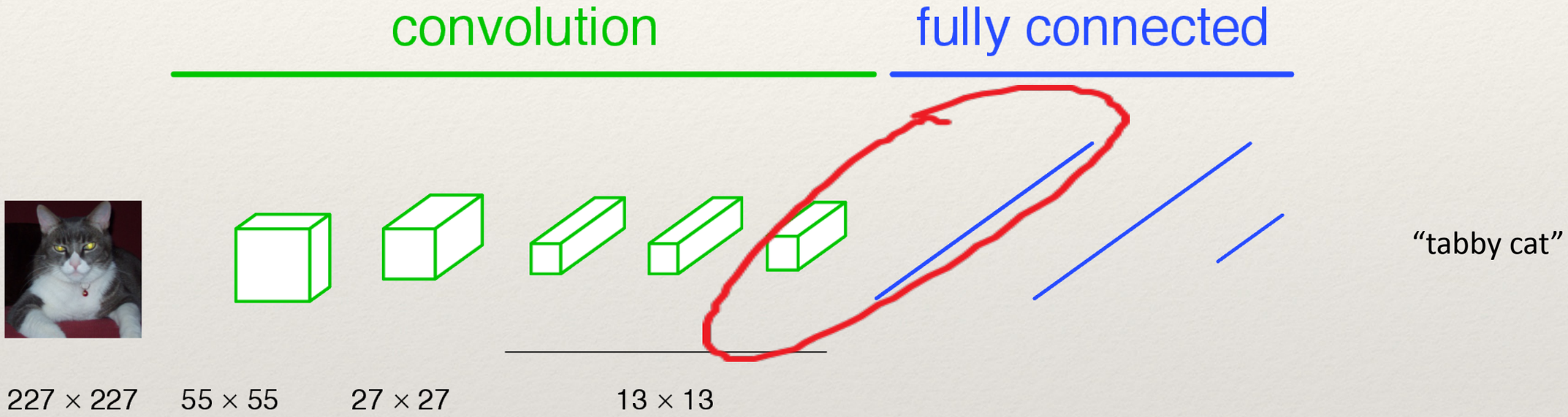
Slides from Long, Shelhamer, and Darrell

A classification network…



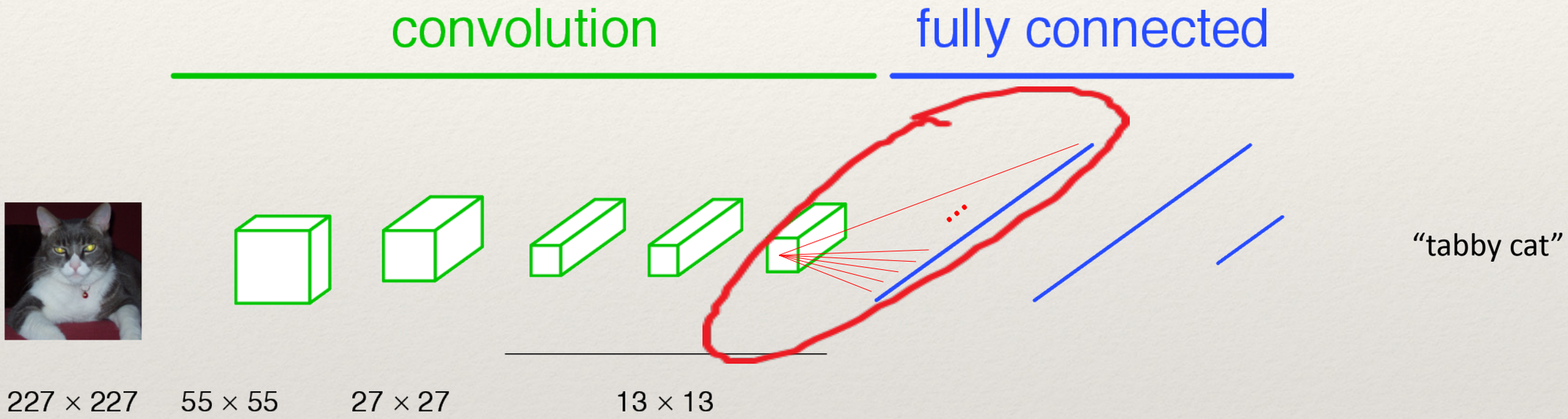
[Long et al.]

A classification network…



[Long et al.]

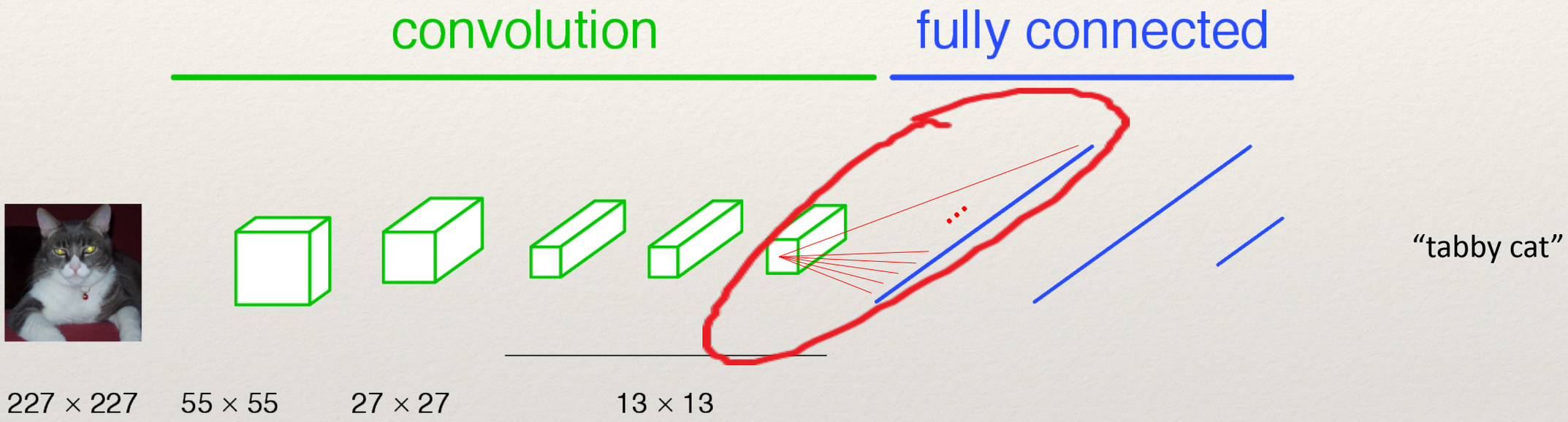
A classification network…



The response of every kernel across all positions are attached densely to the array of perceptrons in the fully-connected layer.

