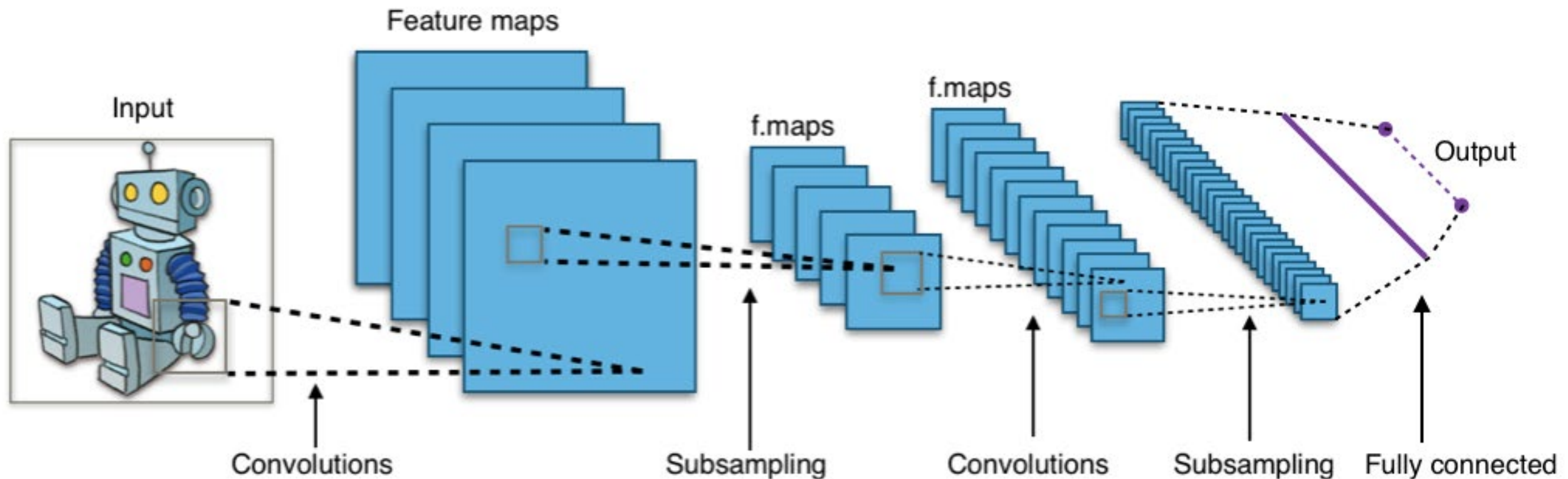


Neural networks



16-705 Computer Vision
Spring 2026, Lecture 7

Course announcements

- Homework assignment 1 has been posted and is due on April 10th.
- Office hours schedule posted on course website and Slack.

Overview of today's lecture

- Perceptron.
- Neural networks.
- Training perceptrons.
- Gradient descent.
- Backpropagation.
- Stochastic gradient descent.
- Convolutional neural networks.

Slide credits

Most of these slides were adapted from:

- Kris Kitani (16-385, Spring 2017).
- Noah Snavely (Cornell University).
- Fei-Fei Li (Stanford University).
- Andrej Karpathy (Stanford University).

Perceptron

1950s Age of the Perceptron

1957 The Perceptron (Rosenblatt)

1969 Perceptrons (Minsky, Papert)

1980s Age of the Neural Network

1986 Back propagation (Hinton)

1990s Age of the Graphical Model

2000s Age of the Support Vector Machine

2010s Age of the Deep Network

deep learning = known algorithms + computing power + big data

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA

† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure¹.

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors². Learning becomes more interesting but

more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input-output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j , to unit j is a linear function of the outputs, y_i , of the units that are connected to j and of the weights, w_{ji} , on these connections

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

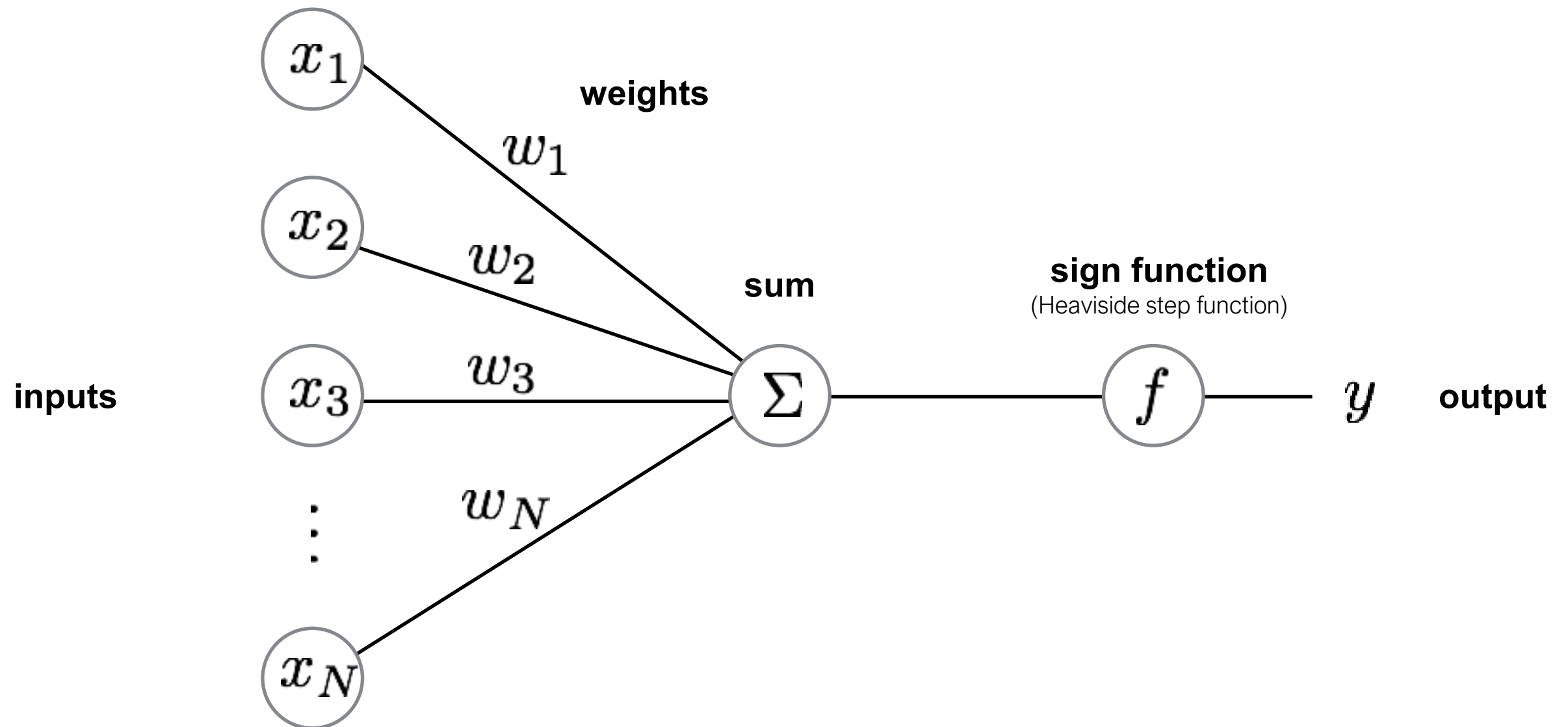
Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j , which is a non-linear function of its total input

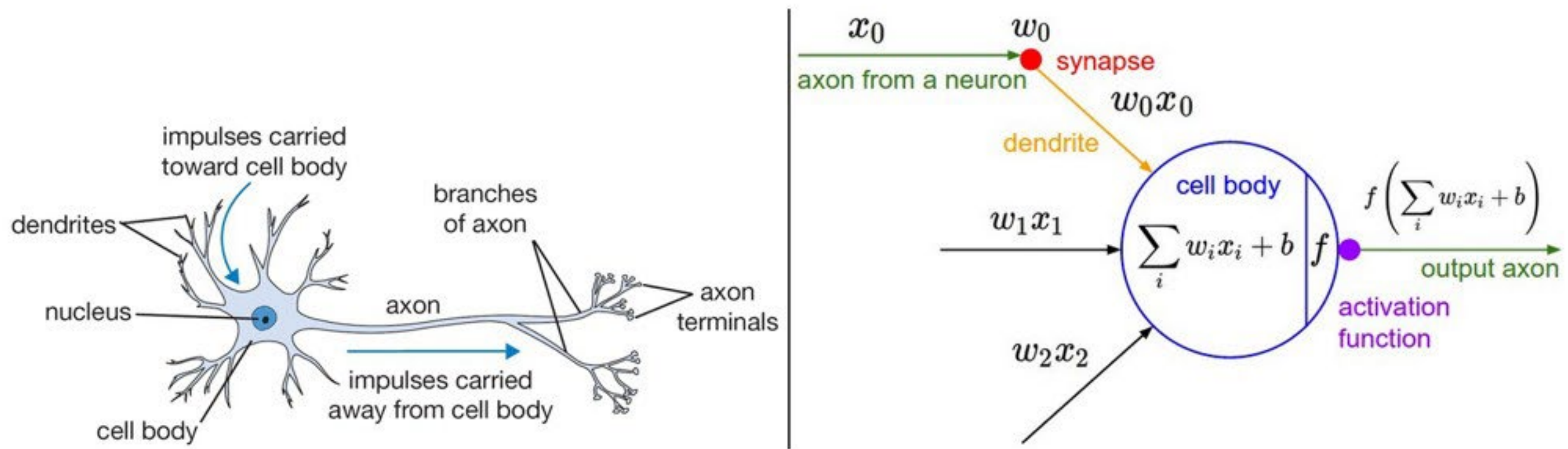
$$y_j = \frac{1}{1 + e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed.

The Perceptron



Aside: Inspiration (?) from Biology



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Neural nets/perceptrons are **loosely** inspired by biology.

But they certainly are **not** a model of how the brain works, or even how neurons work.

1: **function** PERCEPTRON ALGORITHM

2: $\mathbf{w}^{(0)} \leftarrow \mathbf{0}$

3: **for** $t = 1, \dots, T$ **do**

4: **RECEIVE**($\mathbf{x}^{(t)}$) $\mathbf{x} \in \{0, 1\}^N$ N-d binary vector

5: $\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$ perceptron is just one line of code!

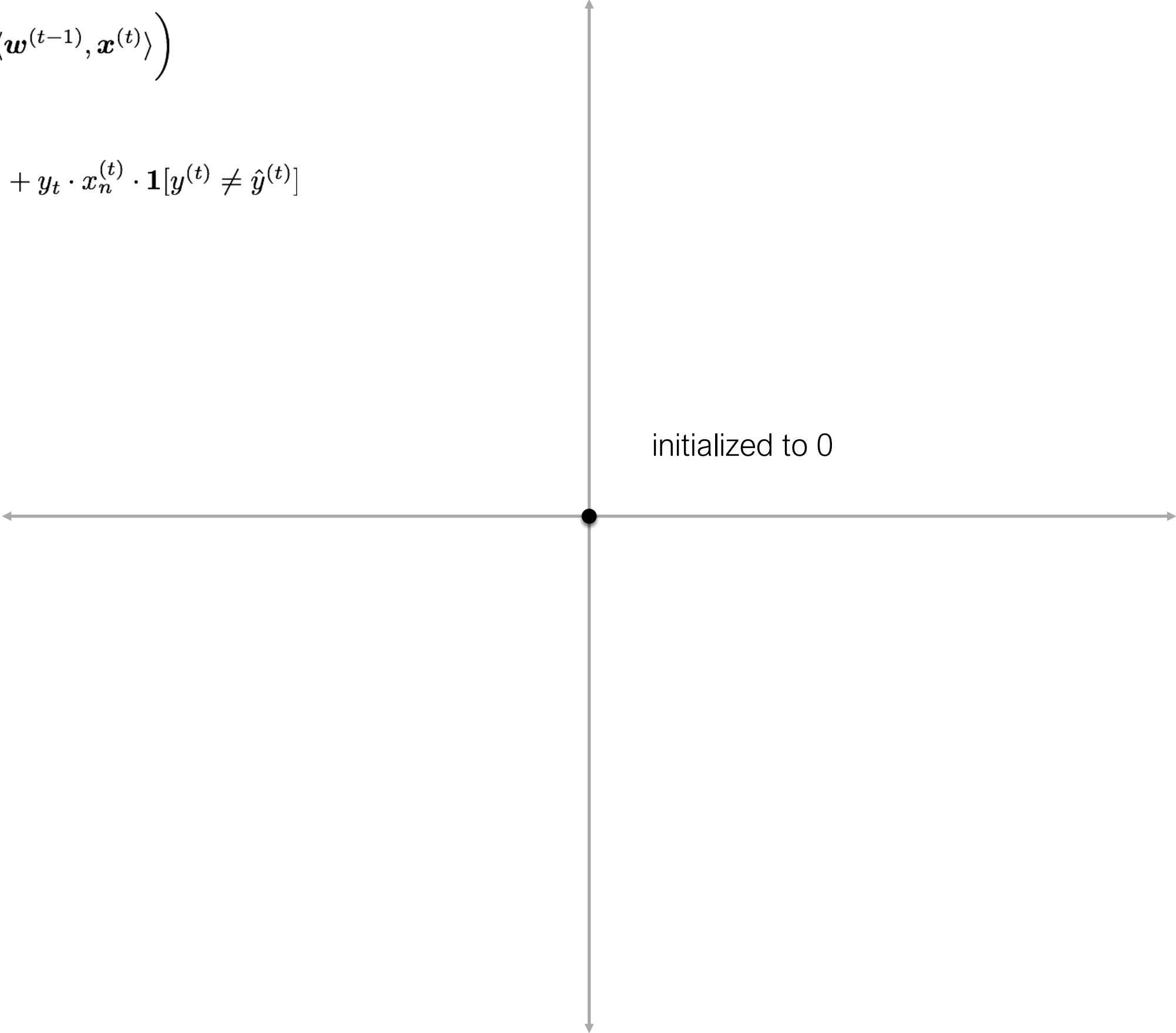
6: **RECEIVE**(y^t) $y \in \{1, -1\}$
$$7: \quad w_n^{(t)} = w_n^{(t-1)} + y_t \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$



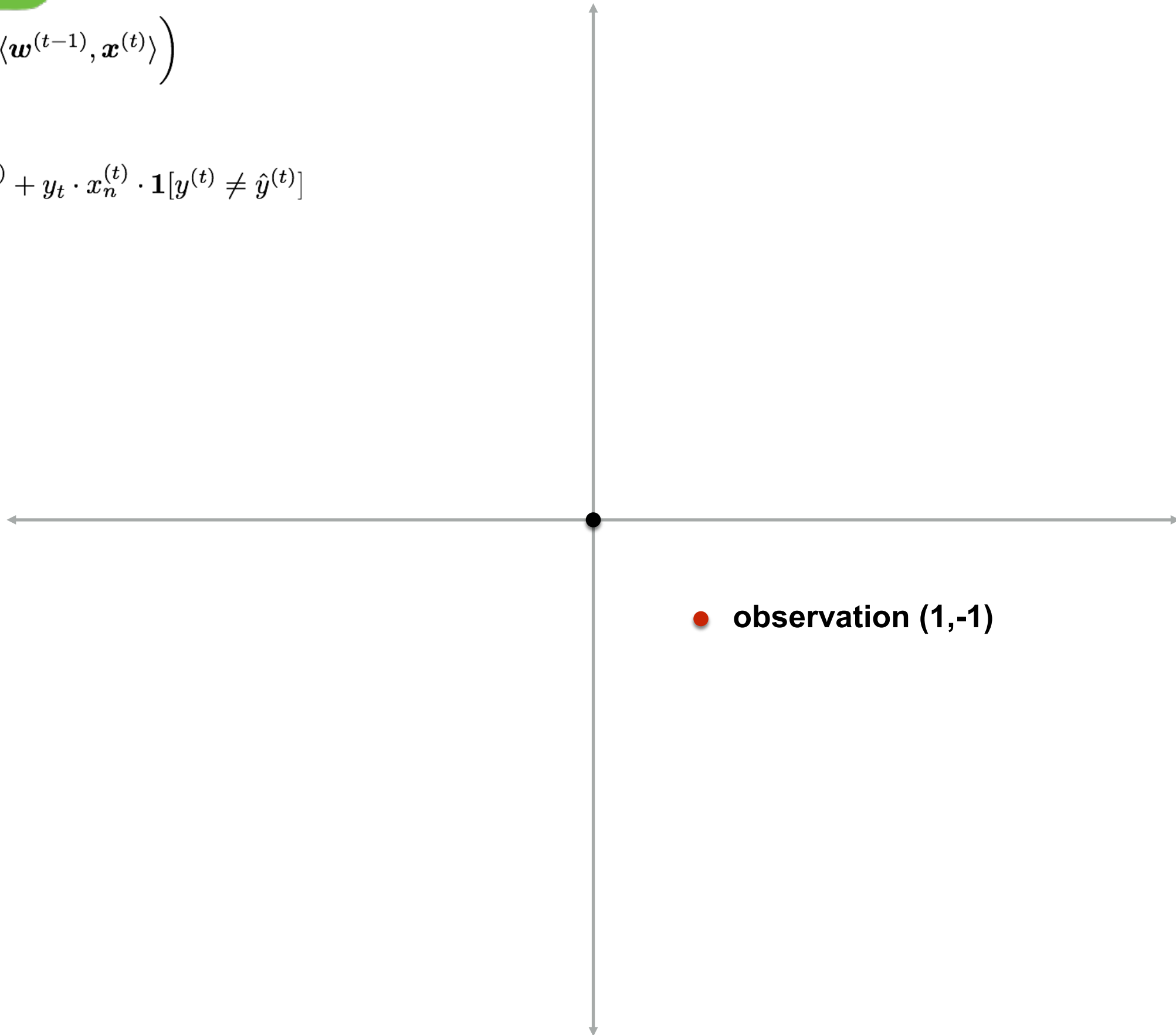
initialized to 0

RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

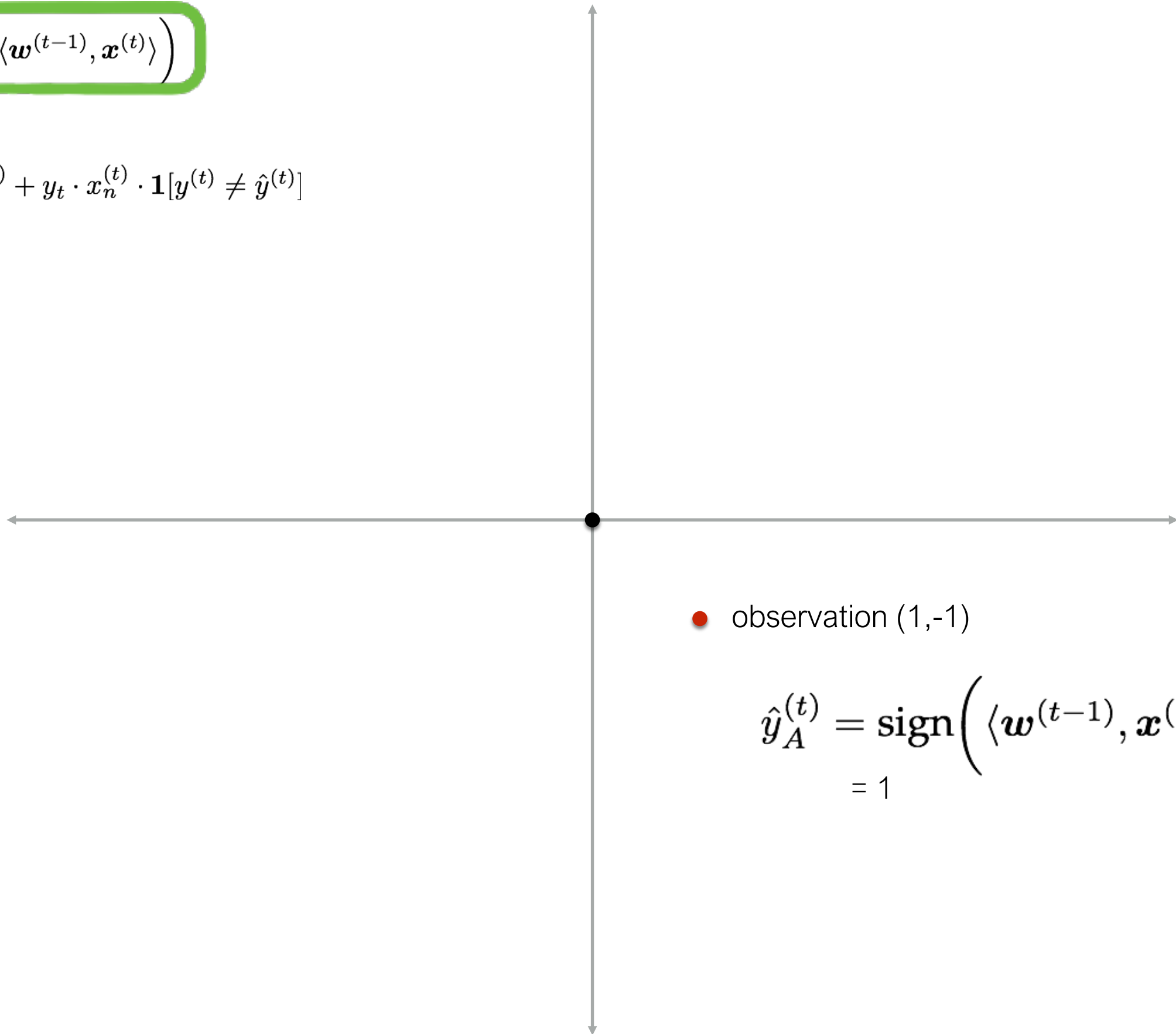


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

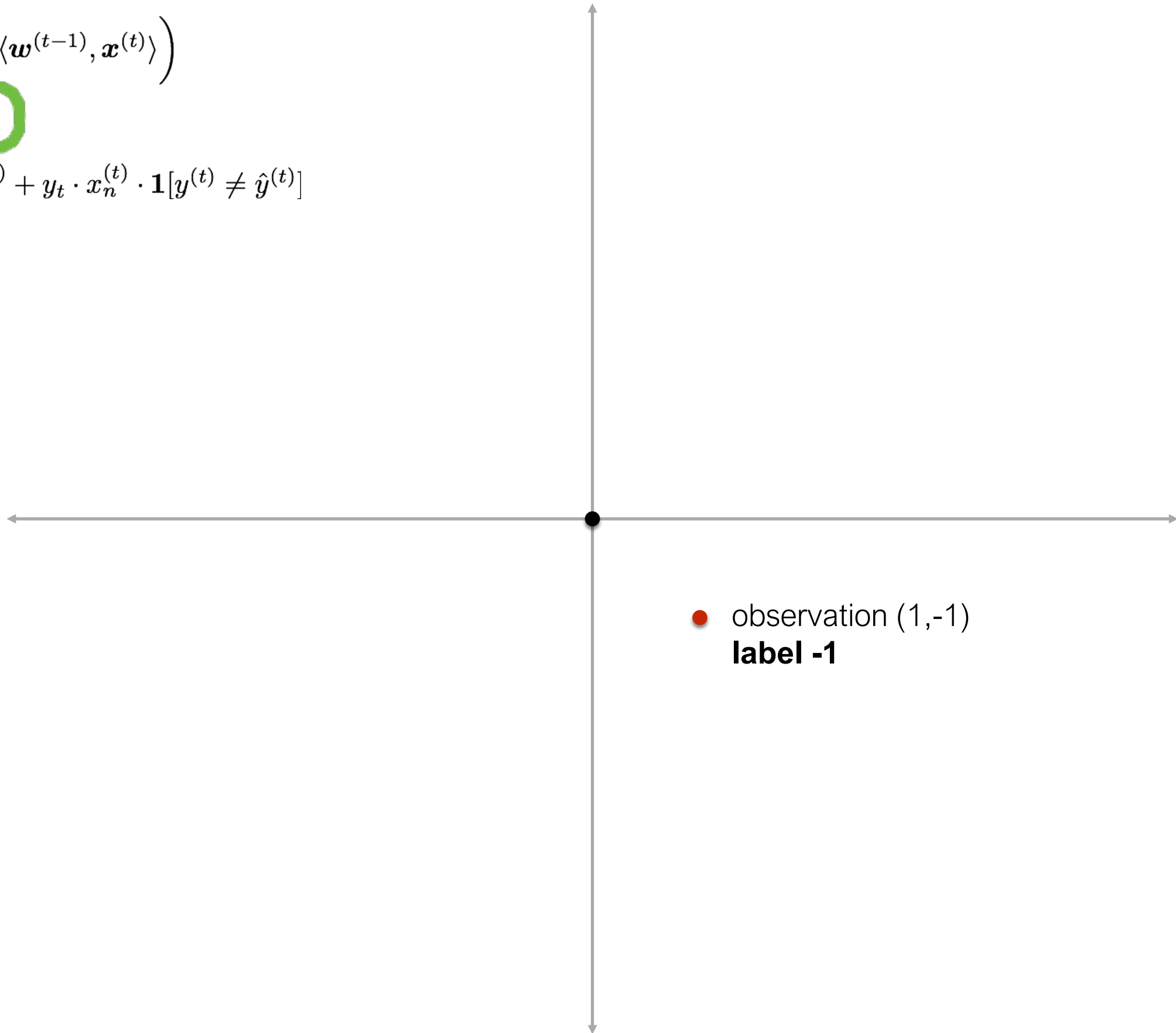


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$



RECEIVE($\mathbf{x}^{(t)}$)

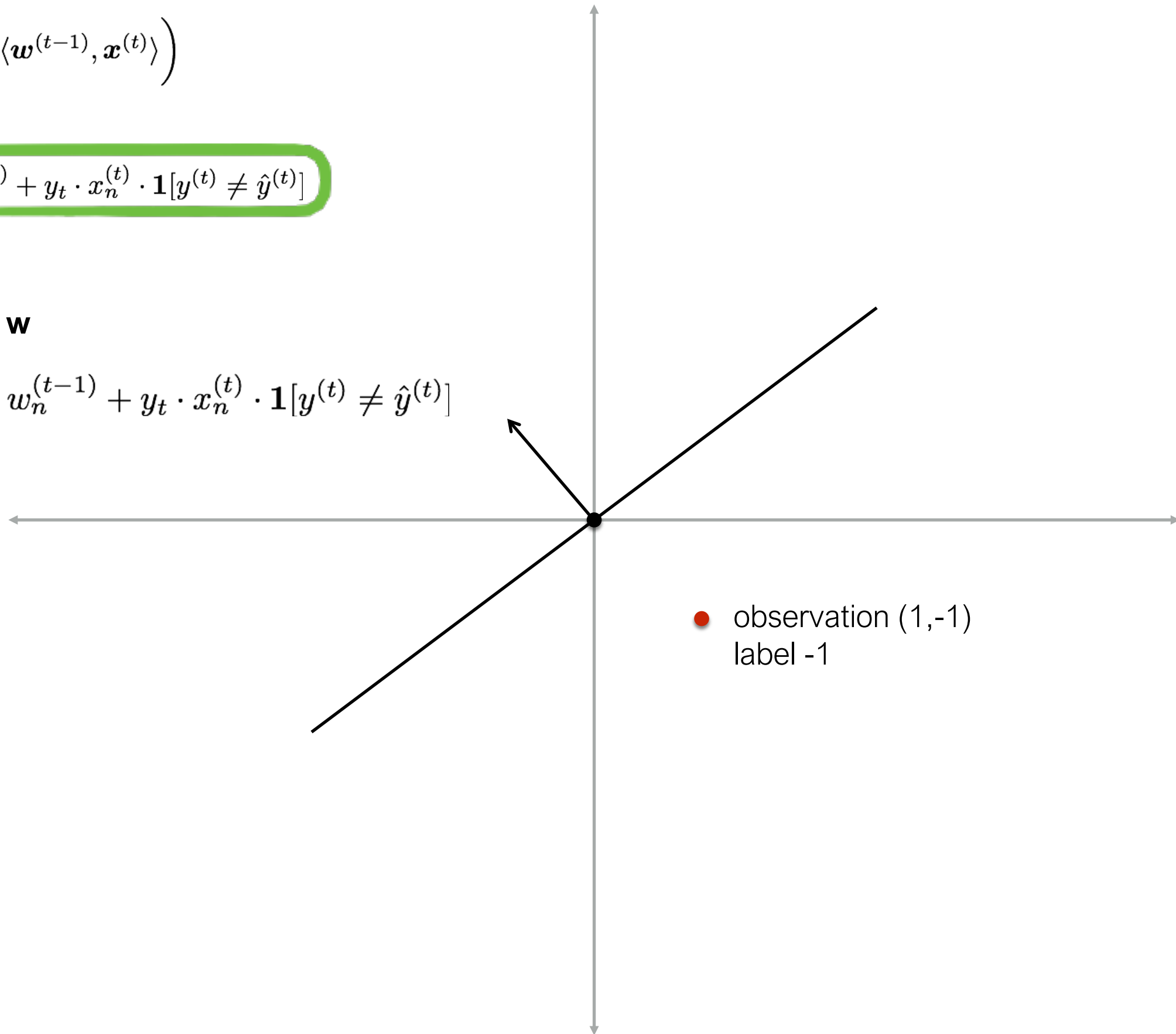
$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

update w

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$



RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

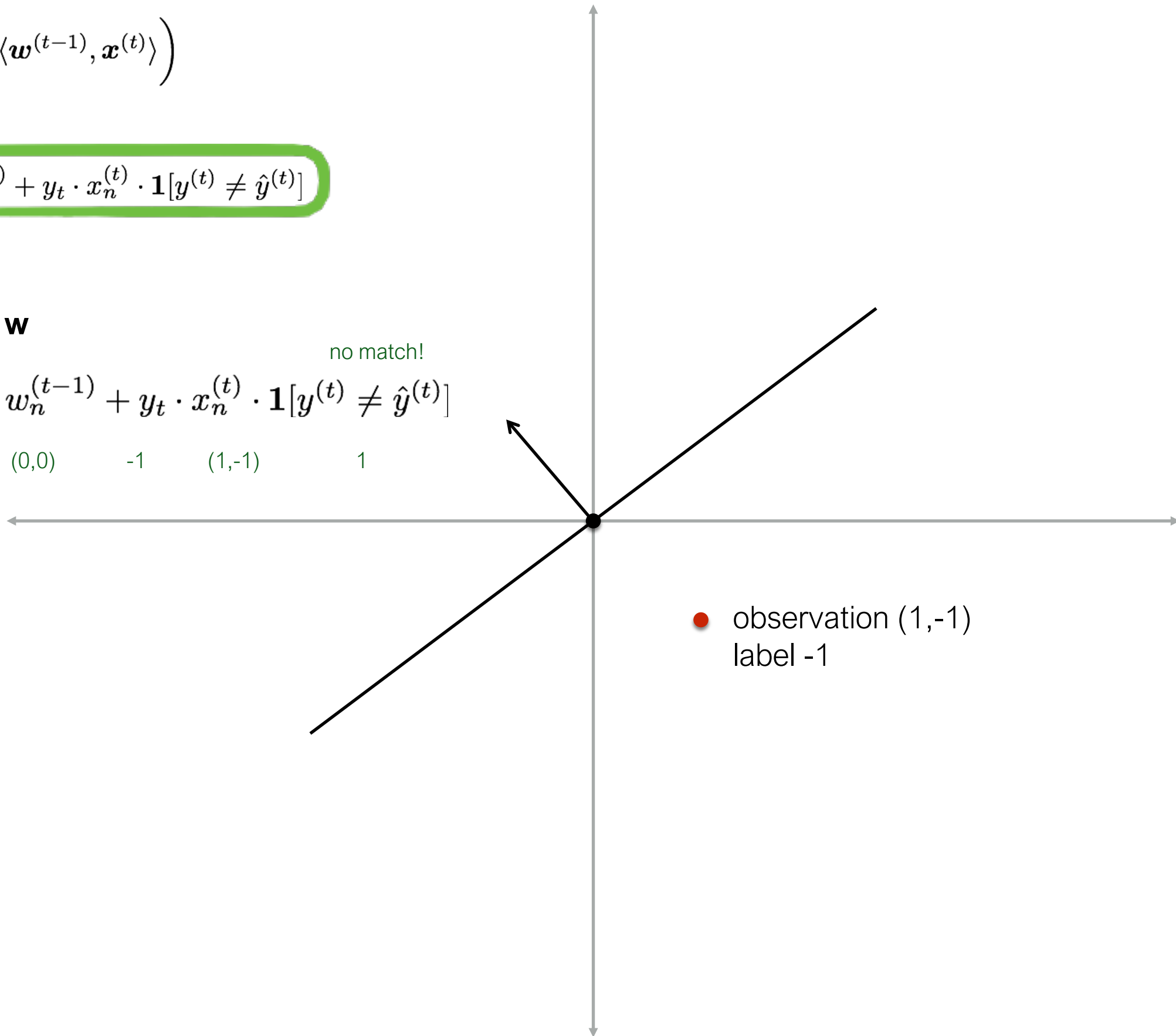
$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

update w

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

(-1,1) (0,0) -1 (1,-1) 1

no match!



RECEIVE($\mathbf{x}^{(t)}$)

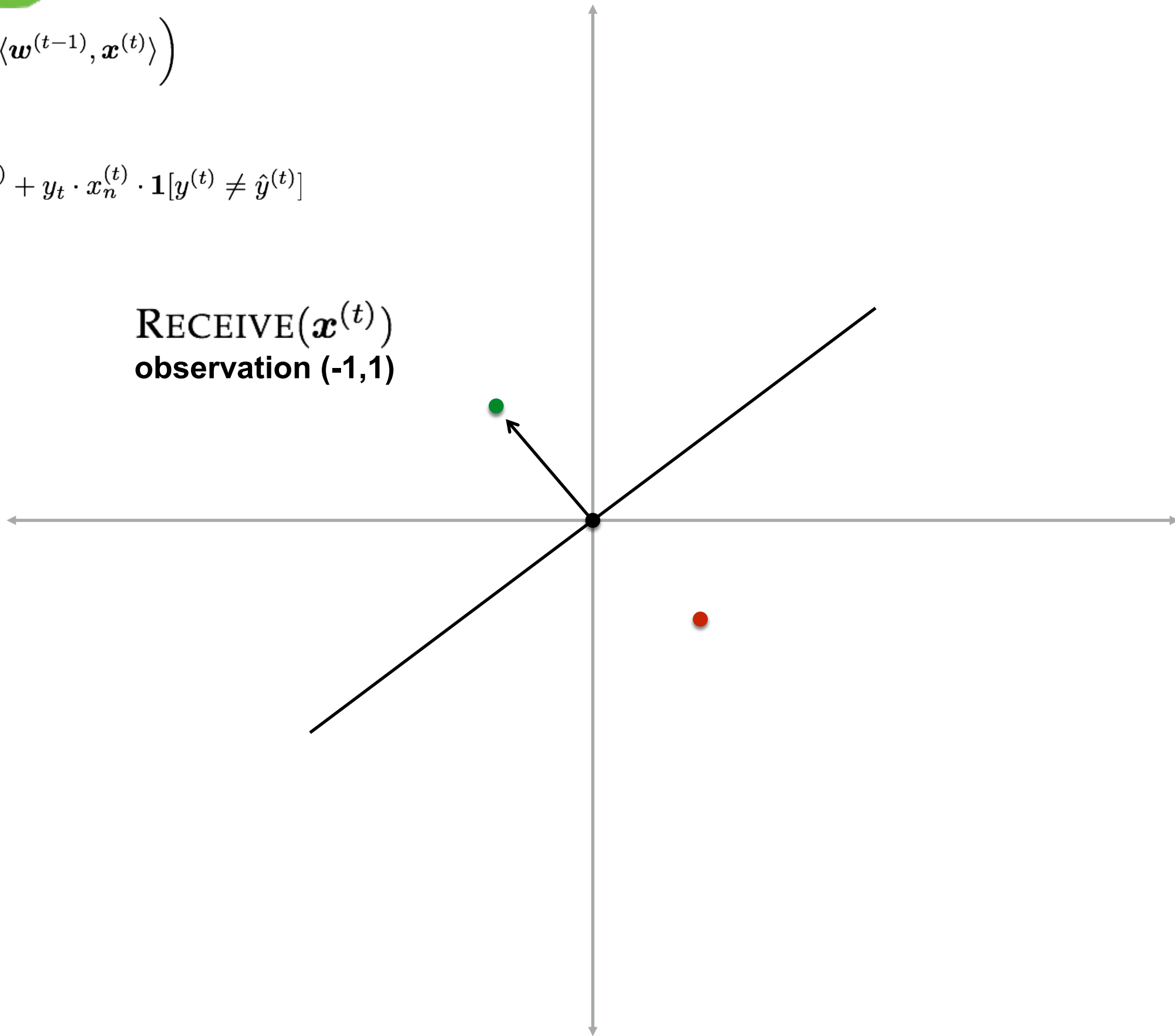
$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

(-1,1)

RECEIVE($\mathbf{x}^{(t)}$)
observation (-1,1)



RECEIVE($\mathbf{x}^{(t)}$)

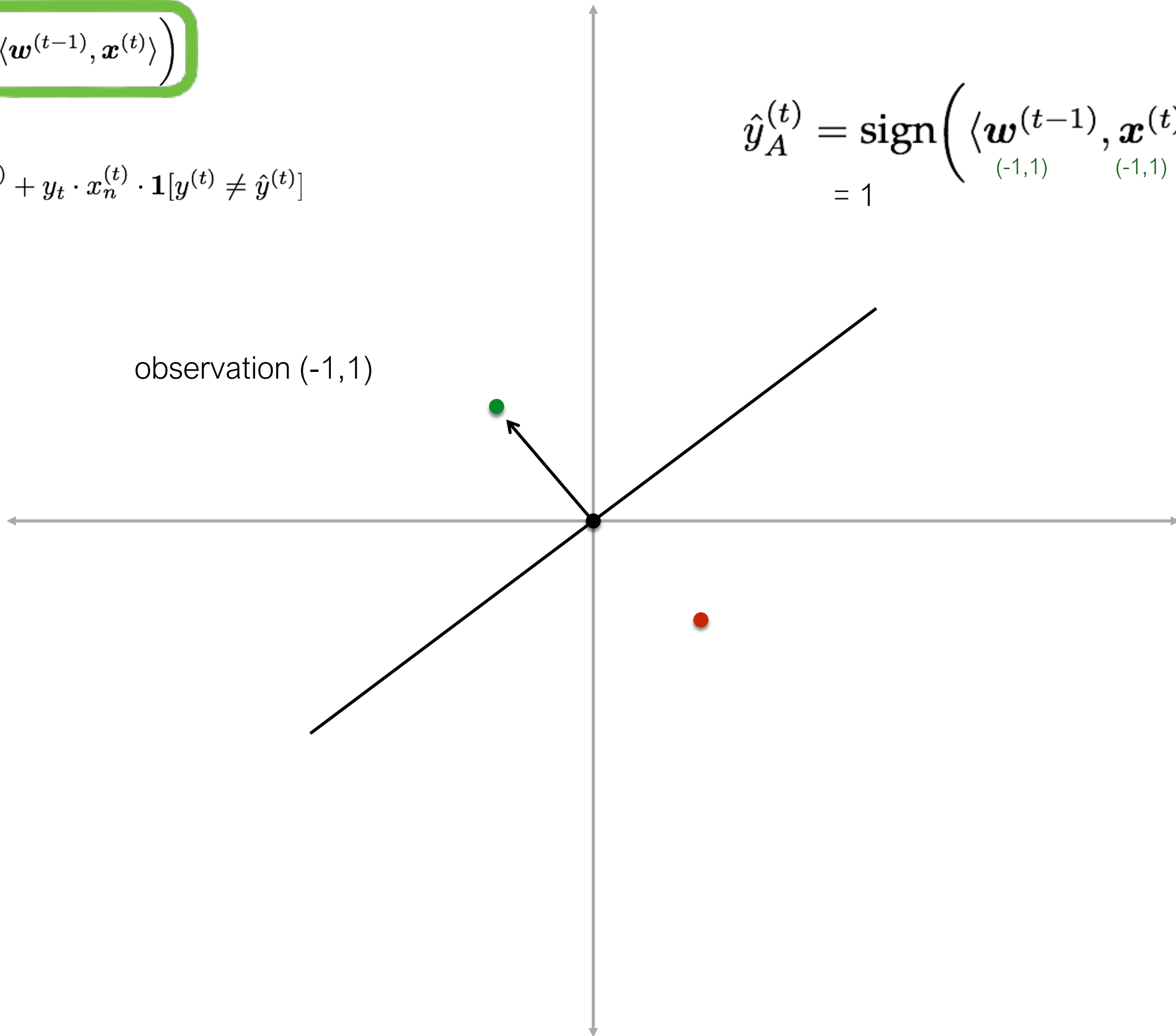
$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\underset{(-1,1)}{w_n^{(t)}} = w_n^{(t-1)} + y_t \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

$$\underset{=1}{\hat{y}_A^{(t)}} = \text{sign}\left(\underset{(-1,1)}{\langle \mathbf{w}^{(t-1)}, \underset{(-1,1)}{\mathbf{x}^{(t)} \rangle}\right)$$

observation (-1,1)



RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

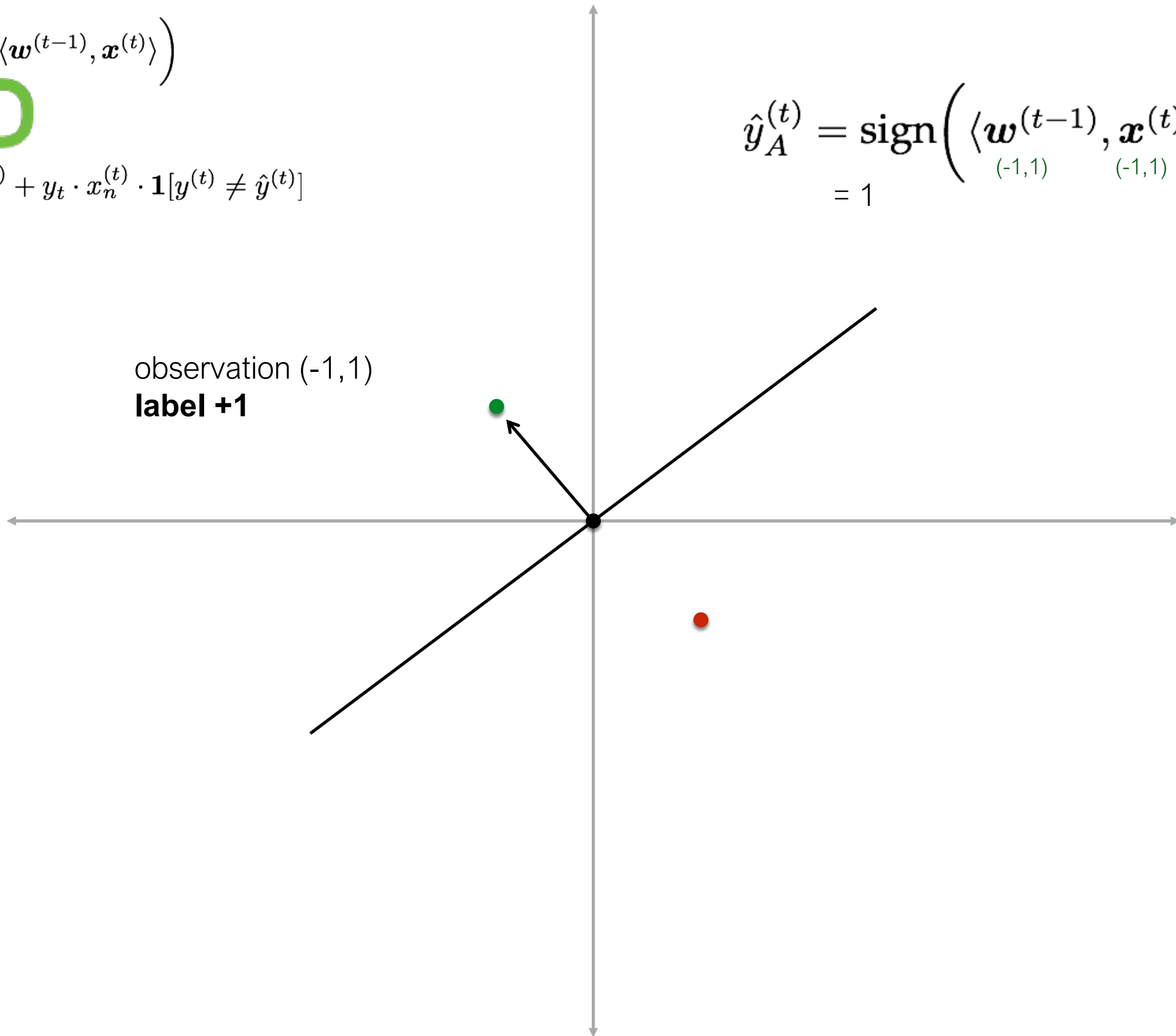
$$w_n^{(t)} = w_n^{(t-1)} + y_t \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

(-1,1)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \underset{(-1,1)}{\mathbf{w}^{(t-1)}}, \underset{(-1,1)}{\mathbf{x}^{(t)}} \rangle\right)$$

= 1

observation (-1,1)
label +1



RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

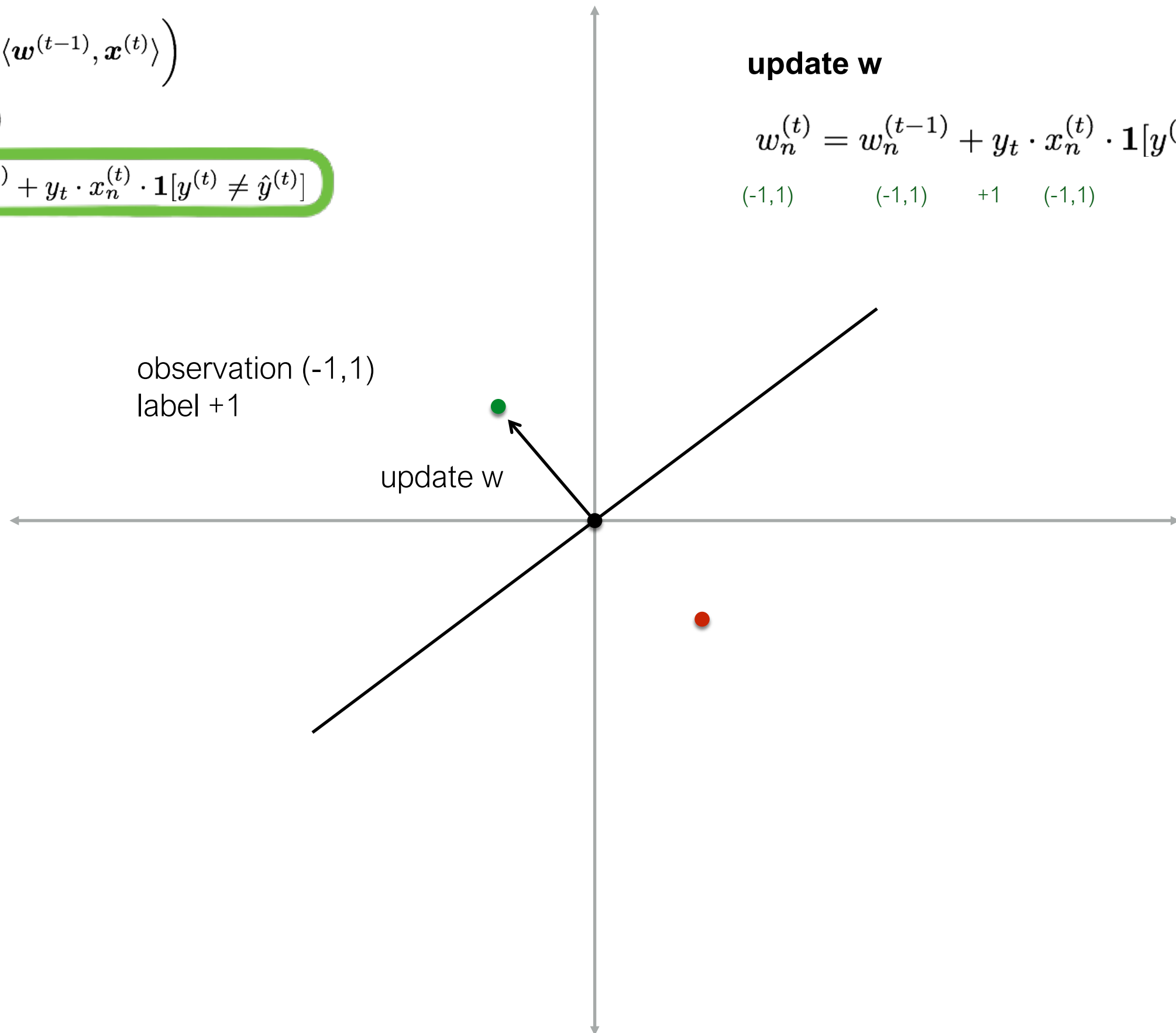
$$w_n^{(t)} = w_n^{(t-1)} + y_t \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

update w

$$w_n^{(t)} = w_n^{(t-1)} + y_t \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

match!