[Long et al.]

A classification network…



The response of every kernel across all positions are attached densely to the array of perceptrons in the fully-connected layer.

AlexNet: 256 filters over 6x6 response map
Each 2,359,296 response is attached to one of 4096 perceptrons, leading to 37 mil params.

[Long et al.]

Problem

We want a label at every pixel

Current network gives us a label for the whole image.

Approach:

- Make CNN for every sub-image size ?
- ‘Convolutionalize’ *all layers* of network, so that we can treat it as one (complex) filter and slide around our full image.

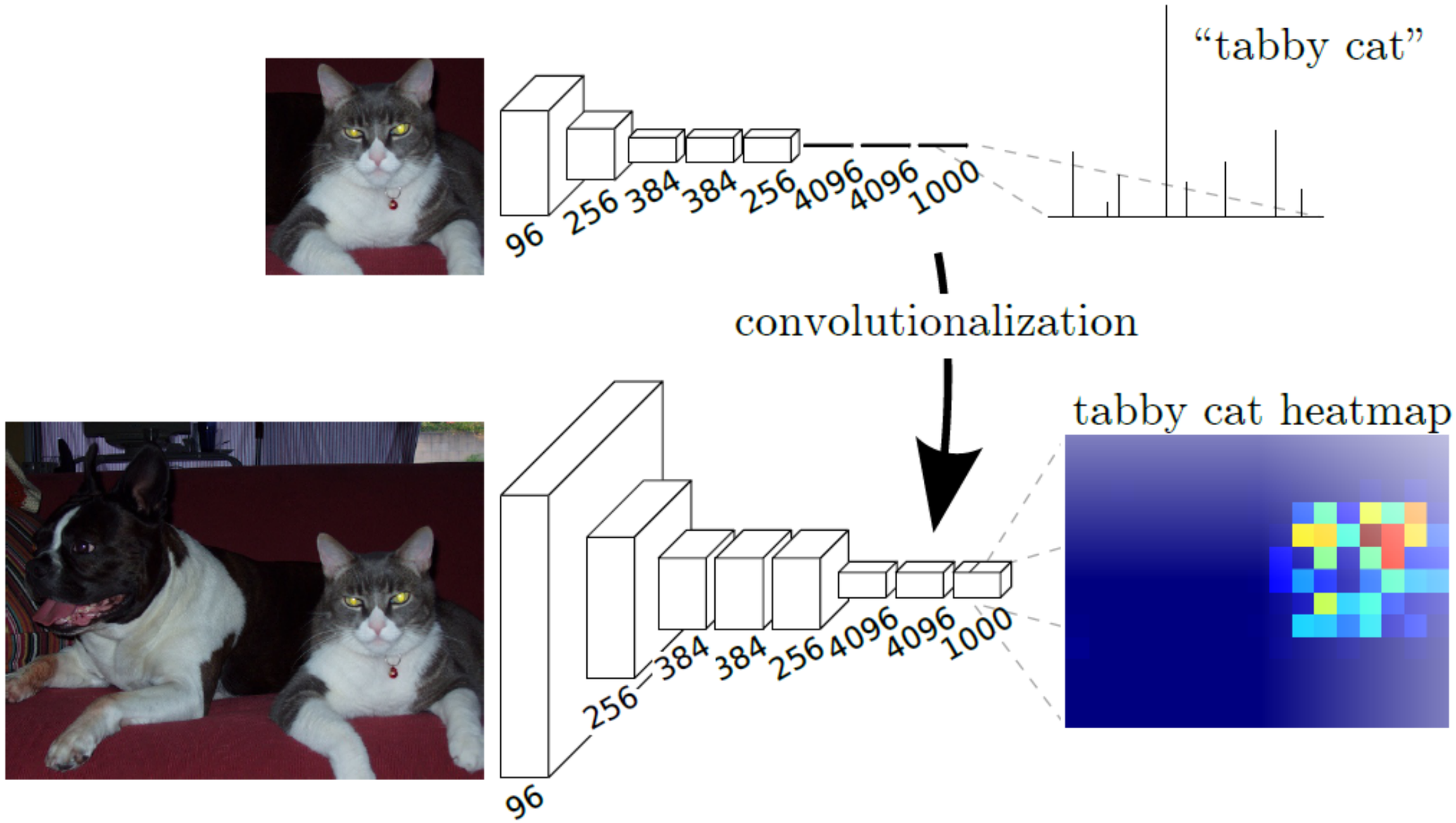


Figure 2. Transforming fully connected layers into convolution layers enables a classification net to output a heatmap. Adding layers and a spatial loss (as in Figure 1) produces an efficient machine for end-to-end dense learning.

Long, Shelhamer, and Darrell 2014

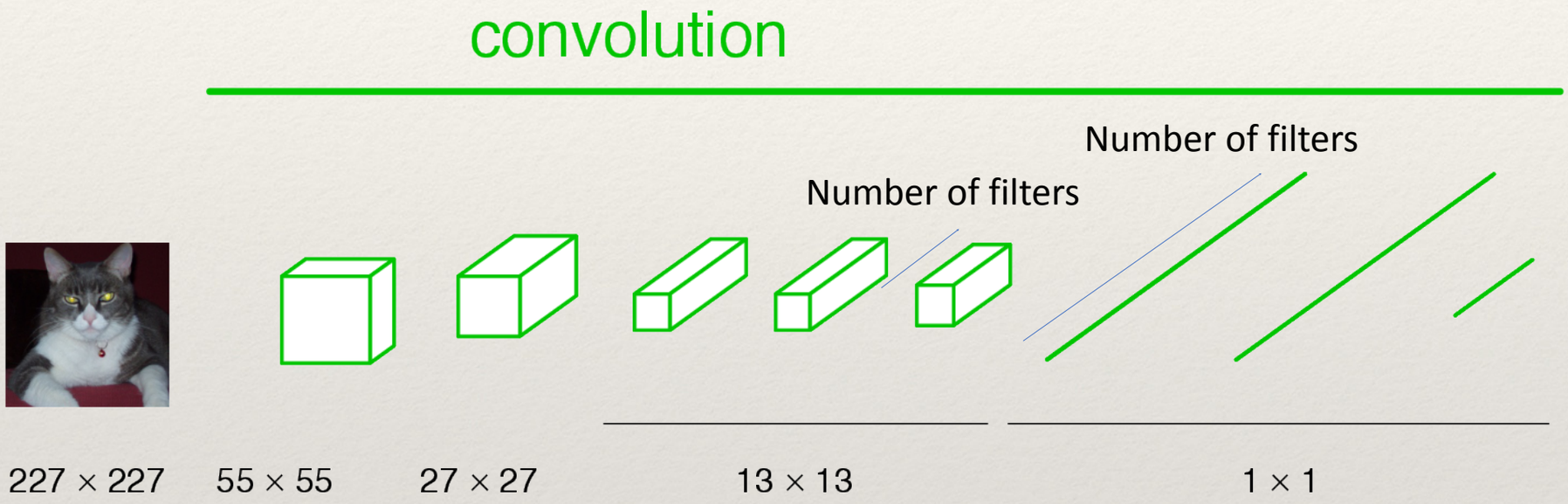
Yann LeCun

6 April 2015 ·

Follow

In Convolutional Nets, there is no such thing as "fully-connected layers". There are only convolution layers with 1x1 convolution kernels and a full connection table.

Convolutionalization



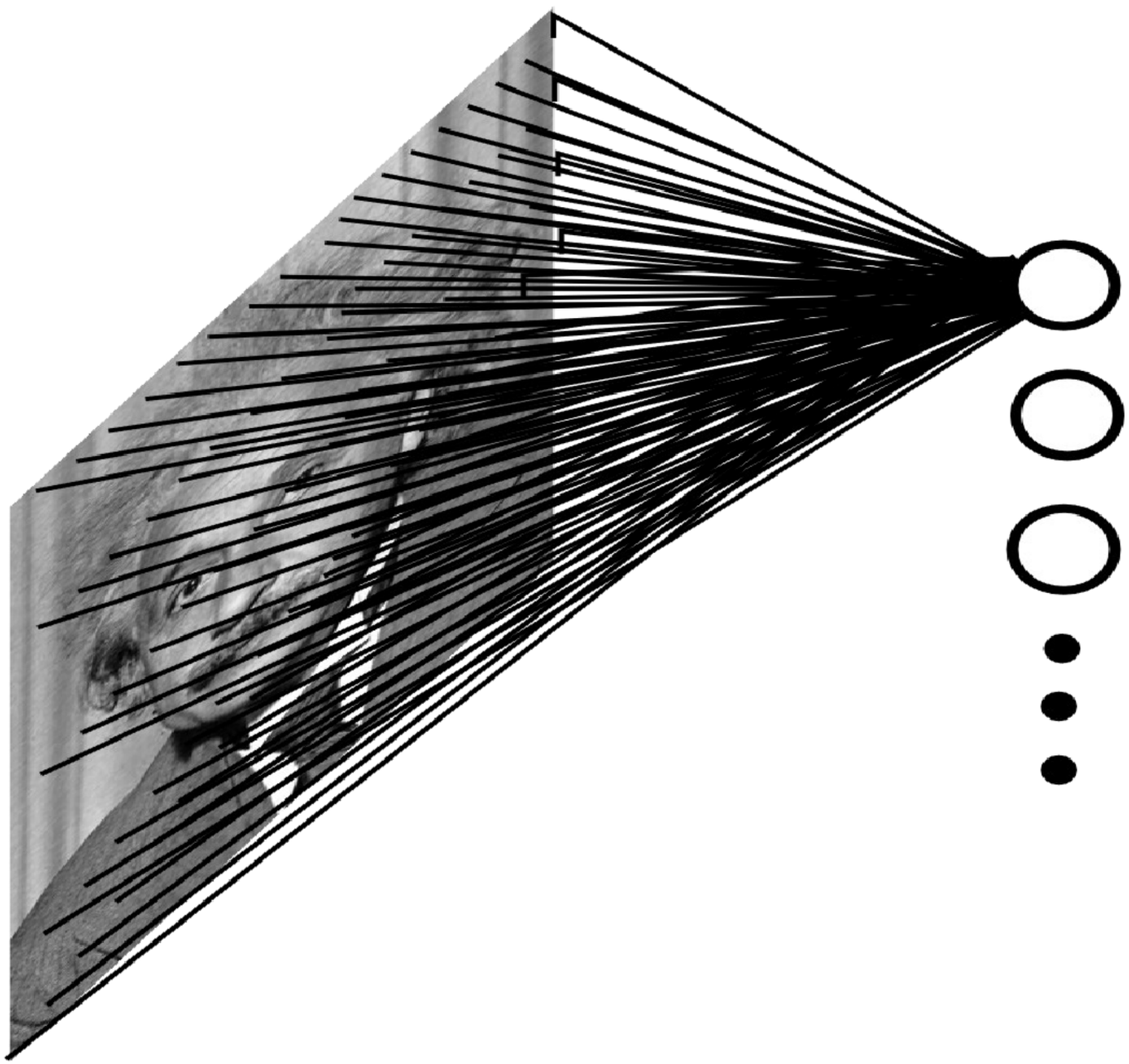
1x1 convolution operates across all filters in the previous layer, and is slid across all positions.

[Long et al.]

Back to the fully-connected perceptron…

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x \leq 0 \\ 1 & \text{if } w \cdot x > 0 \end{cases}$$