(-1,1)	(-1,1)	+1	(-1,1)	0
--------	--------	----	--------	---

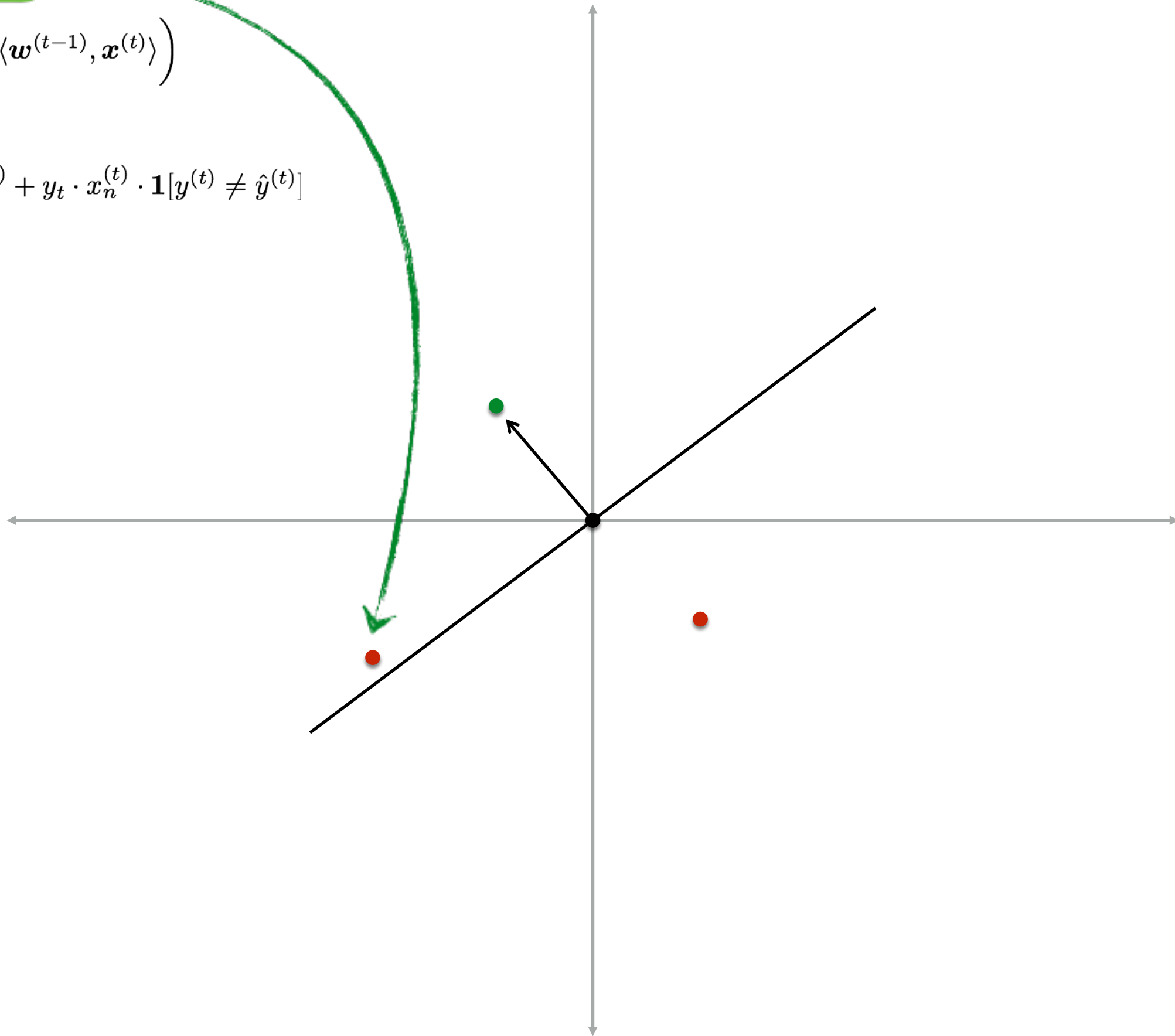


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

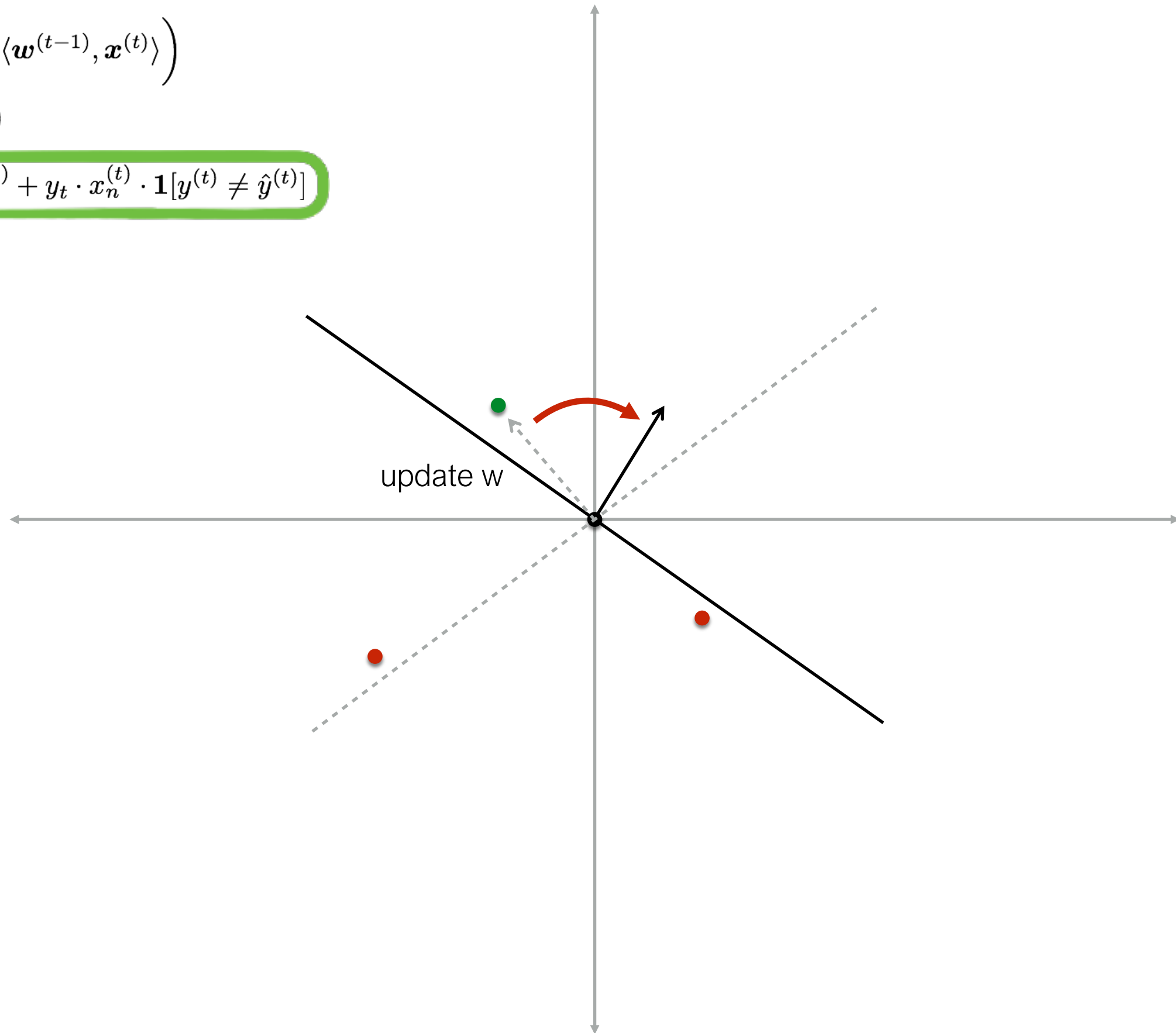


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$w_n^{(t)} = w_n^{(t-1)} + y_t \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

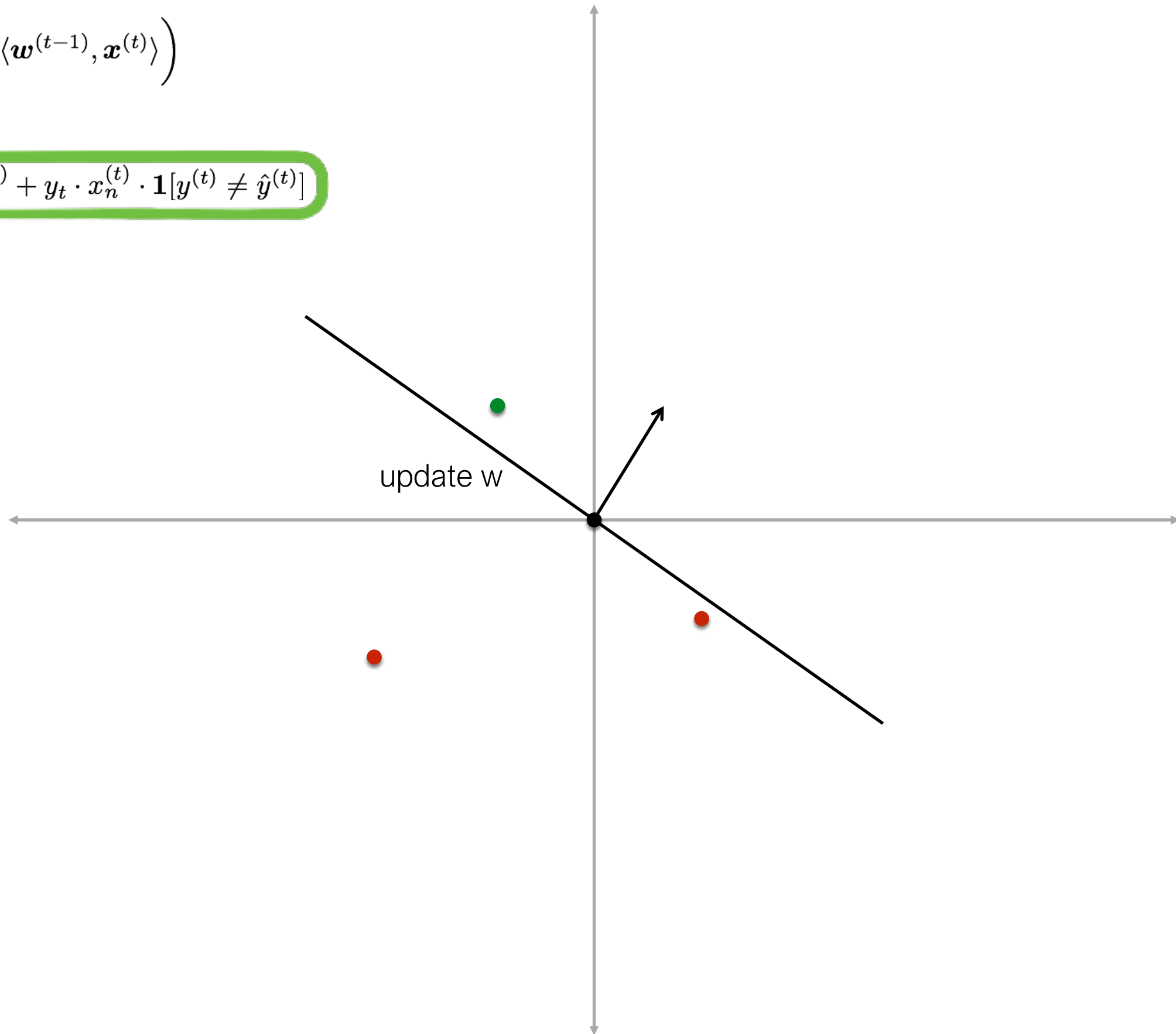


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$w_n^{(t)} = w_n^{(t-1)} + y_t \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

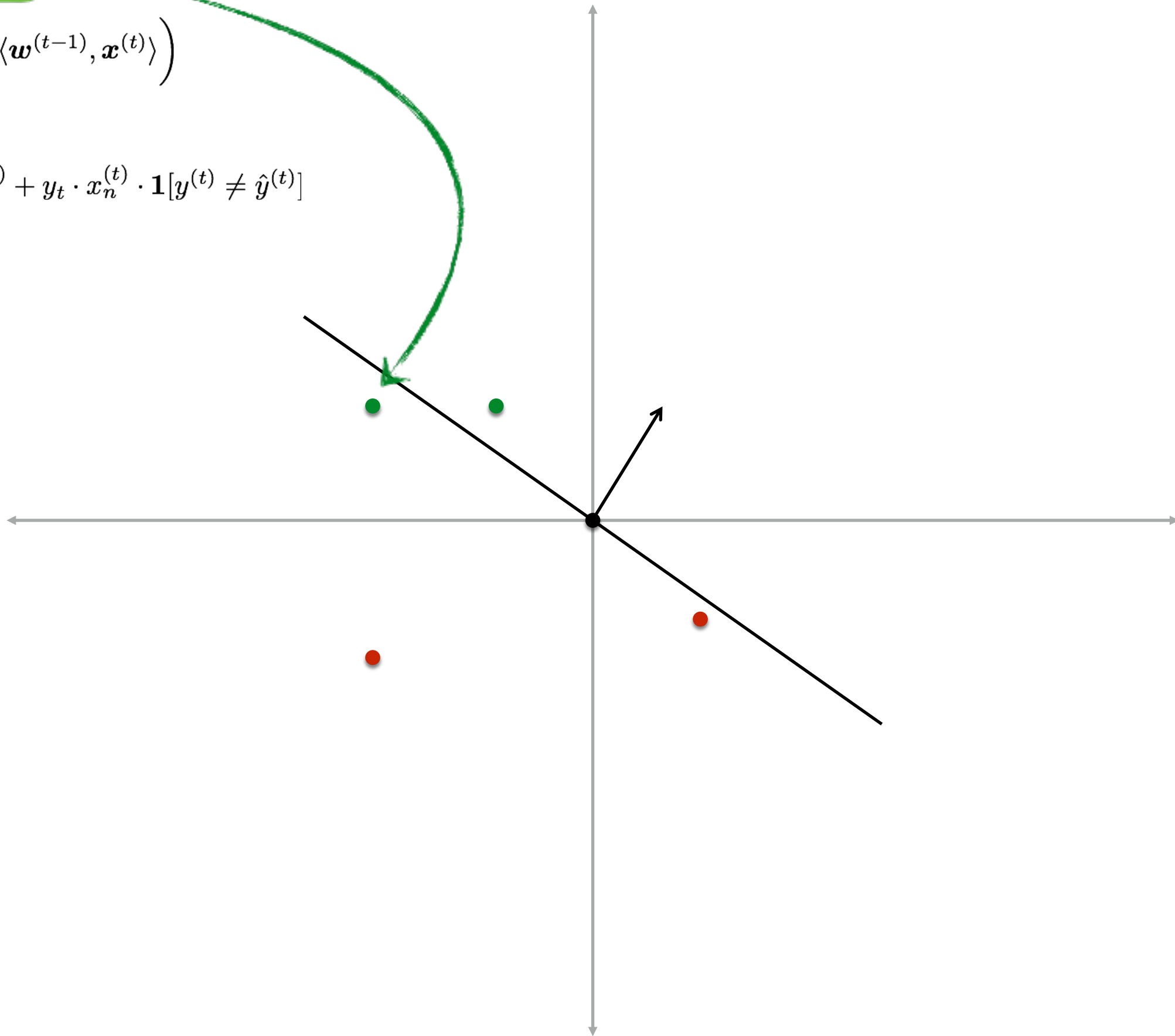


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

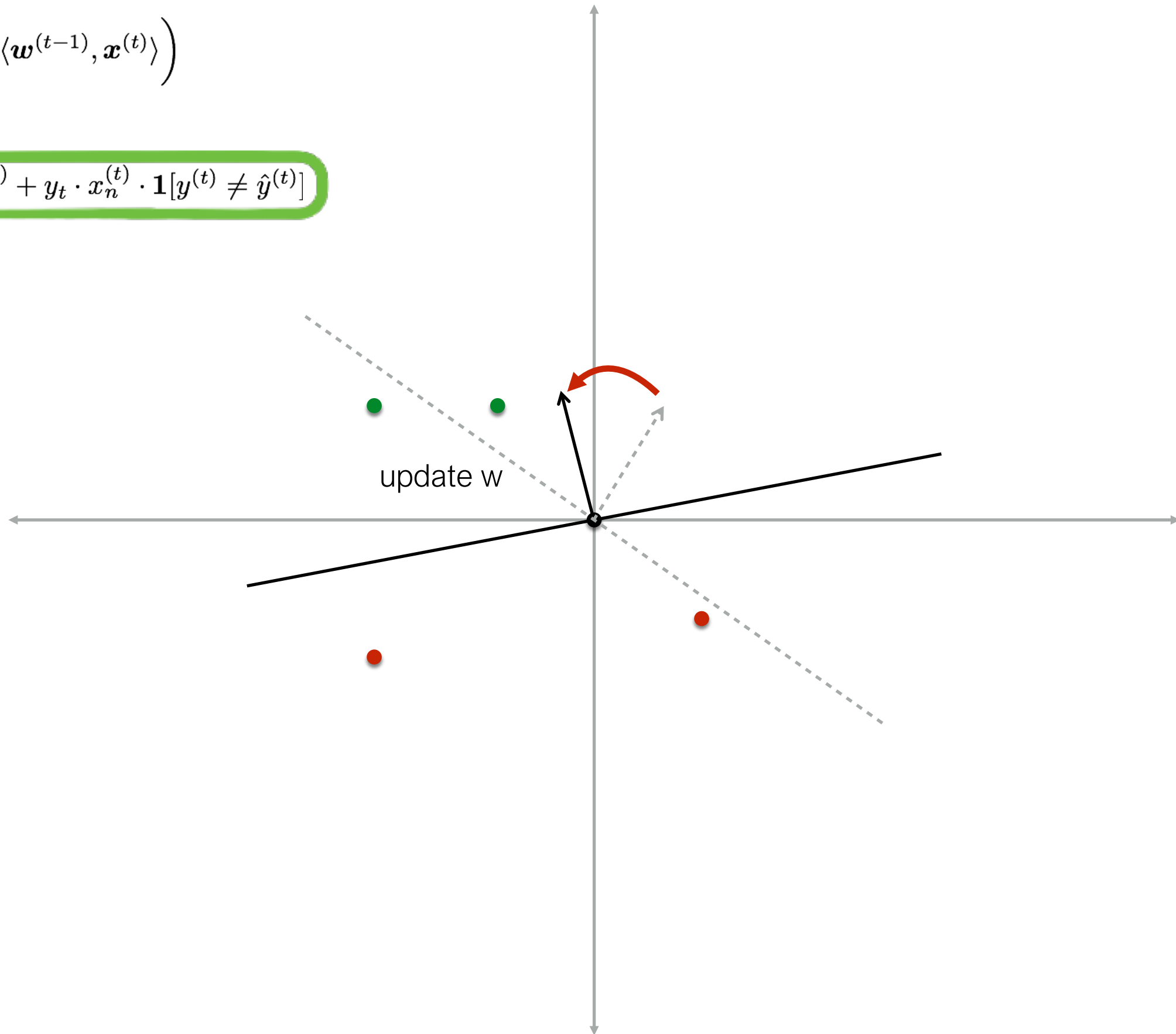


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

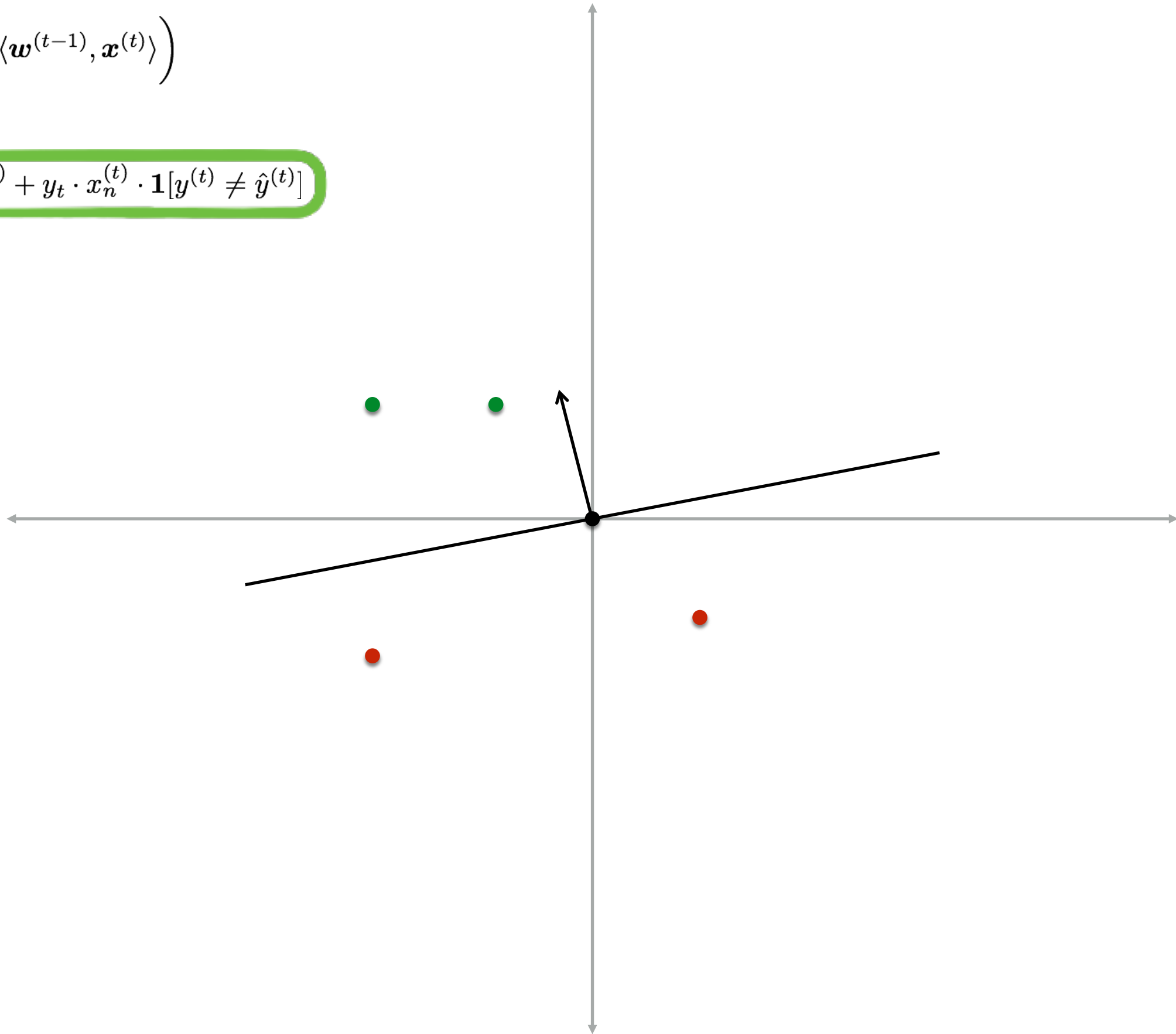


RECEIVE($\mathbf{x}^{(t)}$)

$$\hat{y}_A^{(t)} = \text{sign}\left(\langle \mathbf{w}^{(t-1)}, \mathbf{x}^{(t)} \rangle\right)$$

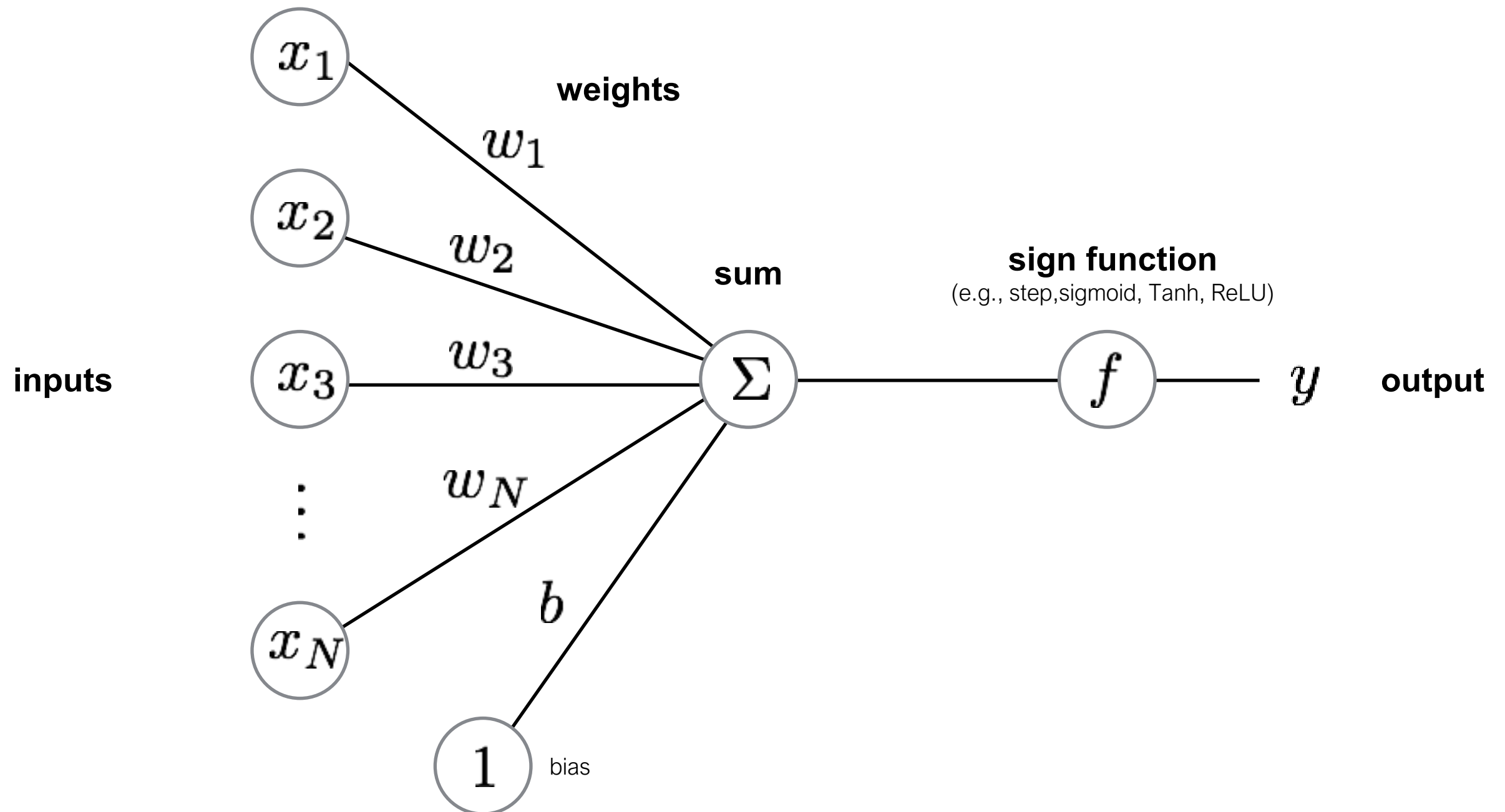
RECEIVE(y^t)

$$\mathbf{w}_n^{(t)} = \mathbf{w}_n^{(t-1)} + y_t \cdot \mathbf{x}_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$$

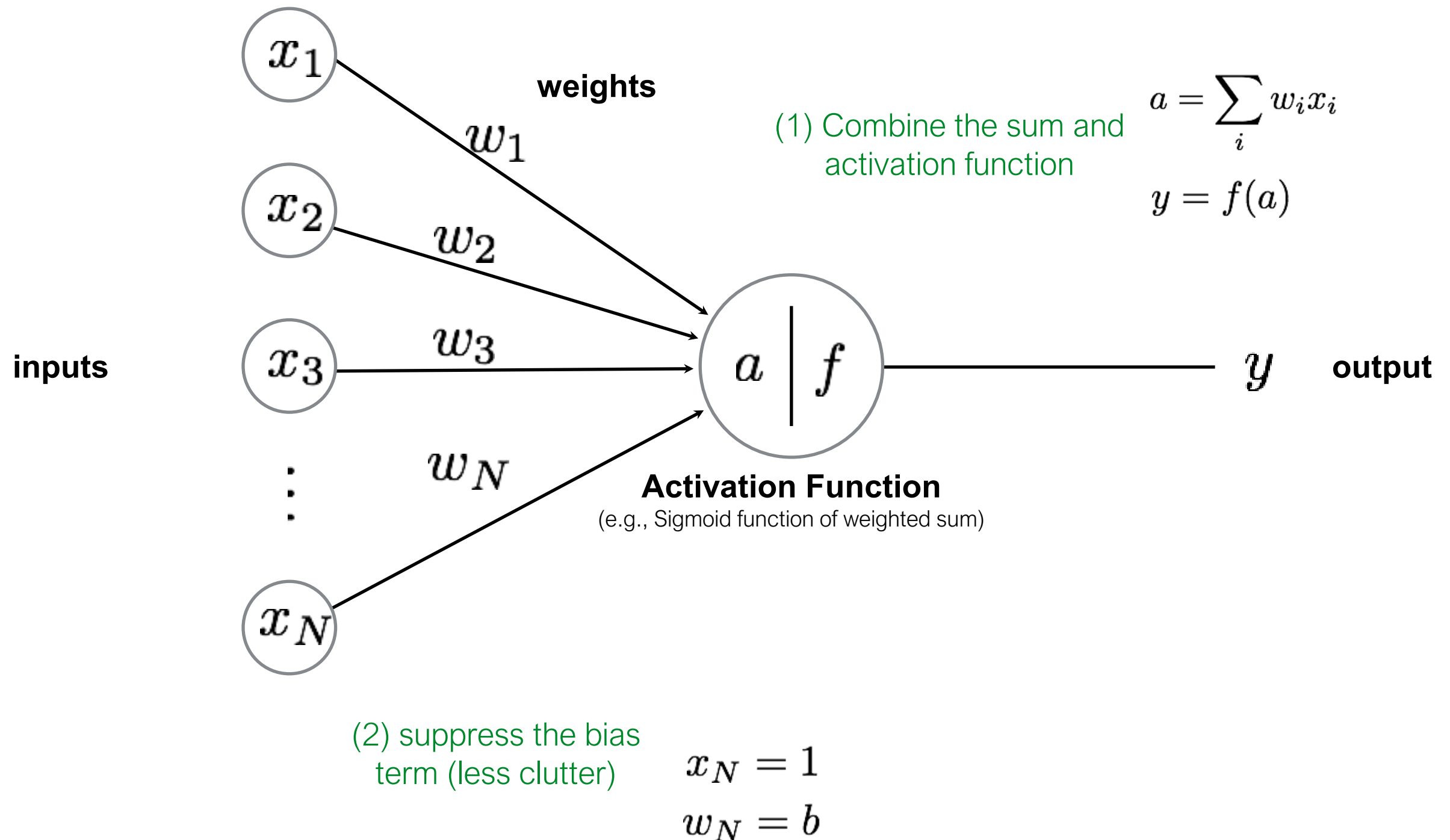


repeat ...

The Perceptron



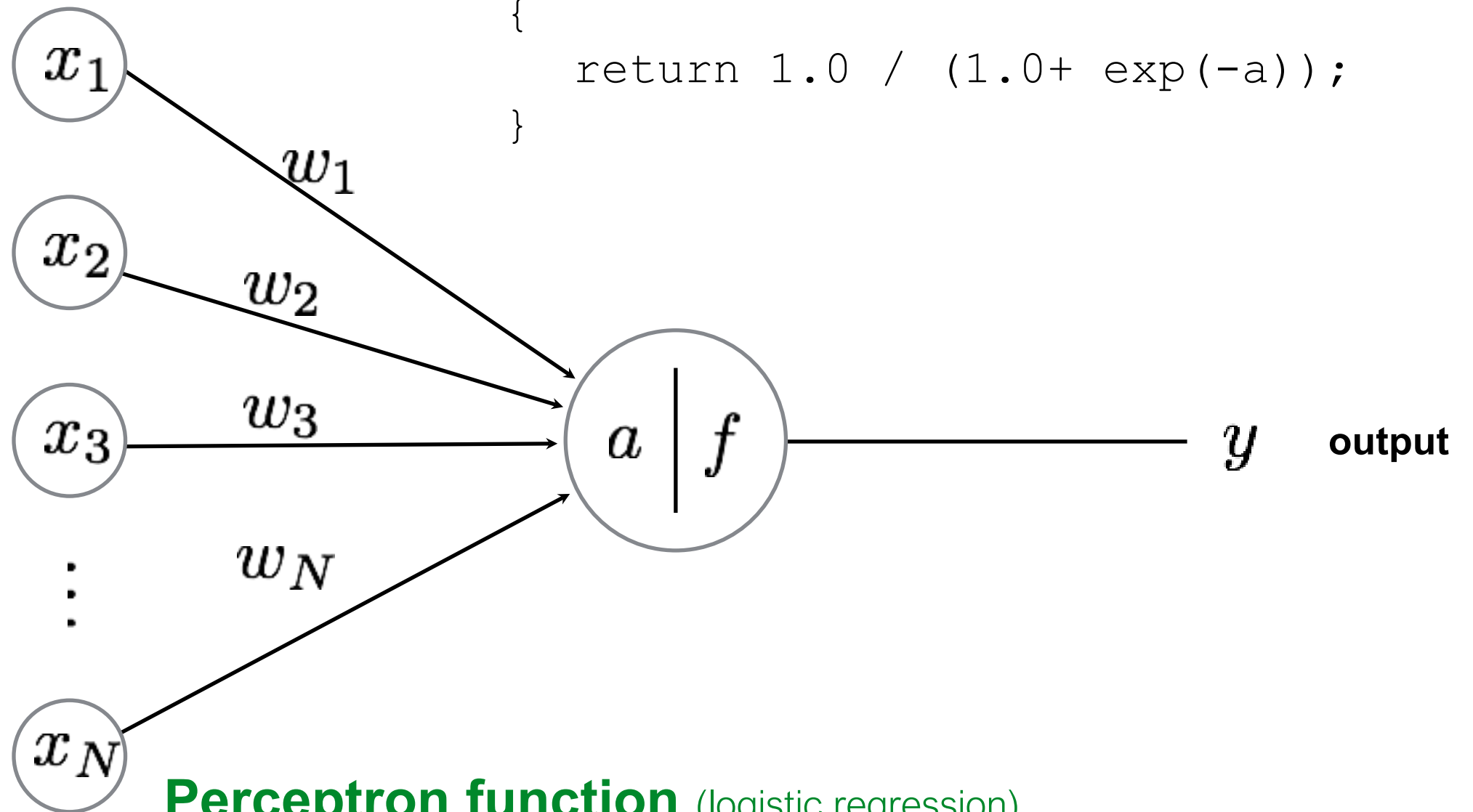
Another way to draw it...



Programming the 'forward pass'

Activation function (sigmoid, logistic function)

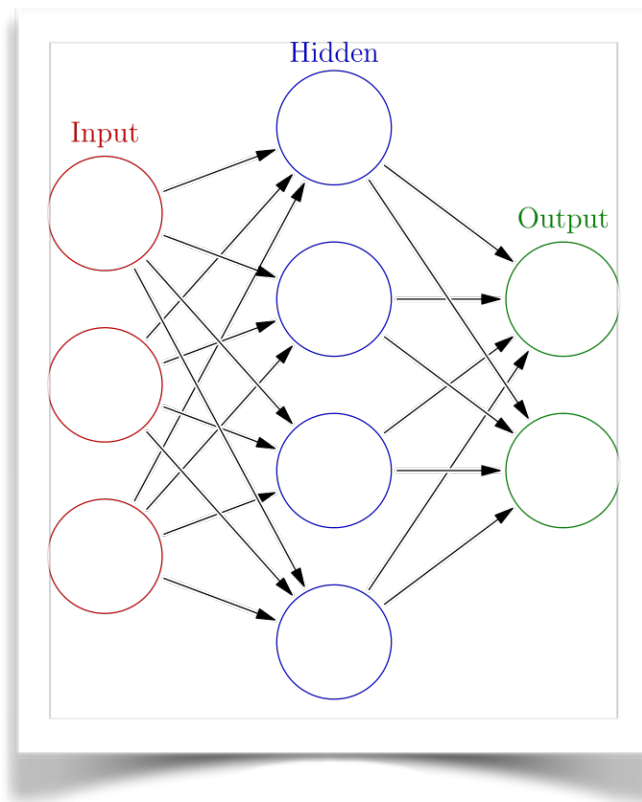
```
float f(float a)
{
    return 1.0 / (1.0 + exp(-a));
}
```



Perceptron function (logistic regression)

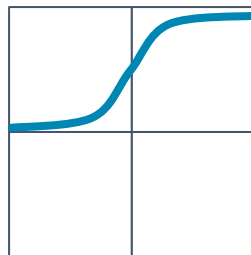
```
float perceptron(vector<float> x, vector<float> w)
{
    float a = dot(x, w);
    return f(a);
}
```

Minor detail: other activation functions



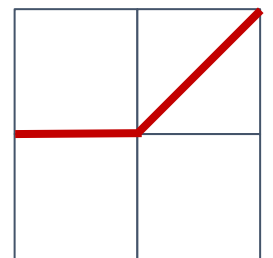
Sigmoid

$$x' = 1/(1 + e^{-x})$$



ReLU

$$x' = \max(0, x)$$



Early neural networks exclusively used sigmoids or 0-1 thresholding, but ReLU was a game-changer (easier to code and gradient signal doesn't disappear)

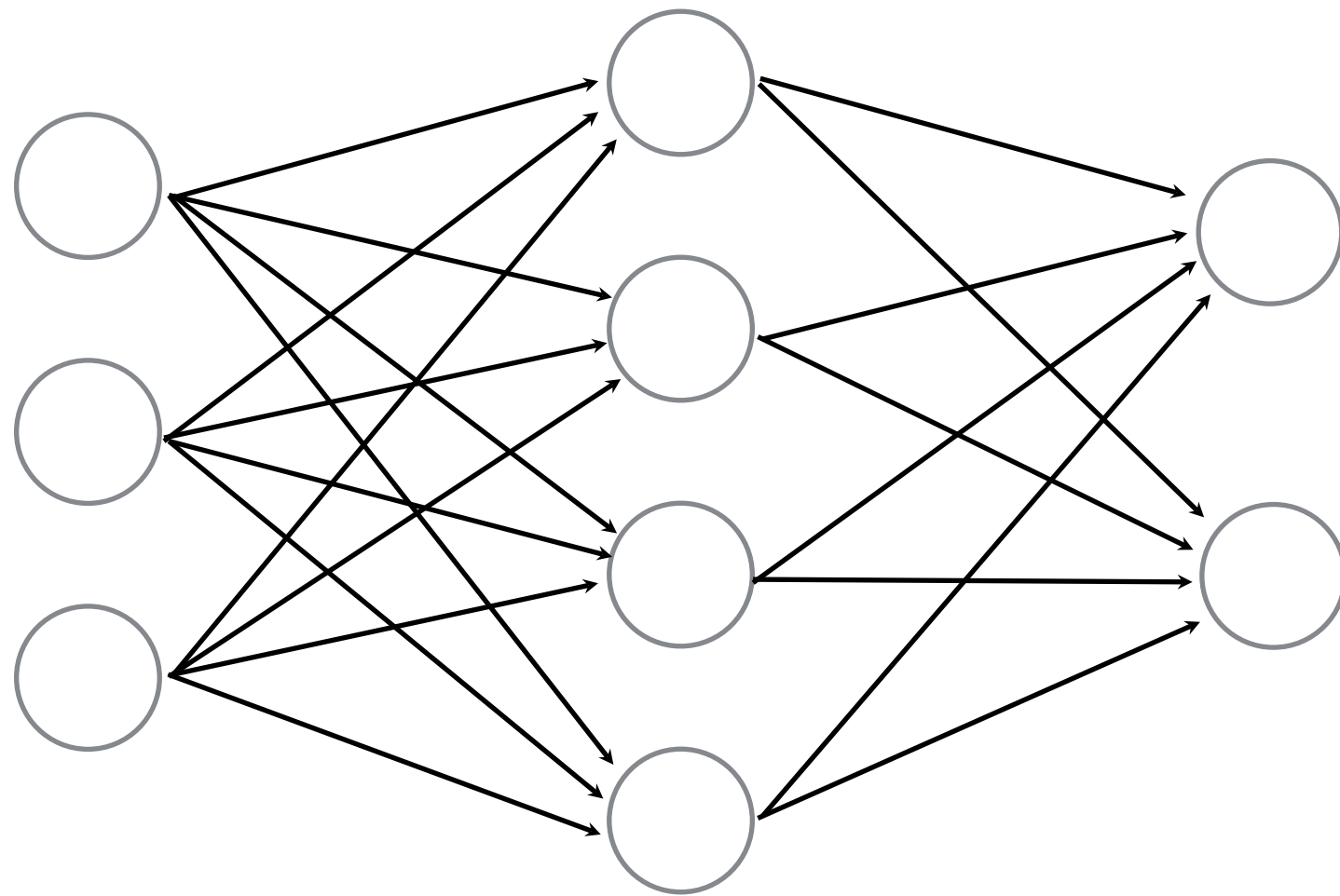
Neural networks

Connect a bunch of perceptrons together ...

Connect a bunch of perceptrons together ...

Neural Network

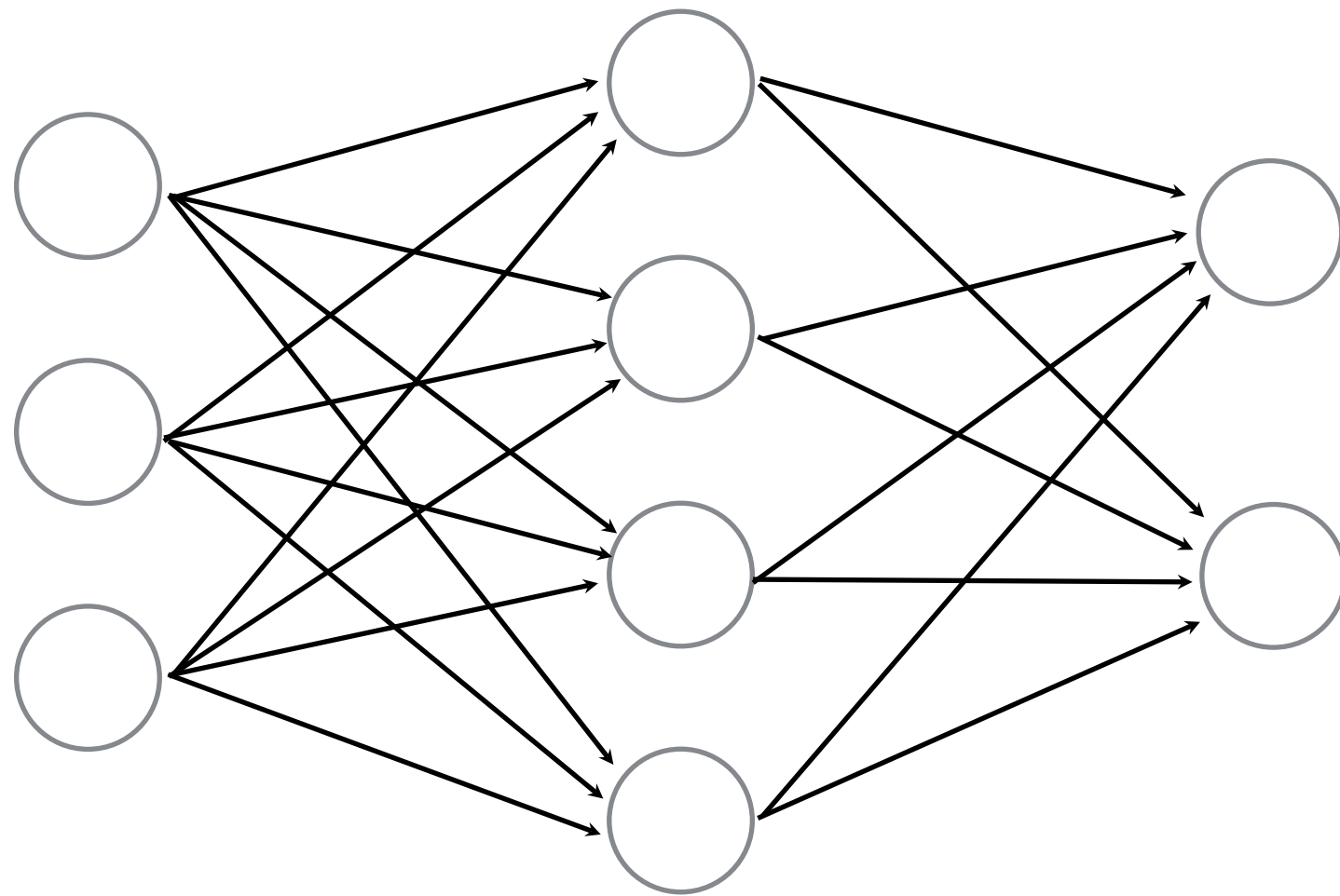
a collection of connected perceptrons



Connect a bunch of perceptrons together ...

Neural Network

a collection of connected perceptrons

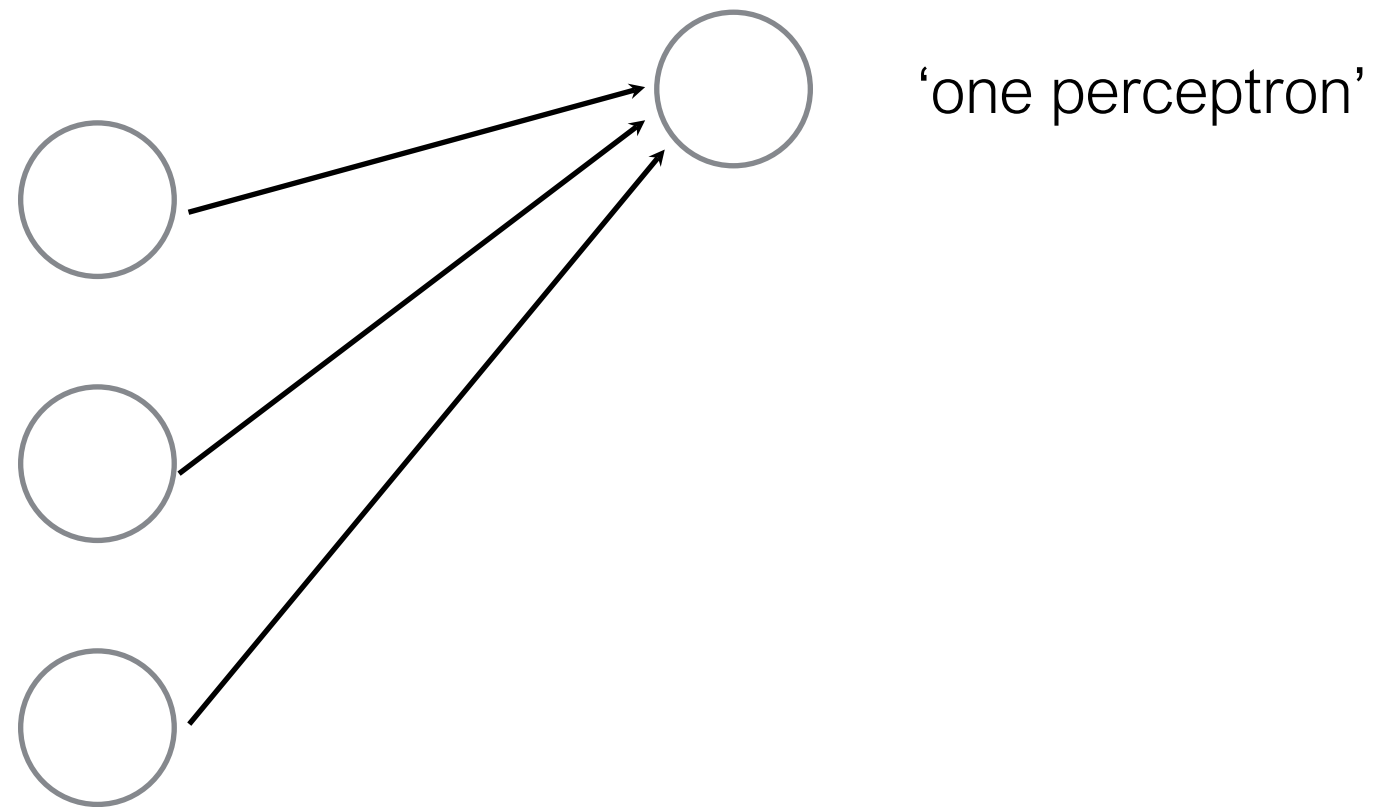


How many perceptrons in this neural network?

Connect a bunch of perceptrons together ...

Neural Network

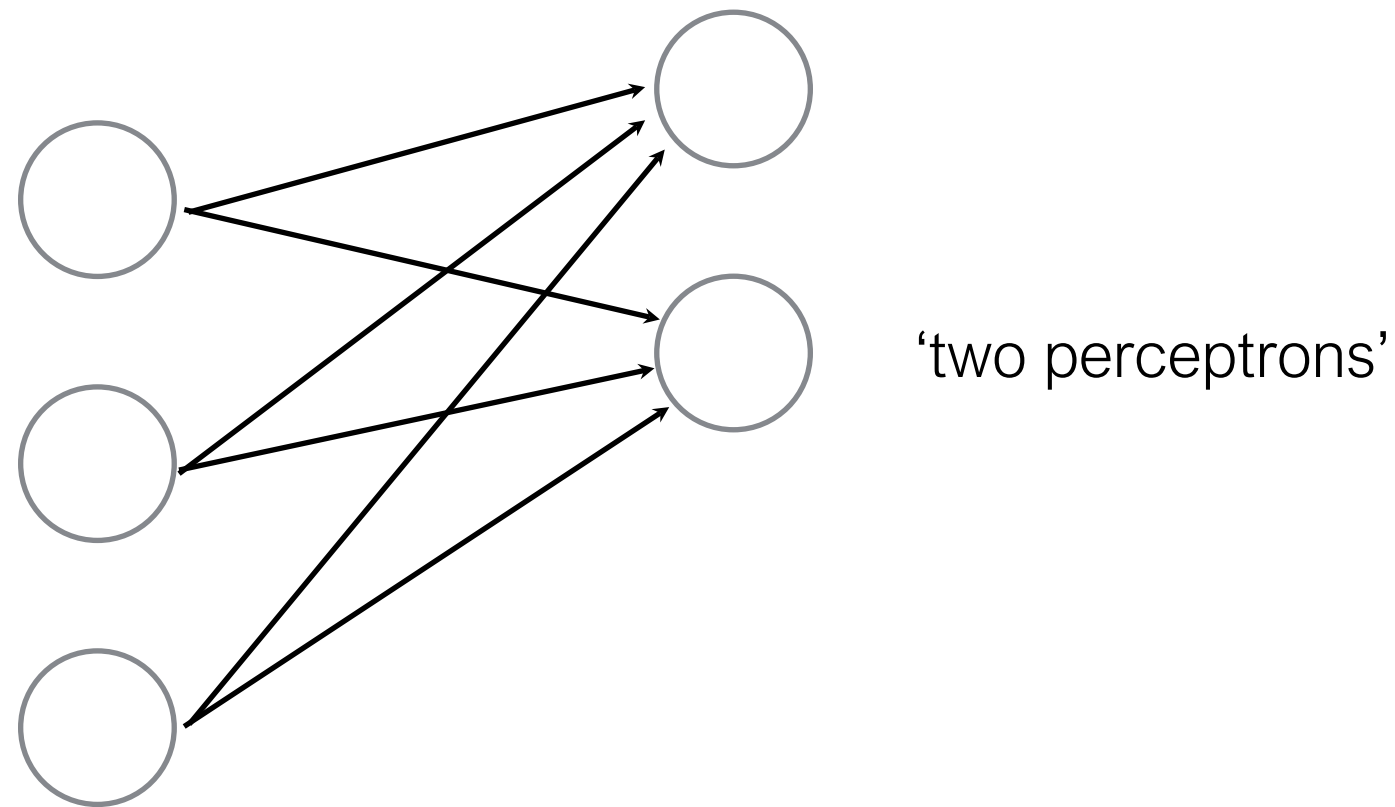
a collection of connected perceptrons



Connect a bunch of perceptrons together ...

Neural Network

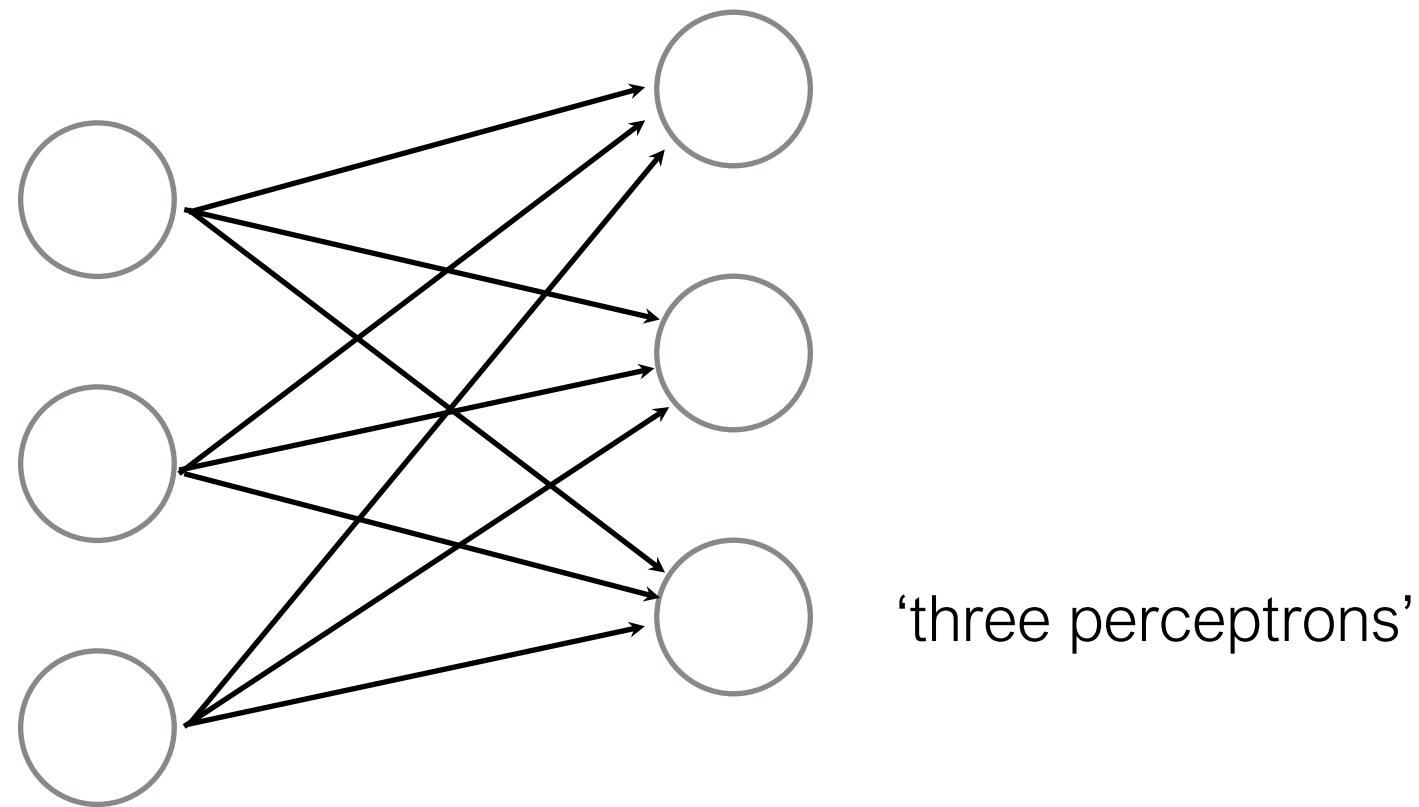
a collection of connected perceptrons



Connect a bunch of perceptrons together ...