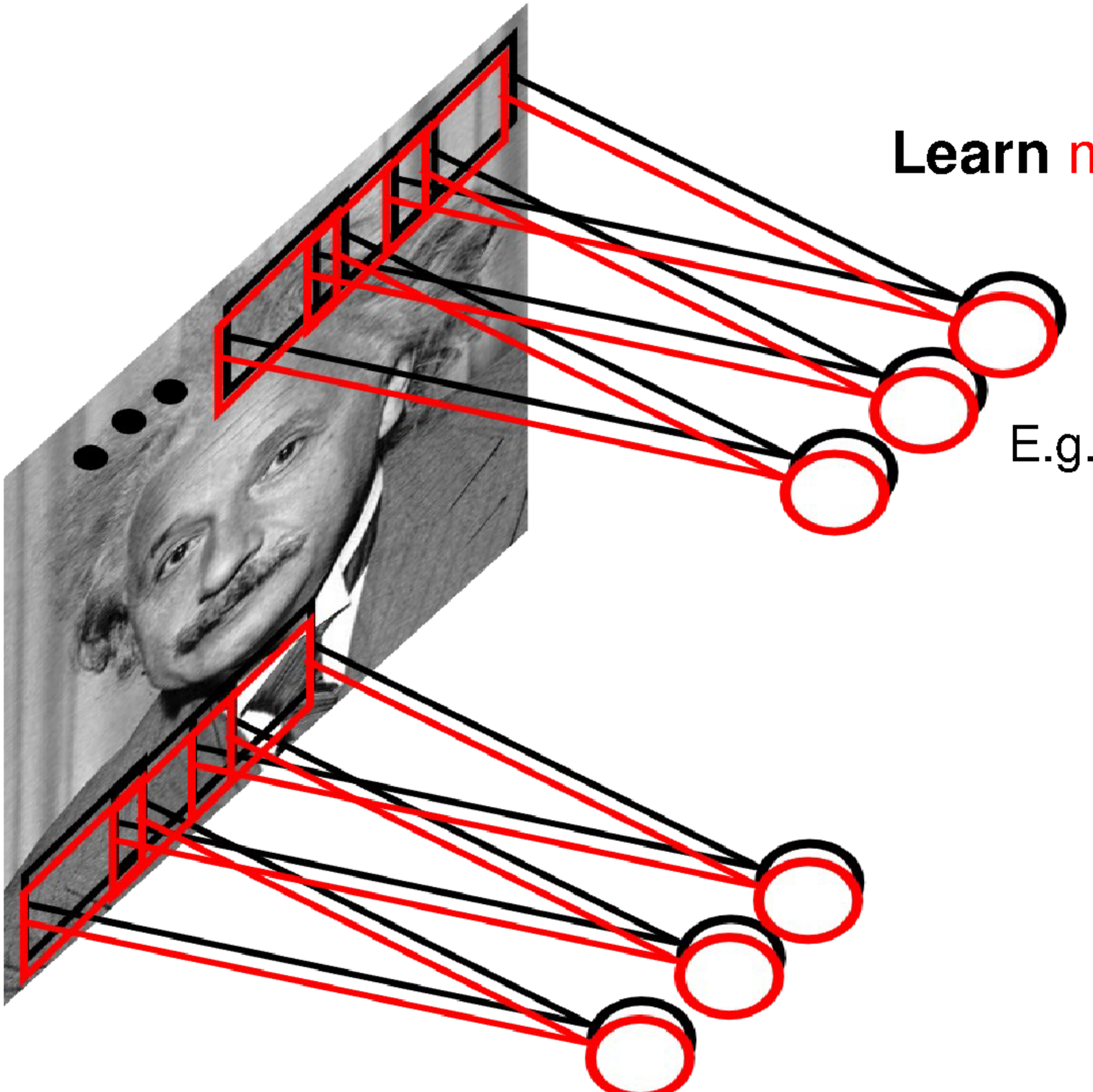
$$w \cdot x \equiv \sum_j w_j x_j$$



Perceptron is connected to every value in the previous layer (across all channels; 1 visible).

[Long et al.]

Convolutional Layer



Learn multiple filters.

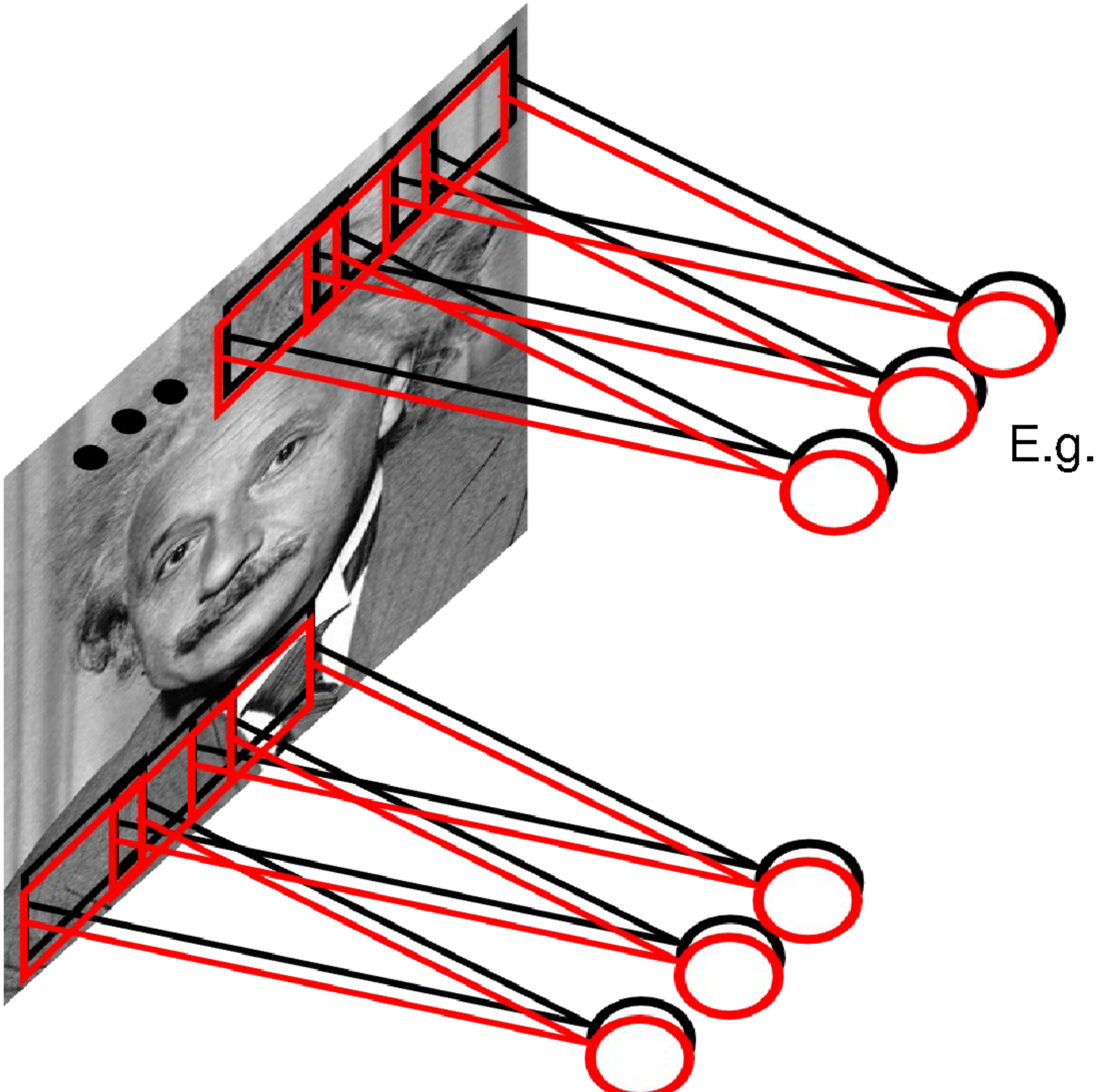
E.g.: 200x200 image

100 Filters

Filter size: 10x10

10K parameters

Convolutional Layer



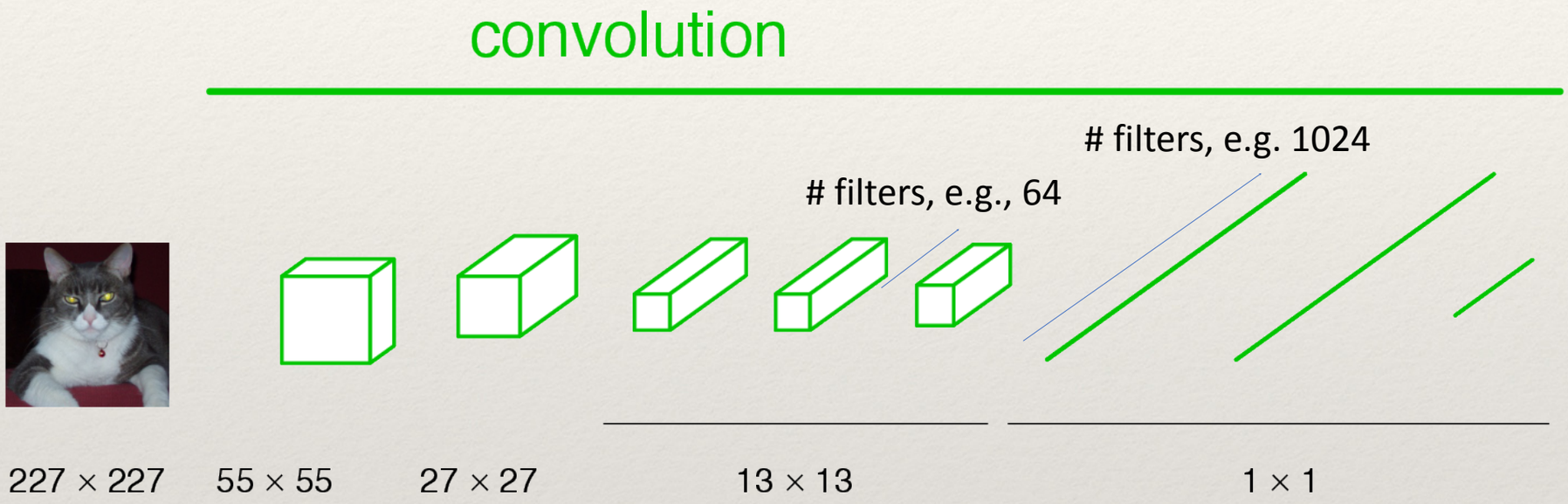
E.g.: 200x200 image

100 Filters

Filter size: 1x1

100 parameters

Convolutionalization

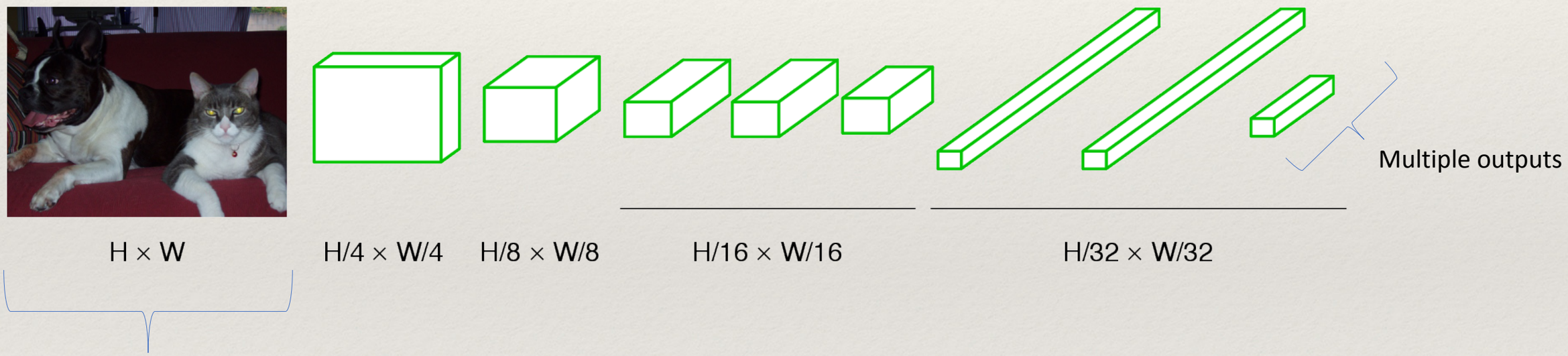


1x1 convolution operates across all filters in the previous layer, and is slid across all positions.

e.g., 64x1x1 kernel, with shared weights over 13x13 output, x1024 filters = 11mil params.

[Long et al.]

Becoming fully convolutional



When we turn these operations into a convolution, the 13x13 just becomes another parameter and our output size adjust dynamically.

Now we have a *vector/matrix* output, and our network acts itself like a complex filter.

[Long et al.]