Neural Network

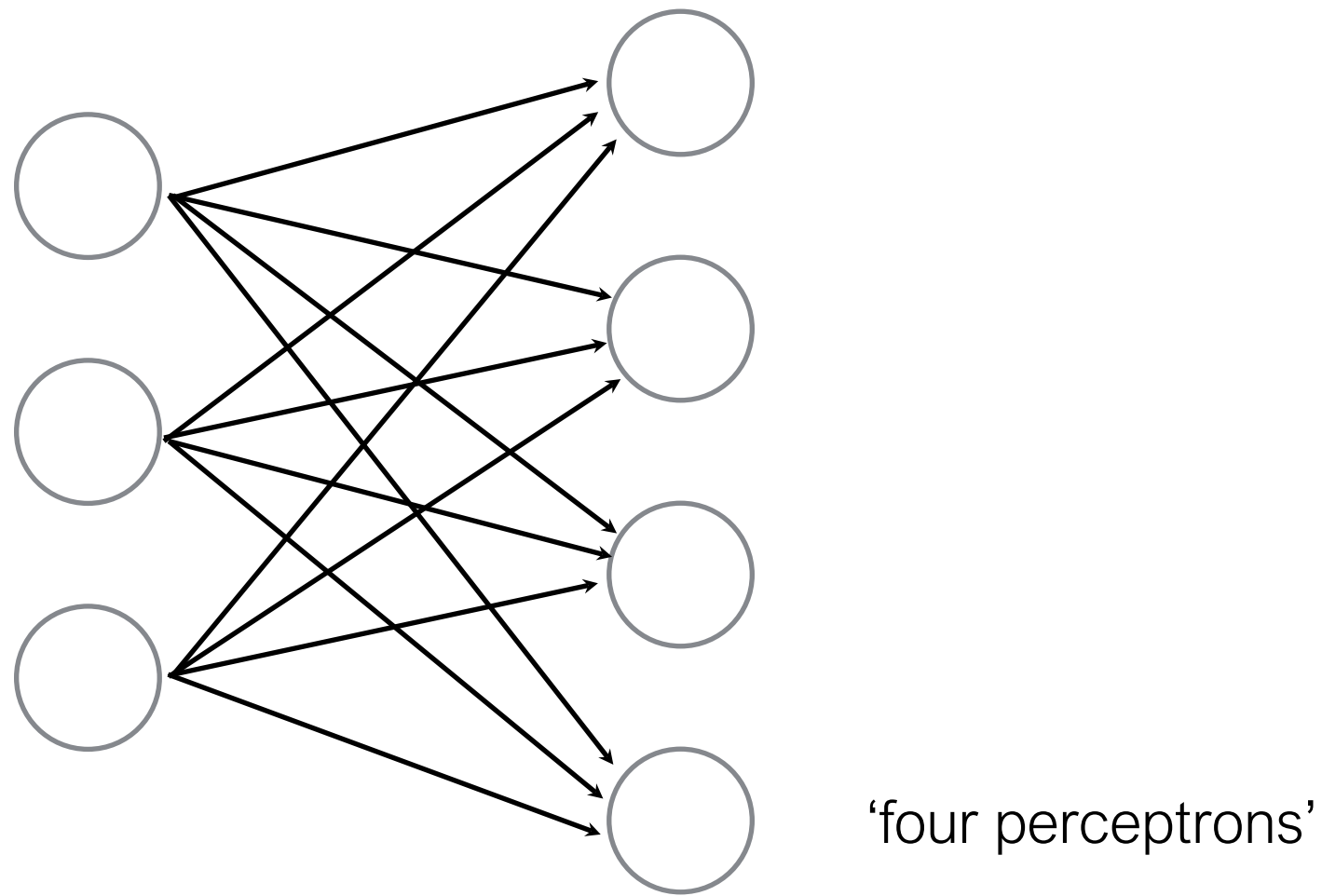
a collection of connected perceptrons



Connect a bunch of perceptrons together ...

Neural Network

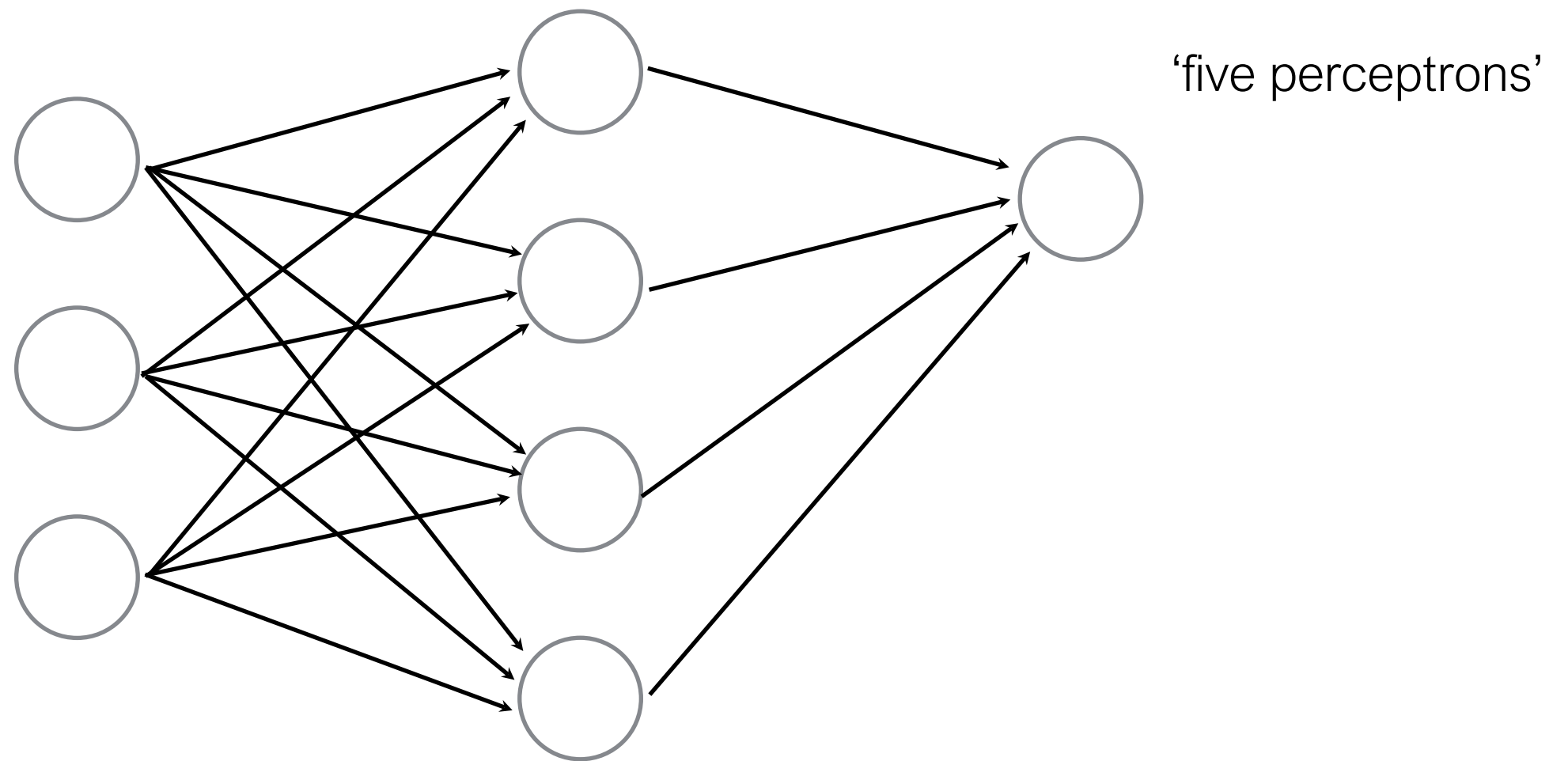
a collection of connected perceptrons



Connect a bunch of perceptrons together ...

Neural Network

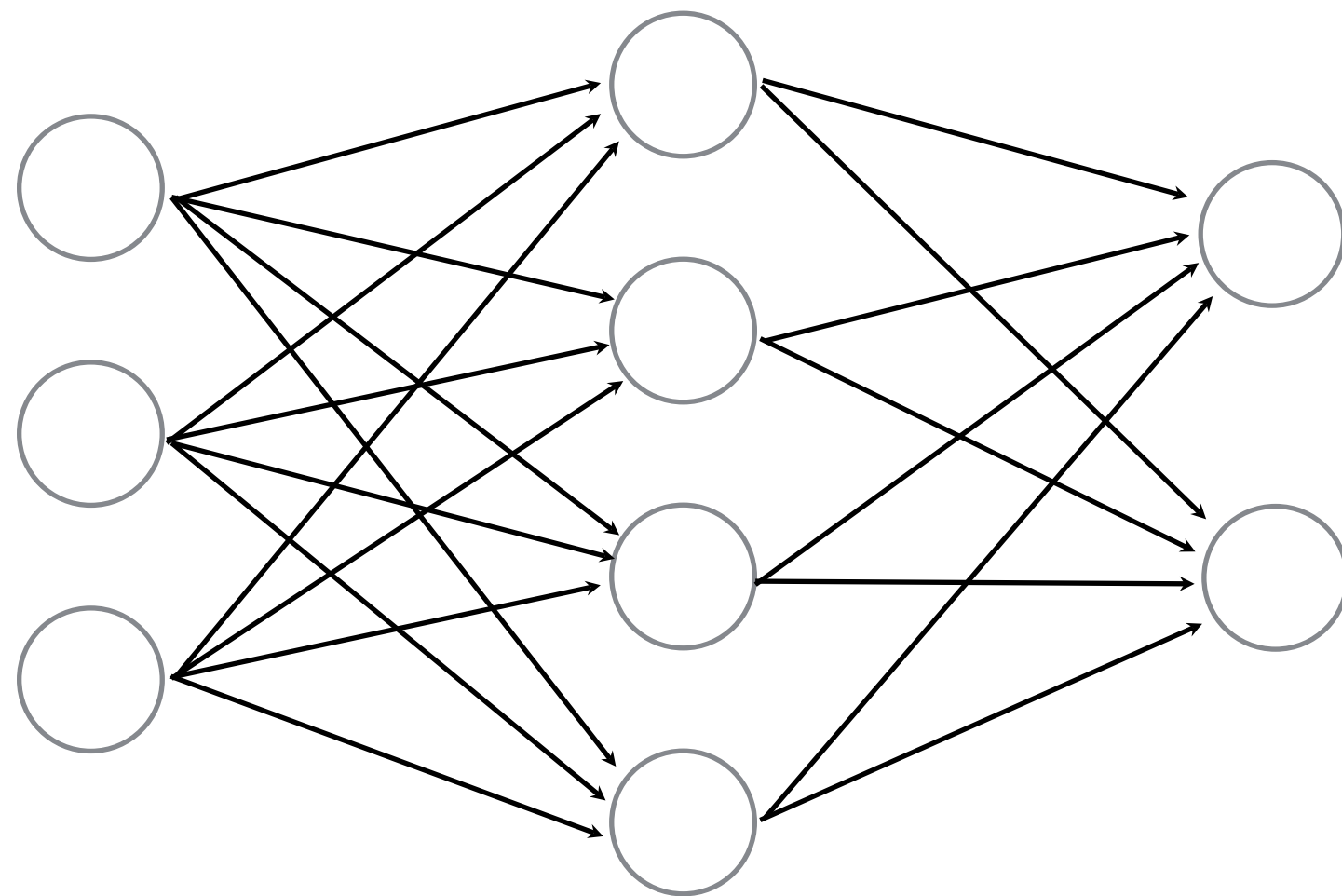
a collection of connected perceptrons



Connect a bunch of perceptrons together ...

Neural Network

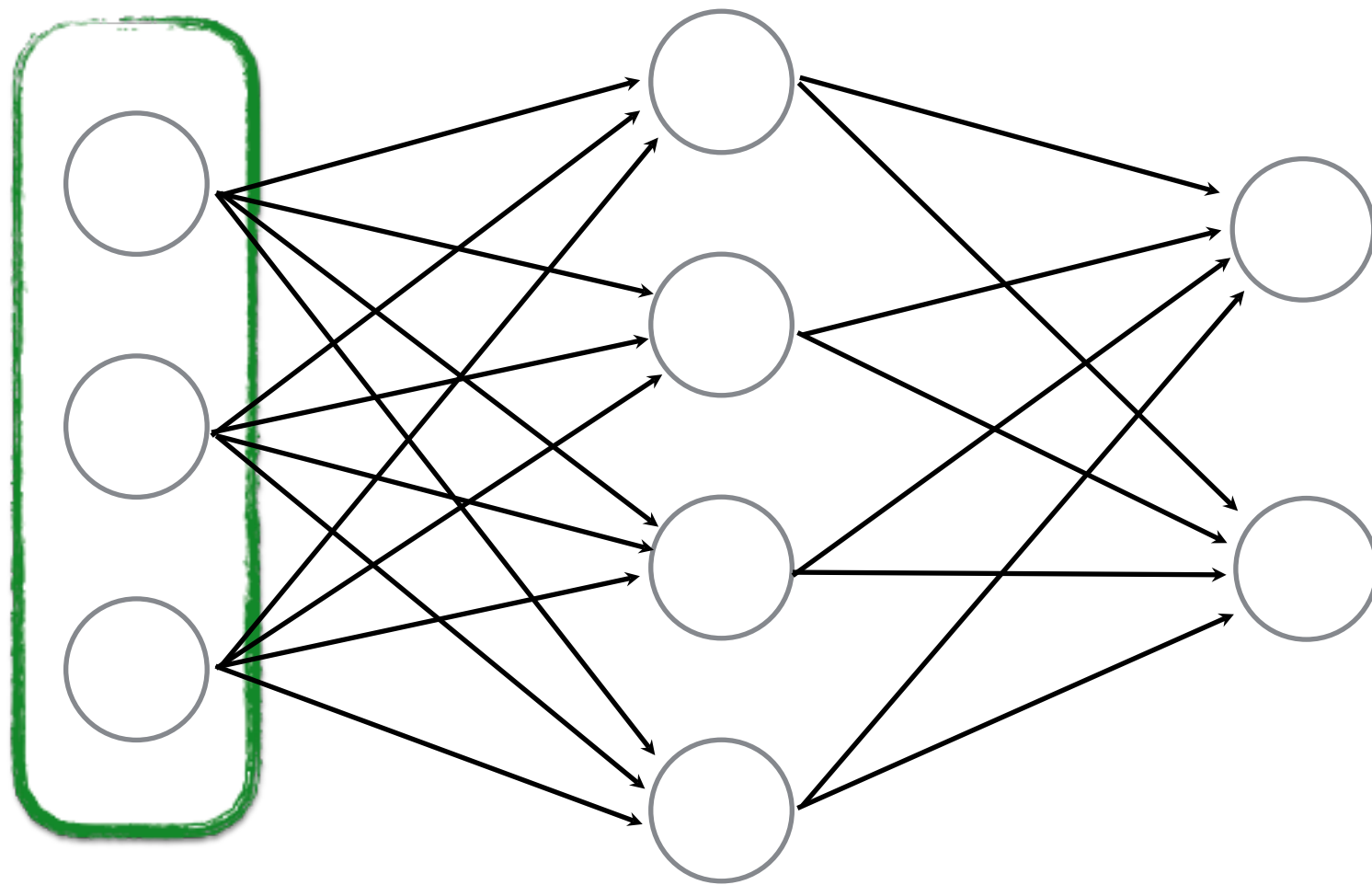
a collection of connected perceptrons



‘six perceptrons’

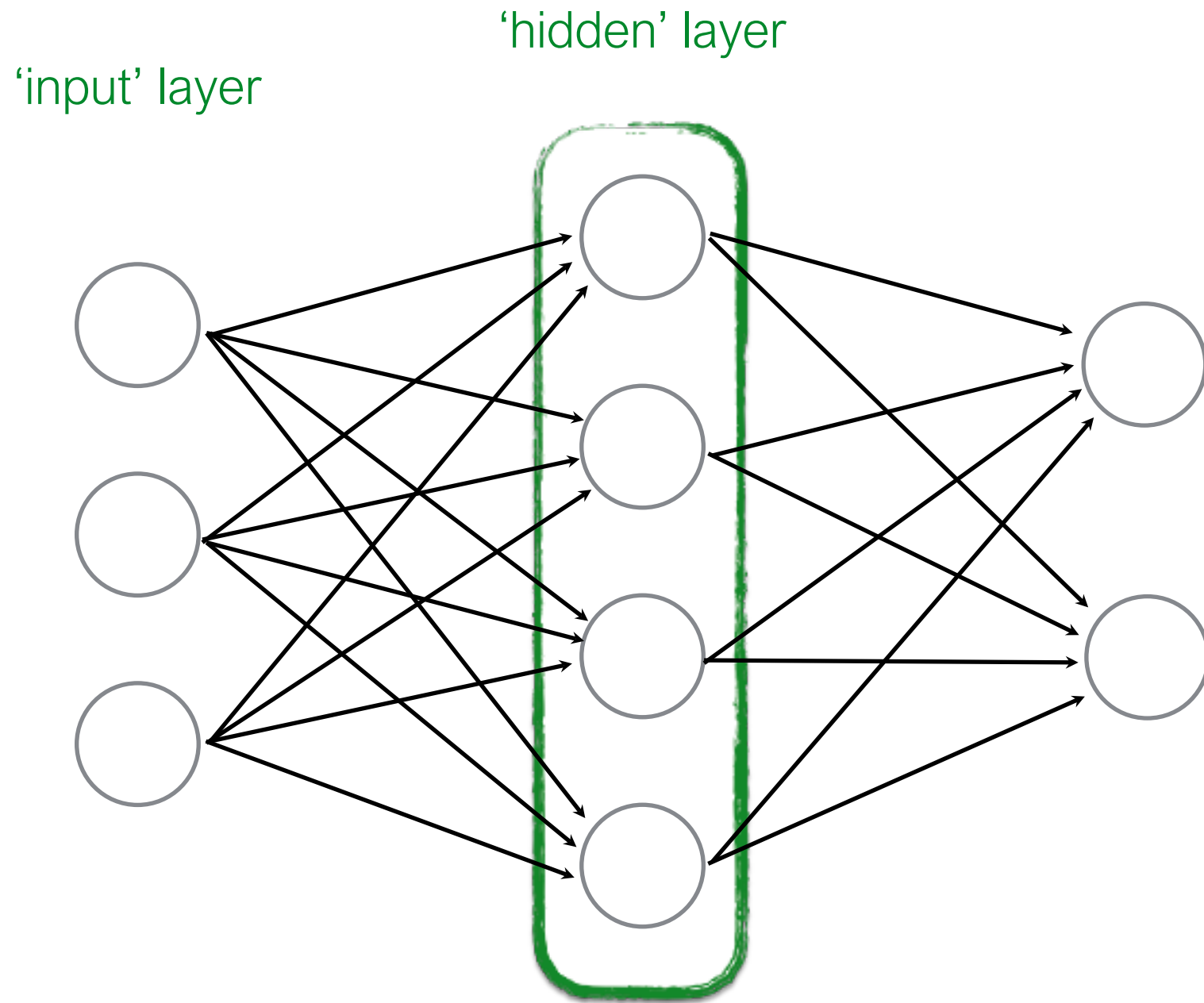
Some terminology...

'input' layer



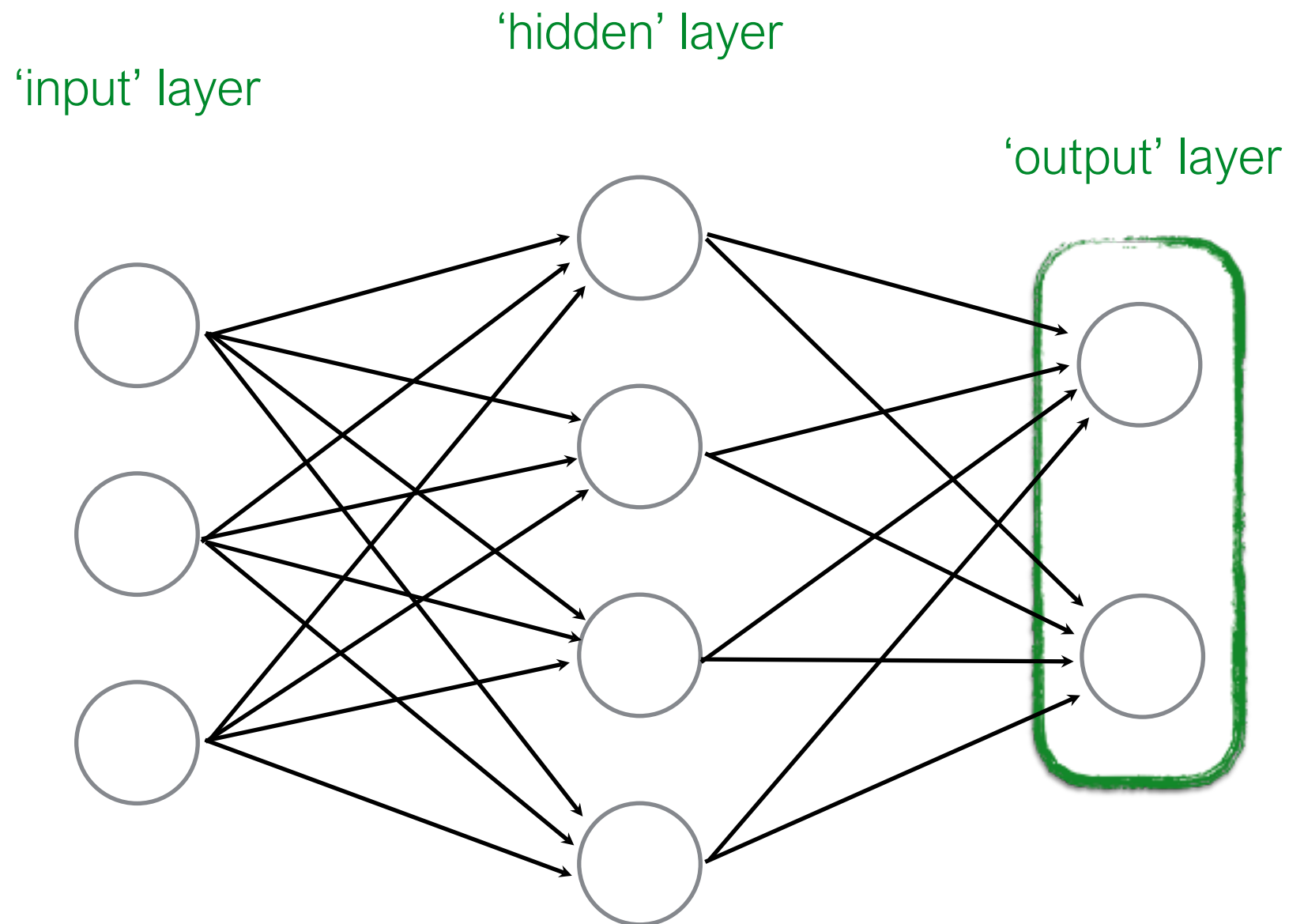
...also called a **Multi-layer Perceptron** (MLP)

Some terminology...



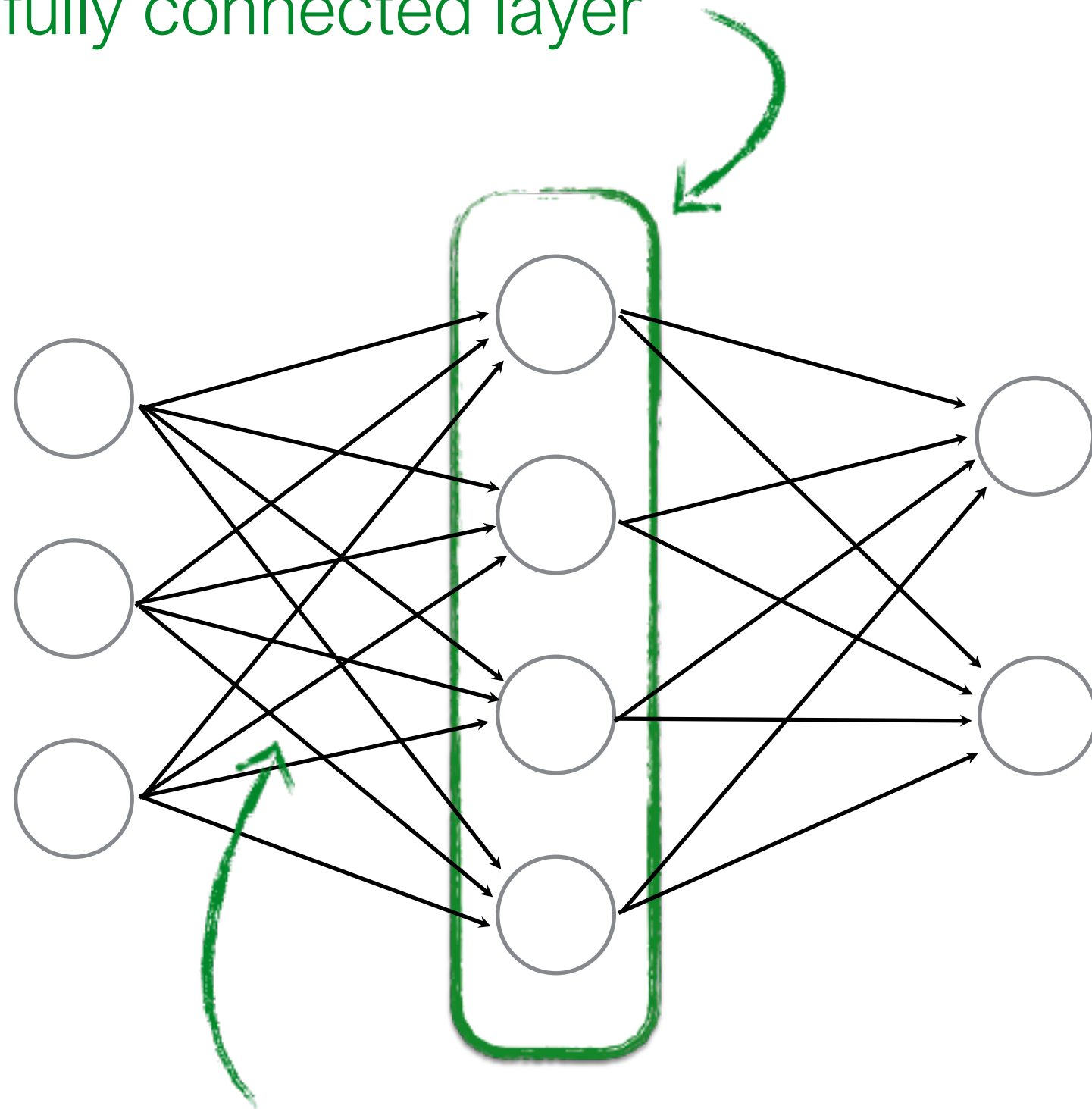
...also called a **Multi-layer Perceptron** (MLP)

Some terminology...

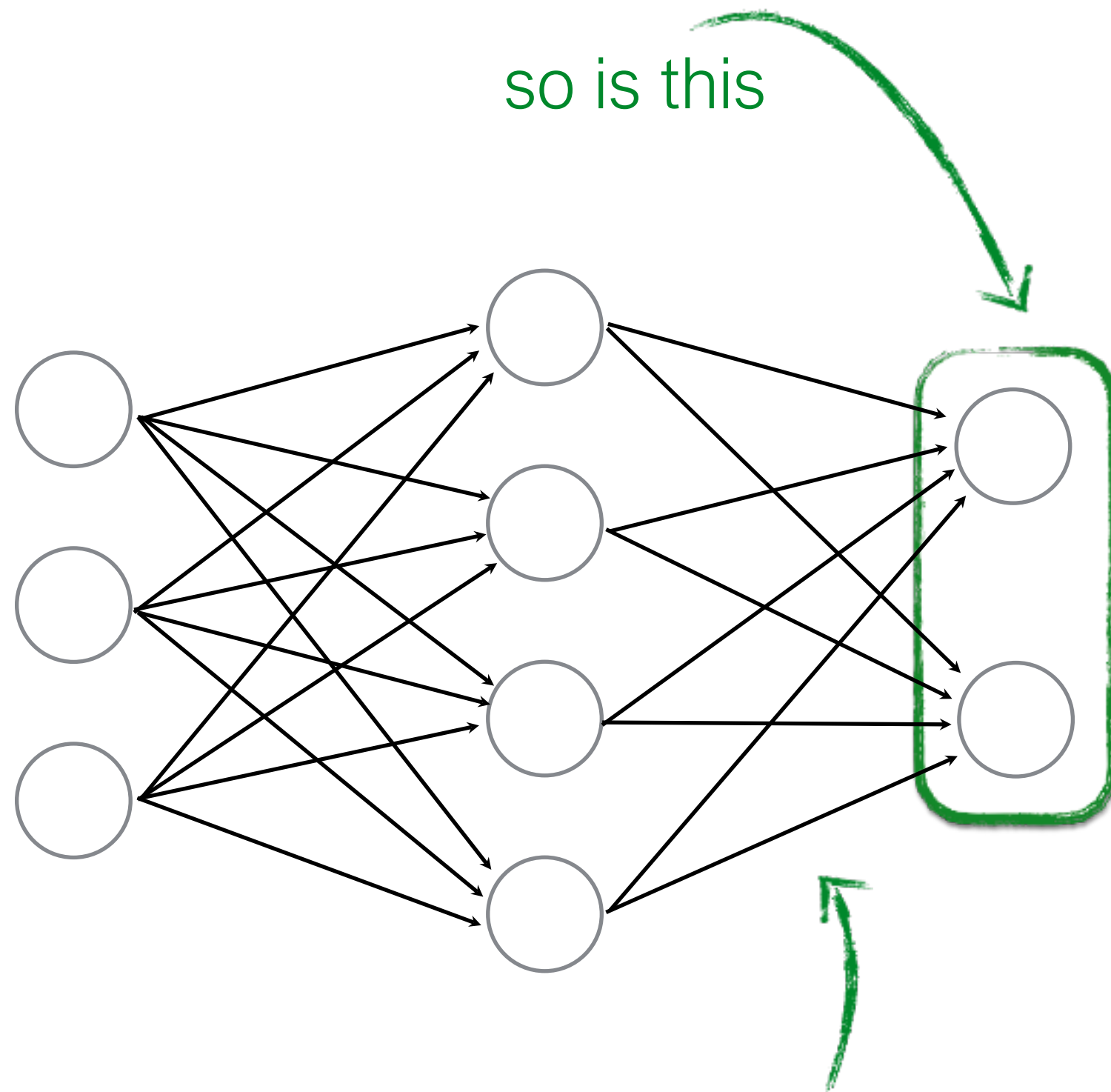


...also called a **Multi-layer Perceptron** (MLP)

this layer is a
'fully connected layer'



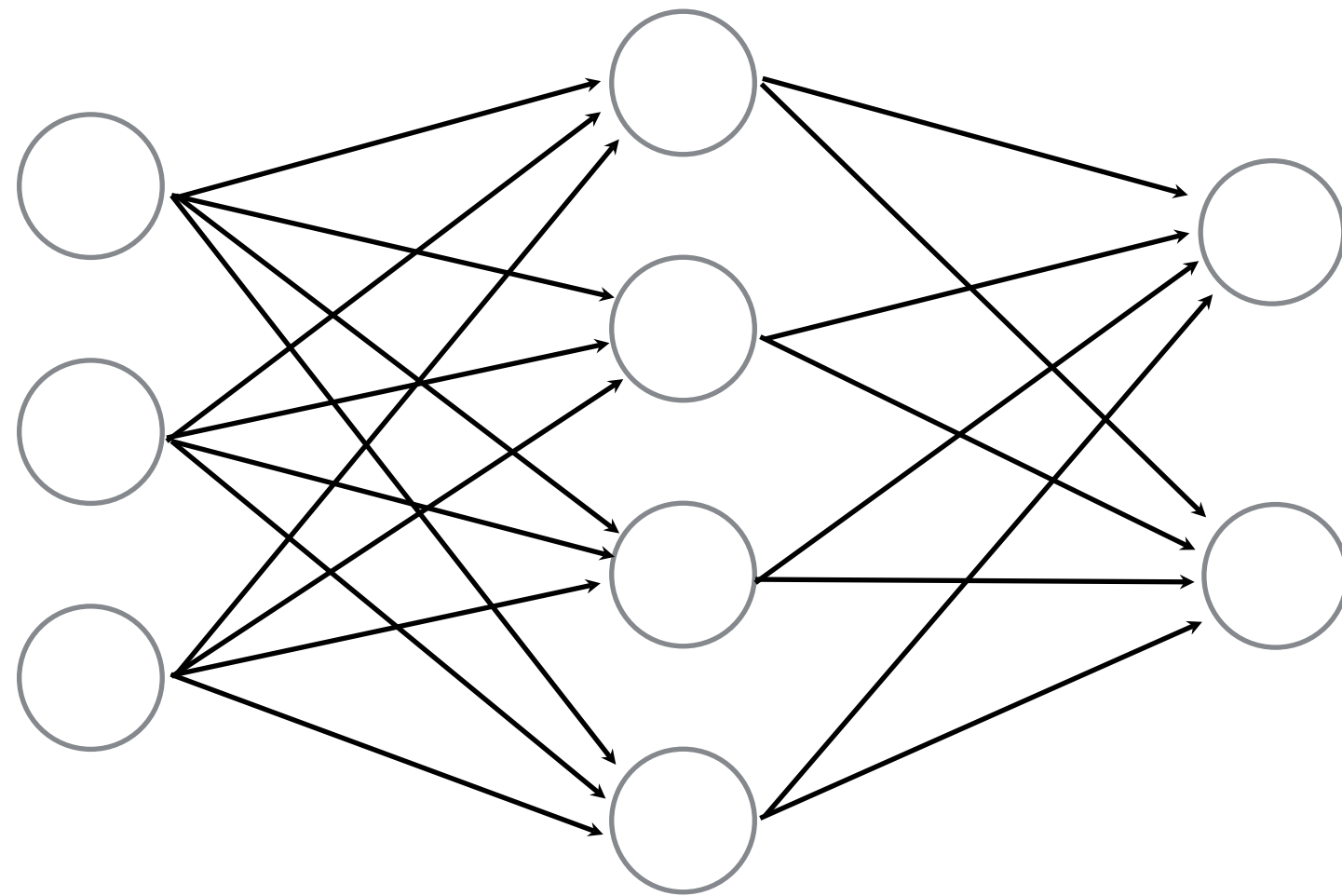
all pairwise neurons between layers are connected



so is this

all pairwise neurons between layers are connected

performance usually tops out at 2-3 layers,
deeper networks don't really improve performance...

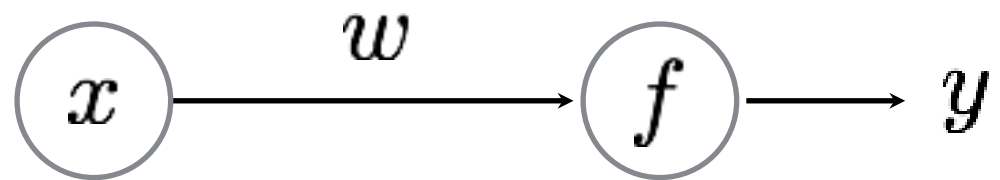


...with the exception of **convolutional** networks for images

Training perceptrons

Let's start easy

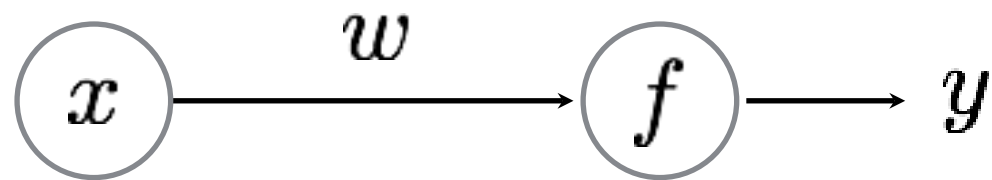
world's smallest perceptron!



$$y = wx$$

What does this look like?

world's smallest perceptron!



$$y = wx$$

(a.k.a. line equation, linear regression)

Learning a Perceptron

Given a set of samples and a Perceptron

$$\{x_i, y_i\}$$

$$y = f_{\text{PER}}(x; w)$$

Estimate the parameters of the Perceptron

$$w$$

Given training data:

x	y
10	10.1
2	1.9
3.5	3.4
1	1.1

What do you think the weight parameter is?

$$y = wx$$

Given training data:

x	y
10	10.1
2	1.9
3.5	3.4
1	1.1

What do you think the weight parameter is?

$$y = wx$$

not so obvious as the network gets more complicated so we use ...

An Incremental Learning Strategy

(gradient descent)

Given several examples

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

and a perceptron

$$\hat{y} = wx$$

An Incremental Learning Strategy

(gradient descent)

Given several examples

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

and a perceptron

$$\hat{y} = wx$$

Modify weight w such that $\hat{y}$ gets '**closer**' to y

An Incremental Learning Strategy

(gradient descent)

Given several examples

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

and a perceptron

$$\hat{y} = wx$$

Modify weight w such that $\hat{y}$ gets '**closer**' to y

perceptron
parameter

perceptron
output

true
label

An Incremental Learning Strategy

(gradient descent)

Given several examples

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

and a perceptron

$$\hat{y} = wx$$

Modify weight w such that $\hat{y}$ gets '**closer**' to y

perceptron
parameter

perceptron
output

*what does
this mean?*

true
label

Before diving into gradient descent, we need to understand ...

Loss Function

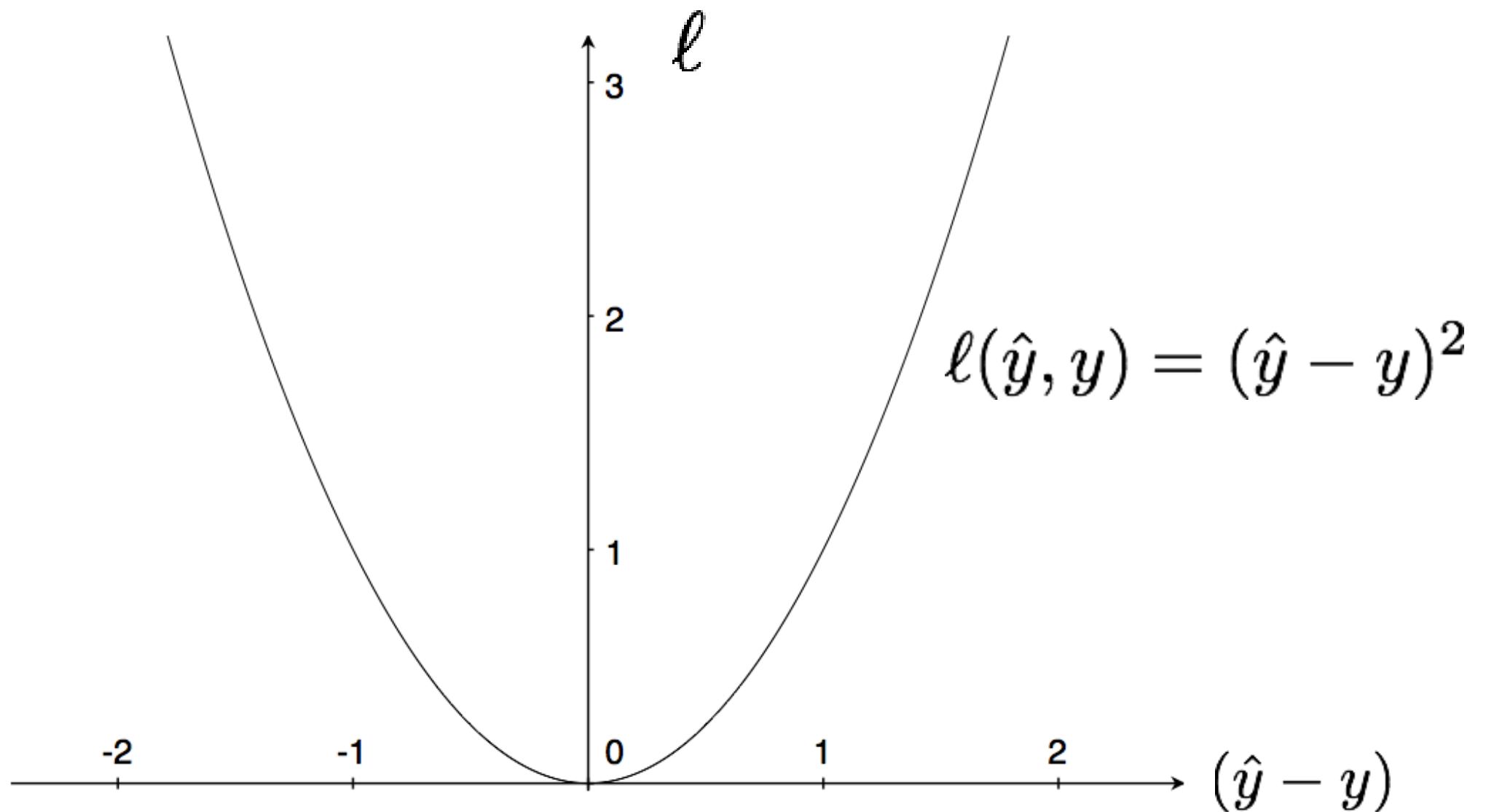
defines what it means to be
close to the true solution

YOU get to choose the loss function!

(some are better than others depending on what you want to do)

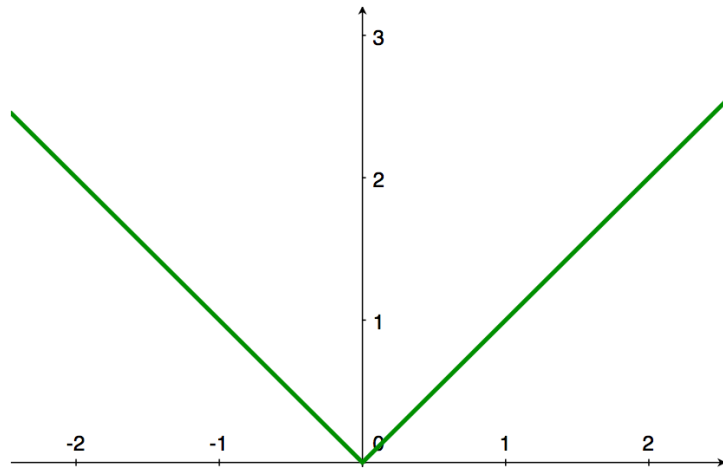
Squared Error (L2)

(a popular loss function) ((why?))



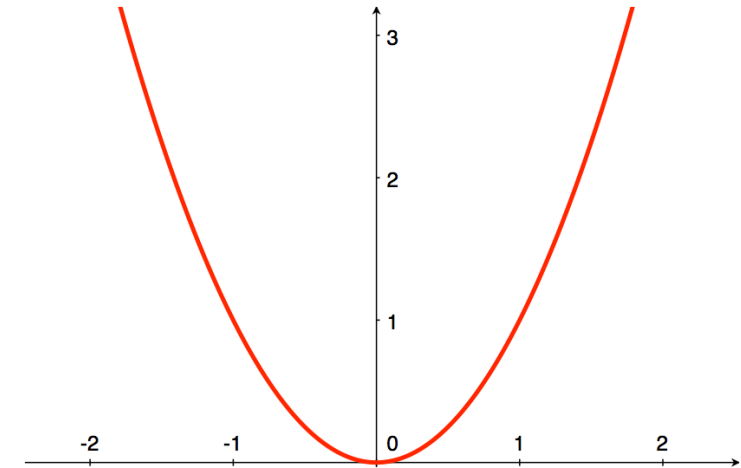
L1 Loss

$$\ell(\hat{y}, y) = |\hat{y} - y|$$



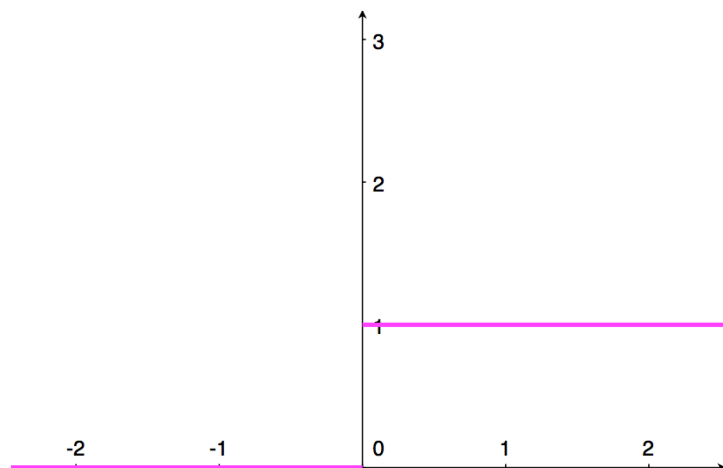
L2 Loss

$$\ell(\hat{y}, y) = (\hat{y} - y)^2$$



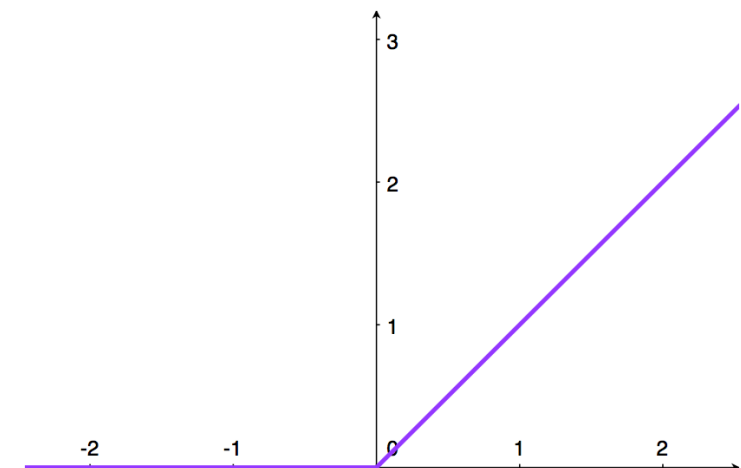
Zero-One Loss

$$\ell(\hat{y}, y) = \mathbf{1}[\hat{y} \neq y]$$



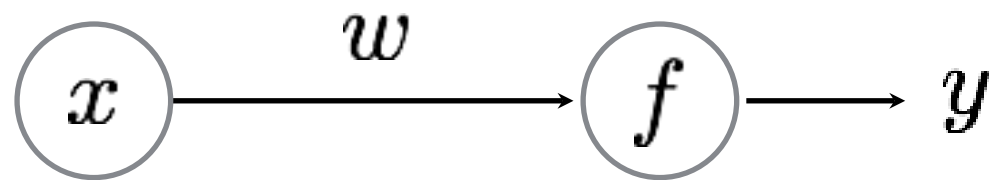
Hinge Loss

$$\ell(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$$



back to the...

World's Smallest Perceptron!



$$y = wx$$

(a.k.a. line equation, linear regression)

function of **ONE** parameter!

Learning a Perceptron

Given a set of samples and a Perceptron

$$\{x_i, y_i\}$$

$$y = f_{\text{PER}}(x; w)$$

what is this 
activation function?

Estimate the parameter of the Perceptron

w

Learning a Perceptron

Given a set of samples and a Perceptron

$$\{x_i, y_i\}$$

$$y = f_{\text{PER}}(x; w)$$

*what is this
activation function?*



linear function! $f(x) = wx$

Estimate the parameter of the Perceptron

$$w$$

Learning Strategy

(gradient descent)

Given several examples

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

and a perceptron

$$\hat{y} = wx$$

Modify weight w such that $\hat{y}$ gets '**closer**' to y

perceptron
parameter

perceptron
output

true
label

Let's demystify this process first...

Code to train your perceptron:

Let's demystify this process first...

Code to train your perceptron:

for $n = 1 \dots N$

$w = w + (y_n - \hat{y})x_i;$

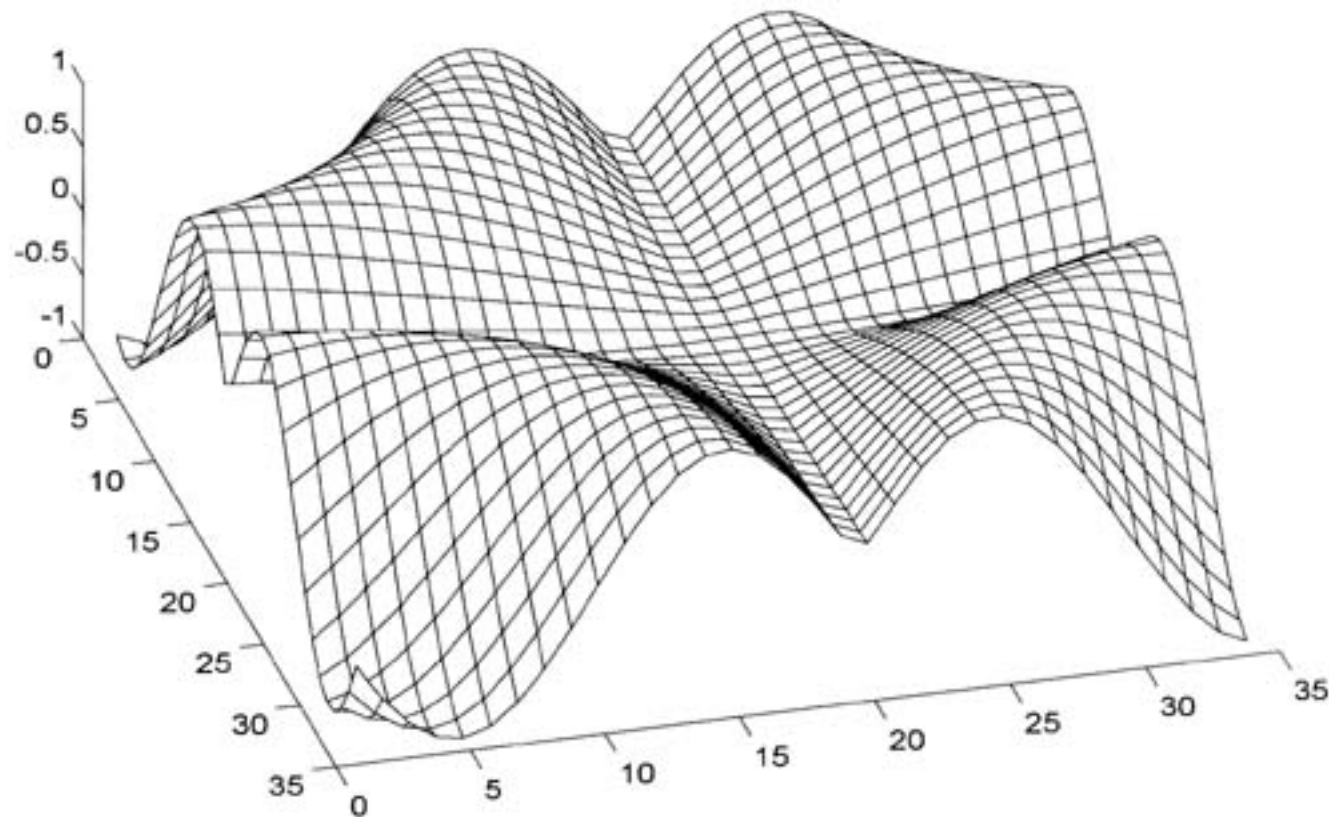
just one line of code!