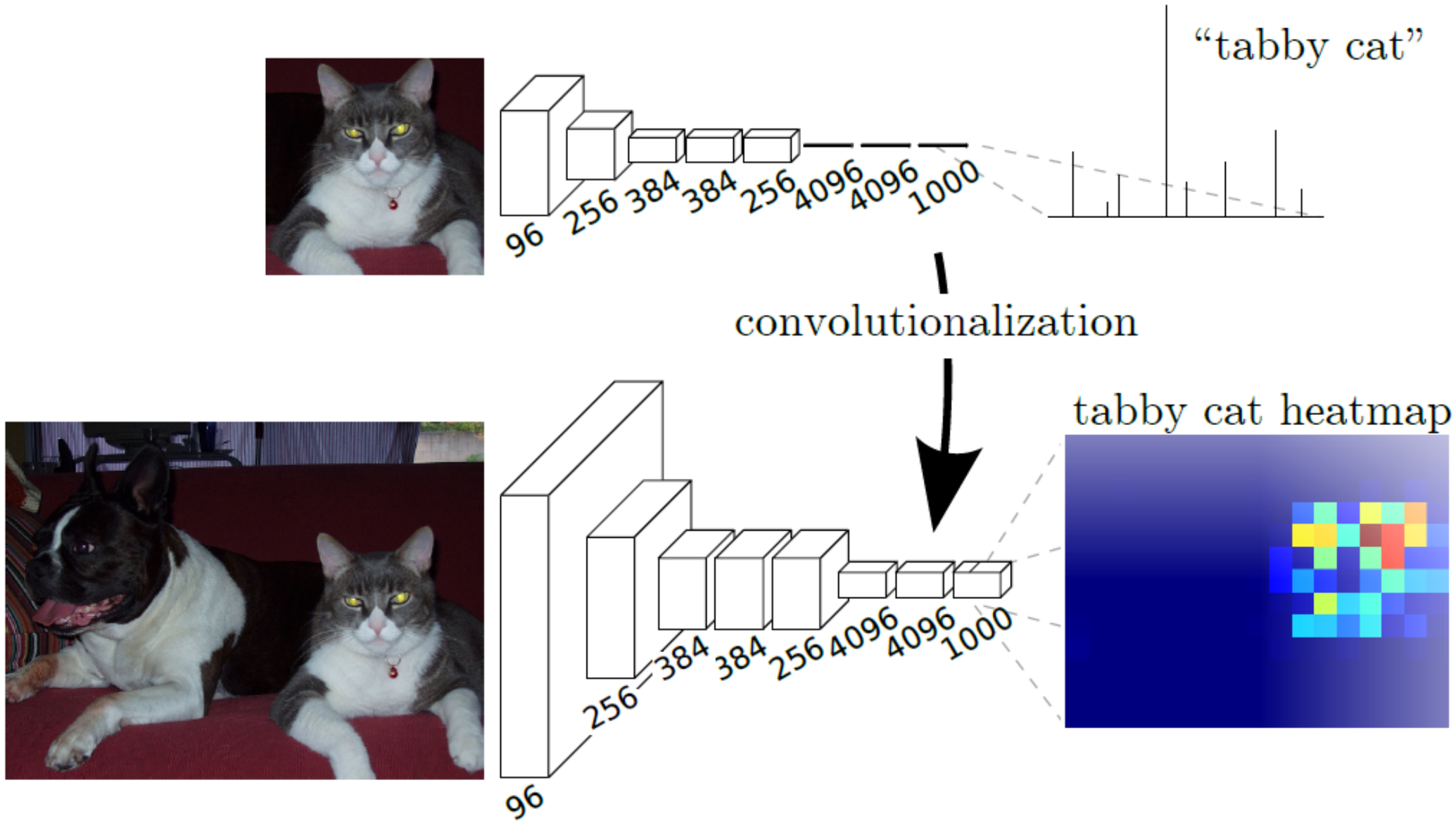
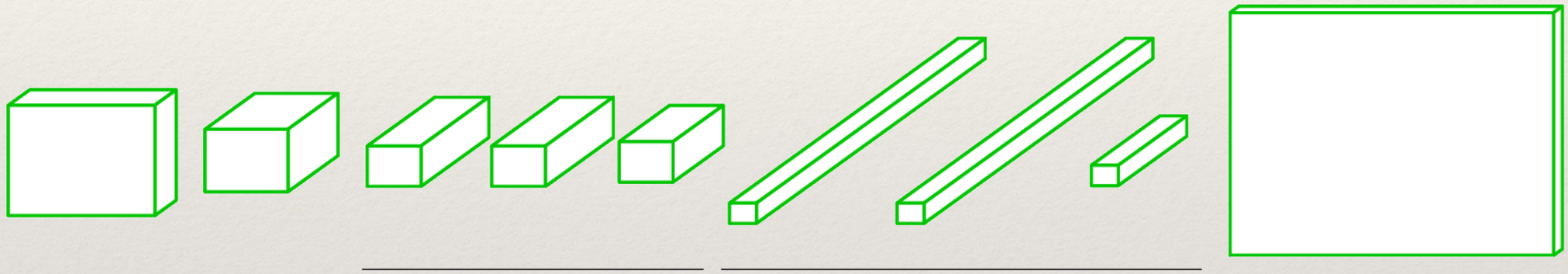


Figure 2. Transforming fully connected layers into convolution layers enables a classification net to output a heatmap. Adding layers and a spatial loss (as in Figure 1) produces an efficient machine for end-to-end dense learning.