Now where does this come from?

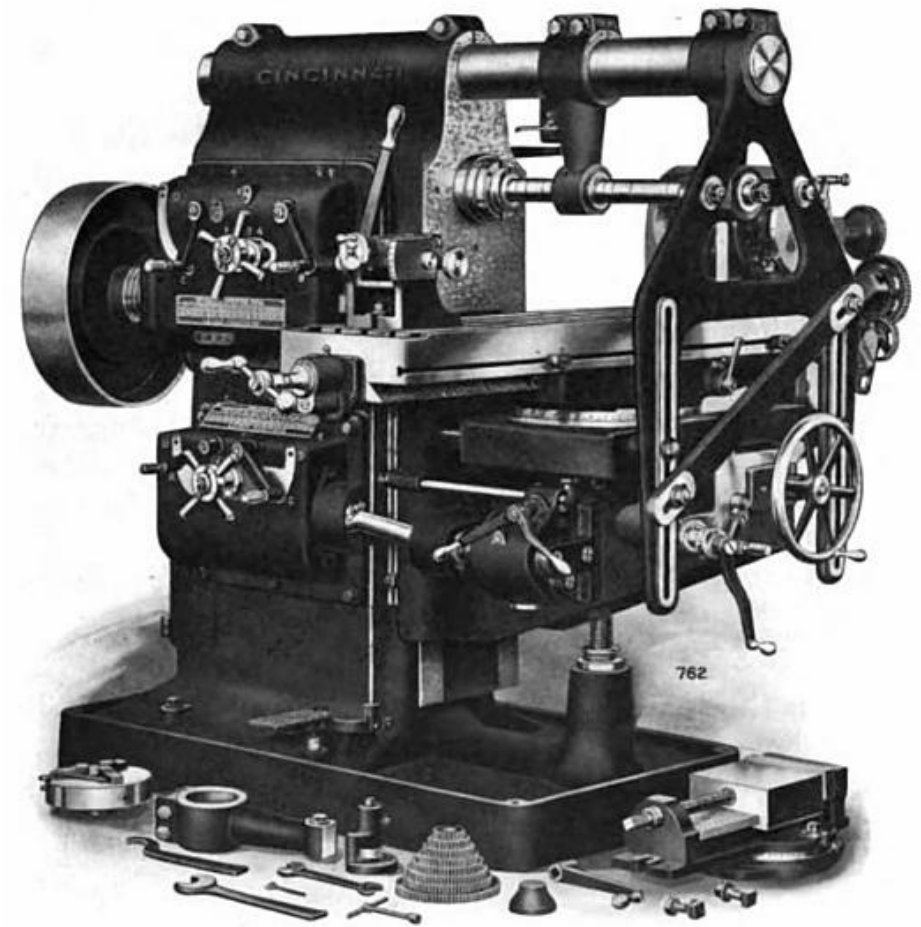
Gradient descent

(partial) derivatives tell us how much
one variable affects another

Two ways to think about them:

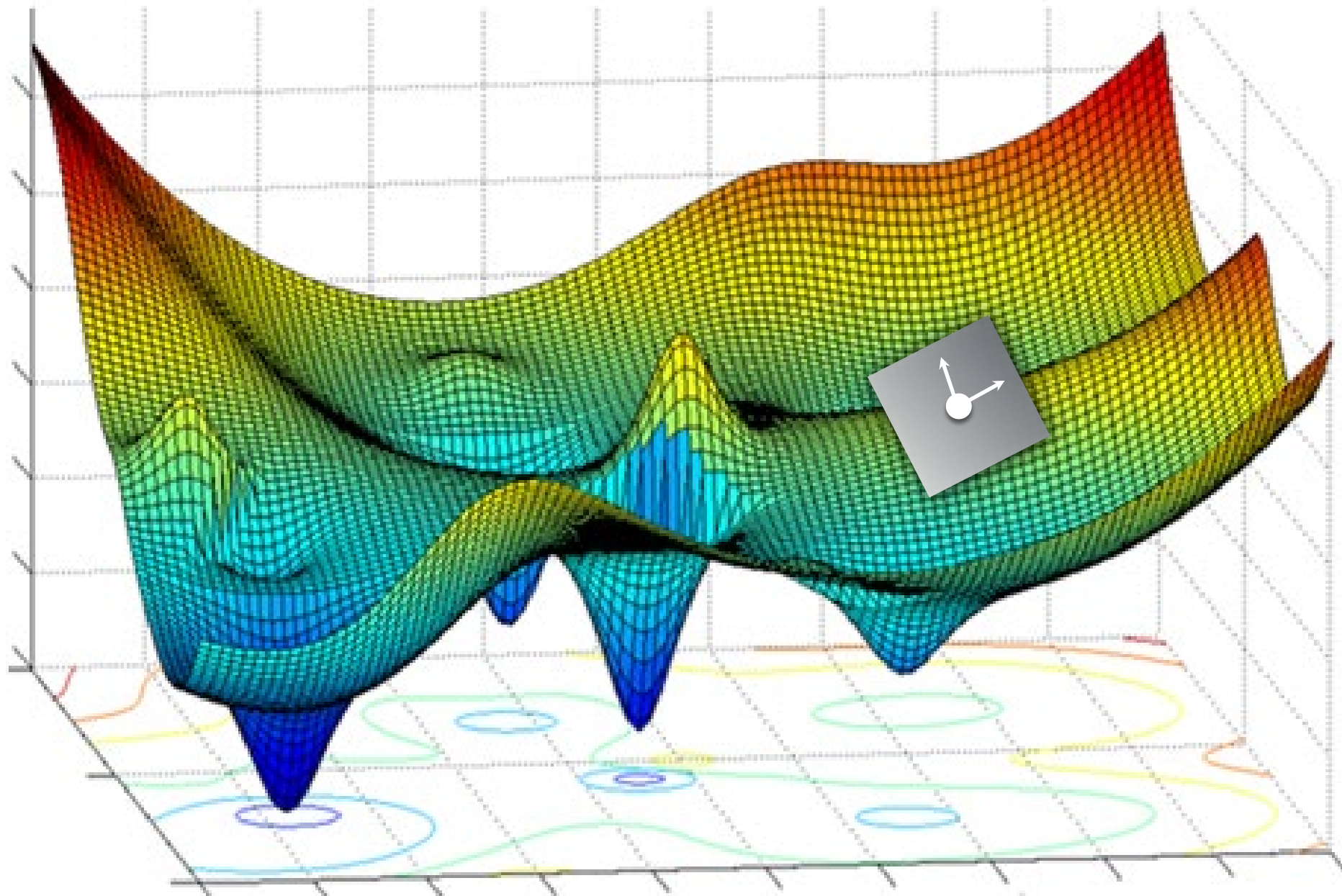


Slope of a function



Knobs on a machine

1. Slope of a function:



$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \left[\frac{\partial f(\mathbf{x})}{\partial x}, \frac{\partial f(\mathbf{x})}{\partial y} \right]$$

describes the slope around a point

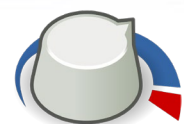
2. Knobs on a machine:



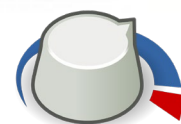
describes how each
'knob' affects the output



$$\frac{\partial f(x)}{\partial w_1}$$



$$\frac{\partial f(x)}{\partial w_2}$$



$$\frac{\partial f(x)}{\partial w_3}$$

small change in parameter

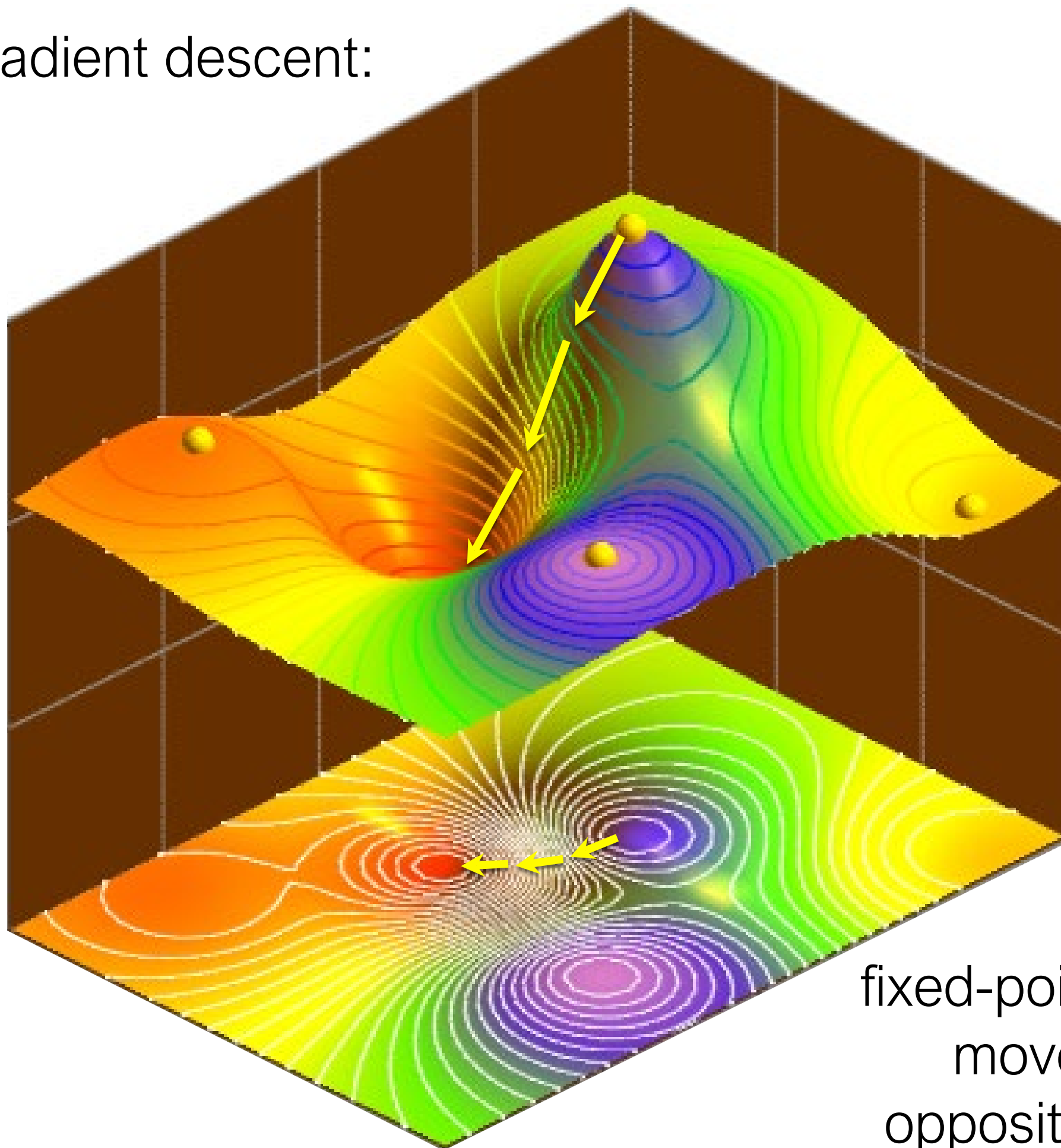
$$\Delta w_1$$



output will change by

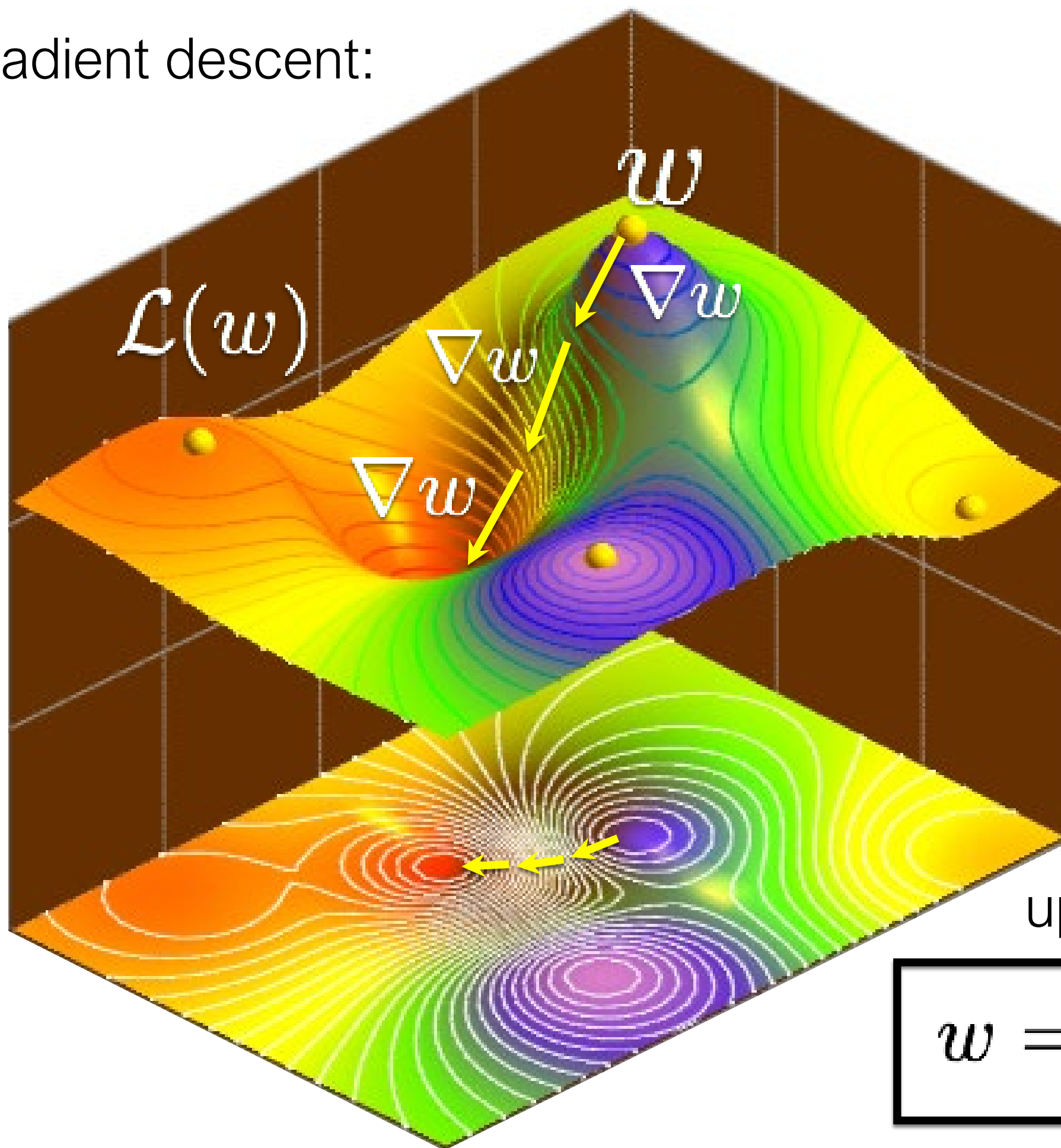
$$\frac{\partial f(x)}{\partial w_1} \Delta w_1$$

Gradient descent:



Given a
fixed-point on a function,
move in the direction
opposite of the gradient

Gradient descent:



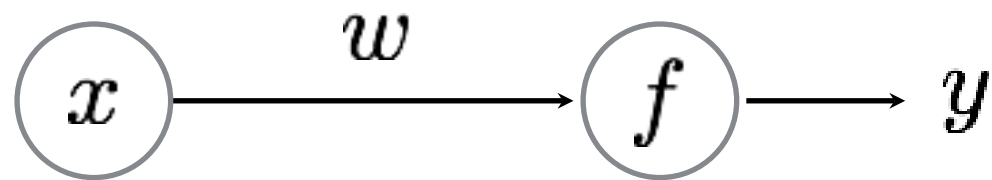
update rule:

$$w = w - \nabla w$$

Backpropagation

back to the...

World's Smallest Perceptron!



$$y = wx$$

(a.k.a. line equation, linear regression)

function of **ONE** parameter!

Training the world's smallest perceptron

for $n = 1 \dots N$

This is just gradient descent, that means...

$$w = w + \underline{(y_n - \hat{y})x_i};$$



this should be the gradient of the loss function

Now where does this come from?

$$\frac{d\mathcal{L}}{dw}$$

...is the rate at which **this** will change...

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$$


the loss function

... per unit change of **this**

$$y = wx$$


the weight parameter

Let's compute the derivative...

Compute the derivative

$$\begin{aligned}\frac{d\mathcal{L}}{dw} &= \frac{d}{dw} \left\{ \frac{1}{2} (y - \hat{y})^2 \right\} \\ &= -(y - \hat{y}) \frac{dw x}{dw} \\ &= -(y - \hat{y}) x = \nabla w \quad \text{just shorthand}\end{aligned}$$

That means the weight update for **gradient descent** is:

$$\begin{aligned}w &= w - \nabla w \quad \text{move in direction of negative gradient} \\ &= w + (y - \hat{y}) x\end{aligned}$$

Gradient Descent (world's smallest perceptron)

For each sample

$$\{x_i, y_i\}$$

1. Predict

a. Forward pass

$$\hat{y} = wx_i$$

b. Compute Loss

$$\mathcal{L}_i = \frac{1}{2}(y_i - \hat{y})^2$$

2. Update

a. Back Propagation

$$\frac{d\mathcal{L}_i}{dw} = -(y_i - \hat{y})x_i = \nabla w$$

b. Gradient update

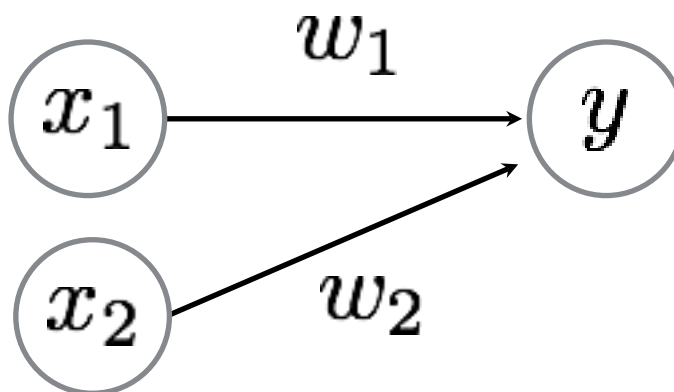
$$w = w - \nabla w$$

Training the world's smallest perceptron

for $n = 1 \dots N$

$$w = w + (y_n - \hat{y})x_i;$$

world's (second) smallest
perceptron!



function of **two** parameters!

Gradient Descent

For each sample

$\{x_i, y_i\}$

1. Predict

a. Forward pass

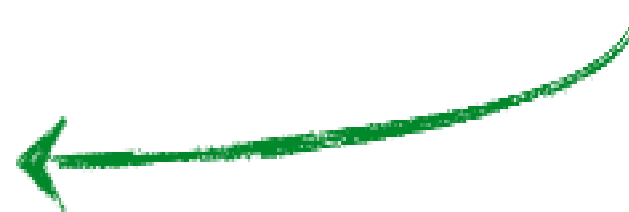
b. Compute Loss

2. Update

a. Back Propagation

b. Gradient update

we just need to compute partial
derivatives for this network



Derivative computation

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (y - \hat{y})^2 \right\} \\ &= -(y - \hat{y}) \frac{\partial \hat{y}}{\partial w_1} \\ &= -(y - \hat{y}) \frac{\partial \sum_i w_i x_i}{\partial w_1} \\ &= -(y - \hat{y}) \frac{\partial w_1 x_1}{\partial w_1} \\ &= -(y - \hat{y}) x_1 = \nabla w_1\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_2} &= \frac{\partial}{\partial w_2} \left\{ \frac{1}{2} (y - \hat{y})^2 \right\} \\ &= -(y - \hat{y}) \frac{\partial \hat{y}}{\partial w_2} \\ &= -(y - \hat{y}) \frac{\partial \sum_i w_i x_i}{\partial w_2} \\ &= -(y - \hat{y}) \frac{\partial w_2 x_2}{\partial w_2} \\ &= -(y - \hat{y}) x_2 = \nabla w_2\end{aligned}$$

Gradient Update

$$\begin{aligned}w_1 &= w_1 - \eta \nabla w_1 \\ &= w_1 + \eta (y - \hat{y}) x_1\end{aligned}$$

$$\begin{aligned}w_2 &= w_2 - \eta \nabla w_2 \\ &= w_2 + \eta (y - \hat{y}) x_2\end{aligned}$$

Gradient Descent

For each sample $\{x_i, y_i\}$

1. Predict

a. Forward pass $\hat{y} = f_{\text{MLP}}(x_i; \theta)$

b. Compute Loss $\mathcal{L}_i = \frac{1}{2}(y_i - \hat{y})$ (side computation to track loss. not needed for backprop)

2. Update

a. Back Propagation

b. Gradient update

(adjustable step size)

two lines now

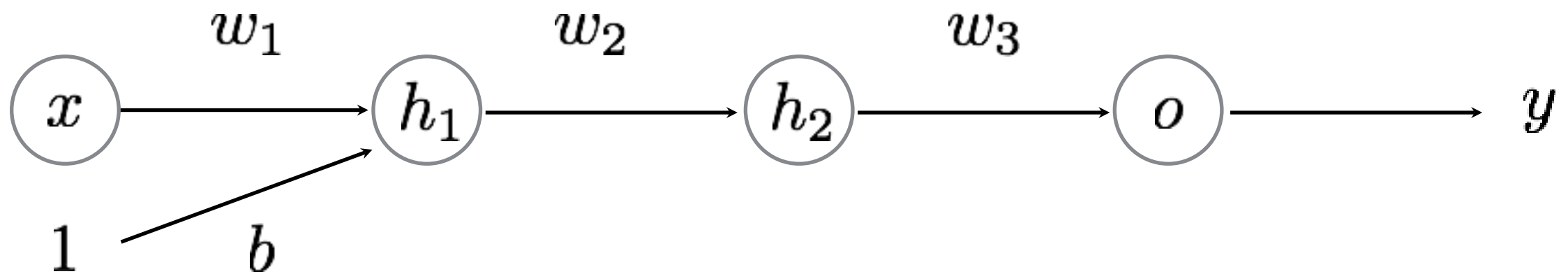
$$\begin{aligned}\nabla w_{1i} &= -(y_i - \hat{y})x_{1i} \\ \nabla w_{2i} &= -(y_i - \hat{y})x_{2i}\end{aligned}$$

$$\begin{aligned}w_{1i} &= w_{1i} + \eta(y - \hat{y})x_{1i} \\ w_{2i} &= w_{2i} + \eta(y - \hat{y})x_{2i}\end{aligned}$$

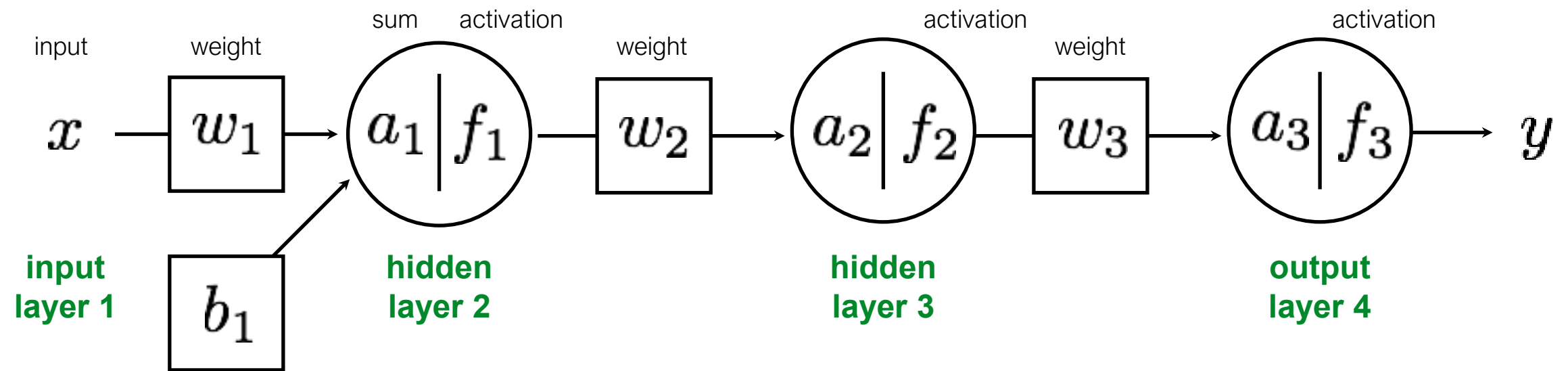


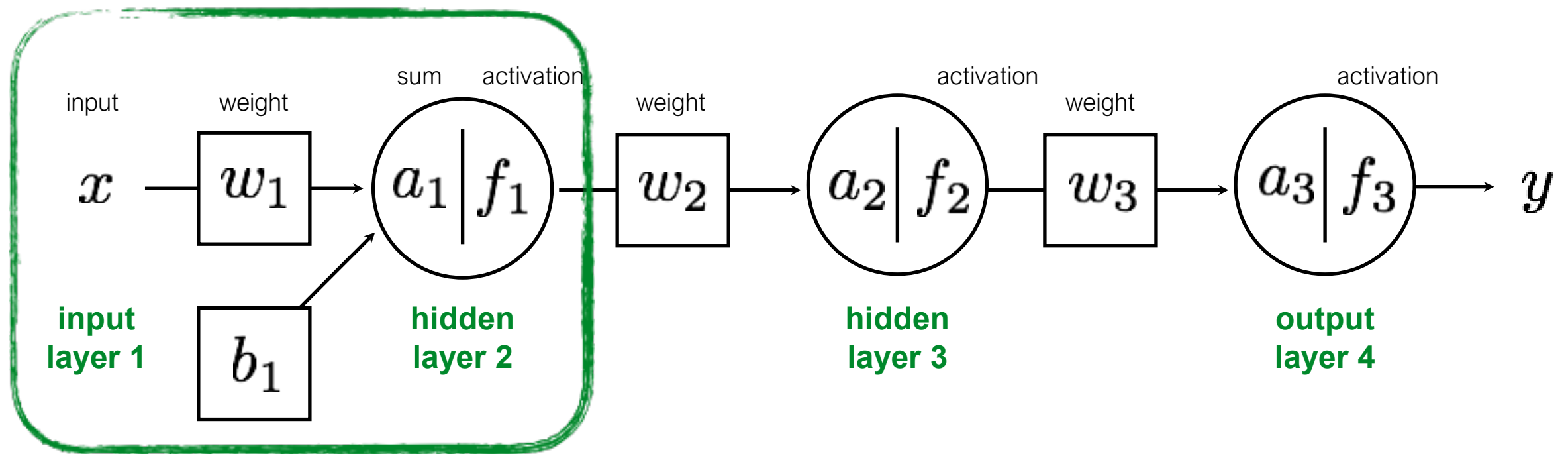
We haven't seen a lot of 'propagation' yet
because our perceptrons only had one layer...

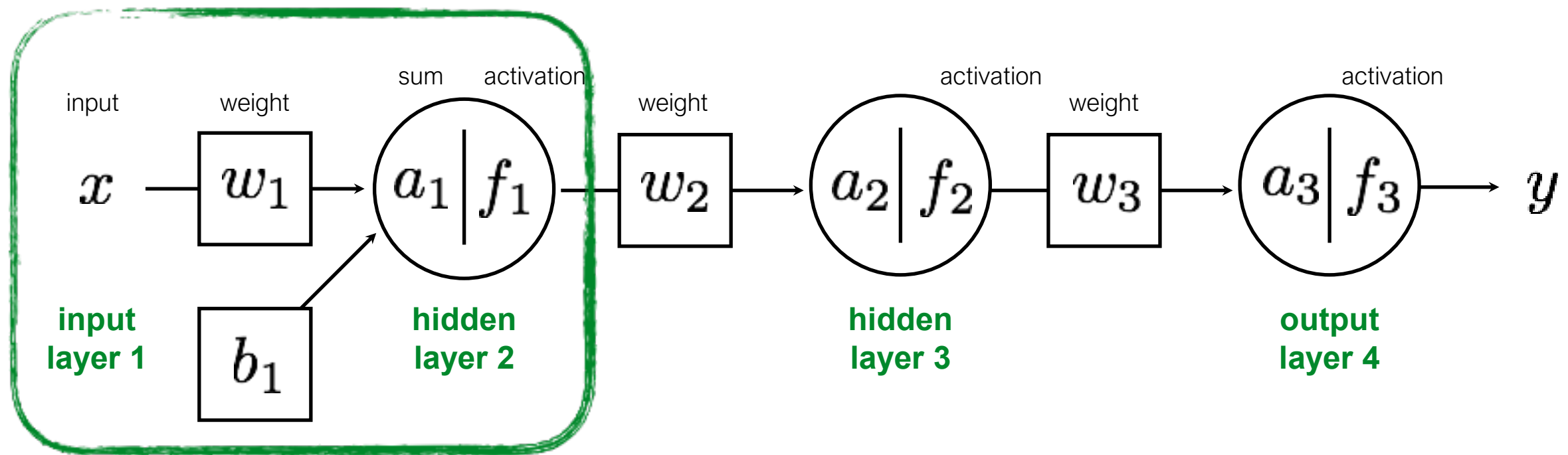
multi-layer perceptron



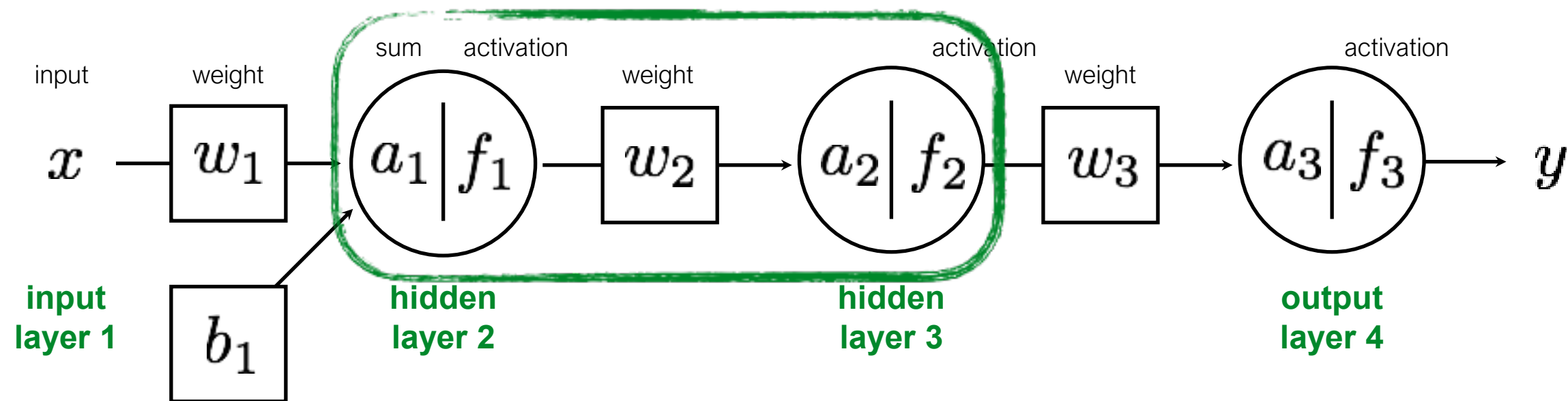
function of **FOUR** parameters and **FOUR** layers!



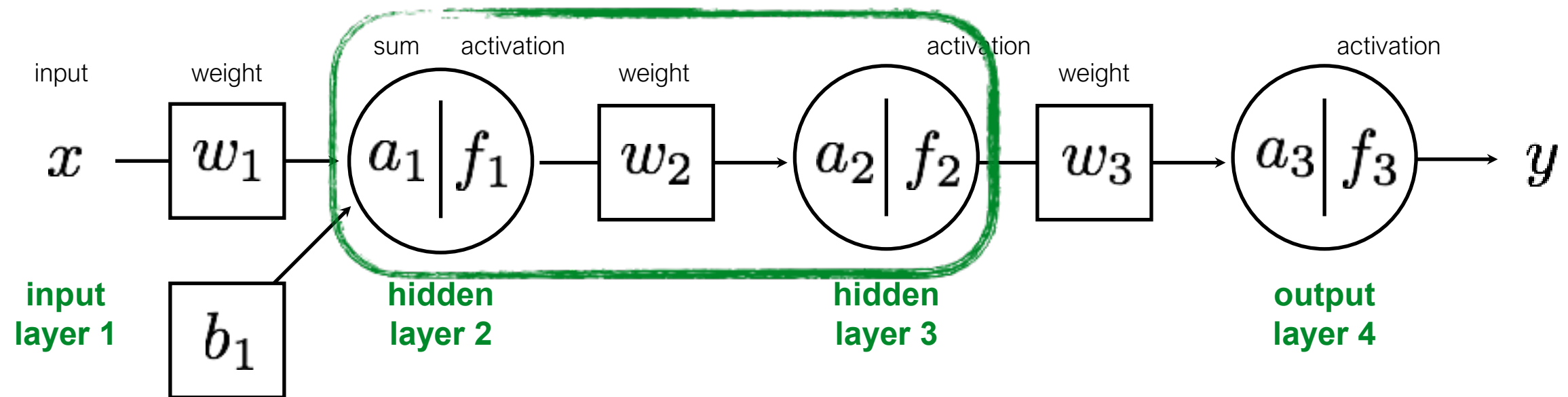




$$a_1 = w_1 \cdot x + b_1$$

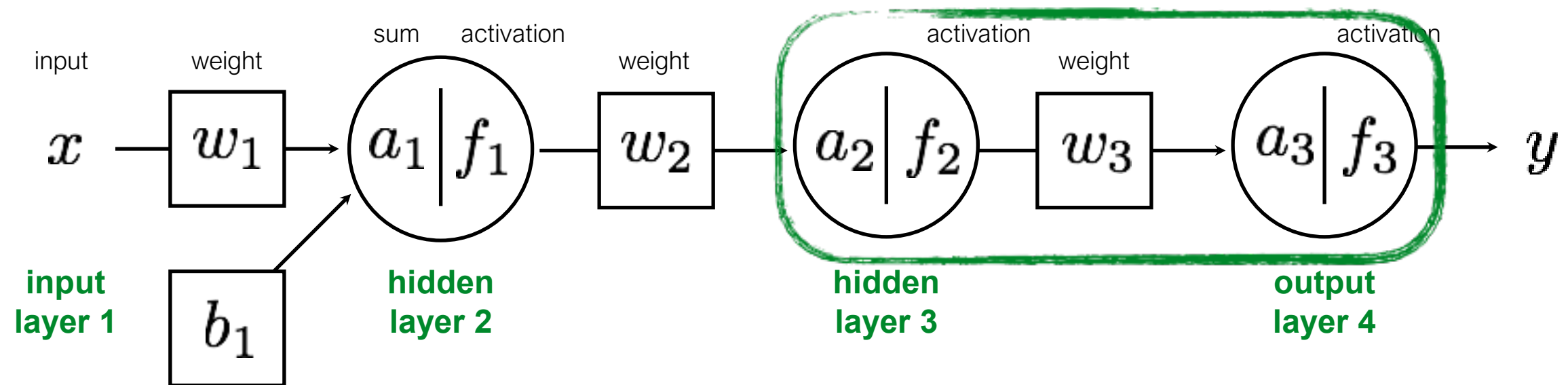


$$a_1 = w_1 \cdot x + b_1$$



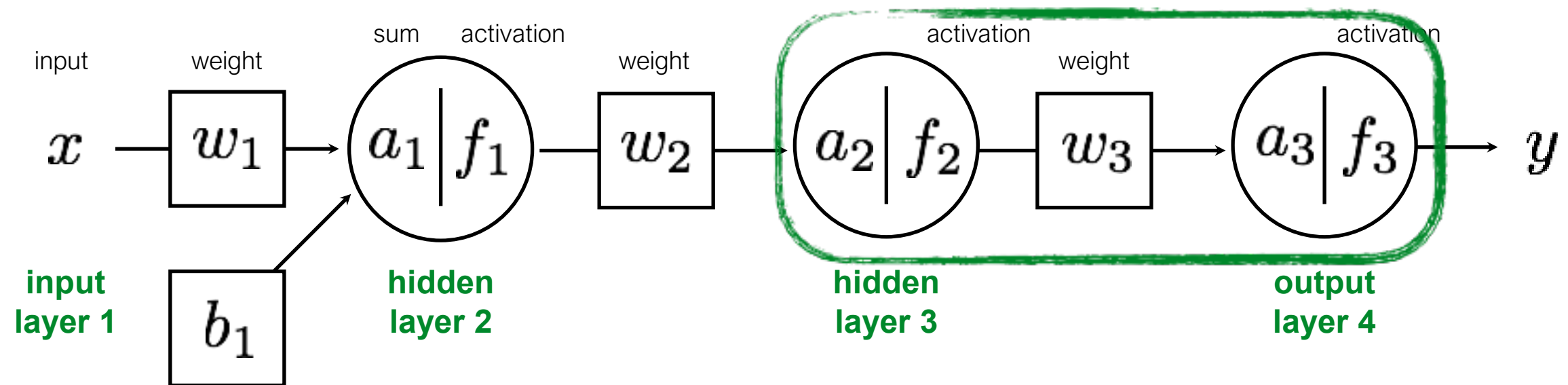
$$a_1 = w_1 \cdot x + b_1$$

$$a_2 = w_2 \cdot f_1(w_1 \cdot x + b_1)$$



$$a_1 = w_1 \cdot x + b_1$$

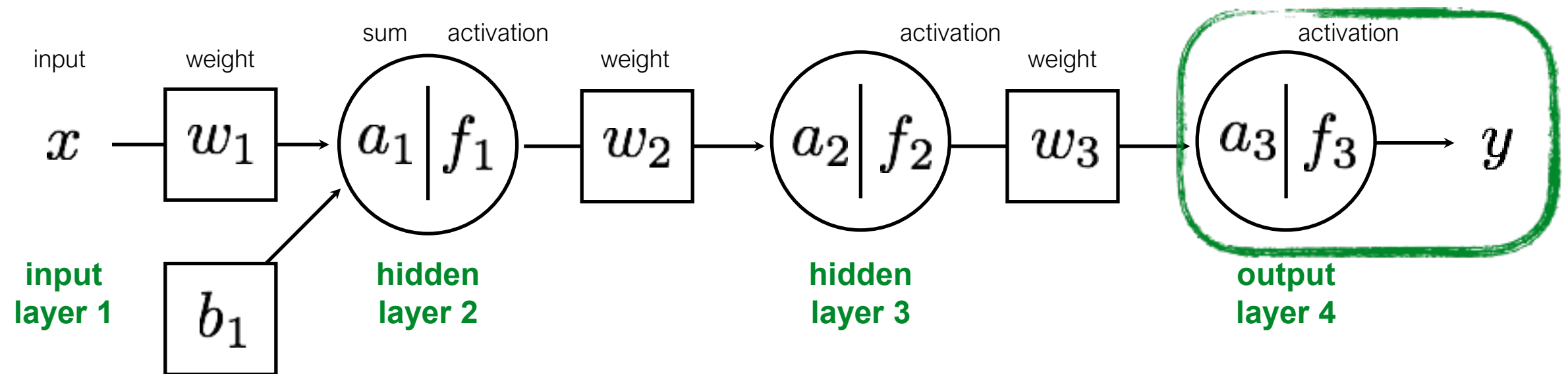
$$a_2 = w_2 \cdot f_1(w_1 \cdot x + b_1)$$



$$a_1 = w_1 \cdot x + b_1$$

$$a_2 = w_2 \cdot f_1(w_1 \cdot x + b_1)$$

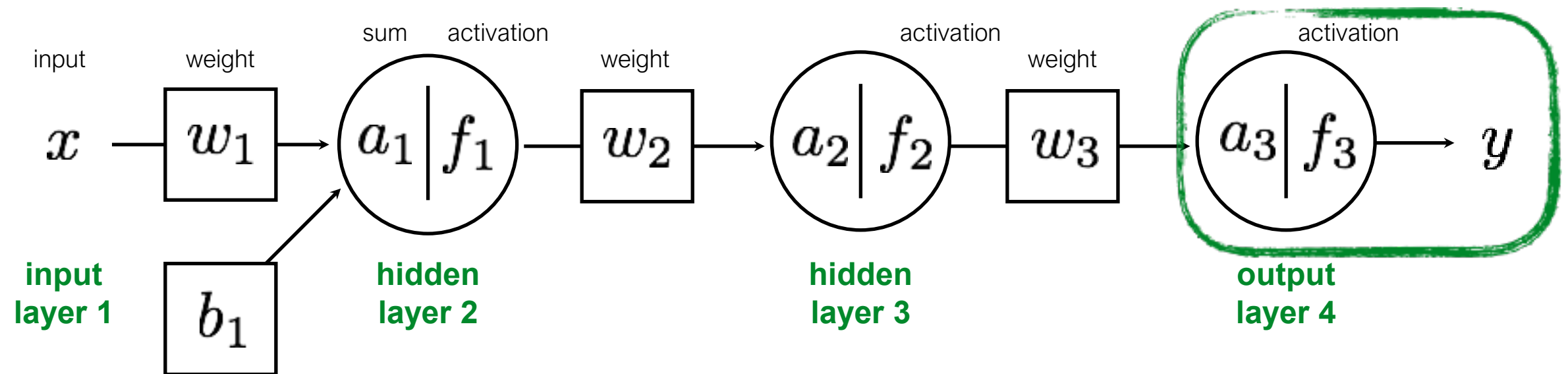
$$a_3 = w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1))$$



$$a_1 = w_1 \cdot x + b_1$$

$$a_2 = w_2 \cdot f_1(w_1 \cdot x + b_1)$$

$$a_3 = w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1))$$



$$a_1 = w_1 \cdot x + b_1$$

$$a_2 = w_2 \cdot f_1(w_1 \cdot x + b_1)$$

$$a_3 = w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1))$$

$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)))$$

Entire network can be written out as one long equation

$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)))$$

We need to train the network:

What is known? What is unknown?

Entire network can be written out as a long equation

$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)))$$



known

We need to train the network:

What is known? What is unknown?

Entire network can be written out as a long equation

$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)))$$

activation function
sometimes has
parameters

unknown

We need to train the network:

What is known? What is unknown?

Learning an MLP

Given a set of samples and a MLP

$$\{x_i, y_i\}$$

$$y = f_{\text{MLP}}(x; \theta)$$

Estimate the parameters of the MLP

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

Gradient Descent

For each **random** sample $\{x_i, y_i\}$

1. Predict

a. Forward pass

$$\hat{y} = f_{\text{MLP}}(x_i; \theta)$$

b. Compute Loss

2. Update

a. Back Propagation

$$\frac{\partial \mathcal{L}}{\partial \theta}$$

vector of parameter partial derivatives

b. Gradient update

$$\theta \leftarrow \theta - \eta \nabla \theta$$

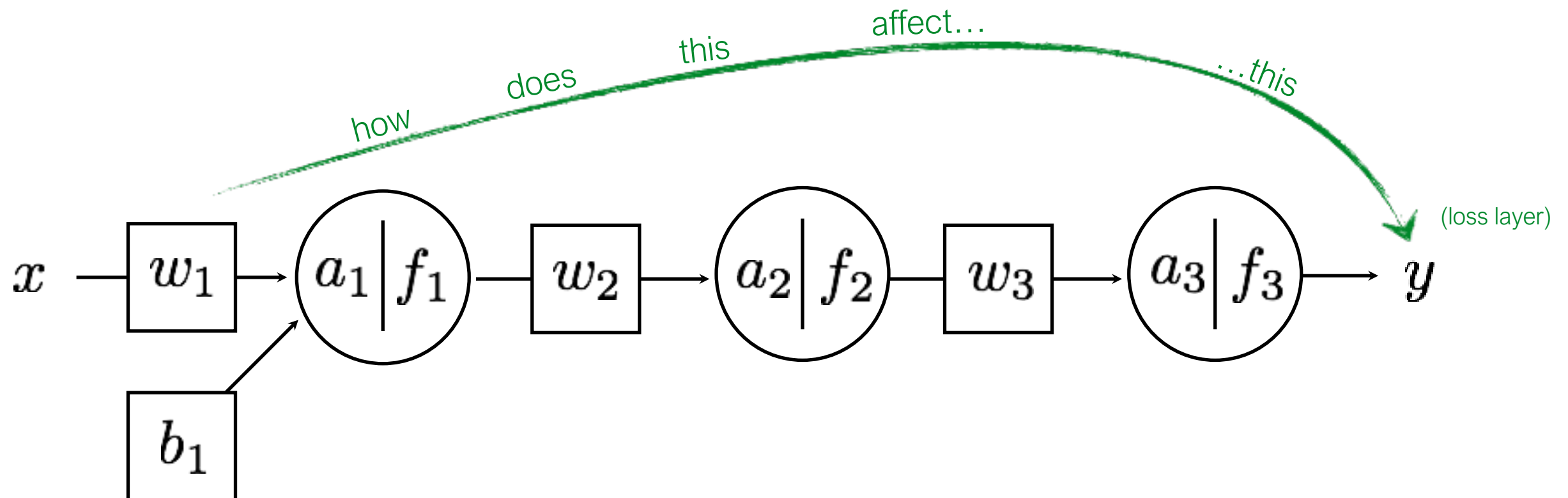
vector of parameter update equations

So we need to compute the partial derivatives

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \mathcal{L}}{\partial w_3} \quad \frac{\partial \mathcal{L}}{\partial w_2} \quad \frac{\partial \mathcal{L}}{\partial w_1} \quad \frac{\partial \mathcal{L}}{\partial b} \right]$$

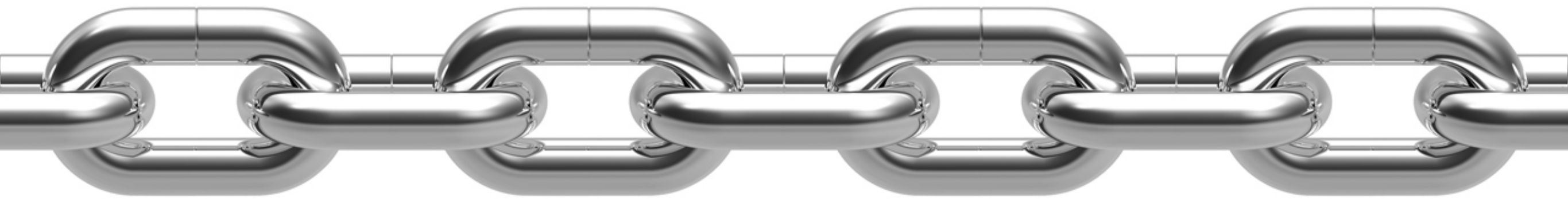
Remember,

Partial derivative $\frac{\partial L}{\partial w_1}$ describes...



So, how do you compute it?

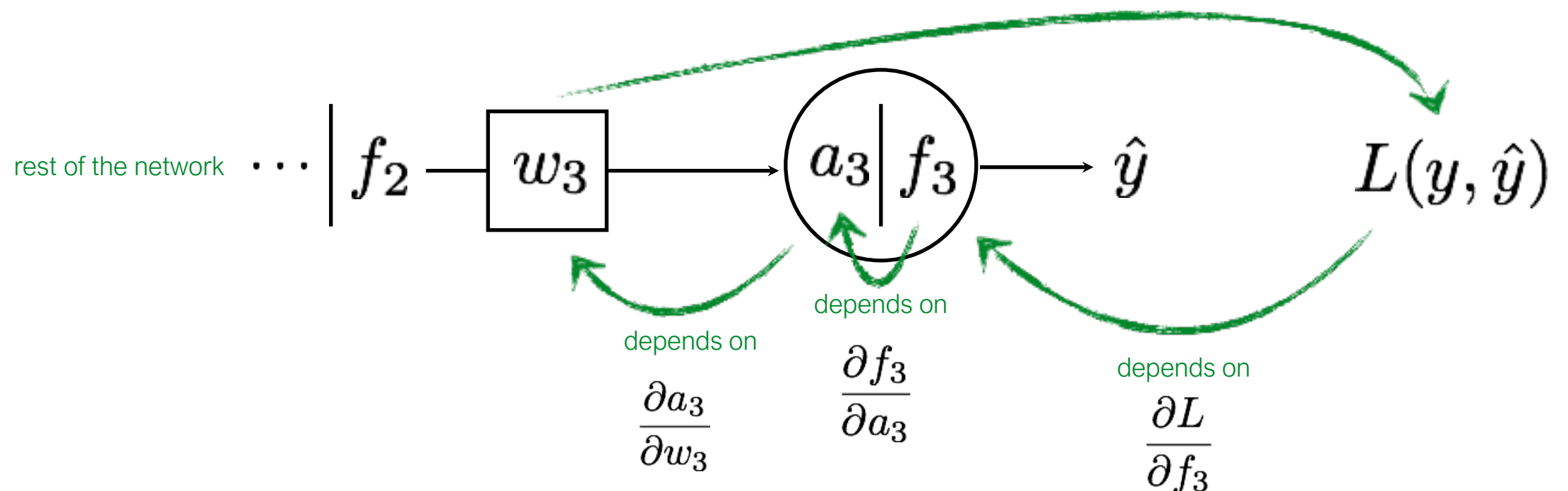
THE CHAIN RULE

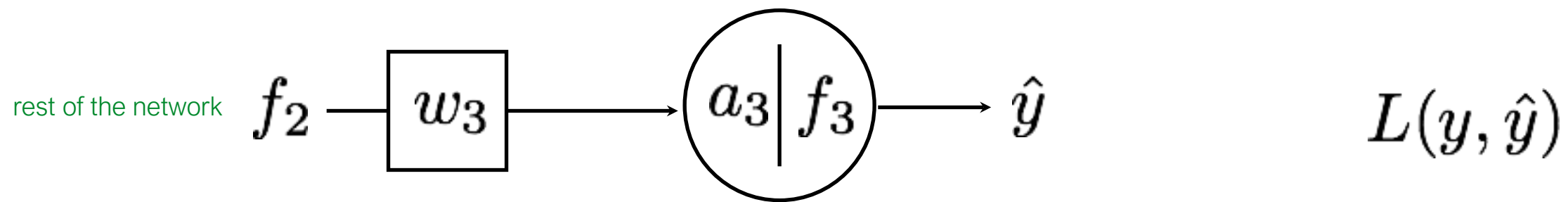


According to the chain rule...