Long, Shelhamer, and Darrell 2014

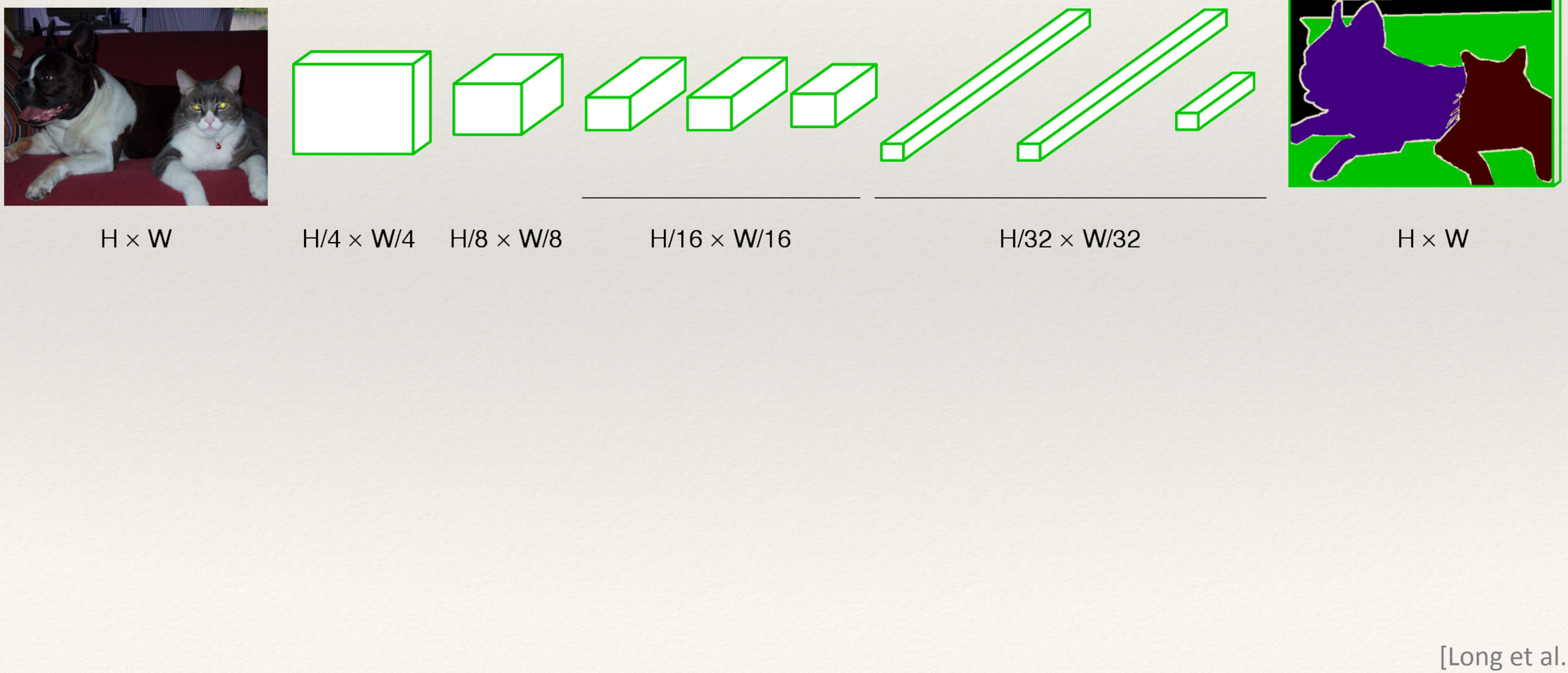
Upsampling the output



Some upsampling algorithm to return us to H x W

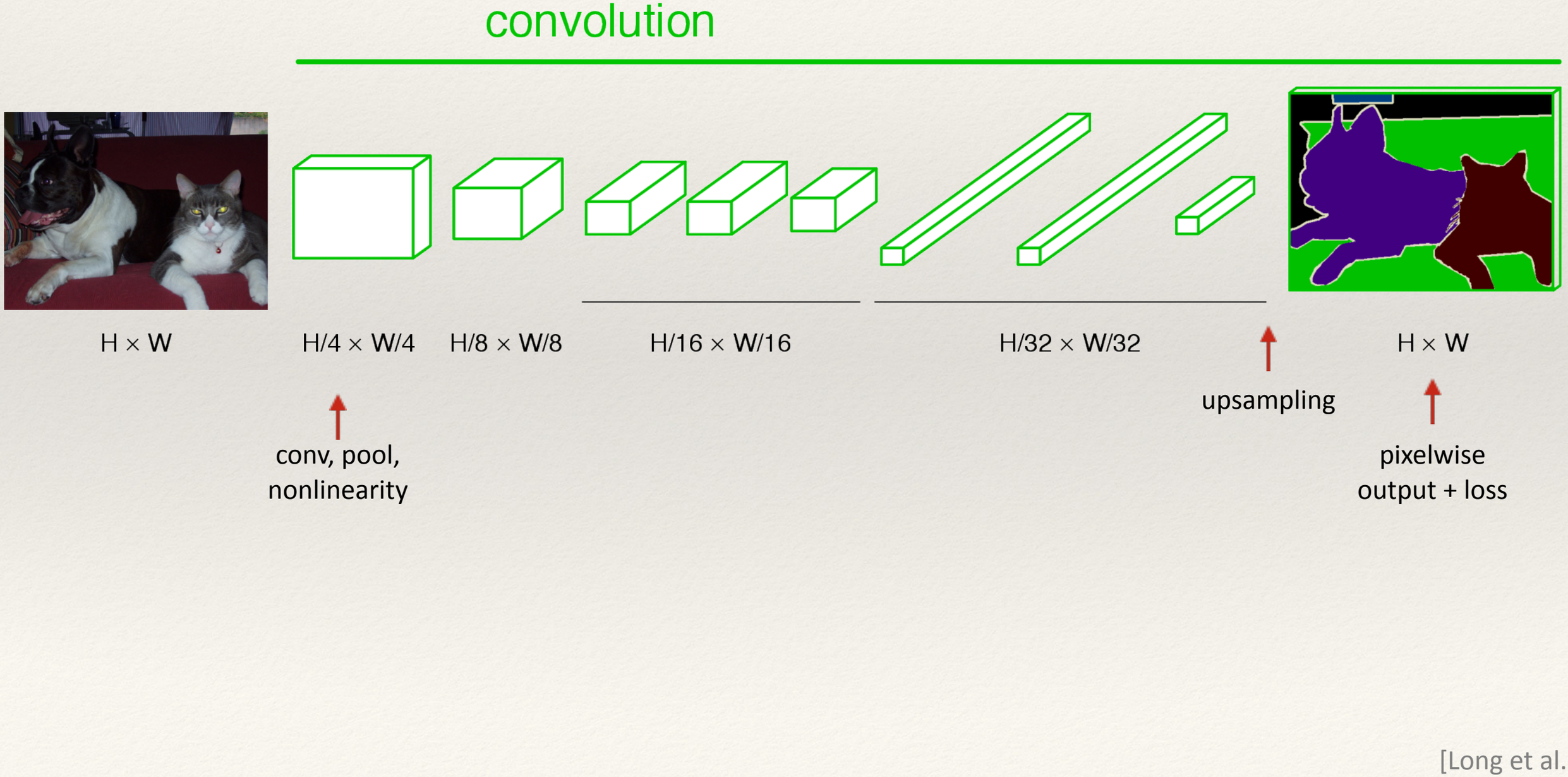
[Long et al.]

End-to-end, pixels-to-pixels network



[Long et al.]