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3}$$

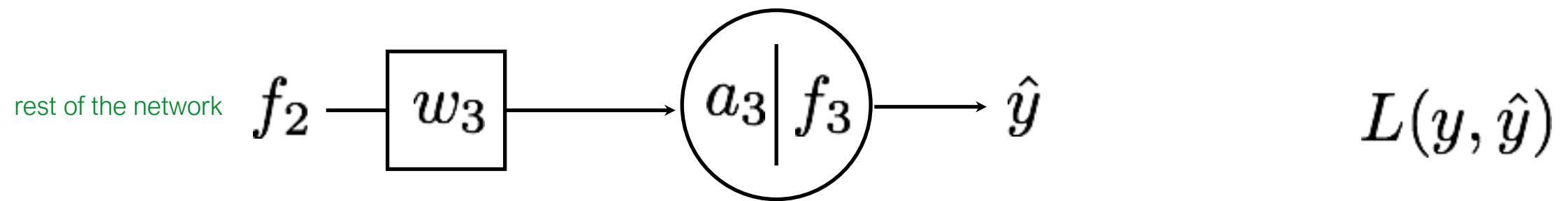
Intuitively, the effect of weight on loss function : $\frac{\partial L}{\partial w_3}$





$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3}$$

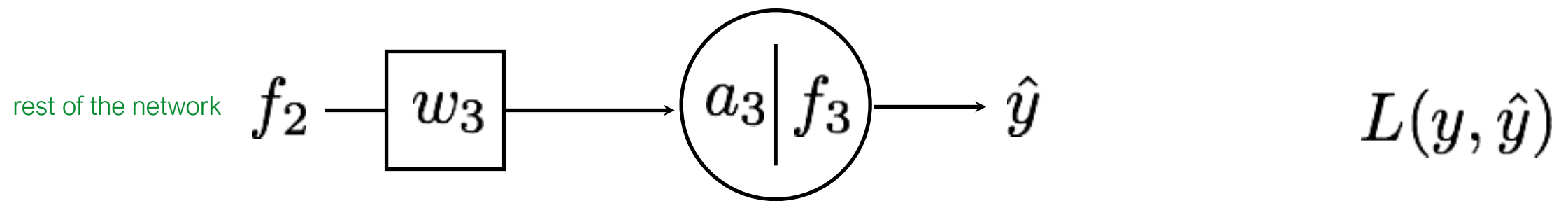
Chain Rule!



$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3}$$

$$= -\eta(y - \hat{y}) \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3}$$

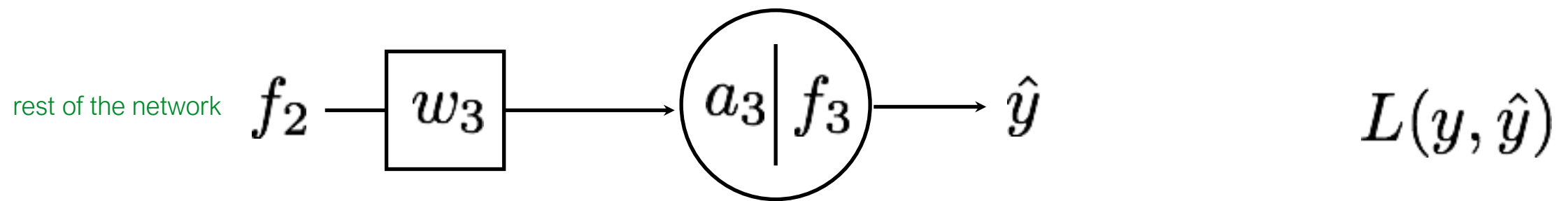
Just the partial
derivative of L2 loss



$$\begin{aligned} \frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \\ &= -\eta(y - \hat{y}) \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \end{aligned}$$

Let's use a Sigmoid function

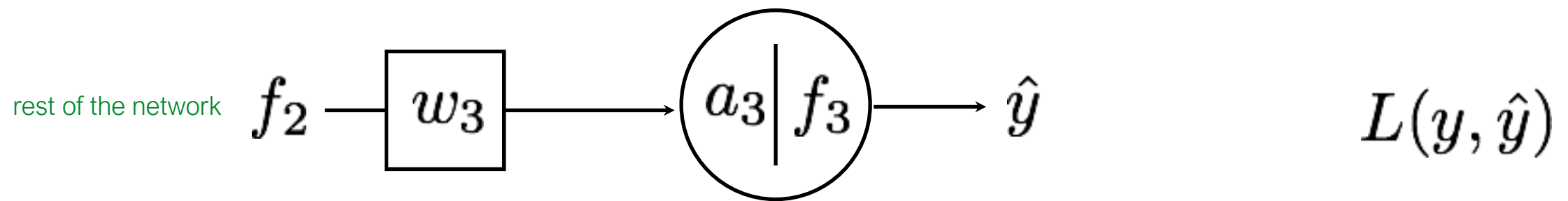
$$\frac{ds(x)}{dx} = s(x)(1 - s(x))$$



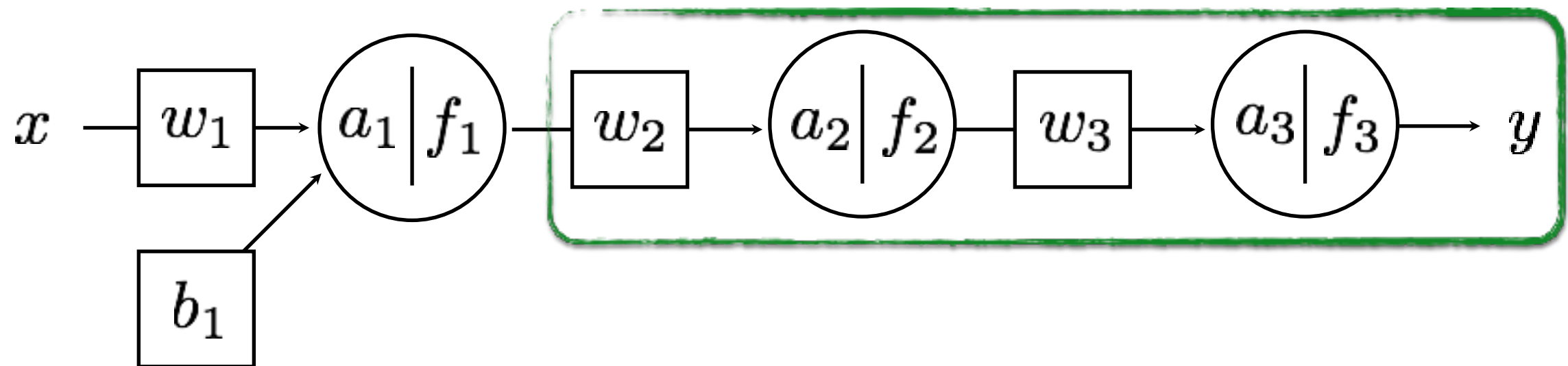
$$\begin{aligned}
 \frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \\
 &= -\eta(y - \hat{y}) \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \\
 &= -\eta(y - \hat{y}) f_3(1 - f_3) \frac{\partial a_3}{\partial w_3}
 \end{aligned}$$

Let's use a Sigmoid function

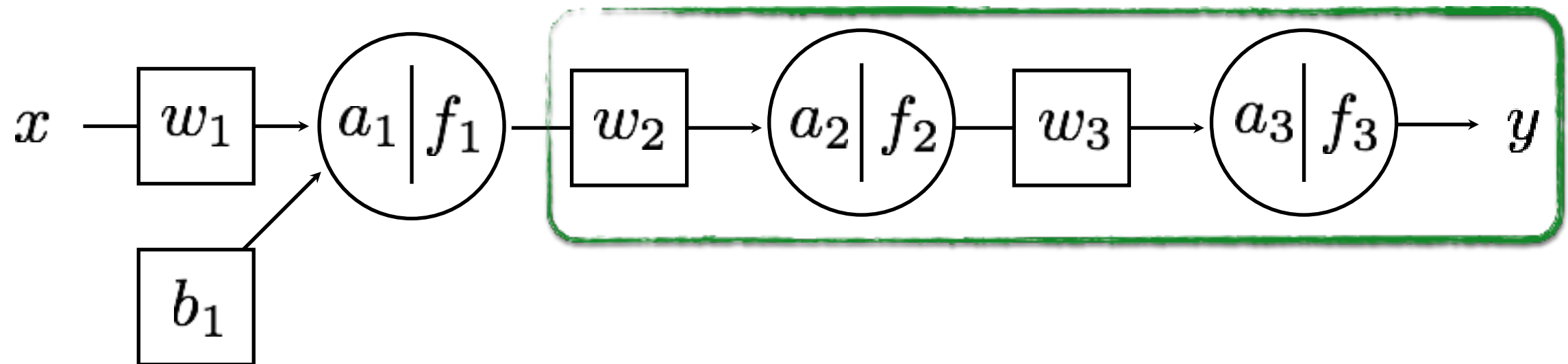
$$\frac{ds(x)}{dx} = s(x)(1 - s(x))$$



$$\begin{aligned}
 \frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \\
 &= -\eta(y - \hat{y}) \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \\
 &= -\eta(y - \hat{y}) f_3(1 - f_3) \frac{\partial a_3}{\partial w_3} \\
 &= -\eta(y - \hat{y}) f_3(1 - f_3) f_2
 \end{aligned}$$



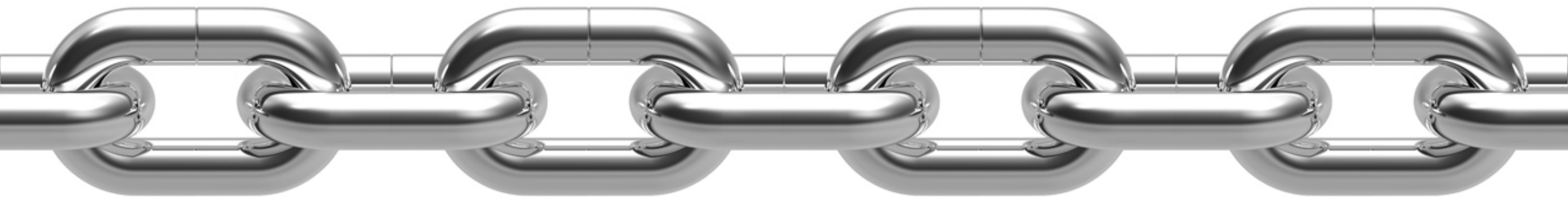
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial w_2}$$



$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial w_2}$$

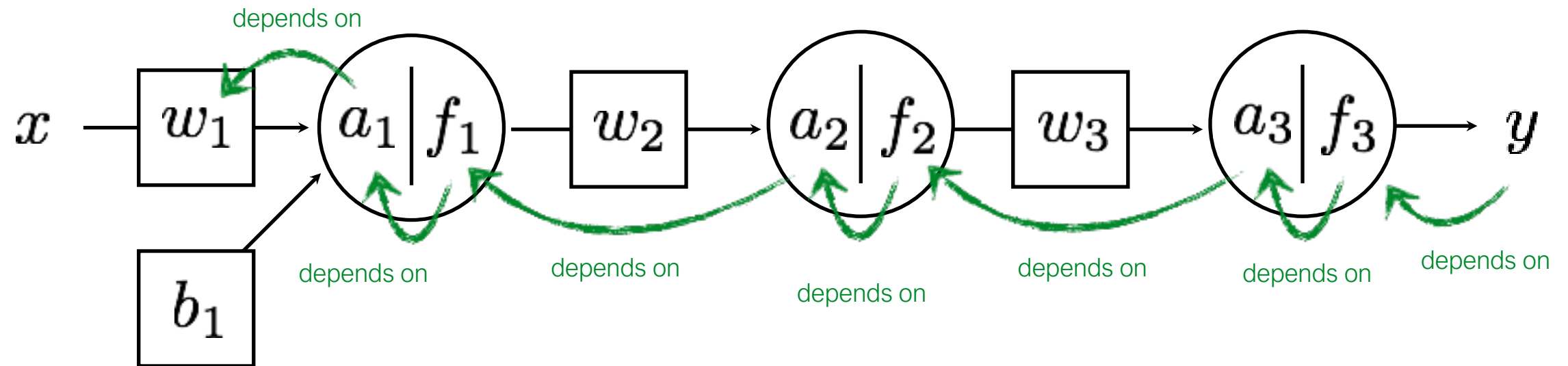
already computed.
re-use (propagate)!

THE CHAIN RULE



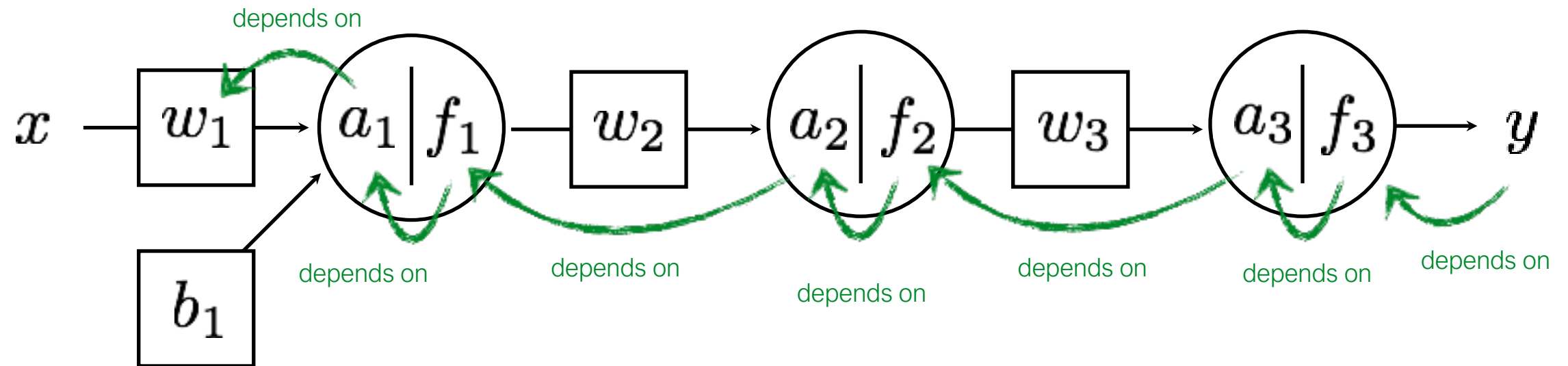
A.K.A. BACKPROPAGATION

The chain rule says...



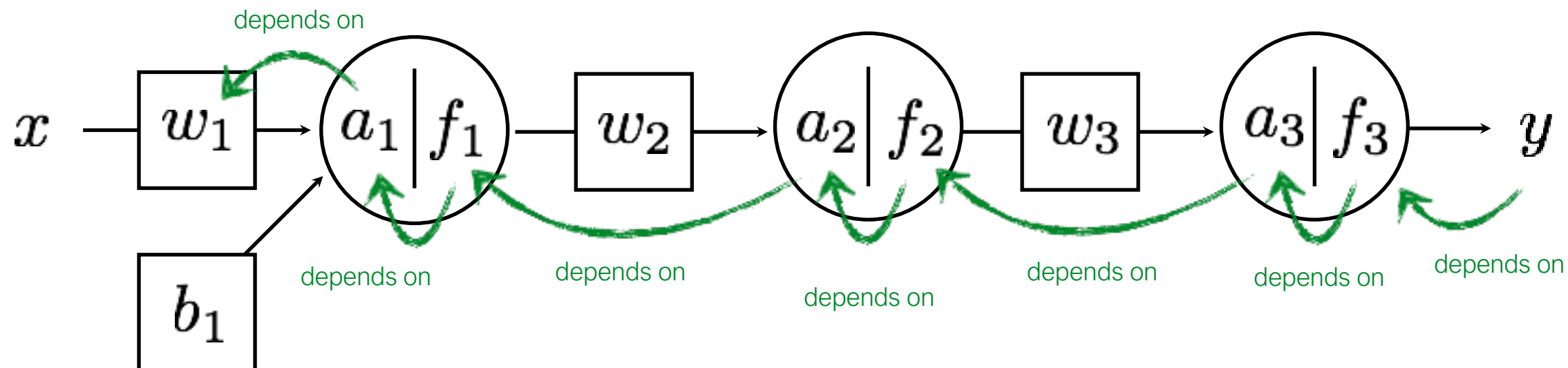
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial w_1}$$

The chain rule says...

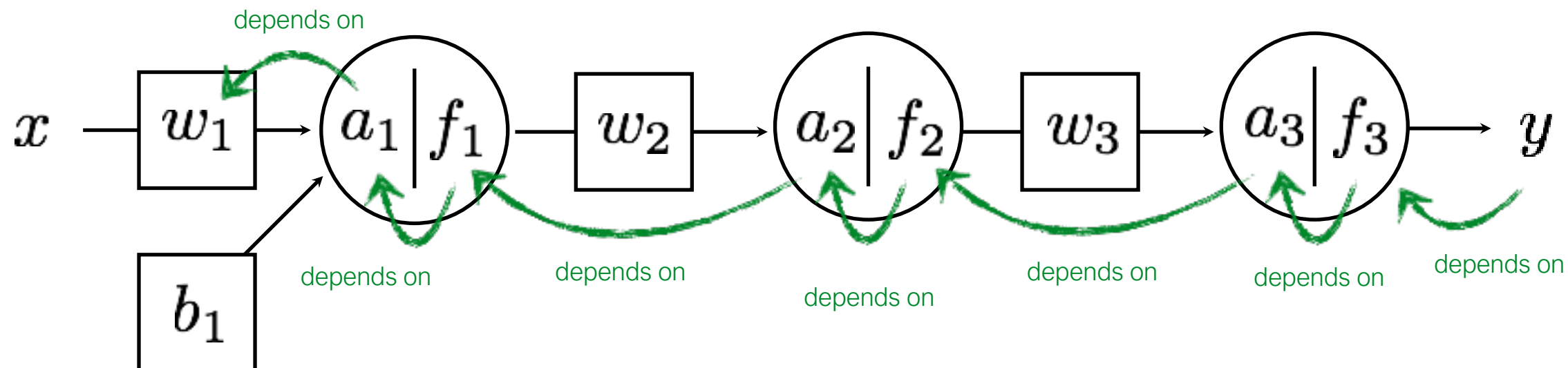


$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial w_1}$$

already computed.
re-use (propagate)!



$$\begin{aligned}
 \frac{\partial \mathcal{L}}{\partial w_3} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \\
 \frac{\partial \mathcal{L}}{\partial w_2} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial w_2} \\
 \frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial w_1} \\
 \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial b}
 \end{aligned}$$

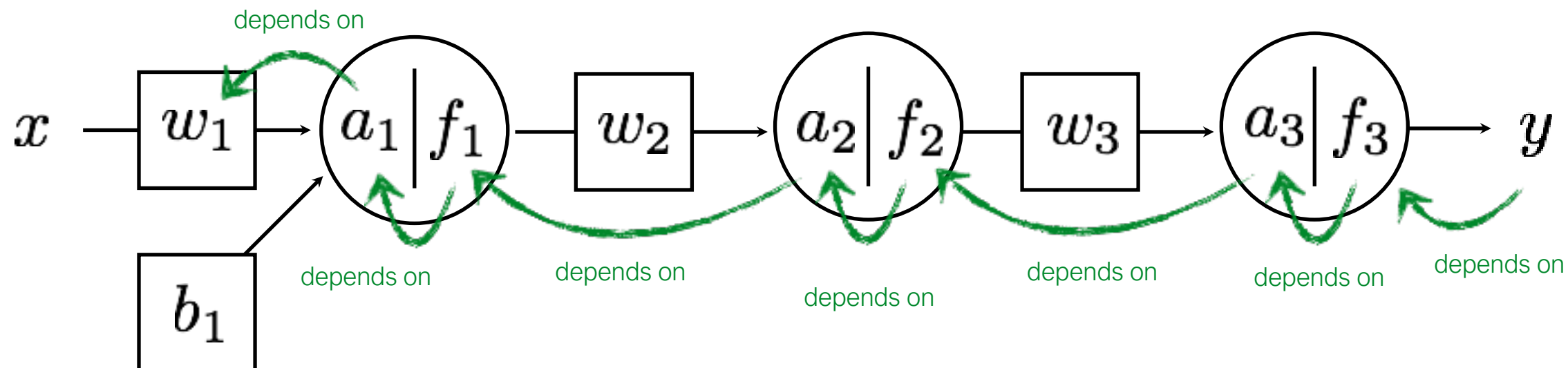


$$\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial w_2}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial w_1}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial b}$$



$$\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial w_2}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial w_1}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial b}$$

Gradient Descent

For each example sample $\{x_i, y_i\}$

1. Predict

a. Forward pass

$$\hat{y} = f_{\text{MLP}}(x_i; \theta)$$

b. Compute Loss

$$\mathcal{L}_i$$

2. Update

a. Back Propagation

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_3} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial w_3} \\ \frac{\partial \mathcal{L}}{\partial w_2} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial w_2} \\ \frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial f_3} \frac{\partial f_3}{\partial a_3} \frac{\partial a_3}{\partial f_2} \frac{\partial f_2}{\partial a_2} \frac{\partial a_2}{\partial f_1} \frac{\partial f_1}{\partial a_1} \frac{\partial a_1}{\partial b}\end{aligned}$$

b. Gradient update

$$w_3 = w_3 - \eta \nabla w_3$$

$$w_2 = w_2 - \eta \nabla w_2$$

$$w_1 = w_1 - \eta \nabla w_1$$

$$b = b - \eta \nabla b$$

Gradient Descent

For each example sample $\{x_i, y_i\}$

1. Predict

a. Forward pass

$$\hat{y} = f_{\text{MLP}}(x_i; \theta)$$

b. Compute Loss

$$\mathcal{L}_i$$

2. Update

a. Back Propagation

$$\frac{\partial \mathcal{L}}{\partial \theta}$$

vector of parameter partial derivatives

b. Gradient update

$$\theta \leftarrow \theta + \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

vector of parameter update equations

Stochastic gradient descent

What we are truly minimizing:

$$\min_{\theta} \sum_{i=1}^N L(y_i, f_{MLP}(x_i))$$

The gradient is:

What we are truly minimizing:

$$\min_{\theta} \sum_{i=1}^N L(y_i, f_{MLP}(x_i))$$

The gradient is:

$$\sum_{i=1}^N \frac{\partial L(y_i, f_{MLP}(x_i))}{\partial \theta}$$

What we use for gradient update is:

What we are truly minimizing:

$$\min_{\theta} \sum_{i=1}^N L(y_i, f_{MLP}(x_i))$$

The gradient is:

$$\sum_{i=1}^N \frac{\partial L(y_i, f_{MLP}(x_i))}{\partial \theta}$$

What we use for gradient update is:

$$\frac{\partial L(y_i, f_{MLP}(x_i))}{\partial \theta} \quad \text{for some } i$$

Stochastic Gradient Descent

For each example sample $\{x_i, y_i\}$

1. Predict

a. Forward pass

$$\hat{y} = f_{\text{MLP}}(x_i; \theta)$$

b. Compute Loss

$$\mathcal{L}_i$$

2. Update

a. Back Propagation

$$\frac{\partial \mathcal{L}}{\partial \theta}$$

vector of parameter partial derivatives

b. Gradient update

$$\theta \leftarrow \theta + \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

vector of parameter update equations

How do we select which sample?

How do we select which sample?

- Select randomly!

Do we need to use only one sample?

How do we select which sample?

- Select randomly!

Do we need to use only one sample?

- You can use a *minibatch* of size $B < N$.

Why not do gradient descent with all samples?

How do we select which sample?

- Select randomly!

Do we need to use only one sample?

- You can use a *minibatch* of size $B < N$.

Why not do gradient descent with all samples?

- It's very expensive when N is large (big data).

Do I lose anything by using stochastic GD?

How do we select which sample?

- Select randomly!

Do we need to use only one sample?

- You can use a *minibatch* of size $B < N$.

Why not do gradient descent with all samples?

- It's very expensive when N is large (big data).

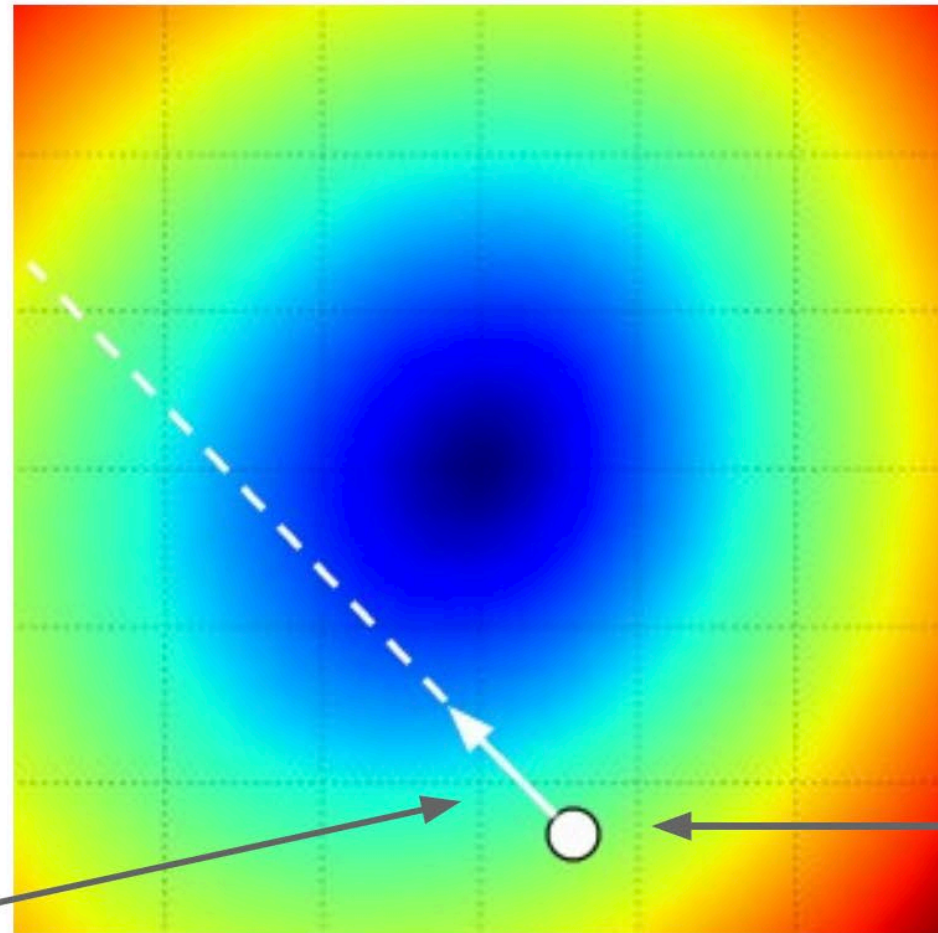
Do I lose anything by using stochastic GD?

- Same convergence guarantees and complexity!
- Better generalization.

Summary

- Always use mini-batch gradient descent
- Incorrectly refer to it as “doing SGD” as everyone else
(or call it batch gradient descent)
- The mini-batch size is a hyperparameter, but it is not very common to cross-validate over it (usually based on practical concerns, e.g. space/time efficiency)

Learning rates



original θ

negative gradient direction

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

Step size: learning rate

Too big: will miss the minimum

Too small: slow convergence

Learning rate scheduling

- Use different learning rate at each iteration.
- Most common choice:

$$\eta_t = \frac{\eta_0}{\sqrt{t}}$$

Need to select initial learning rate η_0 , important!!!

- More modern choice: Adaptive learning rates.

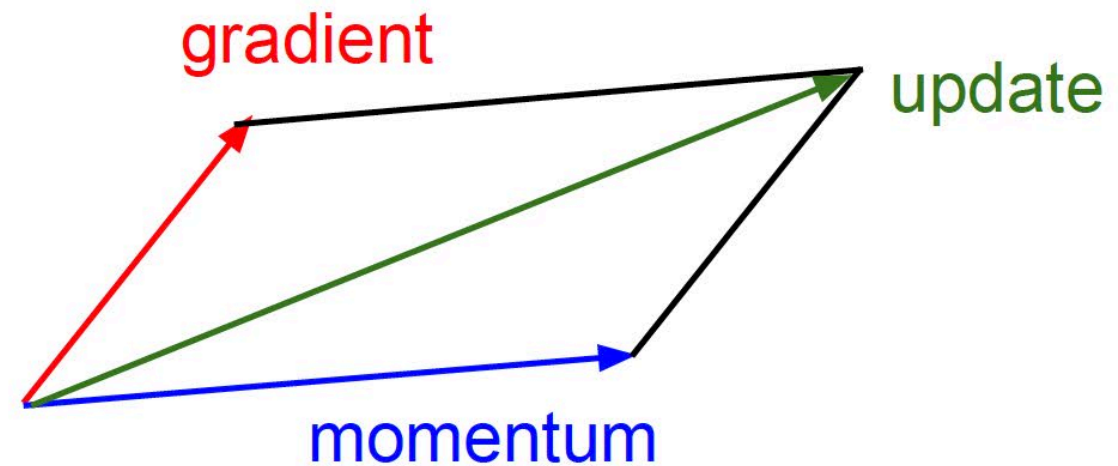
$$\eta_t = G \left(\left\{ \frac{\partial L}{\partial \theta} \right\}_{i=0}^t \right)$$

Many choices for G (Adam, Adagrad, Adadelata).

Momentum Update

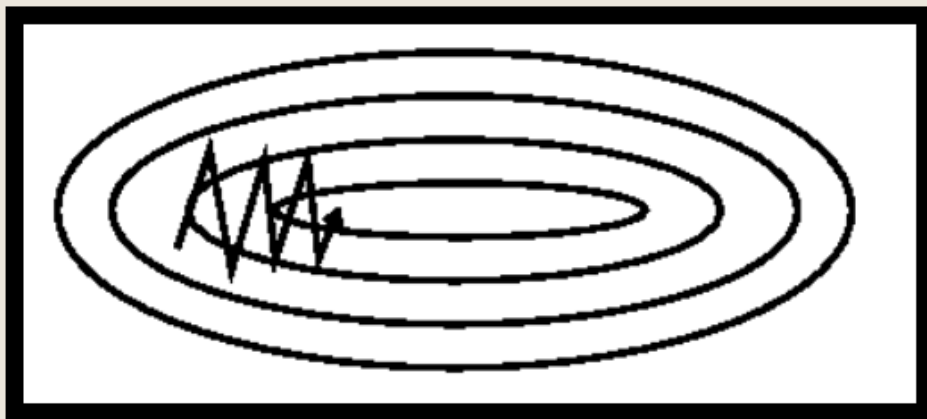
$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

$$\Delta\theta \leftarrow w \frac{\partial L}{\partial \theta} + (1 - w)\Delta\theta$$

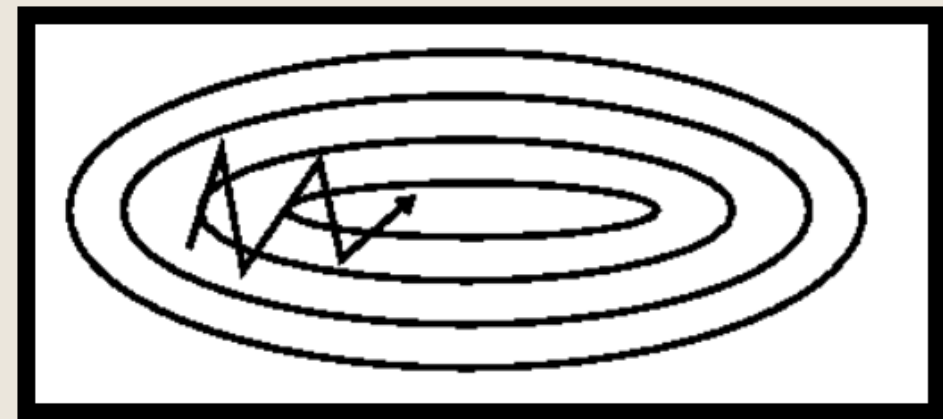


Take direction history into account!

```
weights_grad = evaluate_gradient(loss_fun, data, weights)
vel = vel * 0.9 - step_size * weights_grad
weights += vel
```



(Fig. 2a)



(Fig. 2b)

Many other ways to perform optimization...

- Second order methods that use the Hessian (or its approximation): BFGS, **LBFGS**, etc.
- Currently, the lesson from the trenches is that well-tuned SGD+Momentum is very hard to beat for CNNs.
- No consensus on Adam etc.: Seem to give faster performance to worse local minima.

Derivatives

- Given $f(x)$, where x is vector of inputs
 - Compute gradient of f at x : $\nabla f(x)$

How do we do differentiation?

Numerical differentiation

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x+h) = f(x) + h \frac{df(x)}{dx}$$

Numerical differentiation

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x+h) = f(x) + h \frac{df(x)}{dx}$$

Numerical differentiation is:

- Approximate.
- Slow.
- Numerically unstable.
- Easy to write.

Symbolic differentiation

- What Mathematica does: Automatically derive *analytical* expressions for derivative.
- Often results in very redundant (and expensive to evaluate) expressions.

```
D[Log[1 + Exp[w * x + b]], w]
```

Out[11]=
$$\frac{e^{b+wx} w}{1 + e^{b+wx}}$$

```
In[19]:= D[Log[1 + Exp[w2 * Log[1 + Exp[w1 * x + b1]] + b2]], w1]
```

Out[19]=
$$\frac{e^{b_1+b_2+w_1 x+w_2 \operatorname{Log}\left[1+e^{b_1+w_1 x}\right]} w_2 x}{\left(1+e^{b_1+w_1 x}\right)\left(1+e^{b_2+w_2 \operatorname{Log}\left[1+e^{b_1+w_1 x}\right]}\right)}$$

- Often intractable.

Automatic differentiation (autodiff)

- An autodiff system will convert the program into a sequence of **primitive operations** which have specified routines for computing derivatives.
- In this representation, backprop can be done in a completely mechanical way.

Sequence of primitive operations:

Original program:

$$z = wx + b$$

$$y = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

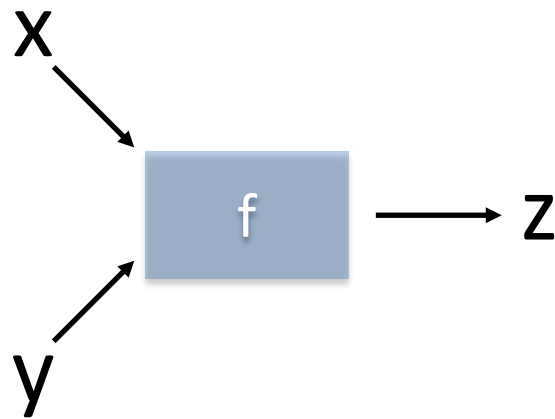
$$y = 1/t_5$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$\mathcal{L} = t_7/2$$

Gamechanger: autograd



```
import torch

class MultiplyFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
        return x * y

    @staticmethod
    def backward(ctx, grad_output):
        a, b = ctx.saved_tensors
        grad_x = grad_output * y
        grad_y = grad_output * x
        return grad_x, grad_y

# Usage
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)
z = MultiplyFunction.apply(x, y)
z.backward()

print(x)      # Output: tensor(2., requires_grad=True)
print(y)      # Output: tensor(3., requires_grad=True)
print(z)      # Output: tensor(6., grad_fn=<MultiplyFunctionBackward>)
print(x.grad) # Output: tensor(3.)
print(y.grad) # Output: tensor(2.)
```

← Need to store dz/dx and dx/dy

← Kickoff backprop (or feed in $dLoss/dz$ from downstream loss)

Forward and backward functions for “ $z = x*y$ ”

MLPs with autodiff (PyTorch)

```
# 4. Define the MLP model
class MLP(nn.Module):
    def __init__(self, inputDim, hiddenDim, numClass):
        super(MLP, self).__init__()
        self.W1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.W2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.W1(x)
        out = self.relu(out)
        out = self.W2(out)
        return out

# Instantiate the model
model = MLP(inputDim, hiddenDim, numClass)

# 5. Define the loss function and optimizer
criterion = nn.CrossEntropyLoss() # Operate on logits
optimizer = optim.SGD(model.parameters(), lr)
```

```
# 6. Training loop
for epoch in range(num_epochs):
    for images, labels in train_loader:

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

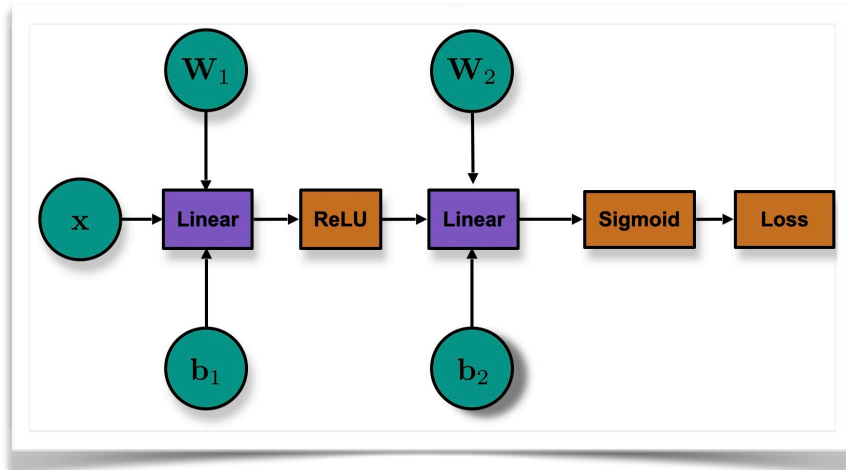
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Loss: {loss.item():.4f}")
```

```
# 7. Test
with torch.no_grad():
    # Grab next batch of test image
    images, labels = next(iter(test_loader))

    # Make predictions
    outputs = model(images)
    _, predicted = torch.max(outputs, 1)
```

Explicit vs autograd



```
# --- Training Loop (SGD) ---
for _ in range(n_epochs):
    loss = 0
    indices = np.random.permutation(len(X_train))
    for i in indices:
        x = X_train[i]      # (64,)
        y_true = y_train[i] # (10,)

        # Forward pass
        z1 = W1 @ x          # (32,)
        h1 = np.maximum(0, z1) # (32,)
        z2 = W2 @ h1         # (10,)
        y_pred = softmax(z2) # (10,)

        # Backward pass
        dz2 = y_pred - y_true # (10,)
        dW2 = np.outer(dz2, h1) # (10,32)
        dh1 = W2.T @ dz2       # (32,)
        dz1 = dh1 * (z1 > 0)   # (32,)
        dW1 = np.outer(dz1, x) # (32,64)

        # Update weights and cross-entropy loss
        W2 -= lr * dW2
        W1 -= lr * dW1
        loss += -np.sum(y_true * np.log(y_pred + 1e-9))

    loss /= len(X_train)
    print(f"Loss: {loss:.4f}")
```

Explicit gradients (NumPy)

```
# 4. Define the MLP model
class MLP(nn.Module):
    def __init__(self, inputDim, hiddenDim, numClass):
        super(MLP, self).__init__()
        self.W1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.W2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.W1(x)
        out = self.relu(out)
        out = self.W2(out)
        return out

# Instantiate the model
model = MLP(inputDim, hiddenDim, numClass)

# 5. Define the loss function and optimizer
criterion = nn.CrossEntropyLoss() # Operate on logits
optimizer = optim.SGD(model.parameters(), lr)
```

```
# 6. Training loop
for epoch in range(num_epochs):
    for images, labels in train_loader:

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Loss: {loss.item():.4f}")
```

Autograd (PyTorch)

Aside: do we really need to store intermediate activations for backprop?

Training Deep Nets with Sublinear Memory Cost

Tianqi Chen¹, Bing Xu², Chiyuan Zhang³, and Carlos Guestrin¹

¹ University of Washington ² Dato, Inc ³ Massachusetts Institute of Technology

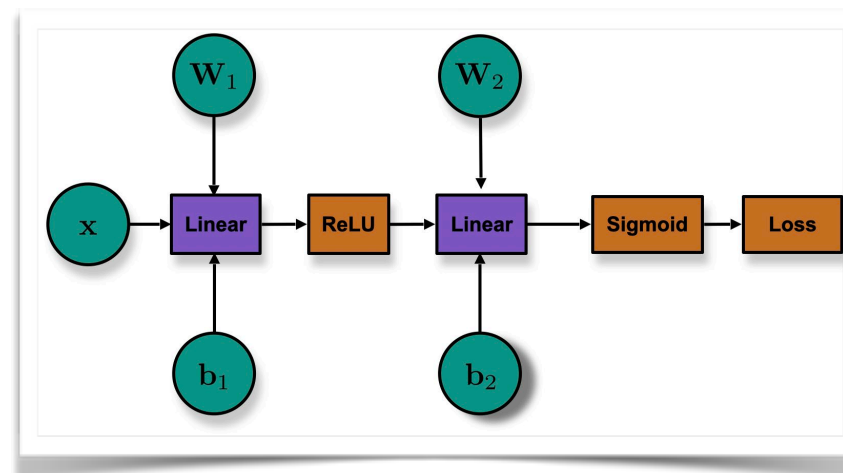
```
# --- Training Loop (SGD) ---
for _ in range(n_epochs):
    loss = 0
    indices = np.random.permutation(len(X_train))
    for i in indices:
        x = X_train[i]          # (64,)
        y_true = y_train[i]     # (10,)

        # Forward pass
        z1 = W1 @ x              # (32,)
        h1 = np.maximum(0, z1)    # (32,)
        z2 = W2 @ h1             # (10,)
        y_pred = softmax(z2)     # (10,)

        # Backward pass
        dz2 = y_pred - y_true     # (10,)
        dW2 = np.outer(dz2, h1)   # (10,32)
        dh1 = W2.T @ dz2         # (32,)
        dz1 = dh1 * (z1 > 0)      # (32,)
        dW1 = np.outer(dz1, x)    # (32,64)

        # Update weights and cross-entropy loss
        W2 -= lr * dW2
        W1 -= lr * dW1
        loss += -np.sum(y_true * np.log(y_pred + 1e-9))

    loss /= len(X_train)
    print(f"Loss: {loss:.4f}")
```



Gradient checkpointing:

don't need to store h_1 ; just store the *checkpoint* z_1 and compute $\max(0, z_1)$ *on-the-fly* (trades off memory for compute)

In summary

- Numerical gradient: easy to implement, bad to use.
- Symbolic gradient: sometimes useful, often intractable.
- Automatic gradient: exact, fast, error-prone.

In practice: Use symbolic gradient for small/trivial programs. Almost always use analytic gradient, but check correctness of implementation with numerical gradient.

- This is called a gradient check.

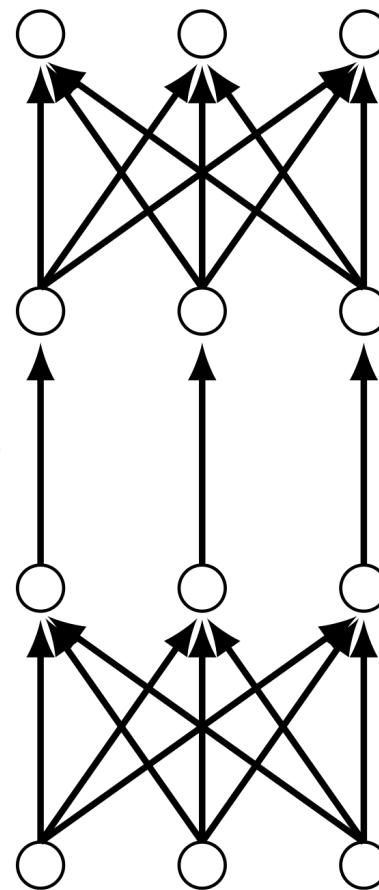
Convolutional Neural Networks

Limitations of MLPs

linear comb. of neurons ▷

neuron-wise nonlinearity ▷

linear comb. of neurons ▷



+ Universal

+ Simple (elegant theory)

+ Embarrassingly parallel

- Weak inductive biases

- Sample inefficient / data hungry

- Dense (fully-connected) linear layers take a lot of compute

CNNs in 2012: “SuperVision” (aka “AlexNet”)

“AlexNet” — Won the ILSVRC2012 Challenge

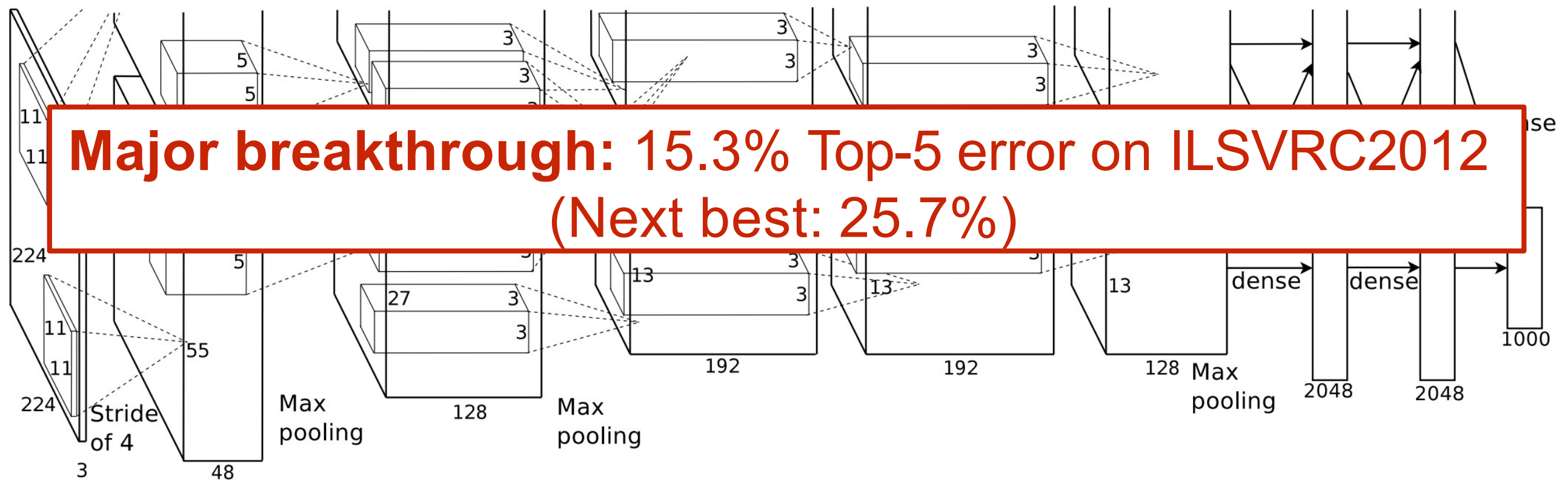
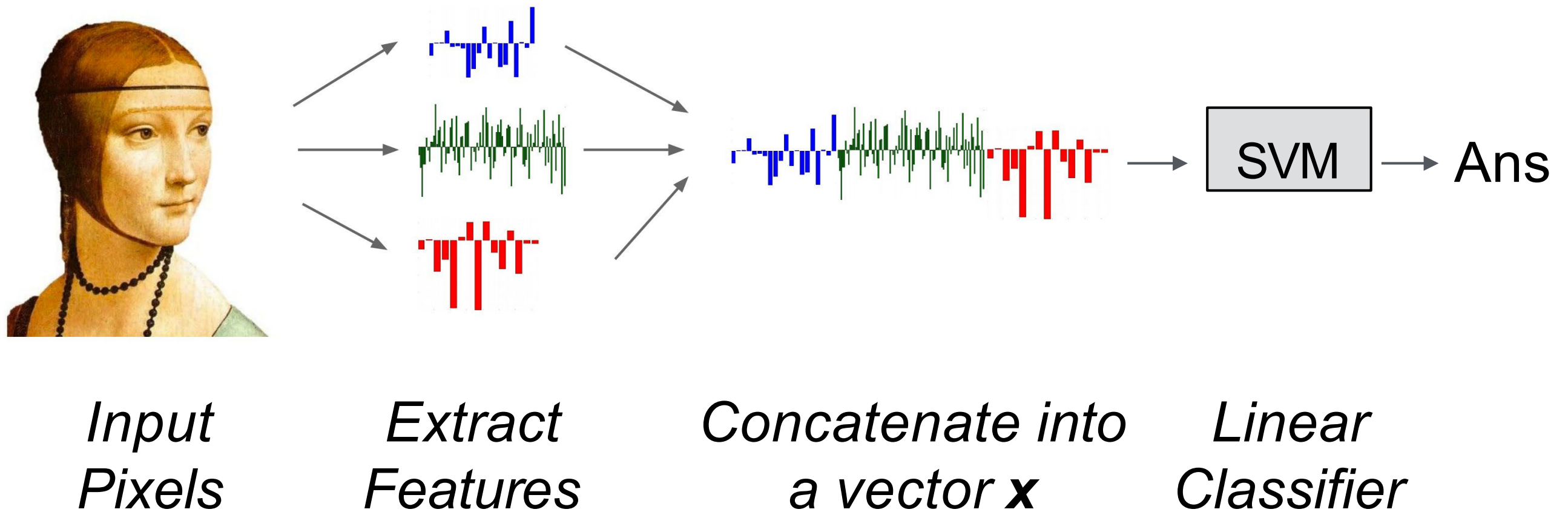


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

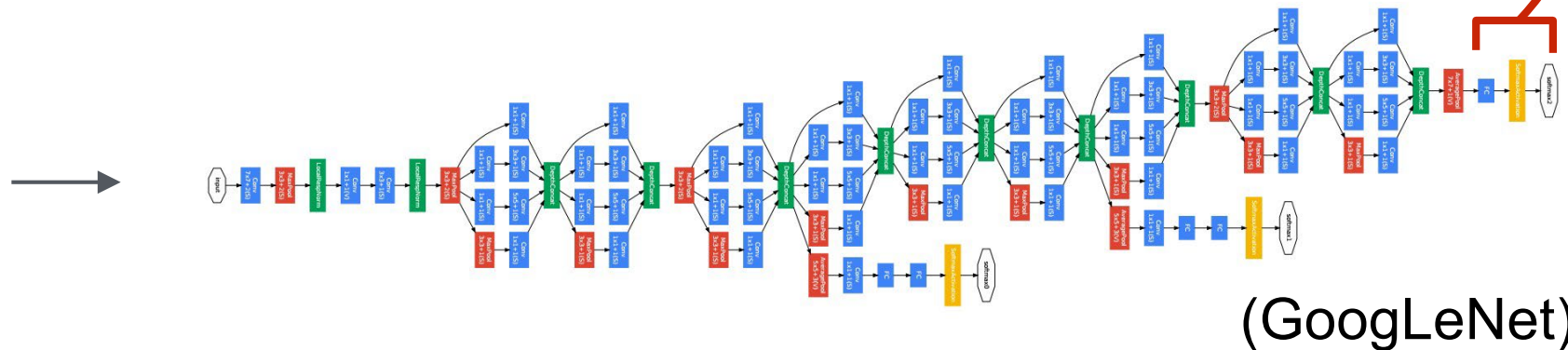
[Krizhevsky, Sutskever, Hinton. NIPS 2012]

Recap: Before Deep Learning



The last layer of (most) CNNs are linear classifiers

This piece is just a linear classifier



→ Ans

*Input
Pixels*

*Perform everything with a big neural
network, trained end-to-end*

Key: perform enough processing so that by the time you get to the end of the network, the classes are linearly separable

ConvNets

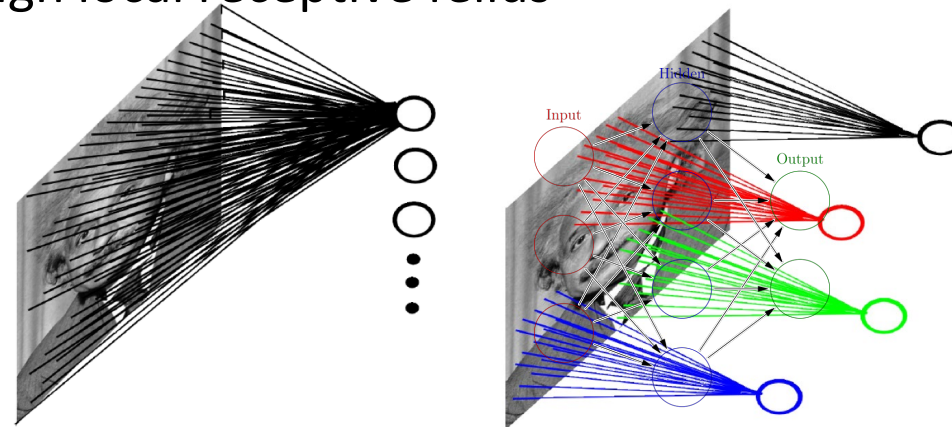
They're just neural networks with
3D activations and weight sharing

What's the core difference between a standard "neural network" (MLP) and a "convolutional neural network" (CNN)?

Given 1000x1000 ($1e6$) input image with 1M ($1e6$) hidden activations, what's size of weight matrix? $1e12$ (1 trillion) weights

[Aside: I will use the terms "(hidden) layers", "neurons", "activation maps" interchangeably]

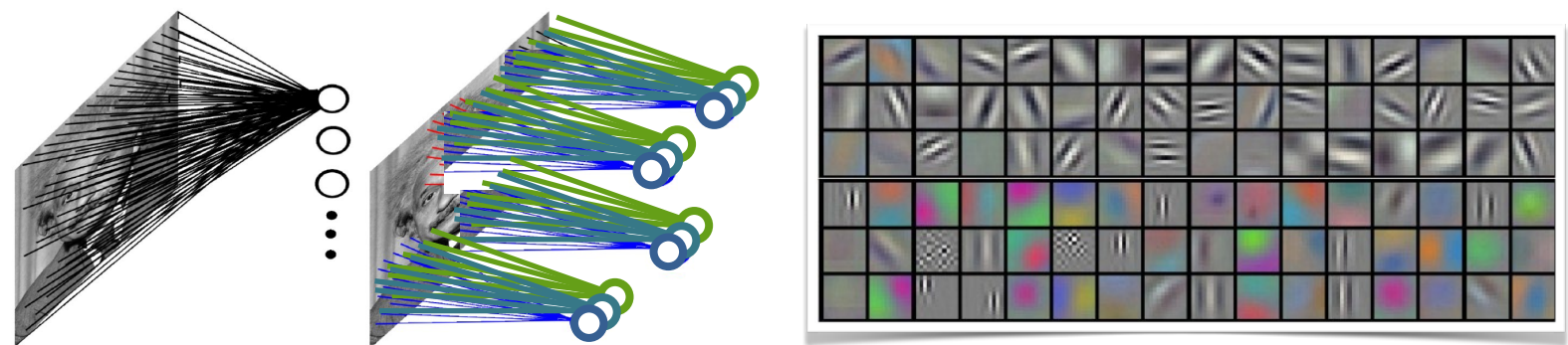
1. Sparsity through local receptive fields



Given 1000x1000 input image with 1M hidden neurons with 10×10 receptive fields, what's the size of weight matrix?

100M weights

2. Weight sharing across locations

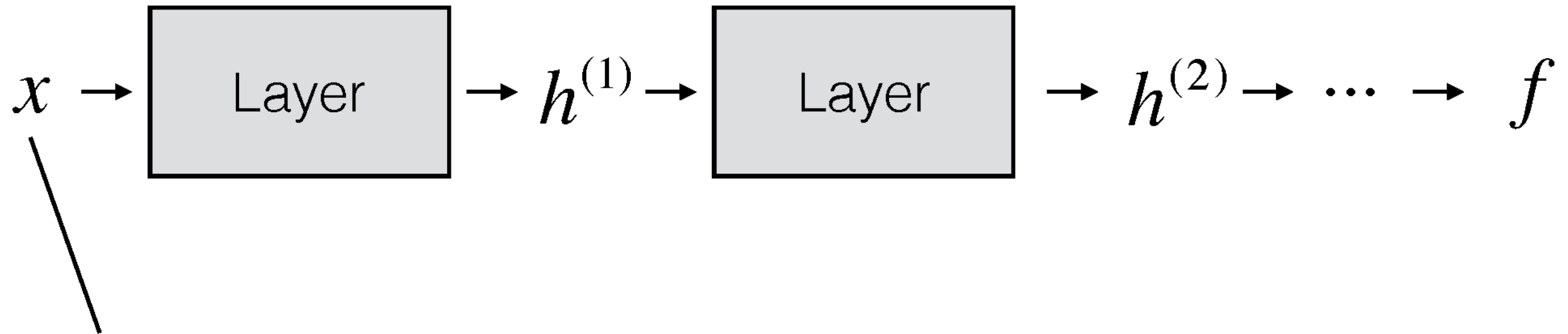


Given 1000x1000 input image with 1M hidden neurons with 10×10 receptive fields *with shared parameters*, what's the size of weight matrix?

Would be interesting to try (1) without (2), but haven't seen it
(2) seems a bit too limiting. Is there a way add back more parameters?

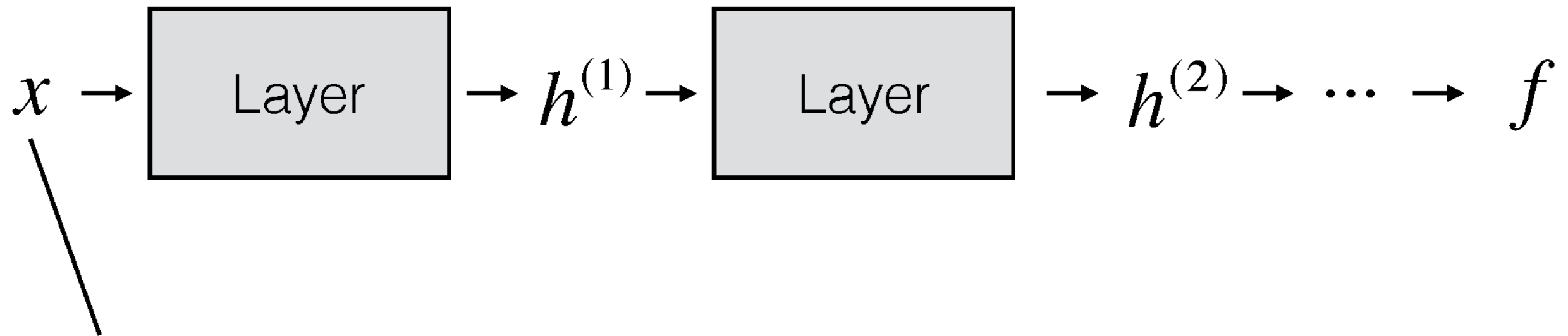
10×10 filter (100 weights)
filter bank

What shape should the activations have?



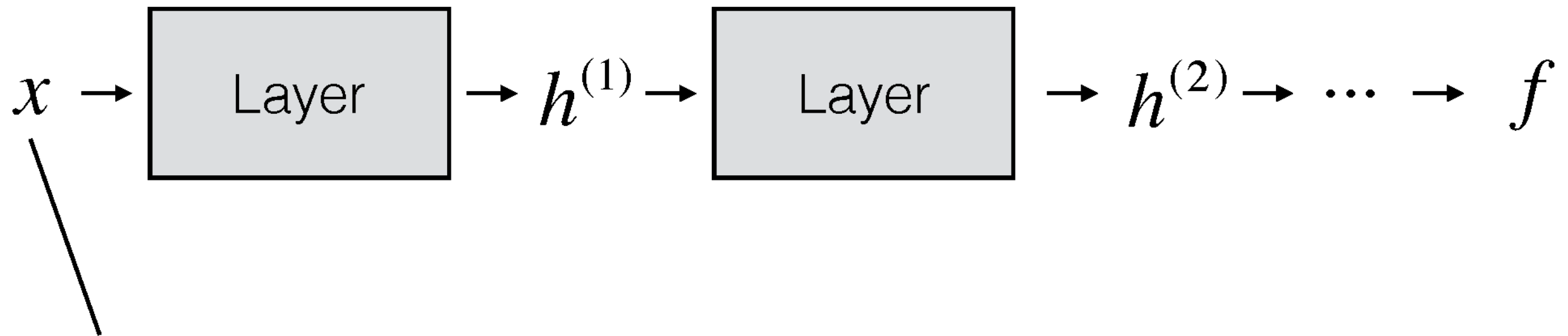
- The input is an image, which is 3D (RGB channel, height, width)

What shape should the activations have?



- The input is an image, which is 3D (RGB channel, height, width)
- We could flatten it to a 1D vector, but then we lose structure

What shape should the activations have?



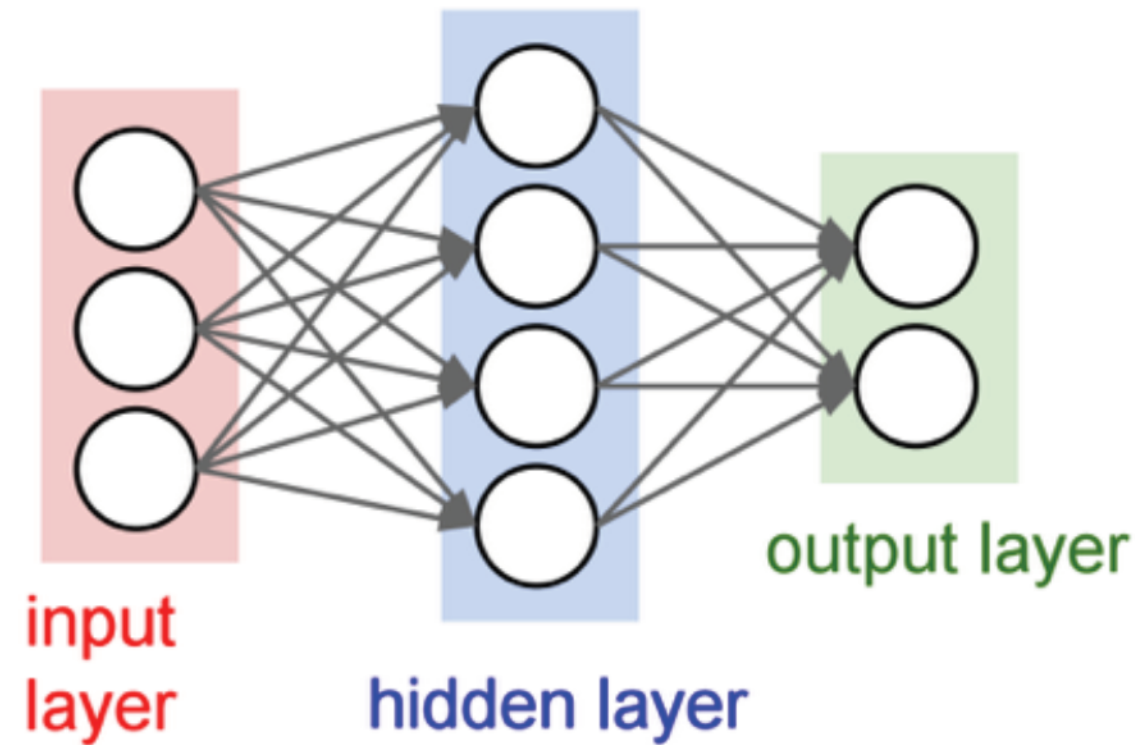
- The input is an image, which is 3D (RGB channel, height, width)
- We could flatten it to a 1D vector, but then we lose structure
- What about keeping everything in 3D?

ConvNets

They're just neural networks with
3D activations and weight sharing

3D Activations

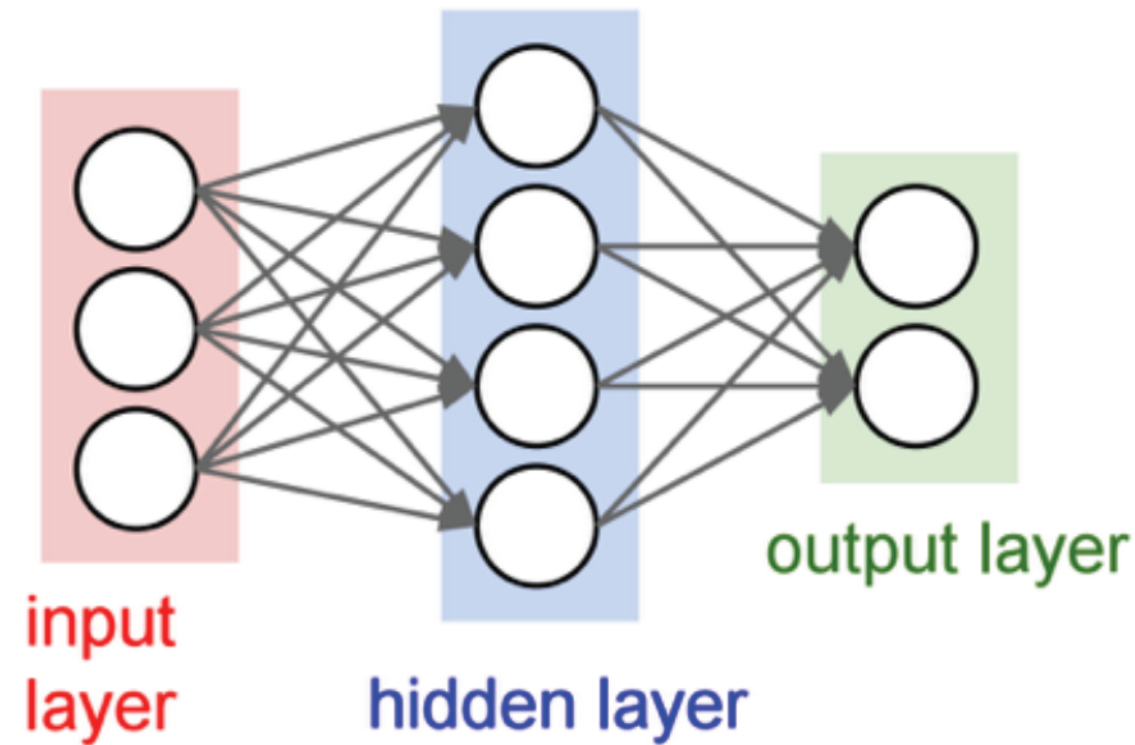
before:



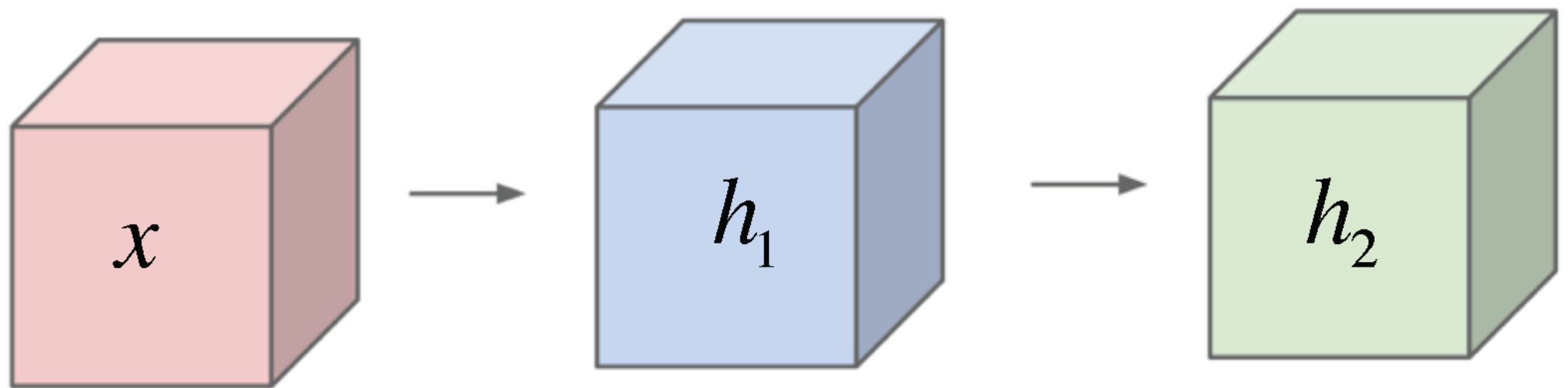
(1D vectors)

3D Activations

before:

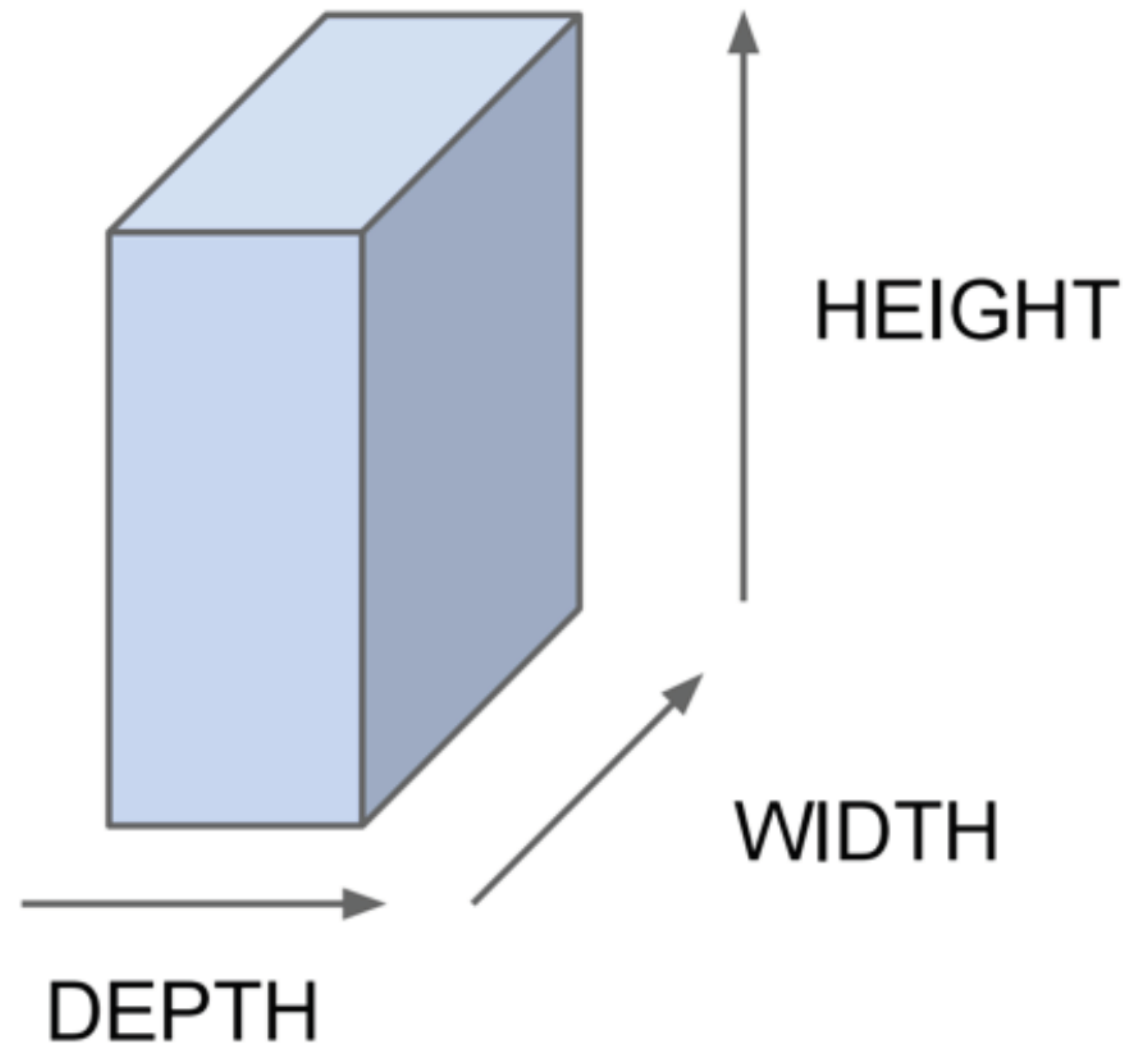


now:



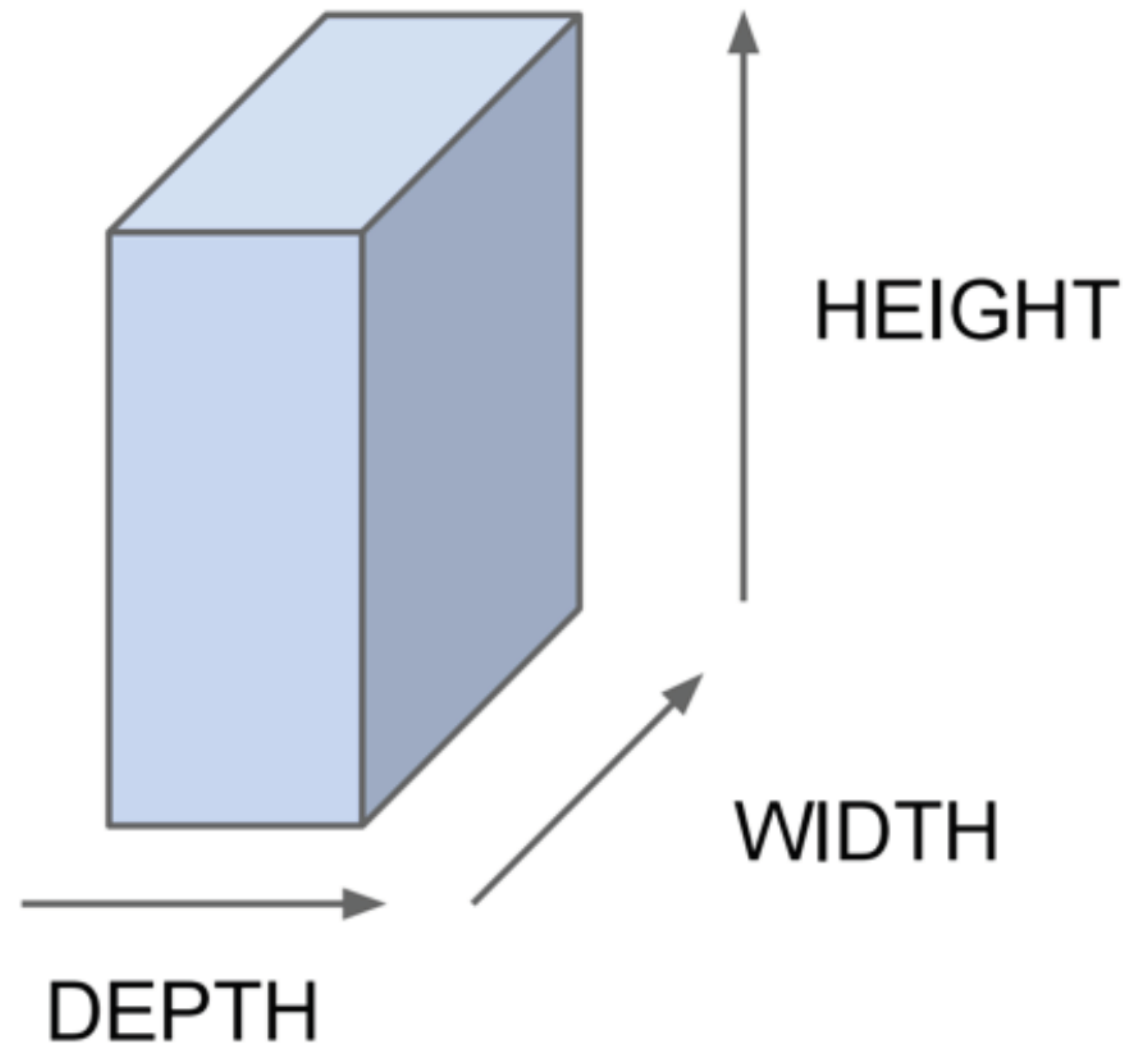
3D Activations

All Neural Net
activations
arranged in **3
dimensions:**



3D Activations

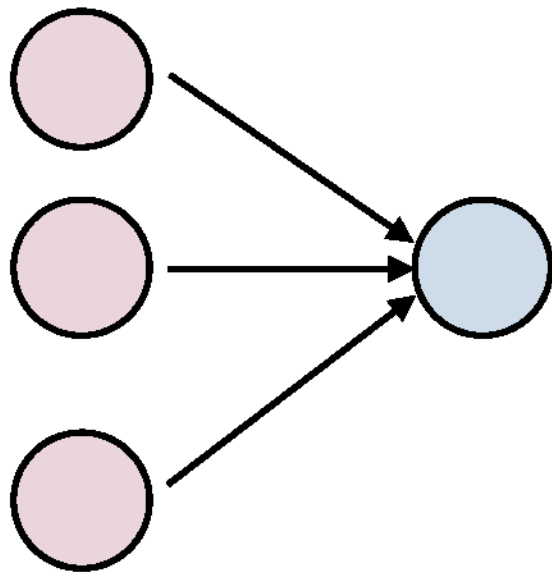
All Neural Net
activations
arranged in **3
dimensions**:



For example, a CIFAR-10 image is a $3 \times 32 \times 32$ volume
(3 depth — RGB channels, 32 height, 32 width)

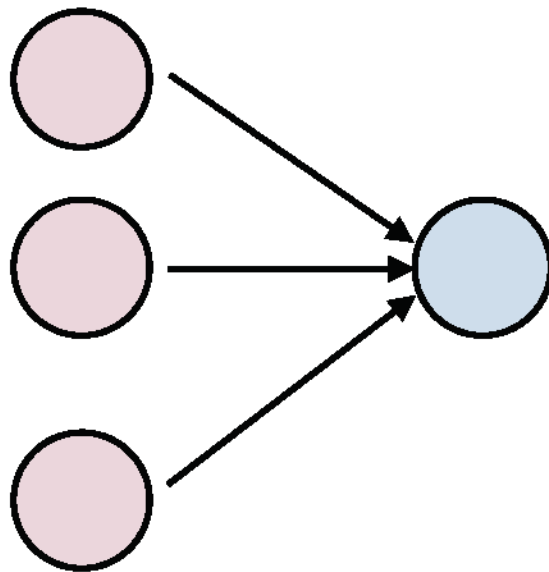
3D Activations

1D Activations:

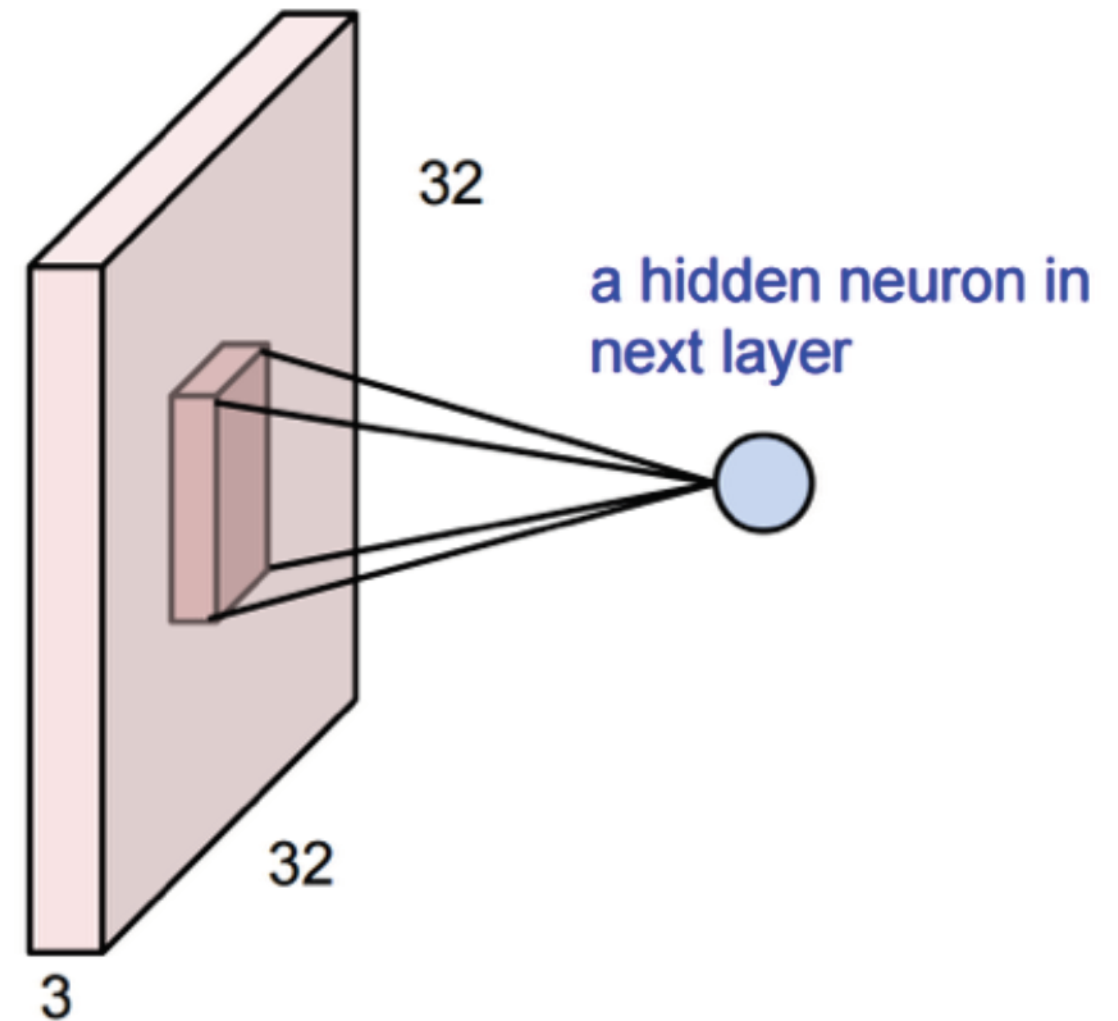


3D Activations

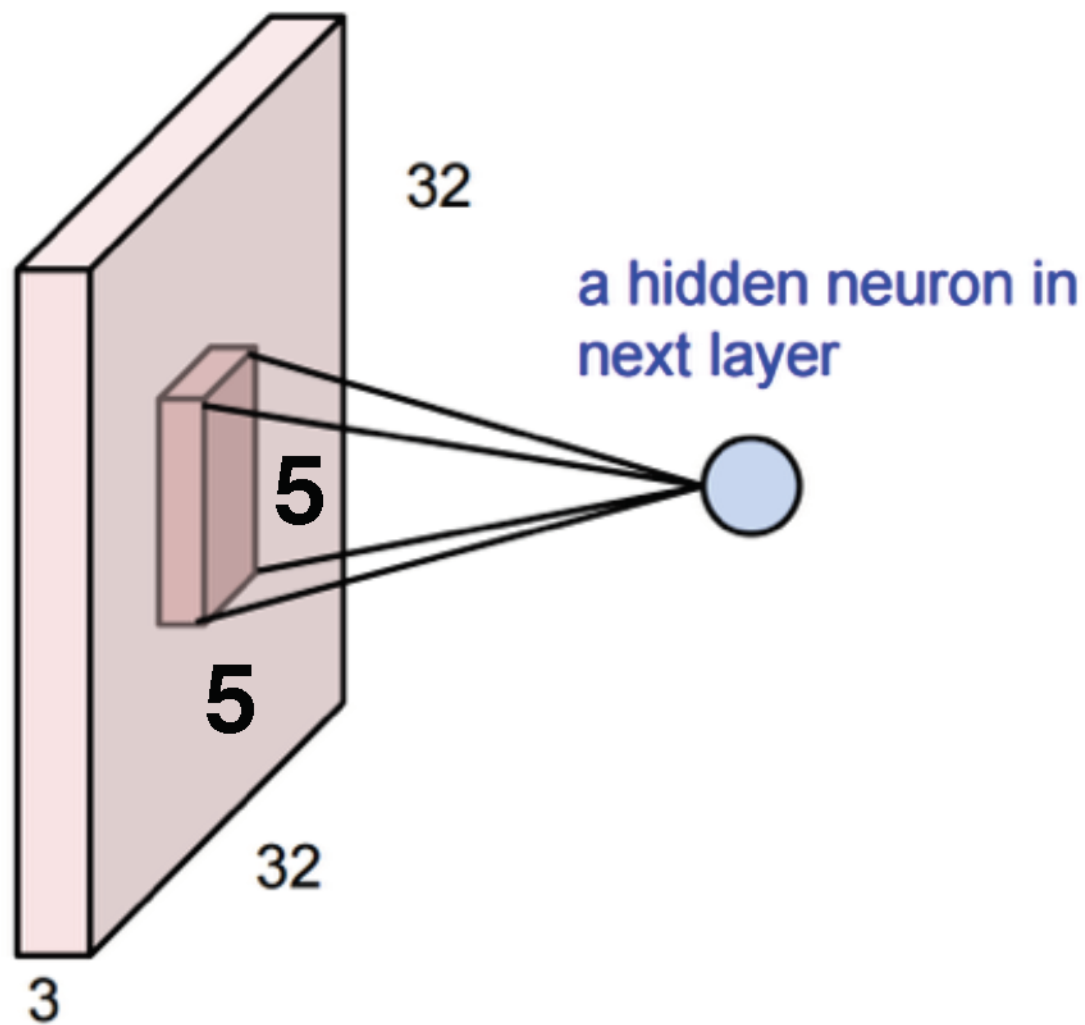
1D Activations:



3D Activations:

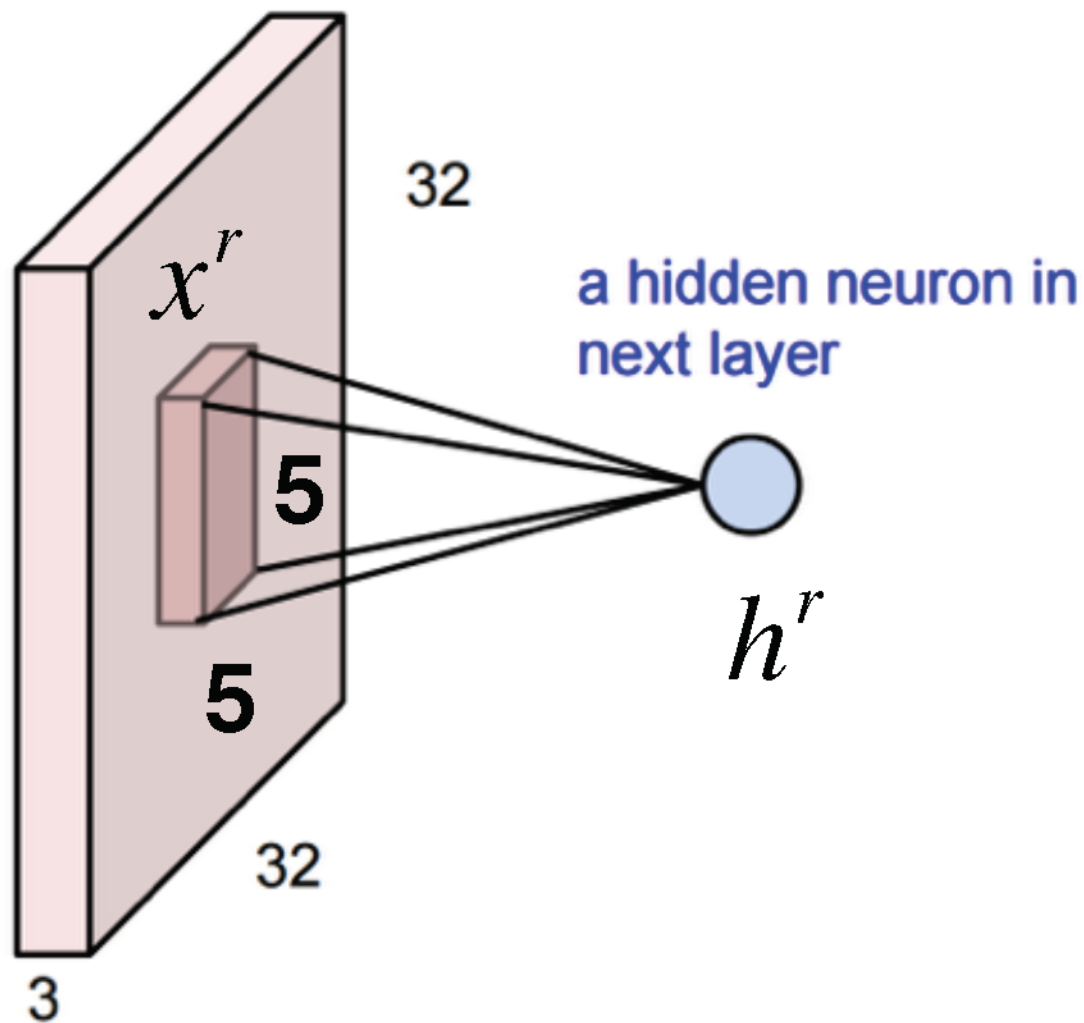


3D Activations



- The input is $3 \times 32 \times 32$
- This neuron depends on a $3 \times 5 \times 5$ chunk of the input
- The neuron also has a $3 \times 5 \times 5$ set of weights and a bias (scalar)

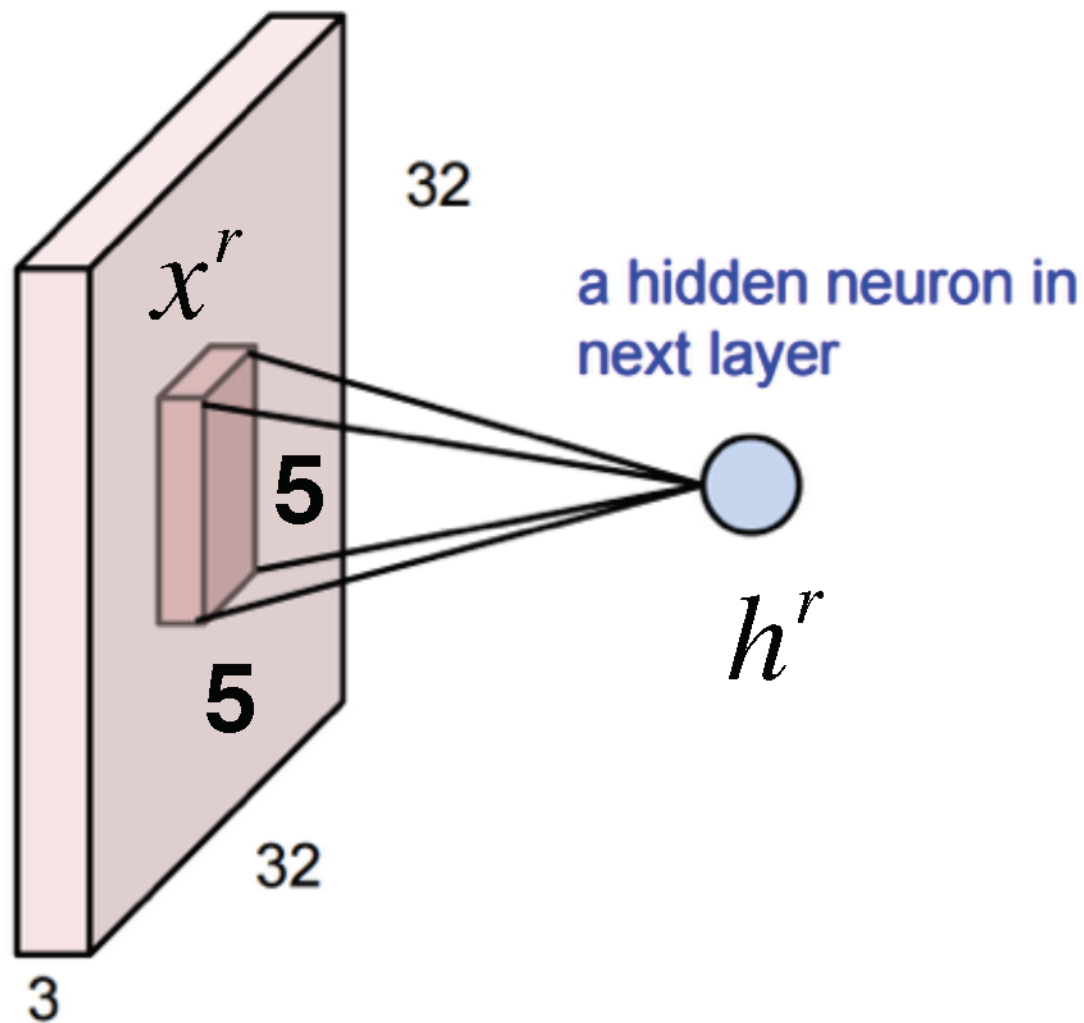
3D Activations



Example: consider the region of the input " x^r "

With output neuron h^r

3D Activations



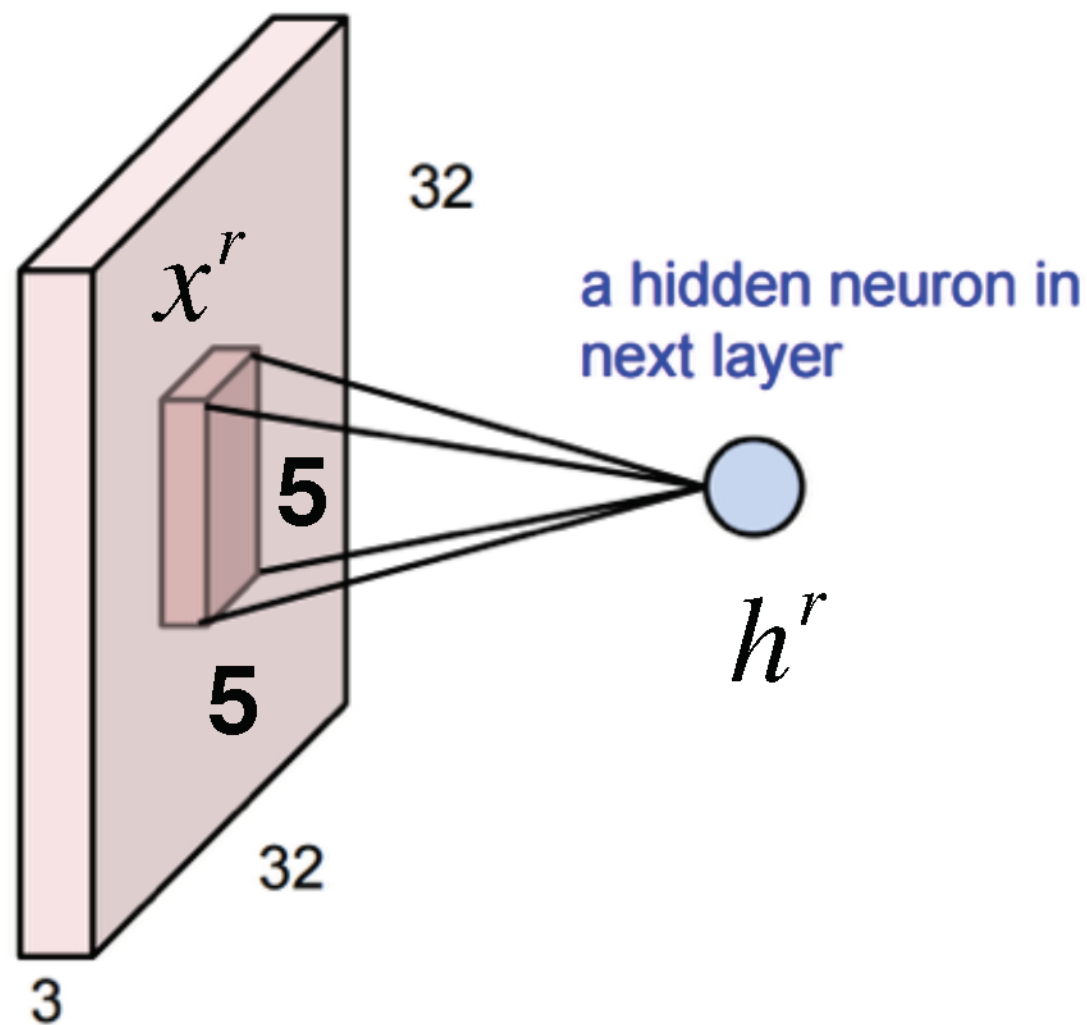
Example: consider the region of the input “ x^r ”

With output neuron h^r

Then the output is:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

3D Activations



Example: consider the region of the input " x^r "

With output neuron h^r

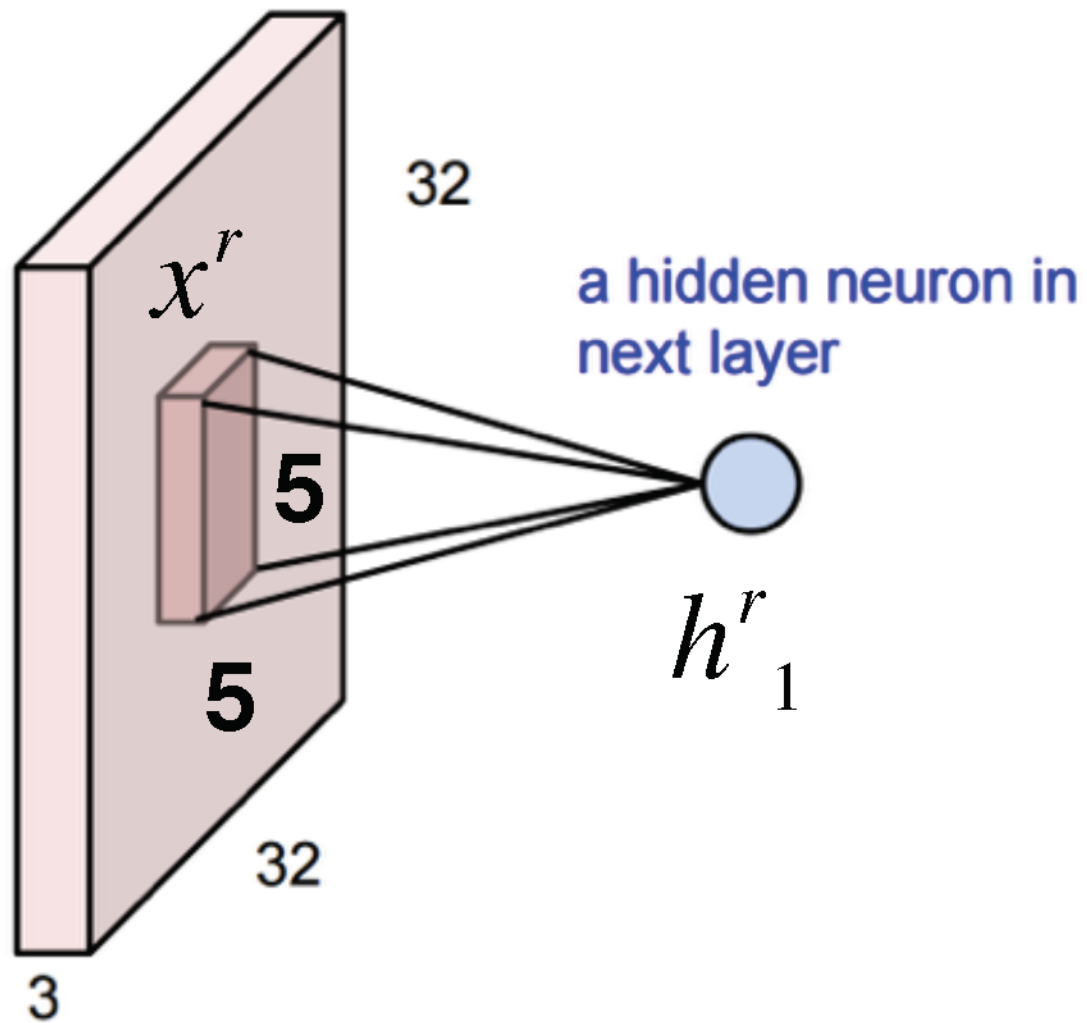
Then the output is:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