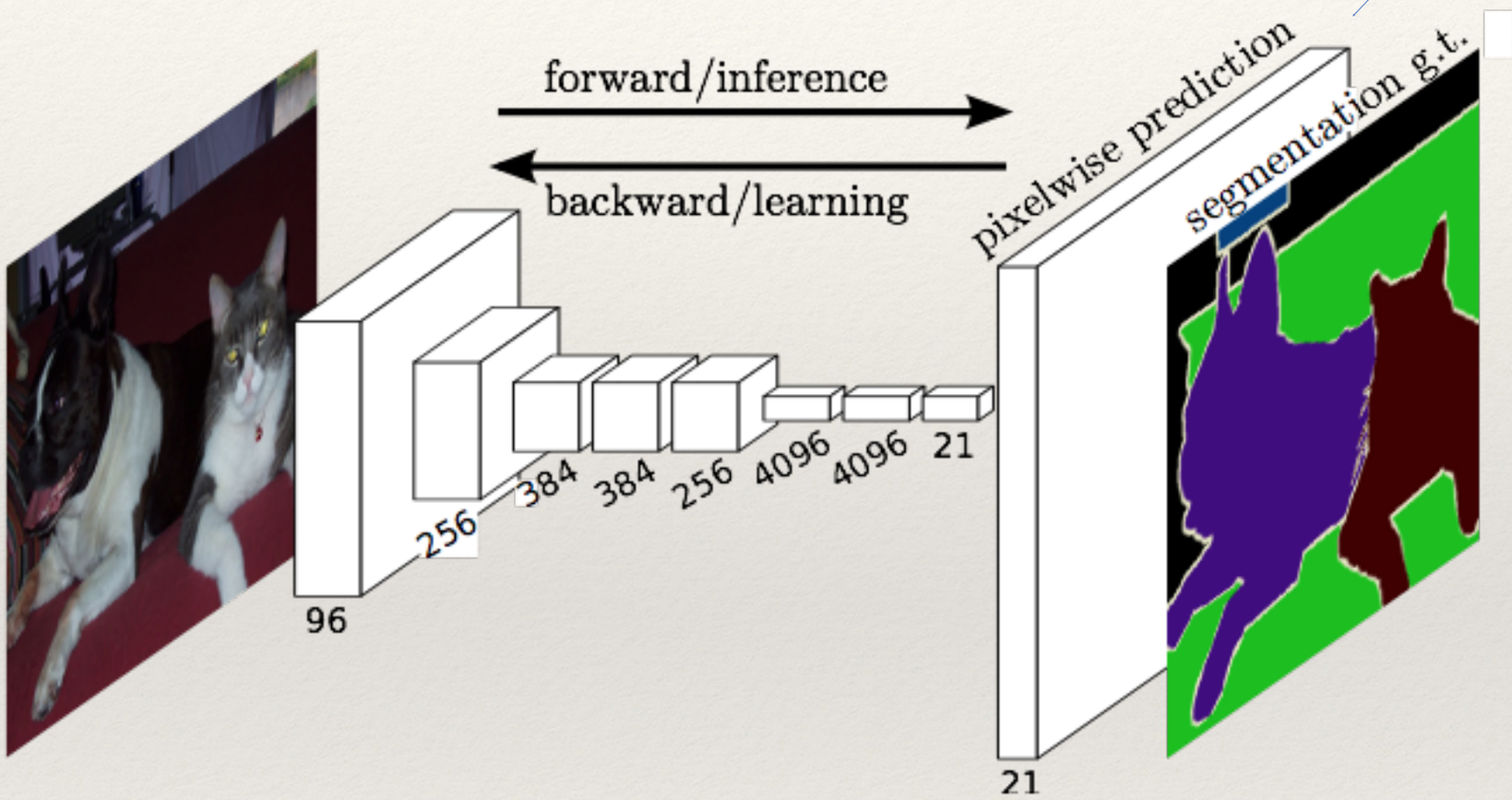
End-to-end, pixels-to-pixels network



[Long et al.]

What is the upsampling layer?

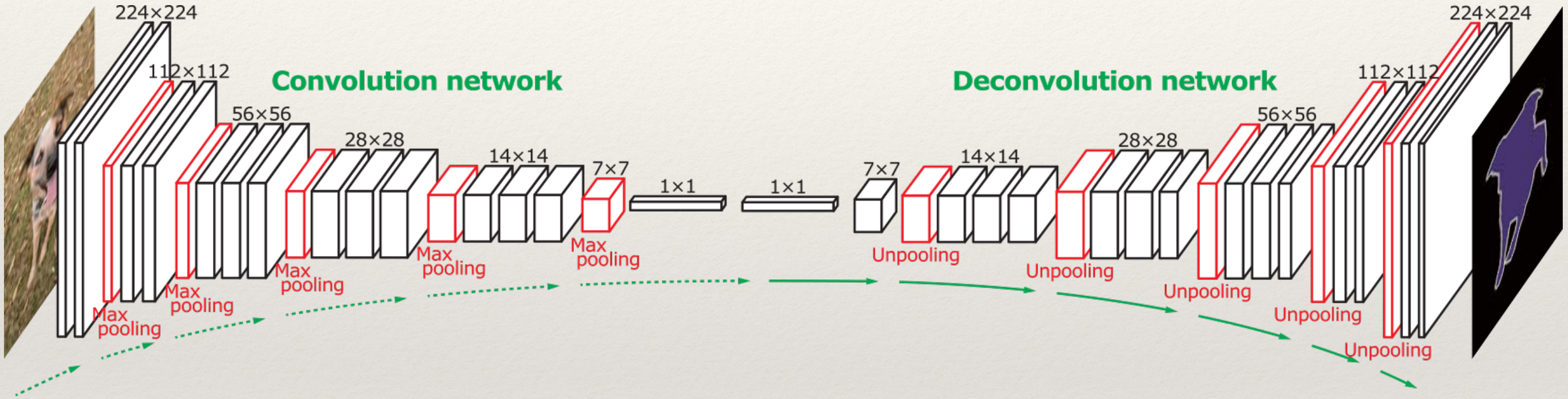
This one.



Hint: it's actually an upsampling *network*

[Long et al.]

‘Deconvolution’ networks learn to upsample



Often called “deconvolution”, but misnomer.
Not the deconvolution that we saw in deblurring -> that is division in the Fourier domain.

‘Transposed convolution’ is better.

Zeiler et al., Deconvolutional Networks, CVPR 2010
Noh et al., Learning Deconvolution Network for Semantic Segmentation, ICCV 2015