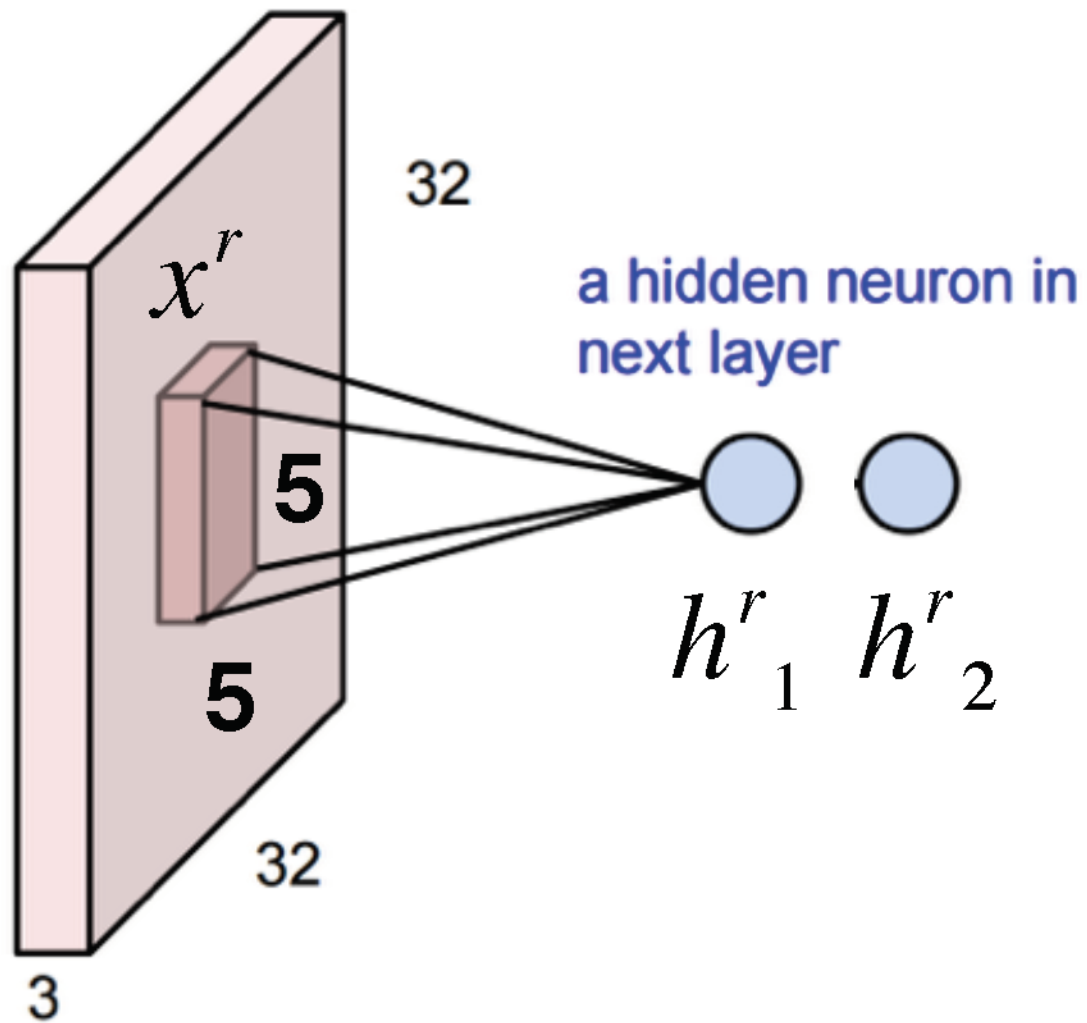


Sum over 3 axes

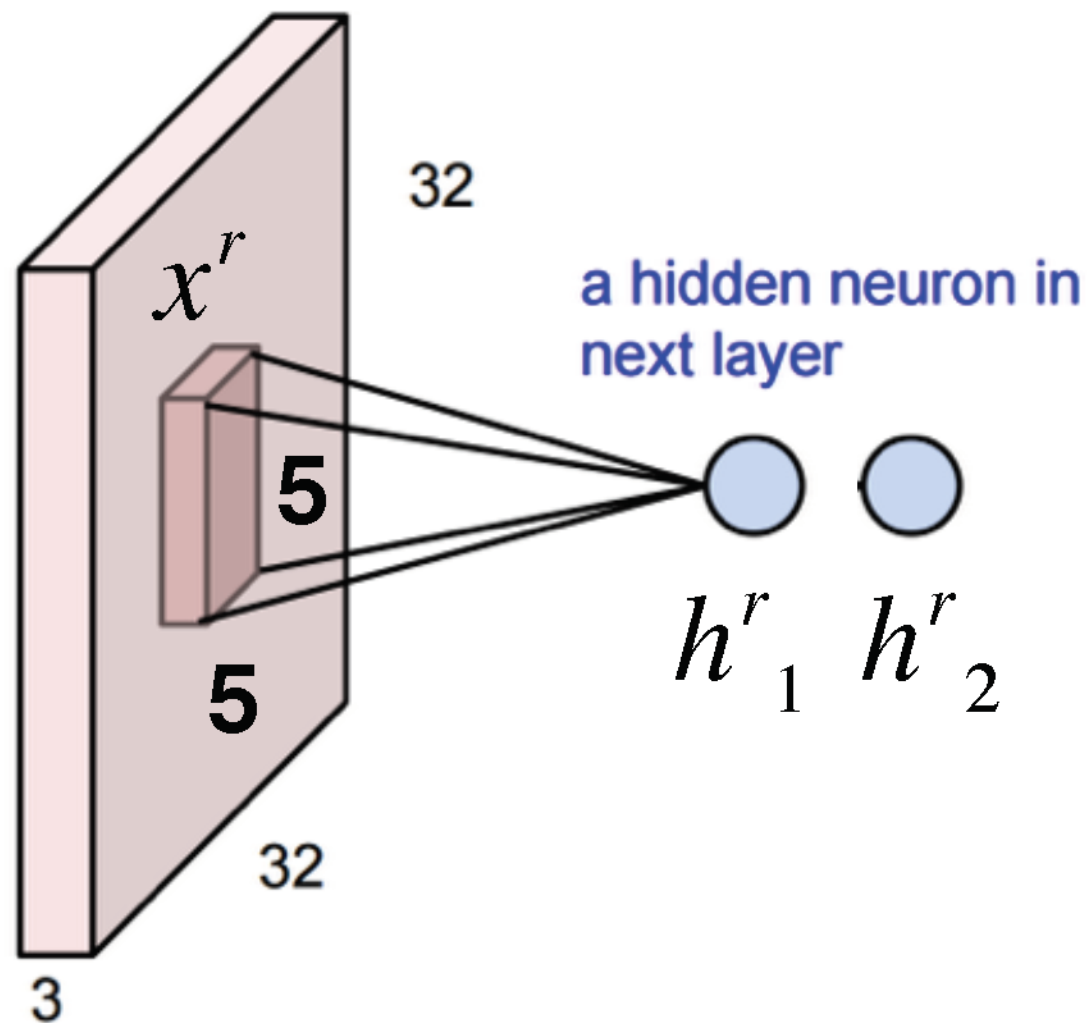
3D Activations



3D Activations



3D Activations

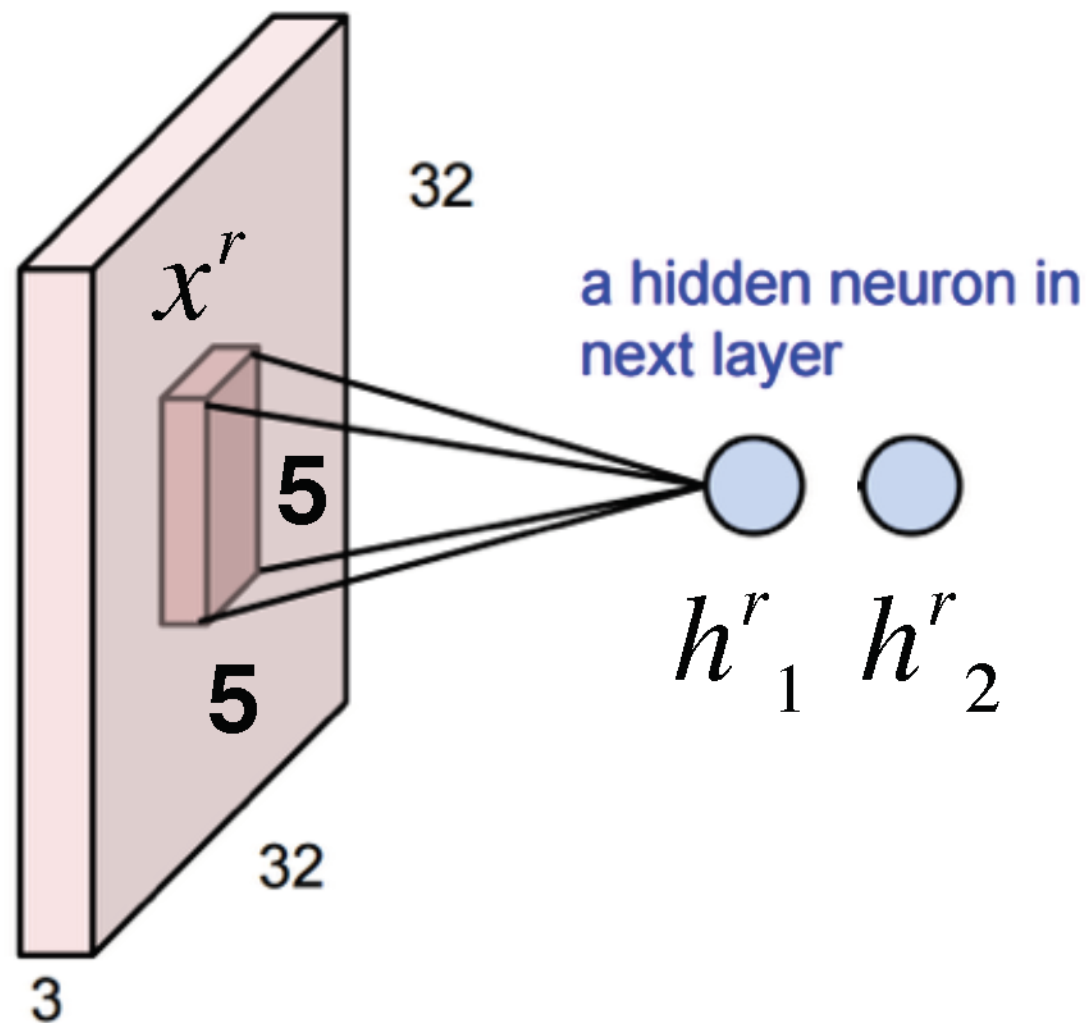


With **2** output neurons

$$h^r_1 = \sum_{ijk} x^r_{ijk} W_{1ijk} + b_1$$

$$h^r_2 = \sum_{ijk} x^r_{ijk} W_{2ijk} + b_2$$

3D Activations



With **2** output neurons

$$h^r_1 = \sum_{ijk} x^r_{ijk} W_{\boxed{1}ijk} + b_{\boxed{1}}$$

$$h^r_2 = \sum_{ijk} x^r_{ijk} W_{\boxed{2}ijk} + b_{\boxed{2}}$$

3D Activations

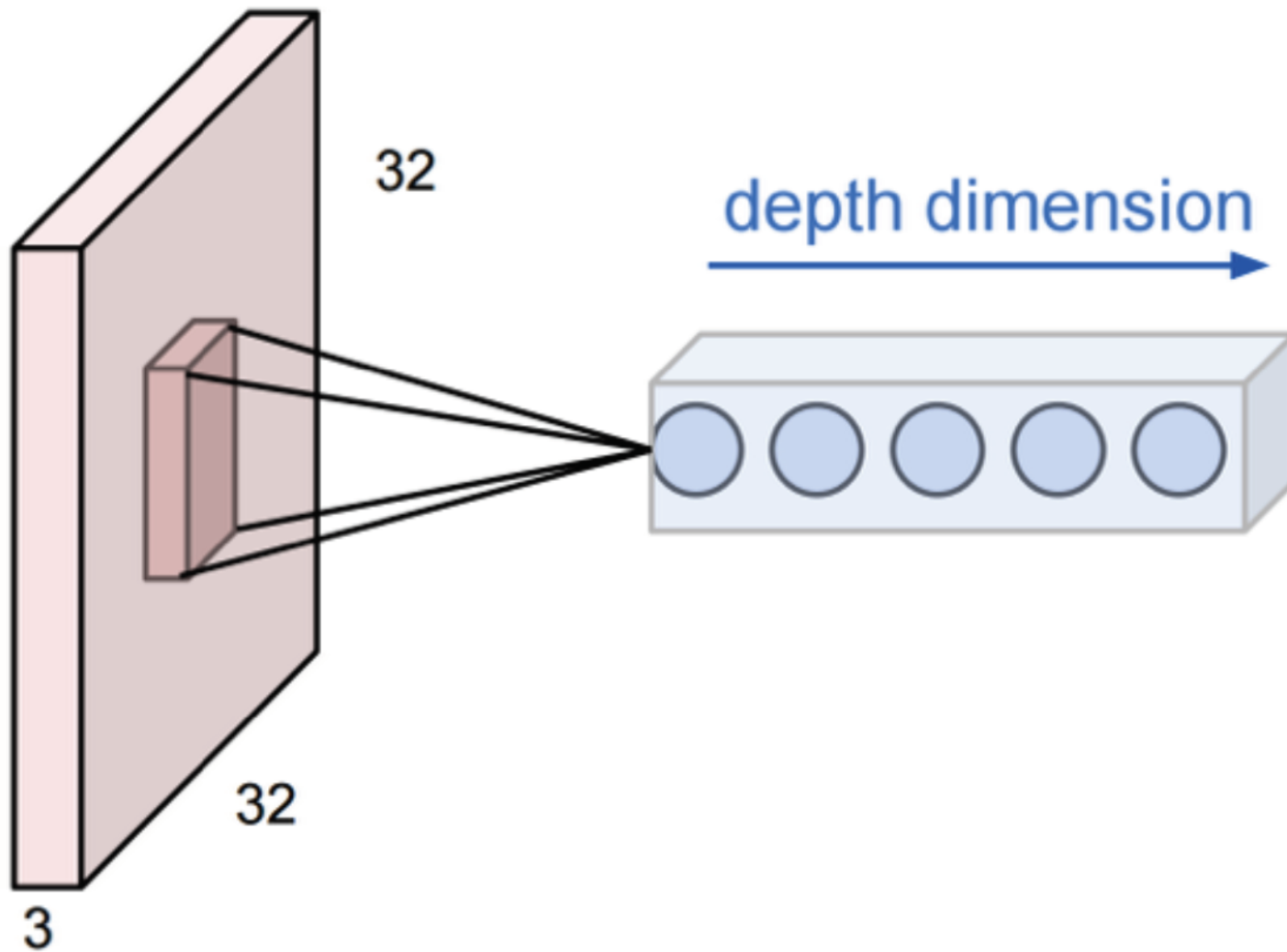
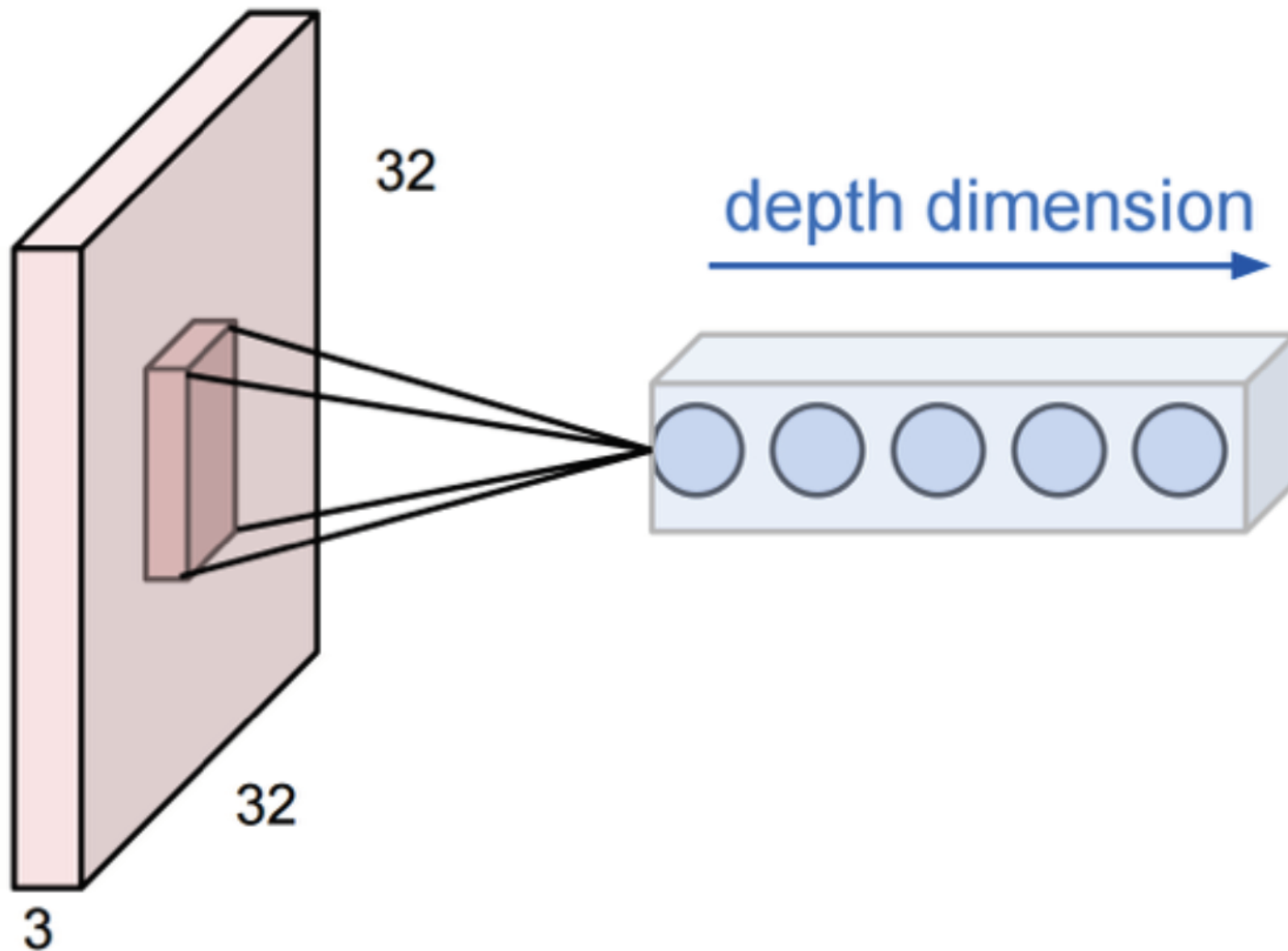


Figure: Andrej Karpathy

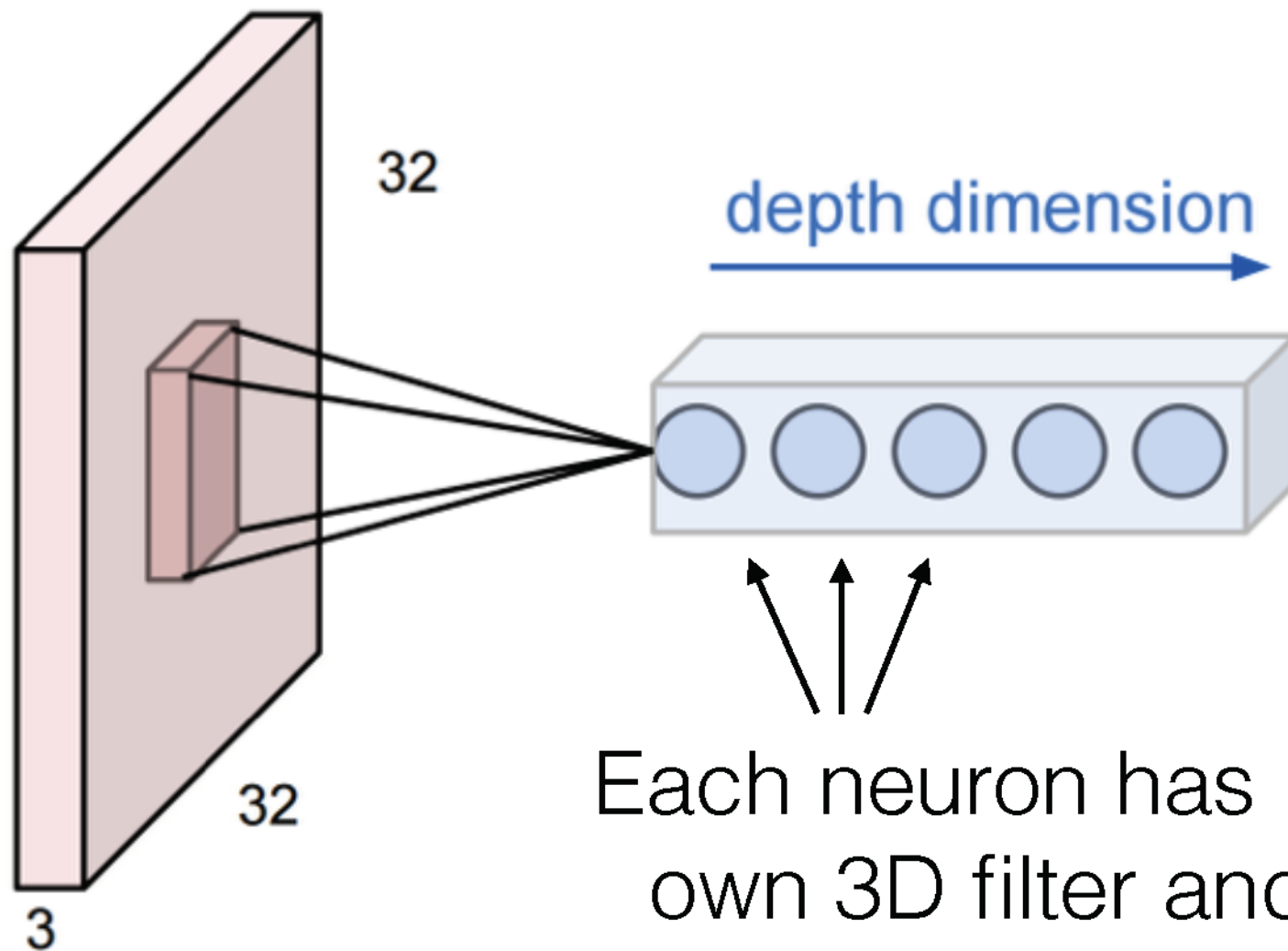
3D Activations



We can keep adding more outputs

These form a column in the output volume:
[depth x 1 x 1]

3D Activations

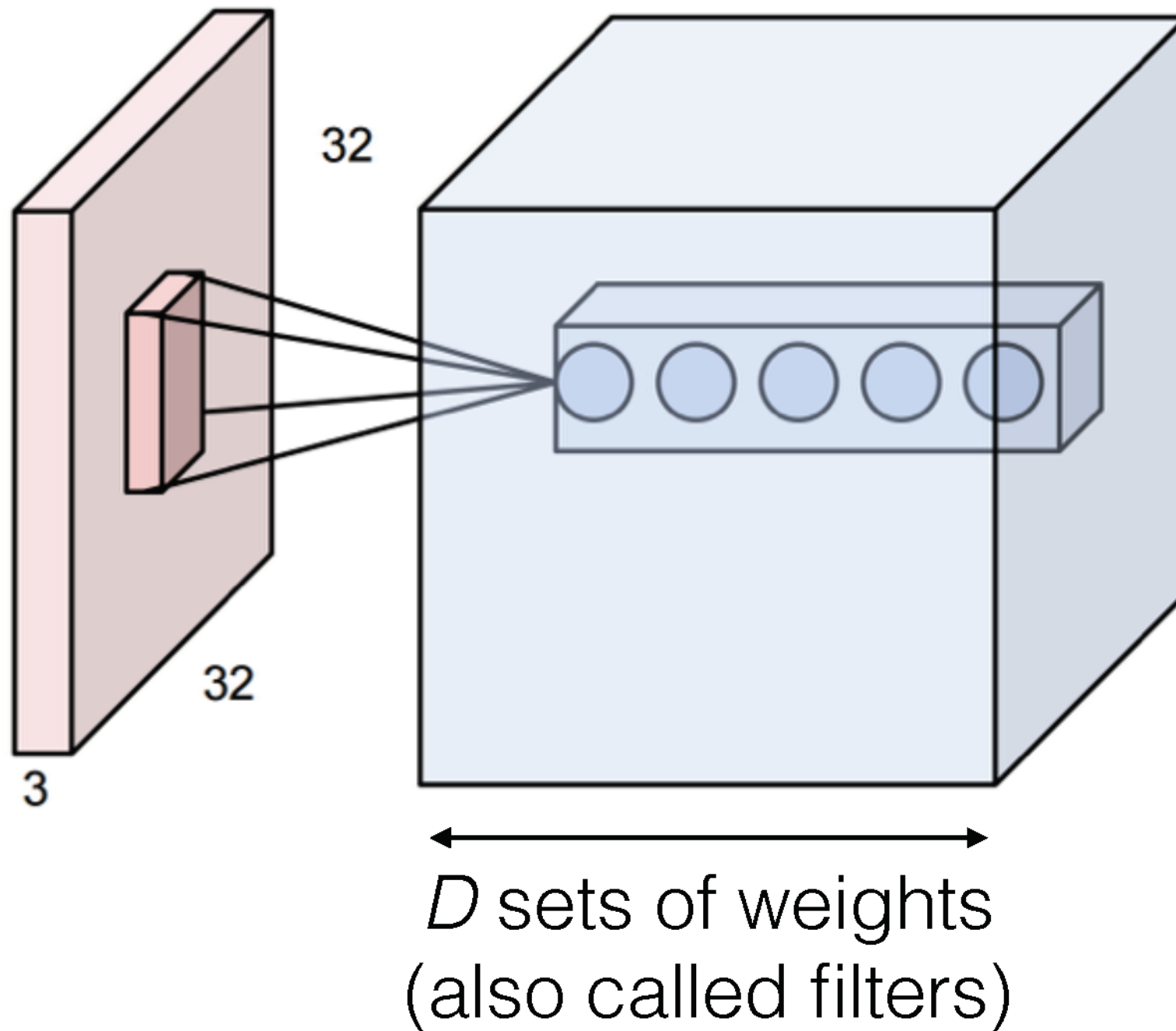


We can keep adding more outputs

These form a column in the output volume:
[depth x 1 x 1]

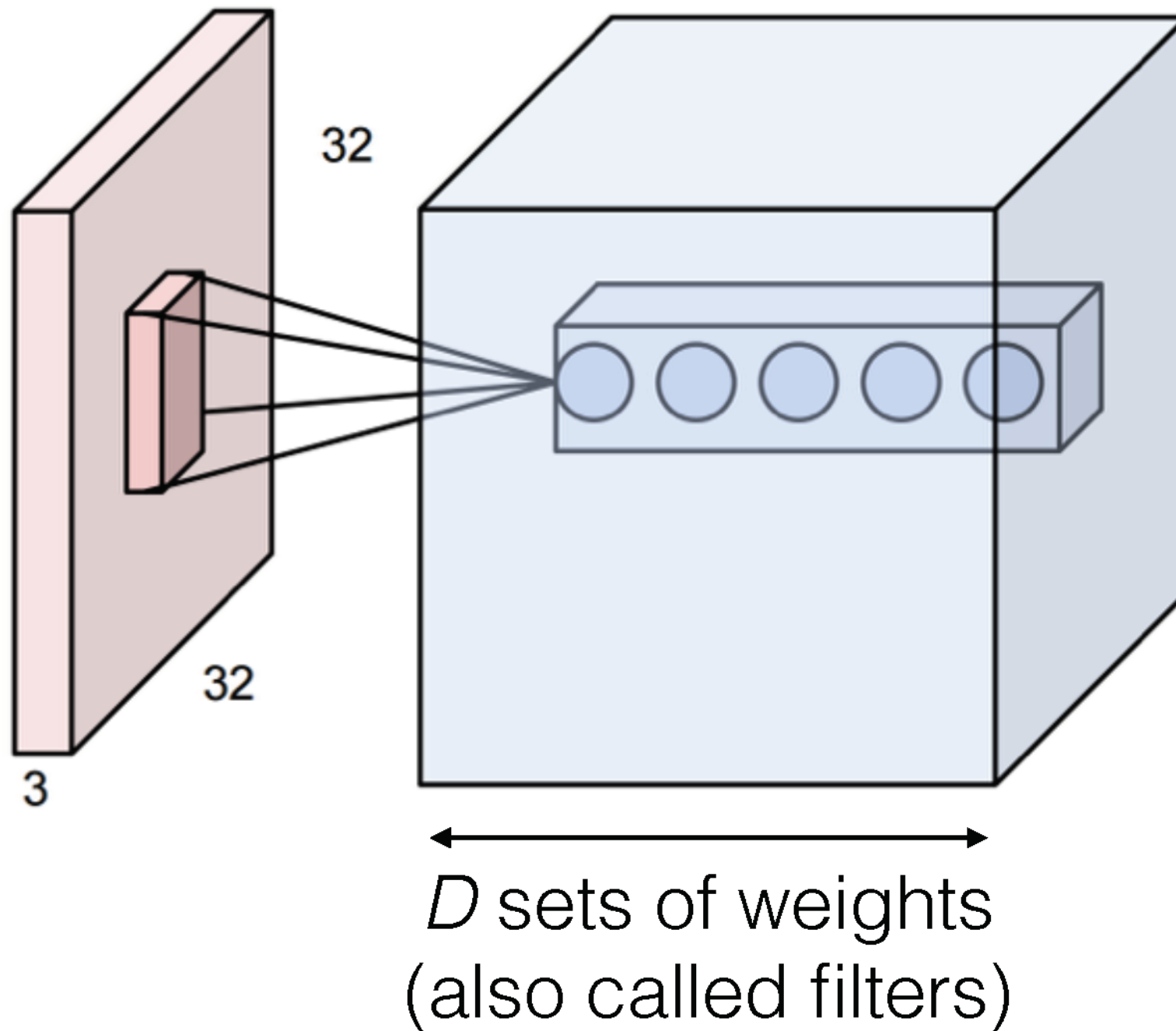
Each neuron has its own 3D filter and own (scalar) bias

3D Activations



Now repeat this
across the input

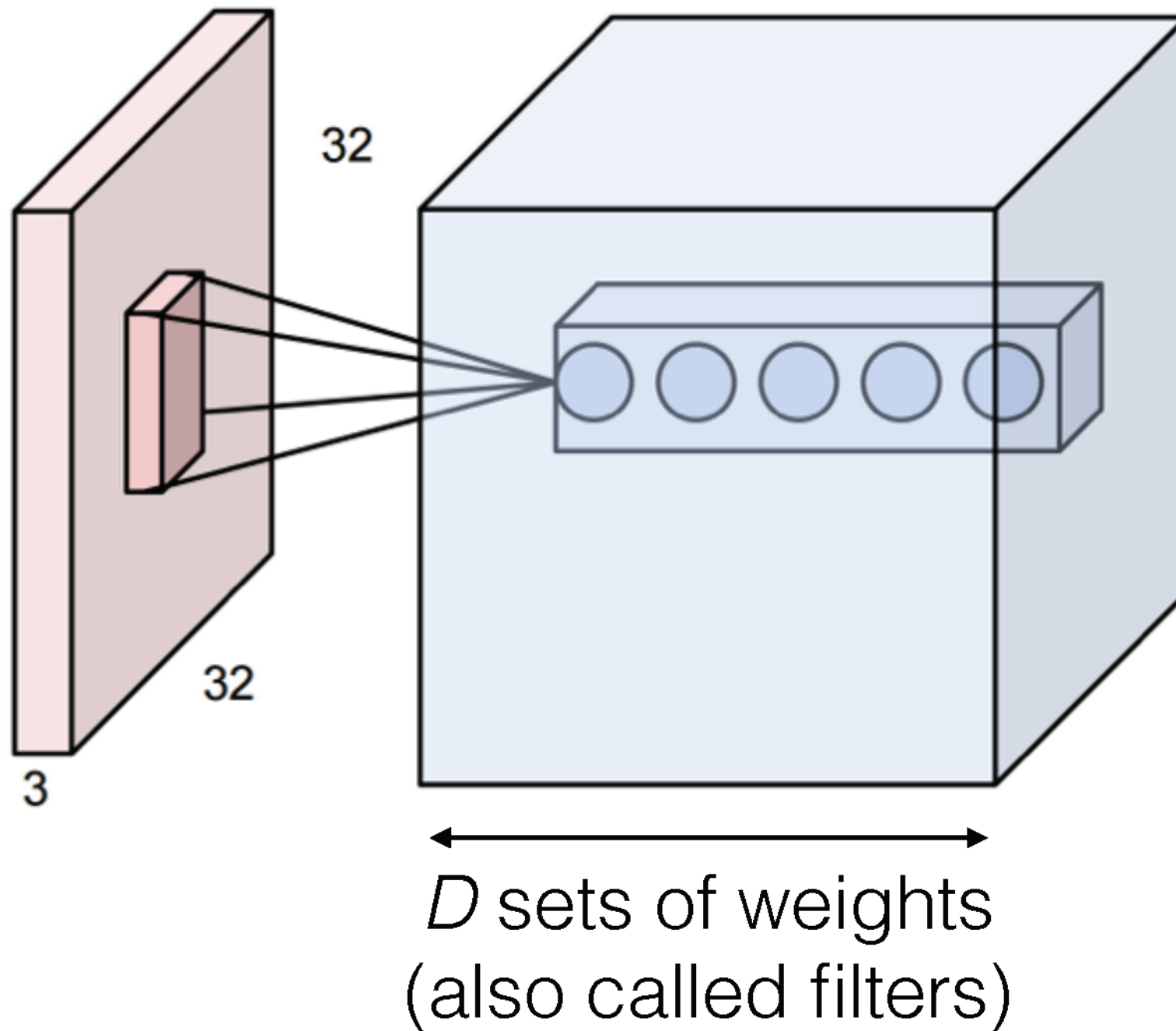
3D Activations



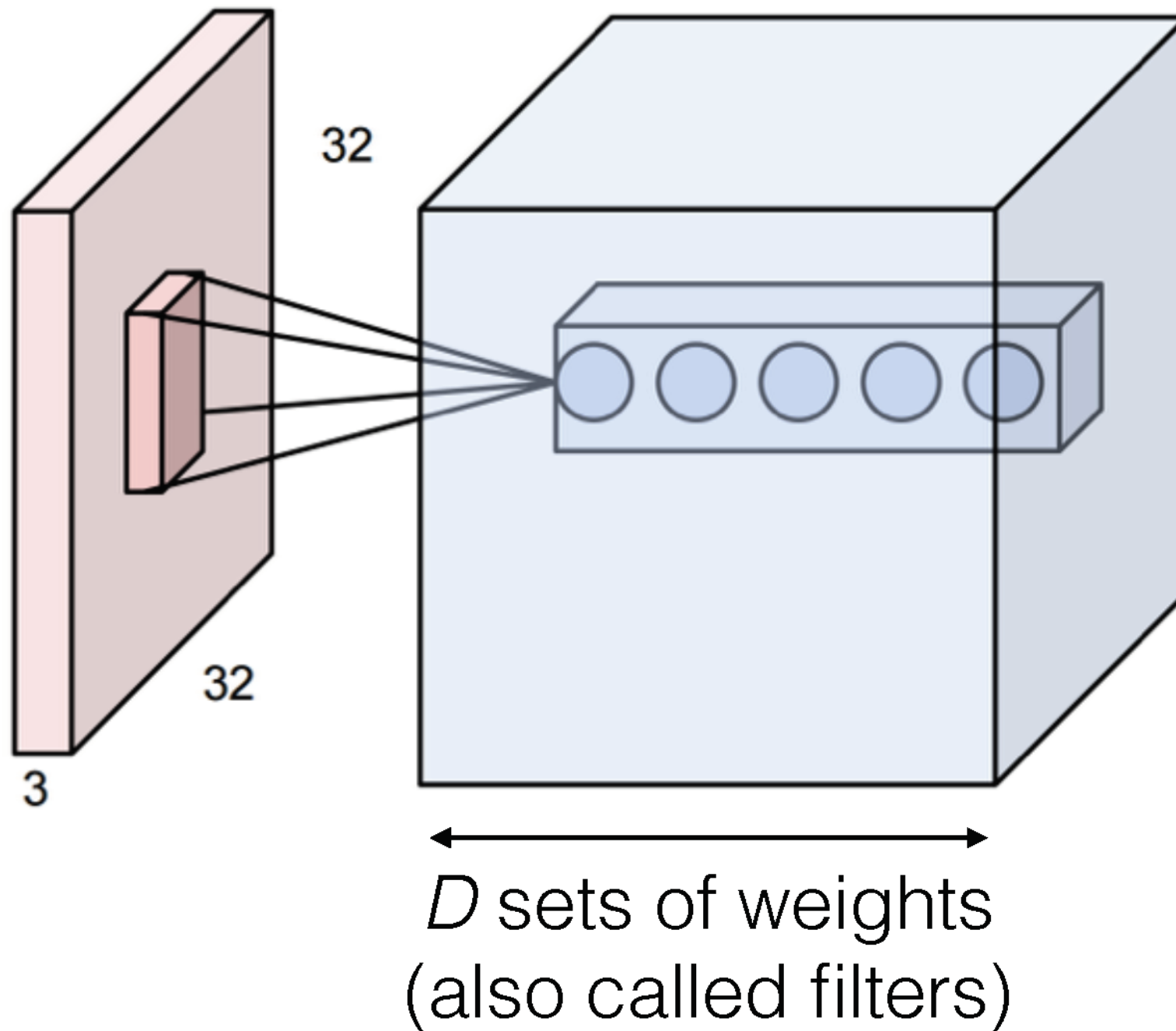
Now repeat this across the input

Weight sharing:
Each filter shares the same weights (but each depth index has its own set of weights)

3D Activations

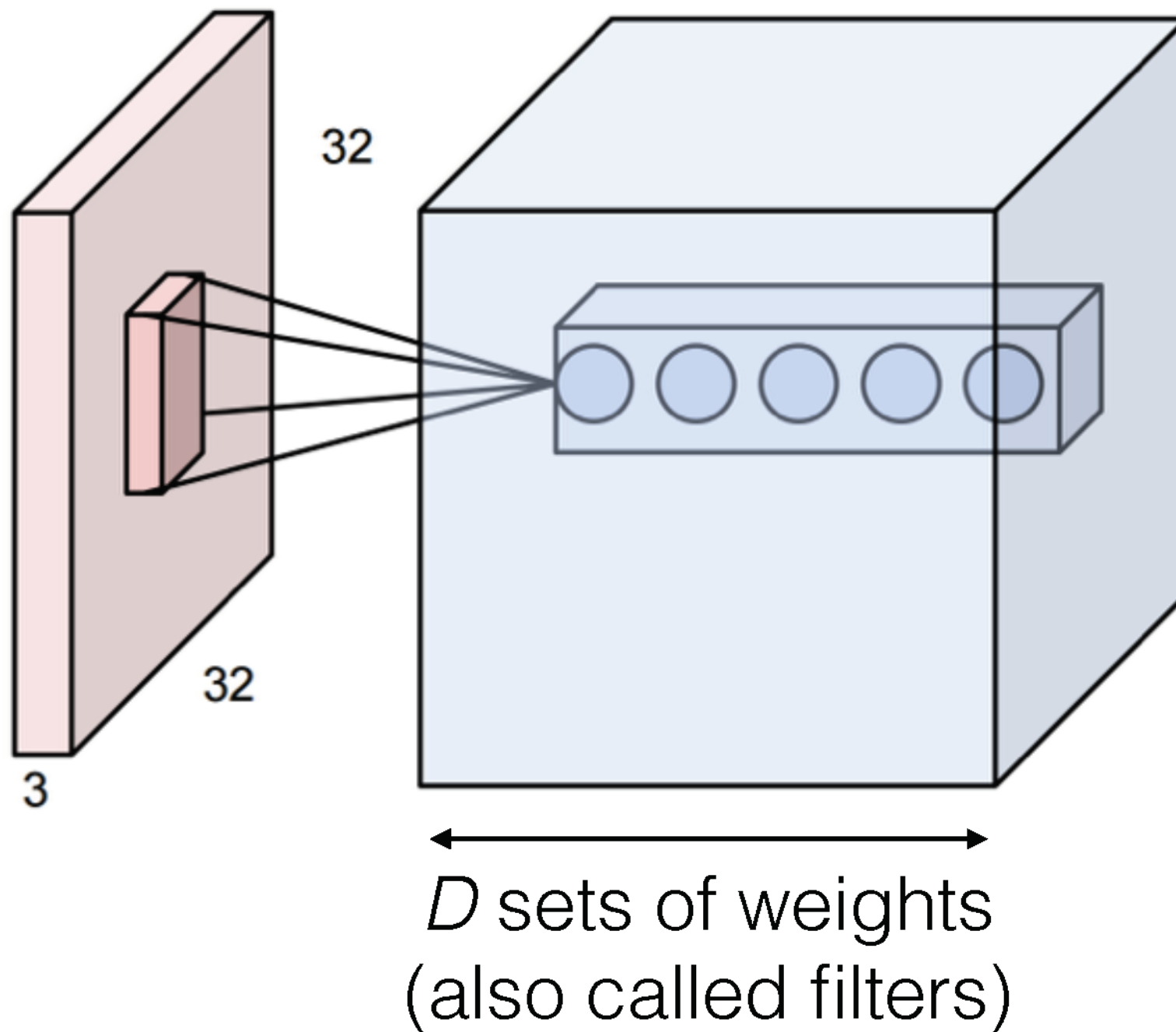


3D Activations



With weight sharing, this is called **convolution**

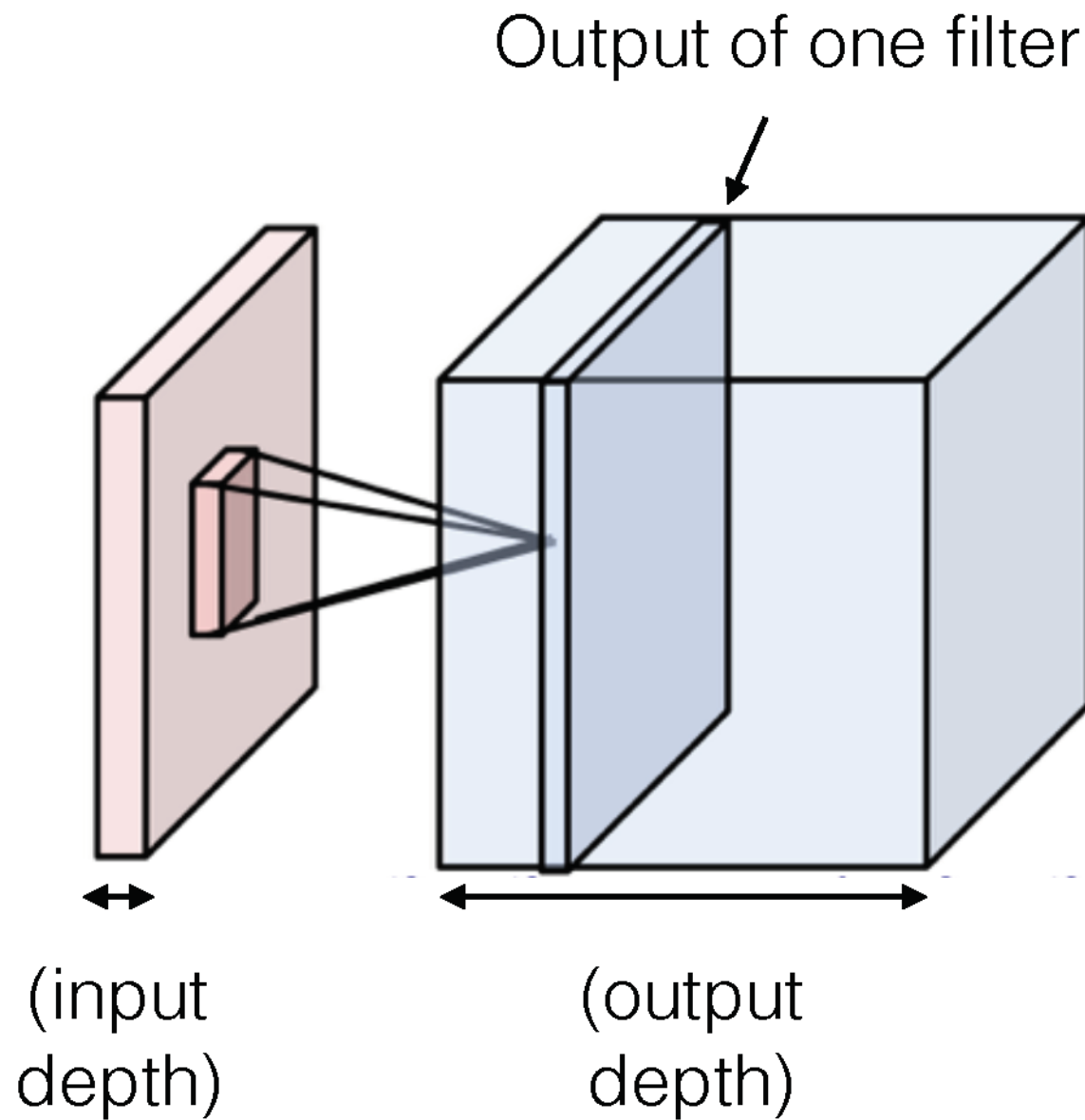
3D Activations



With weight sharing, this is called **convolution**

Without weight sharing, this is called a **locally connected layer**

3D Activations

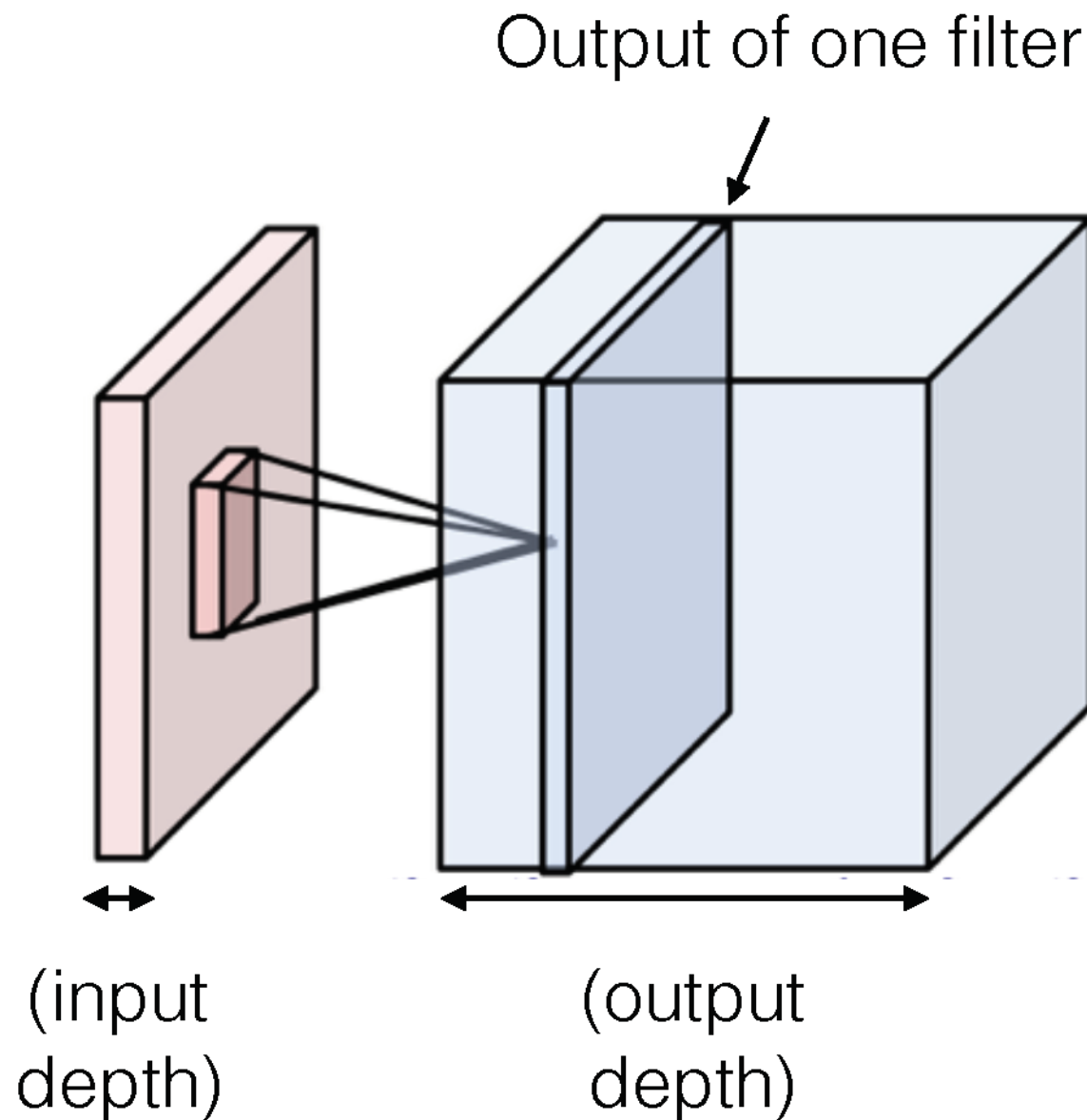


One set of weights gives one slice in the output

To get a 3D output of depth D , use D different filters

In practice, ConvNets use many filters (~ 64 to 1024)

3D Activations



One set of weights gives one slice in the output

To get a 3D output of depth D , use D different filters

In practice, ConvNets use many filters (~ 64 to 1024)

All together, the weights are **4** dimensional:
(output depth, input depth, kernel height, kernel width)

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

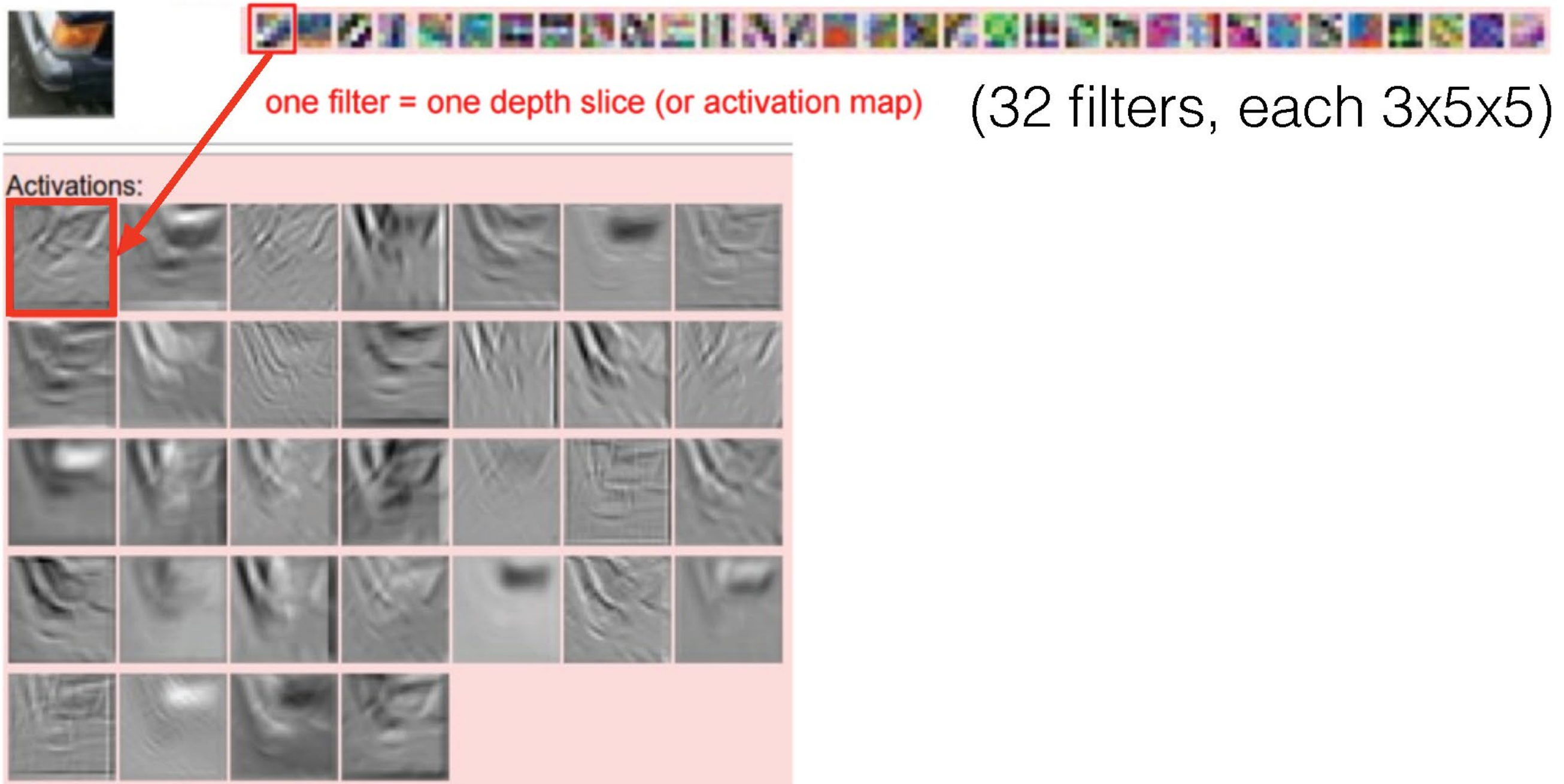


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)



Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

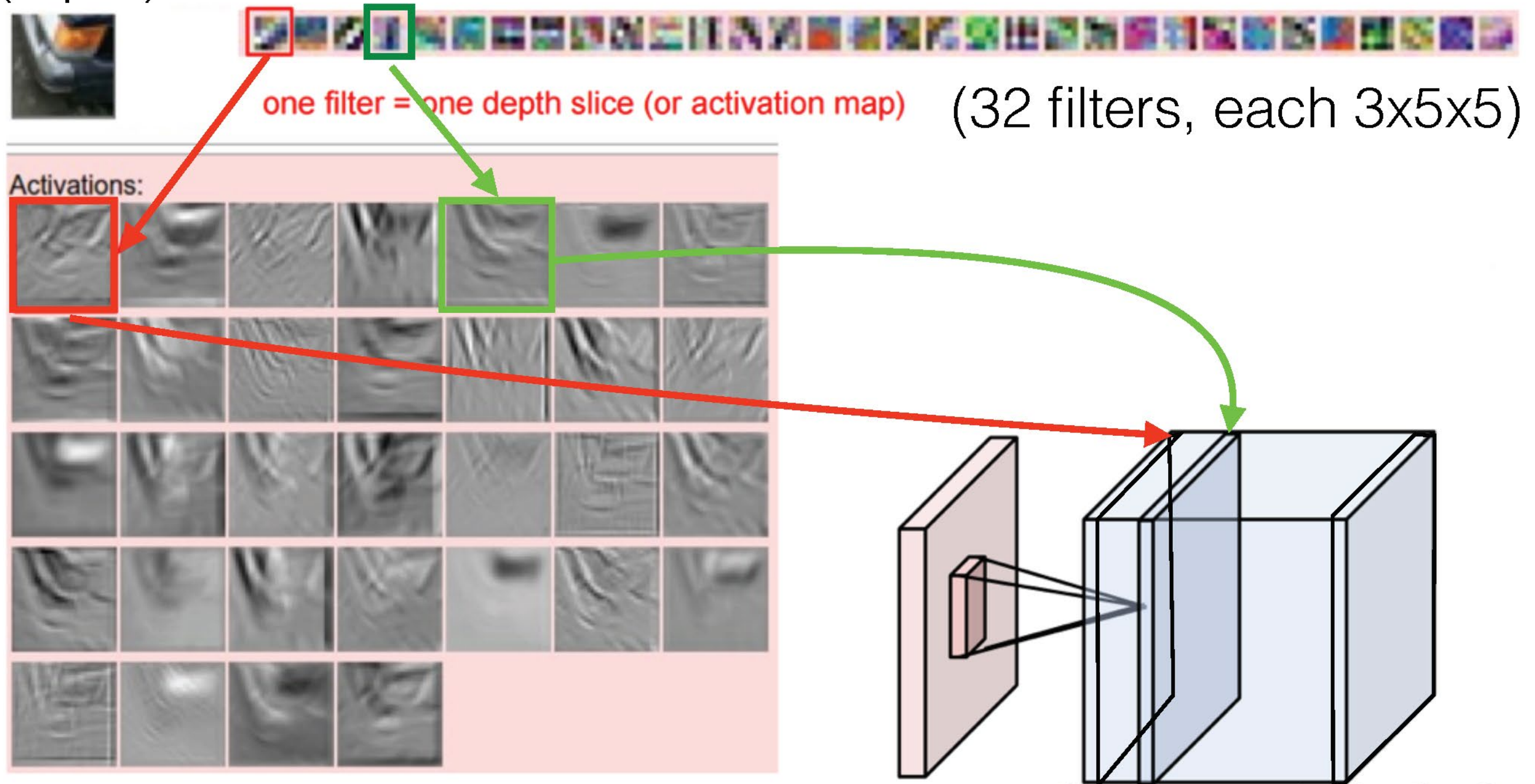


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

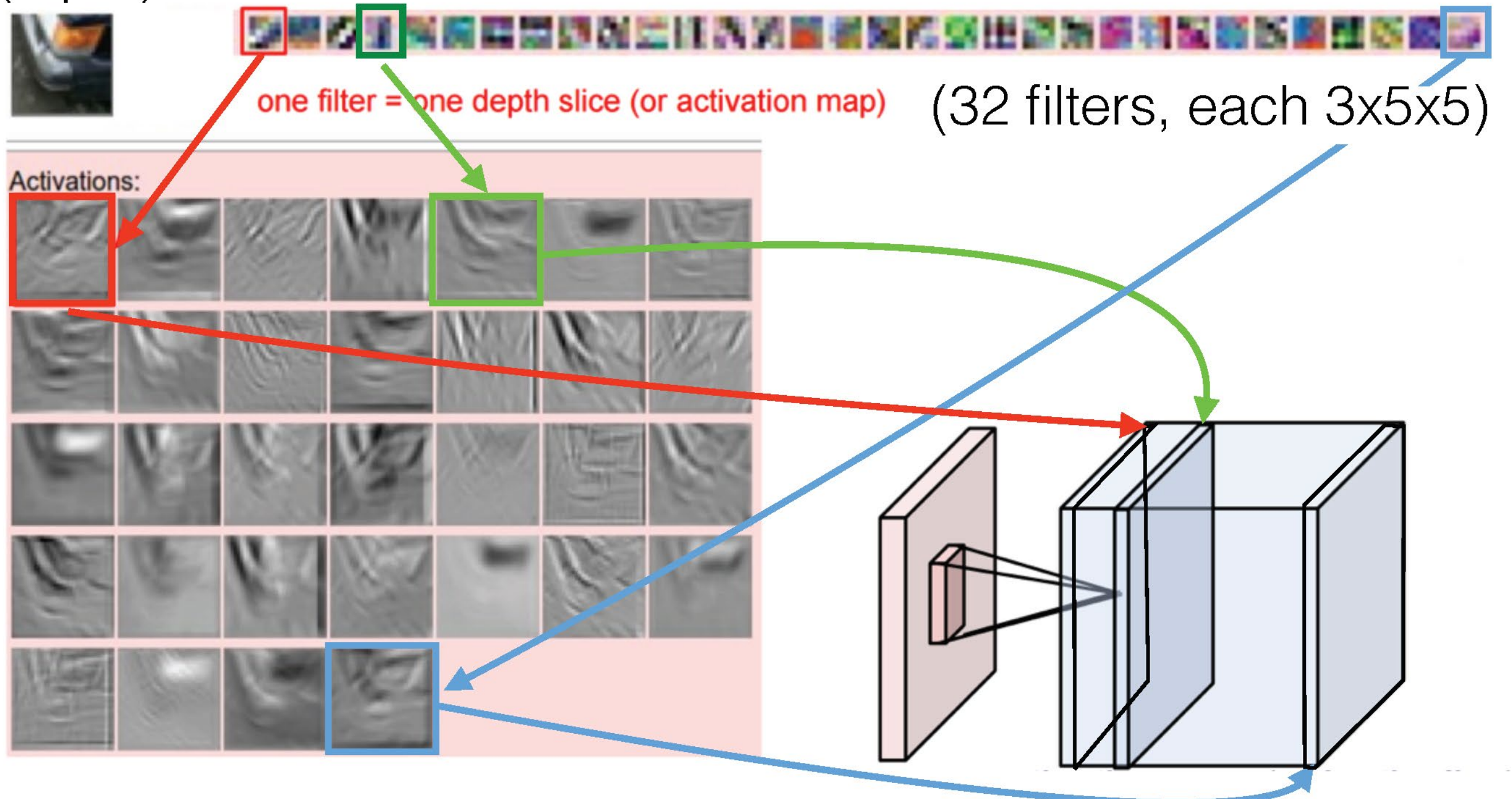
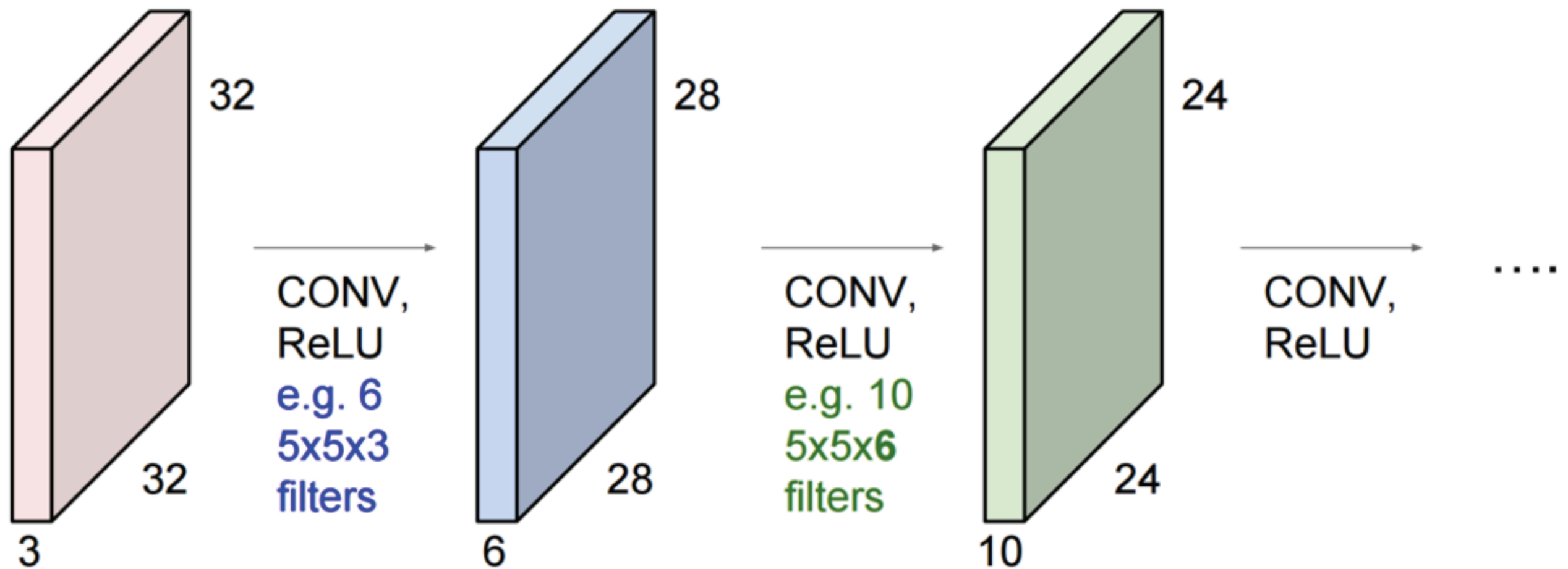


Figure: Andrej Karpathy

(Recap)

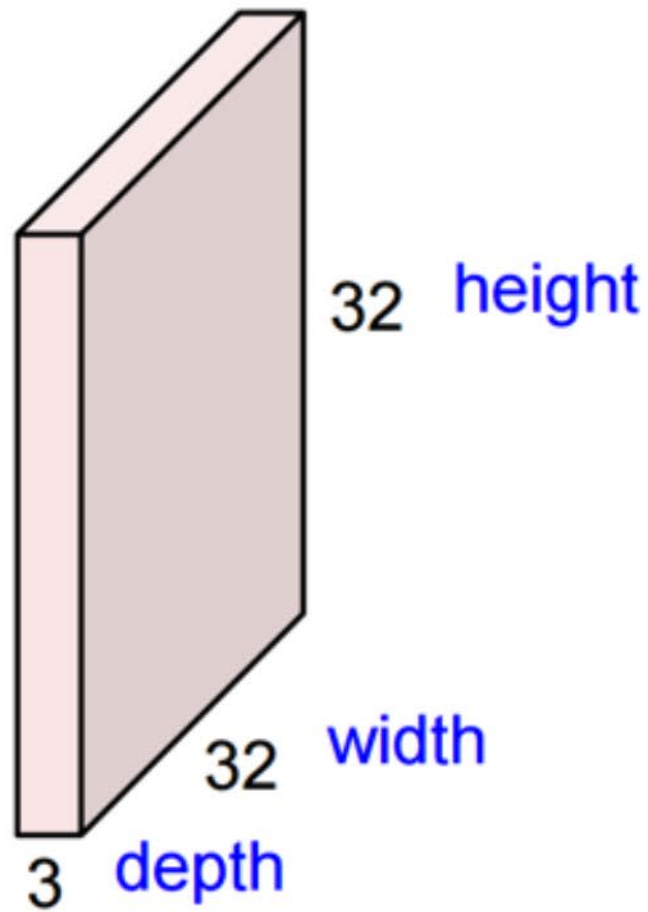
A **ConvNet** is a sequence of convolutional layers, interspersed with activation functions (and possibly other layer types)



(Recap)

Convolution Layer

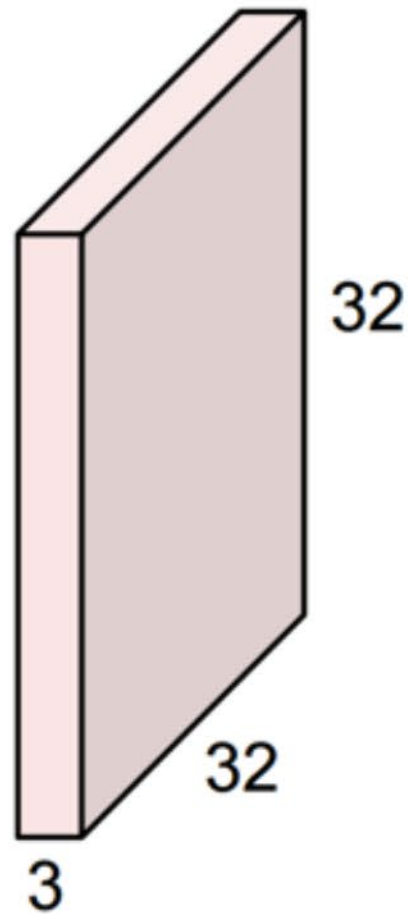
32x32x3 image



(Recap)

Convolution Layer

32x32x3 image



5x5x3 filter

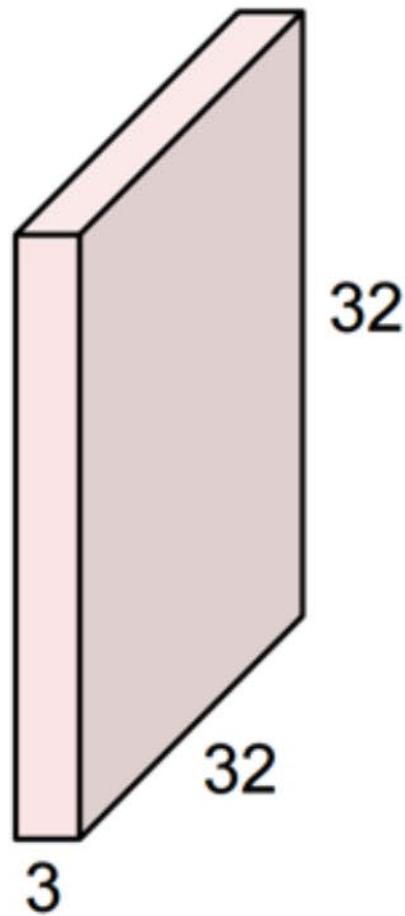


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

(Recap)

Convolution Layer

32x32x3 image



Filters always extend the full depth of the input volume

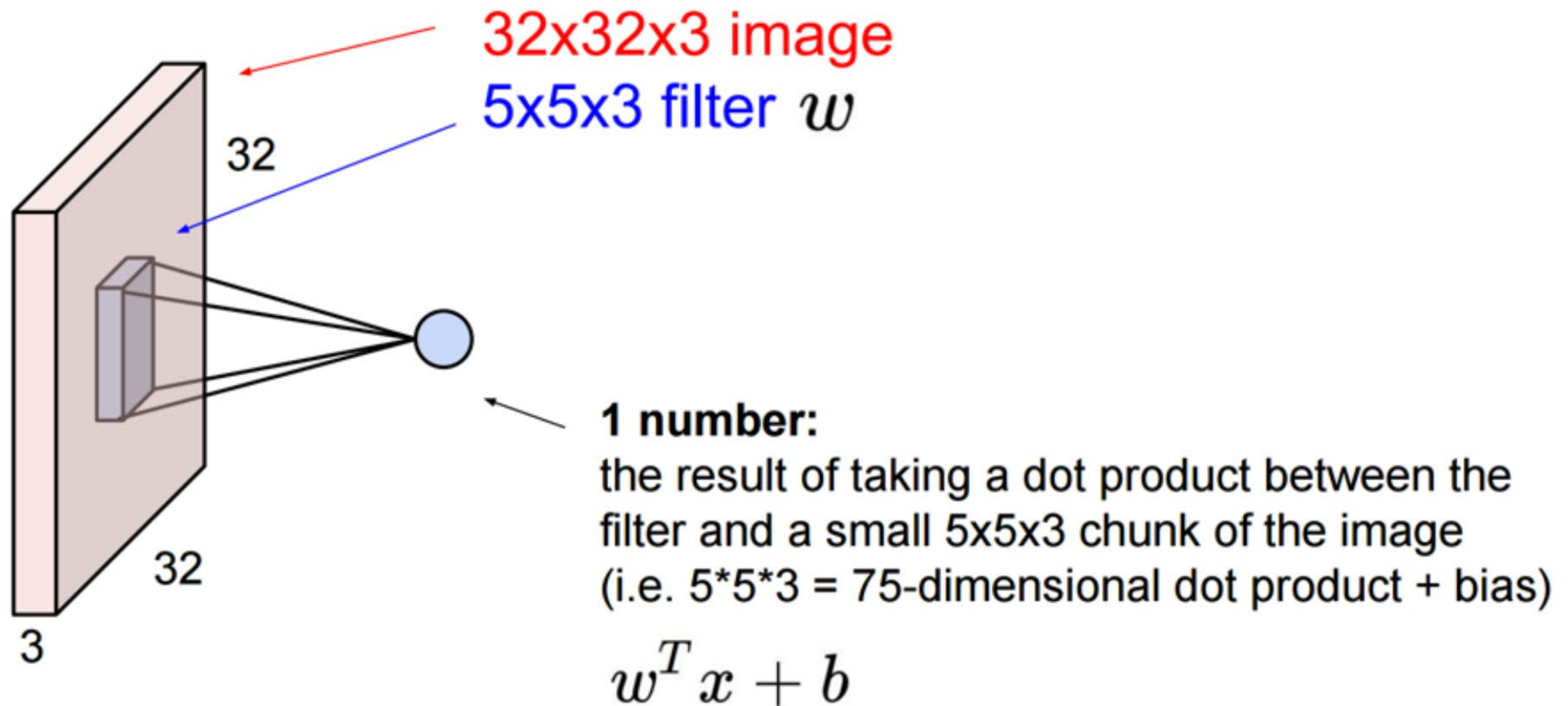
5x5x3 filter



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

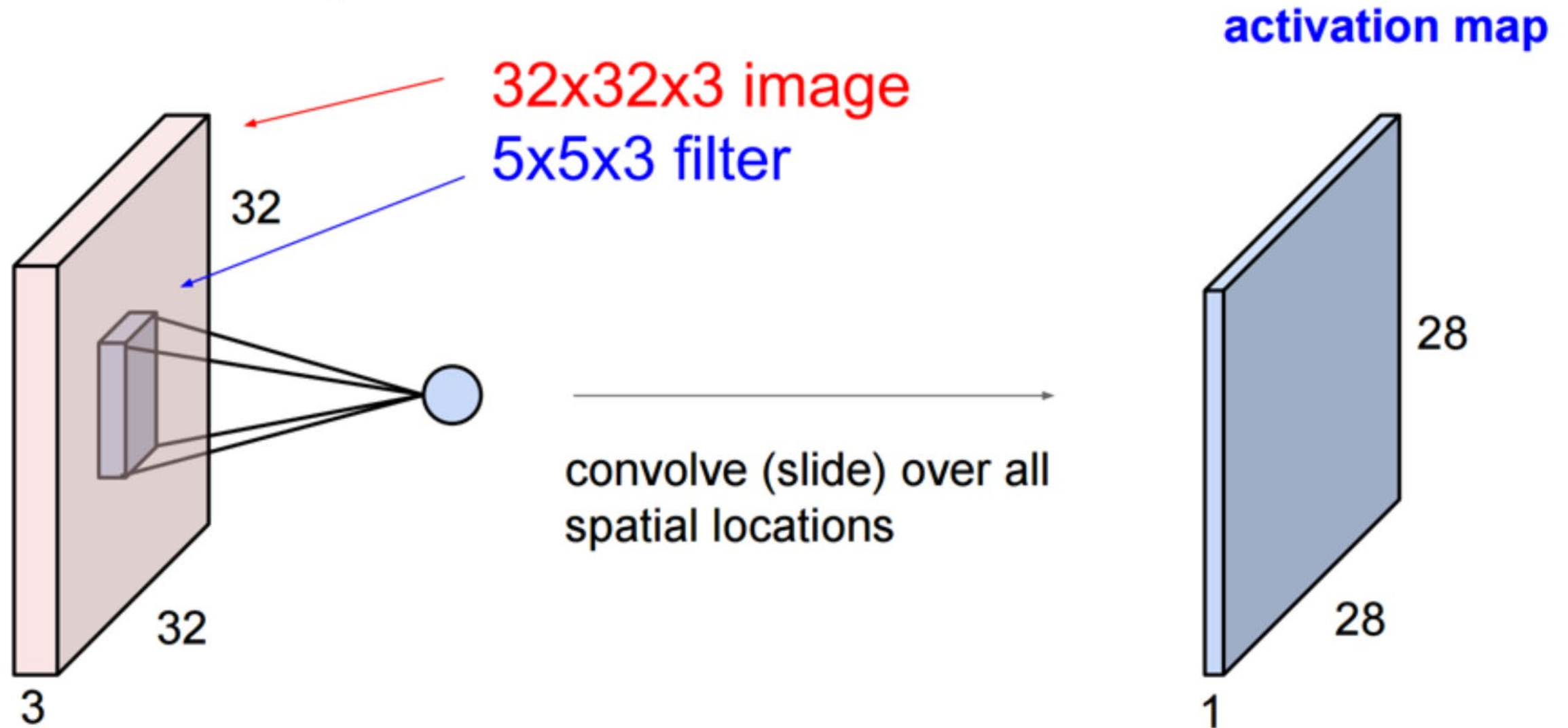
(Recap)

Convolution Layer



(Recap)

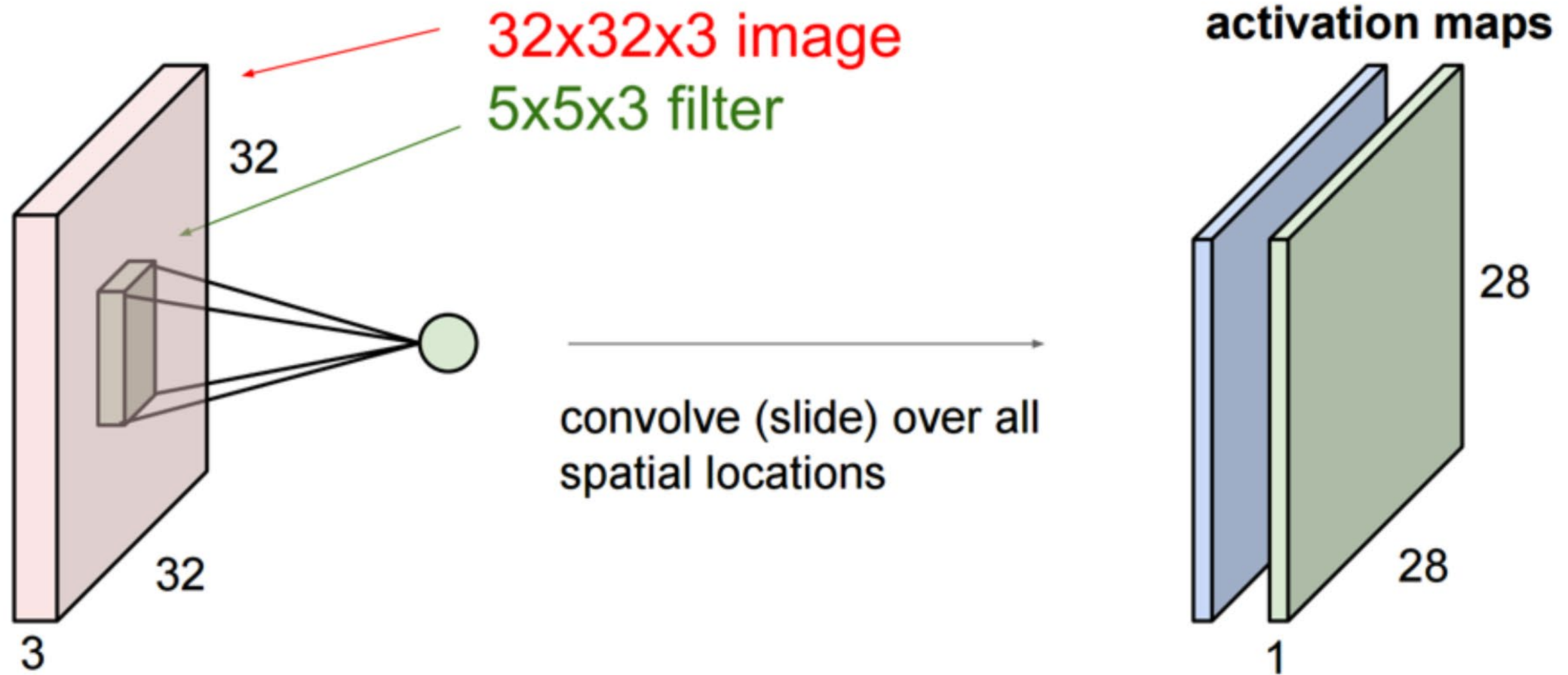
Convolution Layer



(Recap)

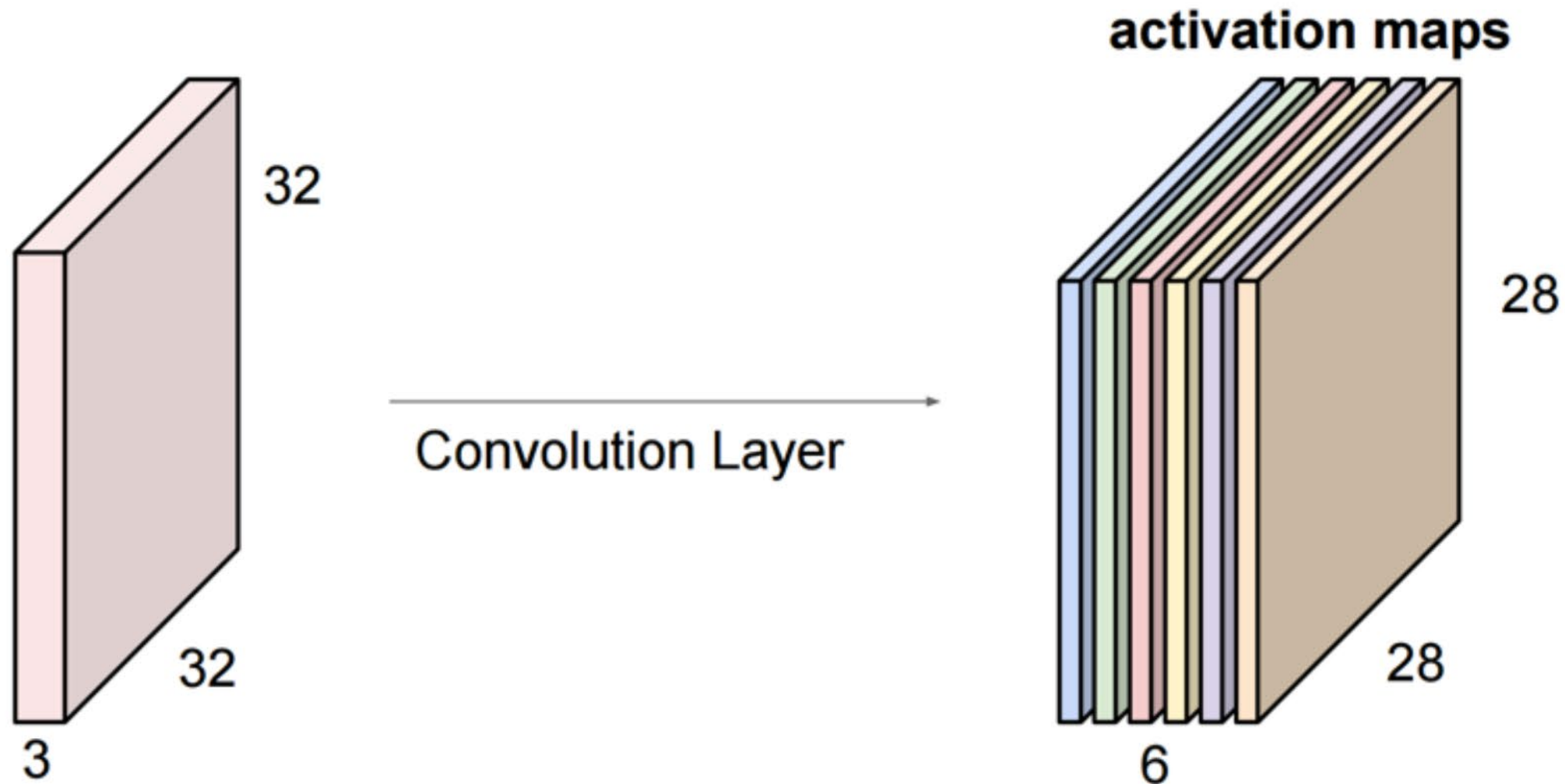
Convolution Layer

consider a second, **green** filter



(Recap)

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

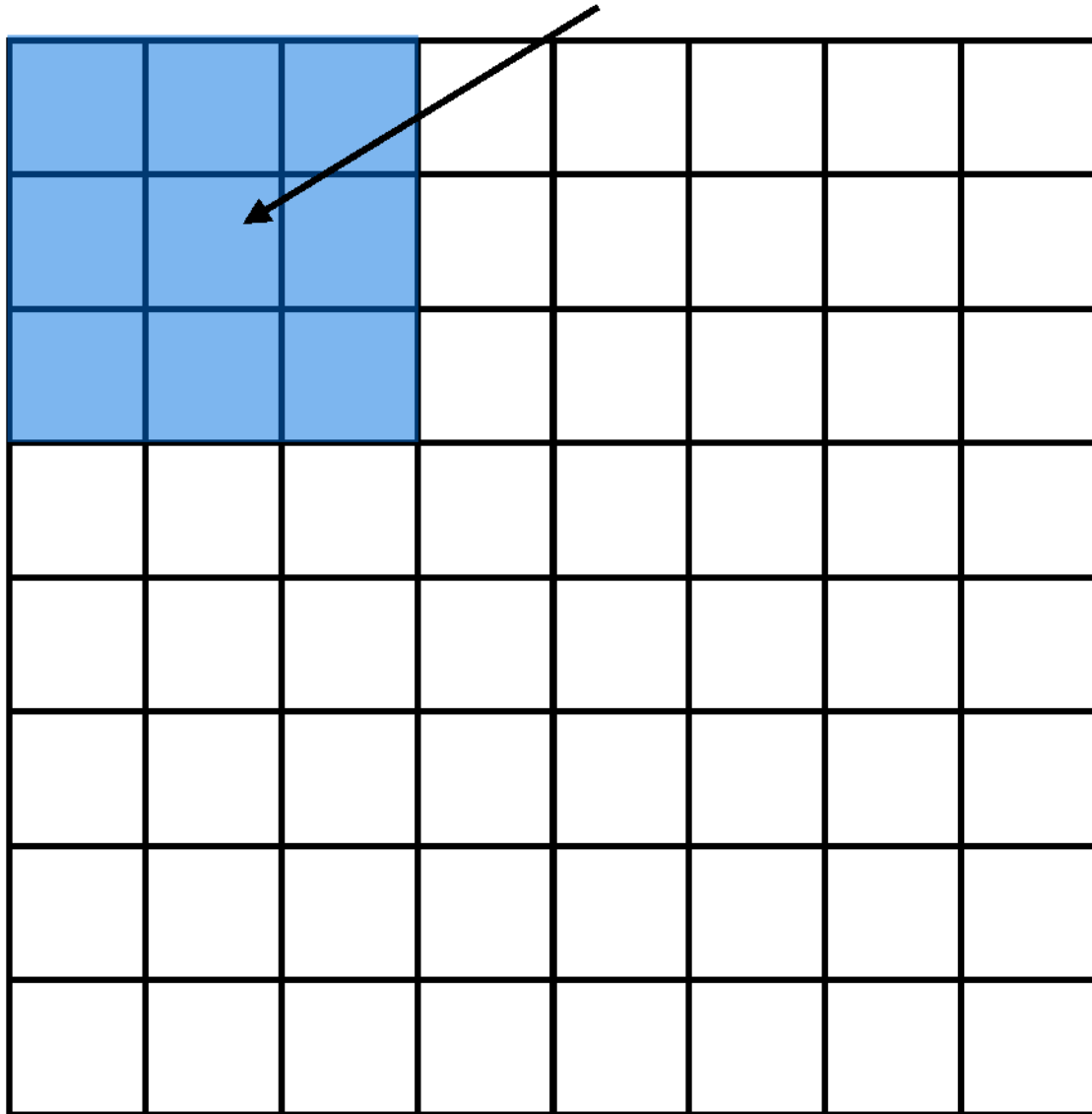


We stack these up to get a “new image” of size 28x28x6!

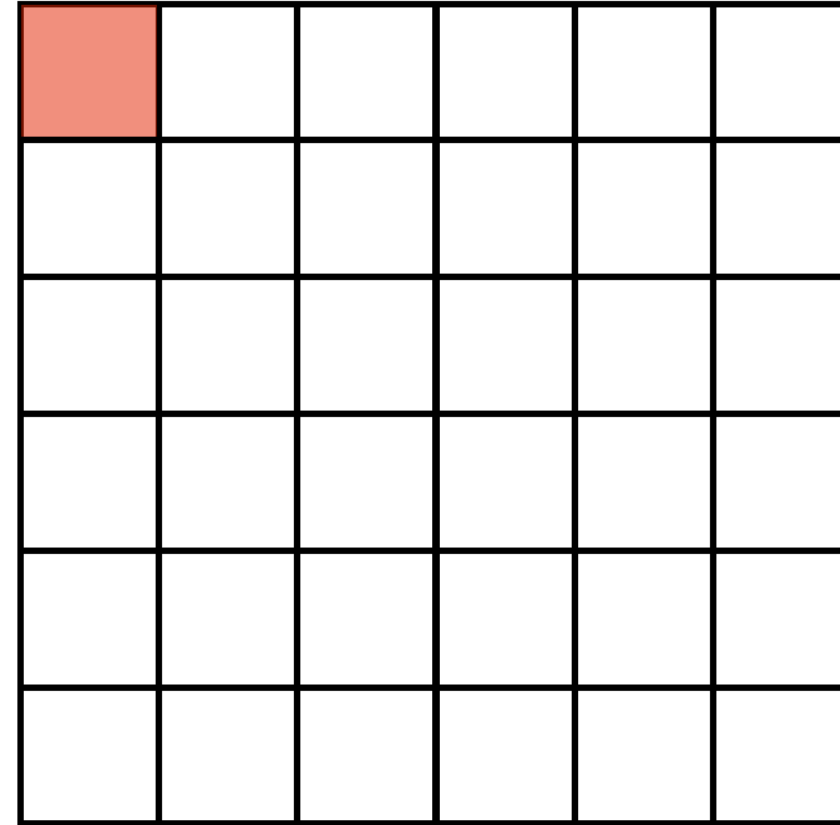
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output

Weights



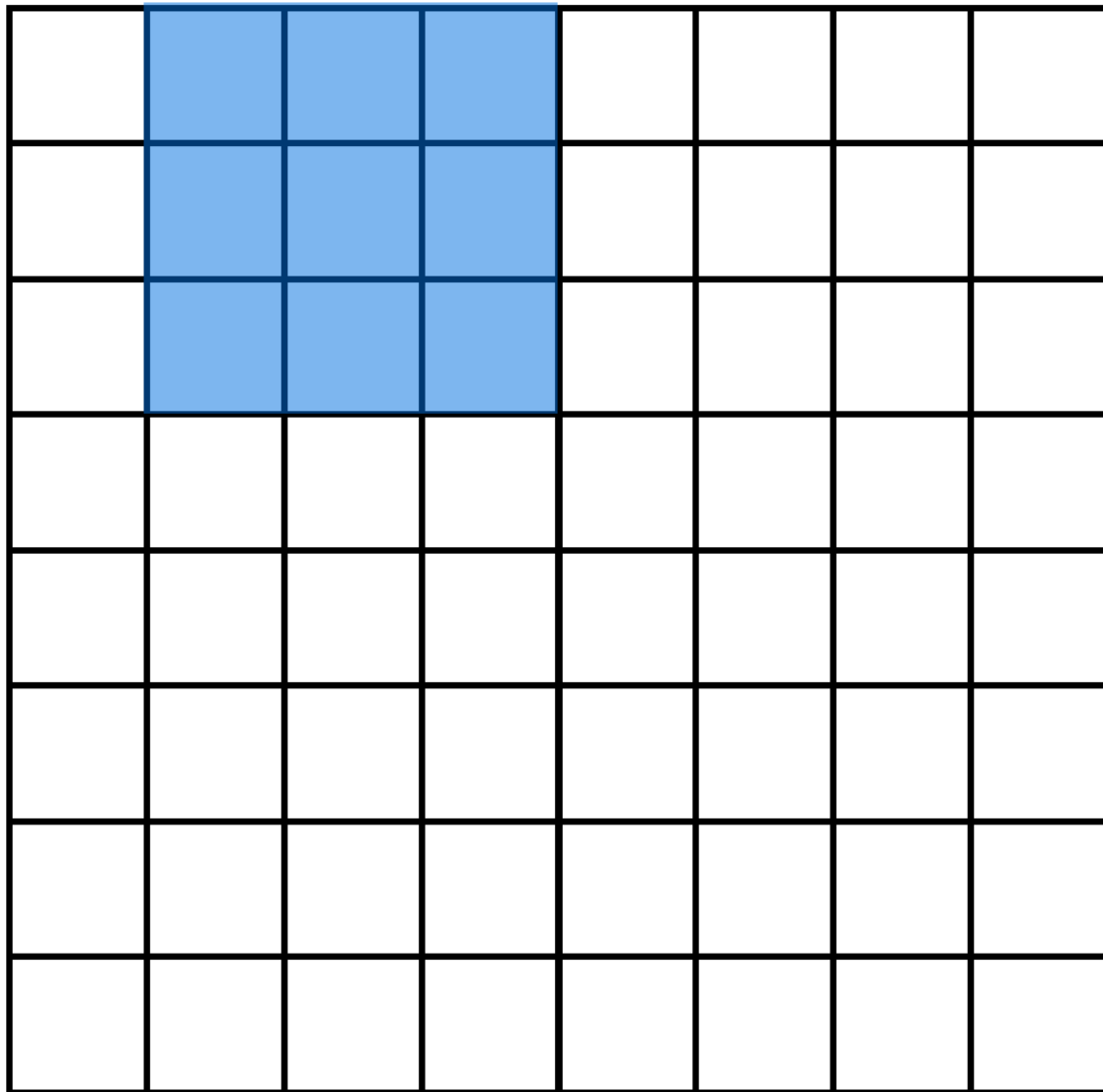
Input



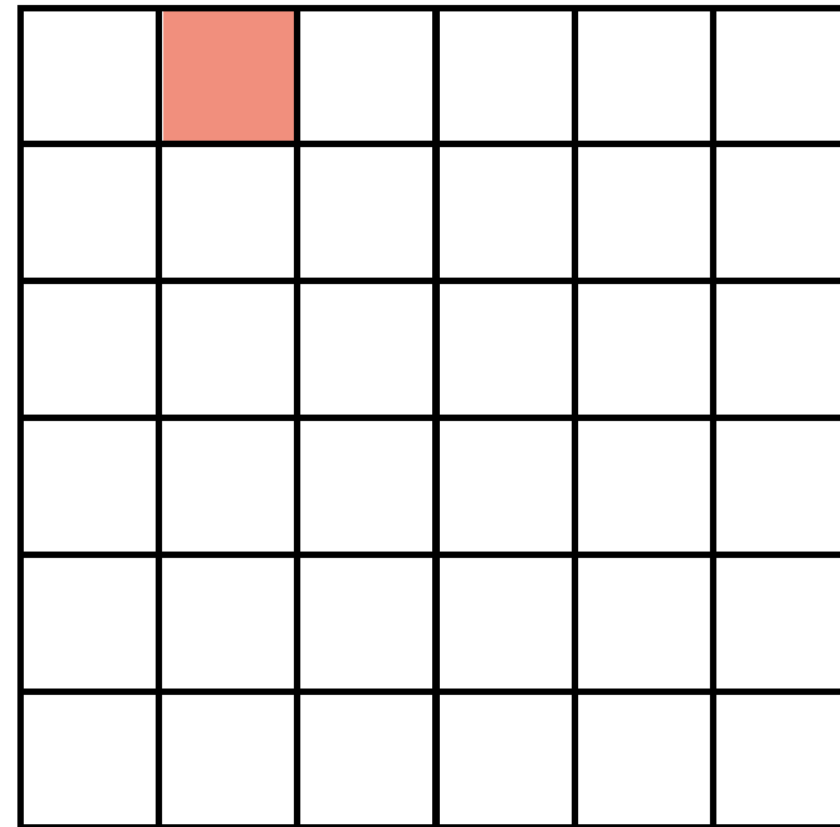
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



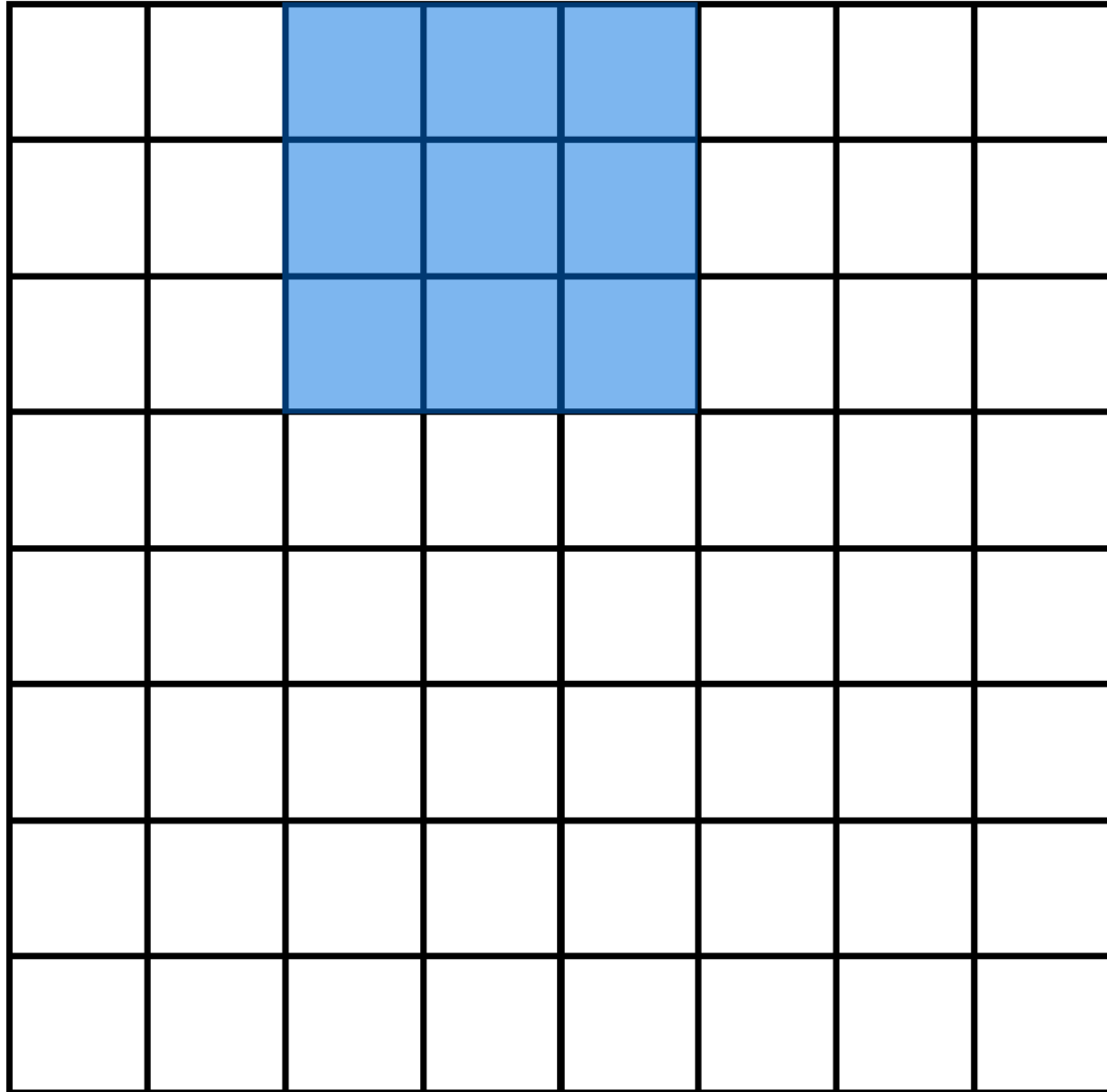
Input



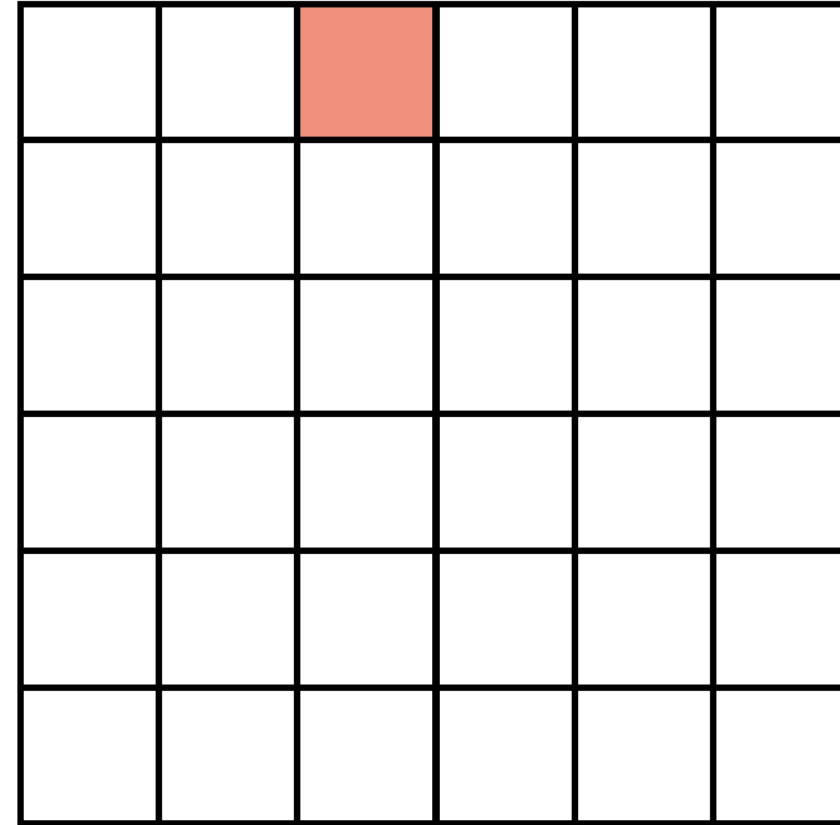
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



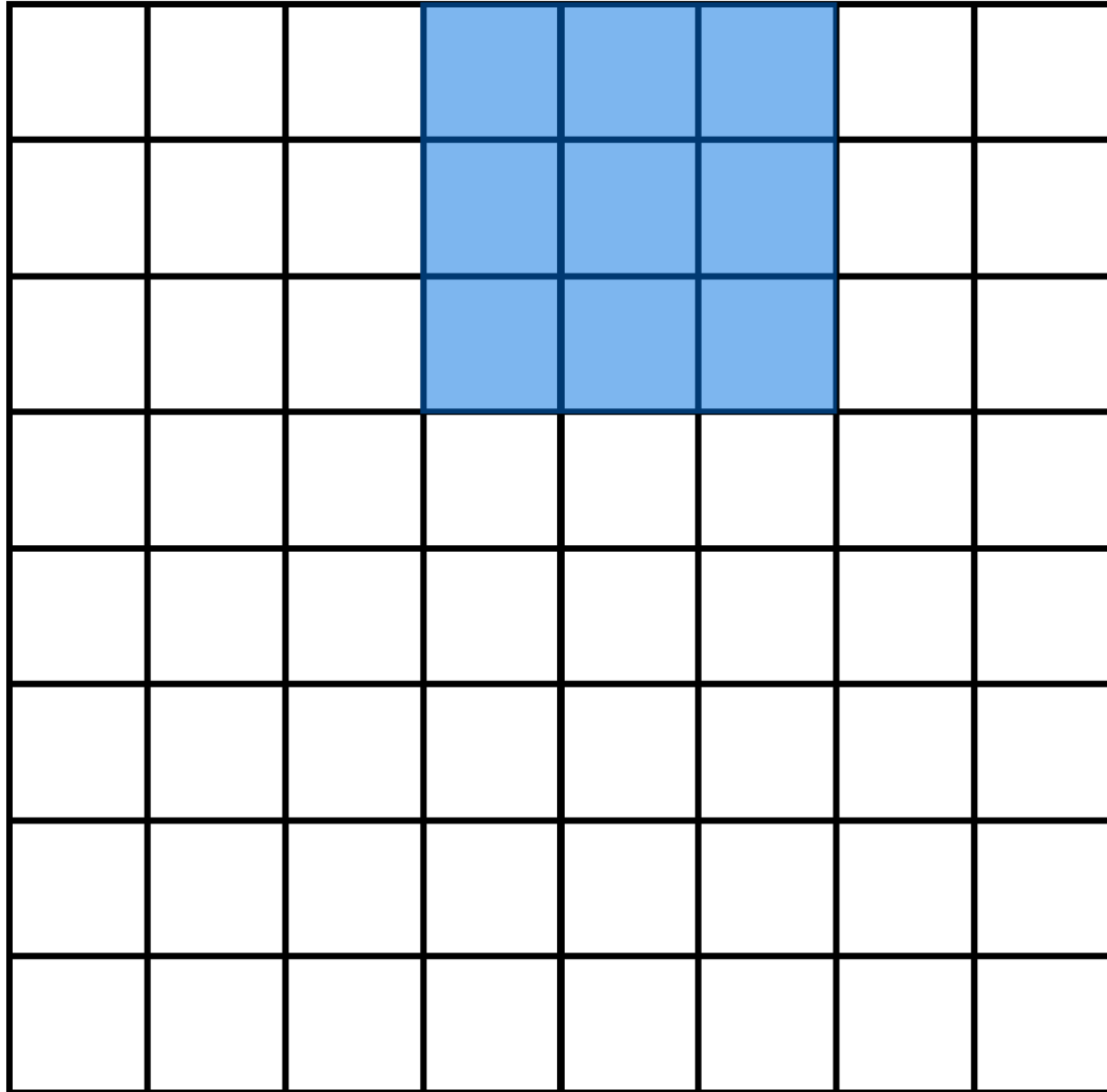
Input



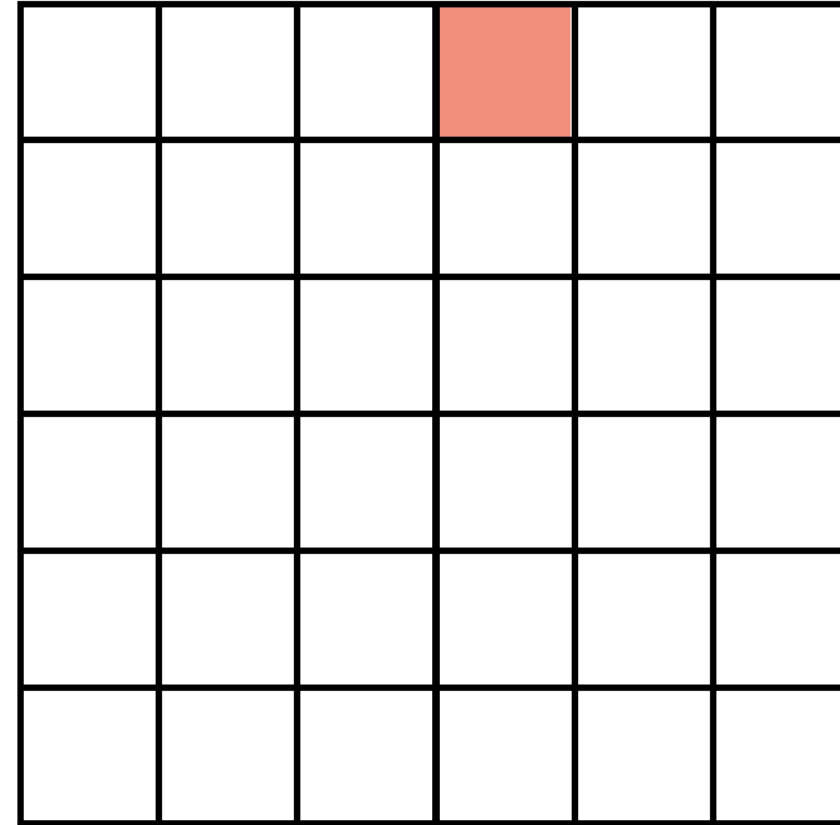
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



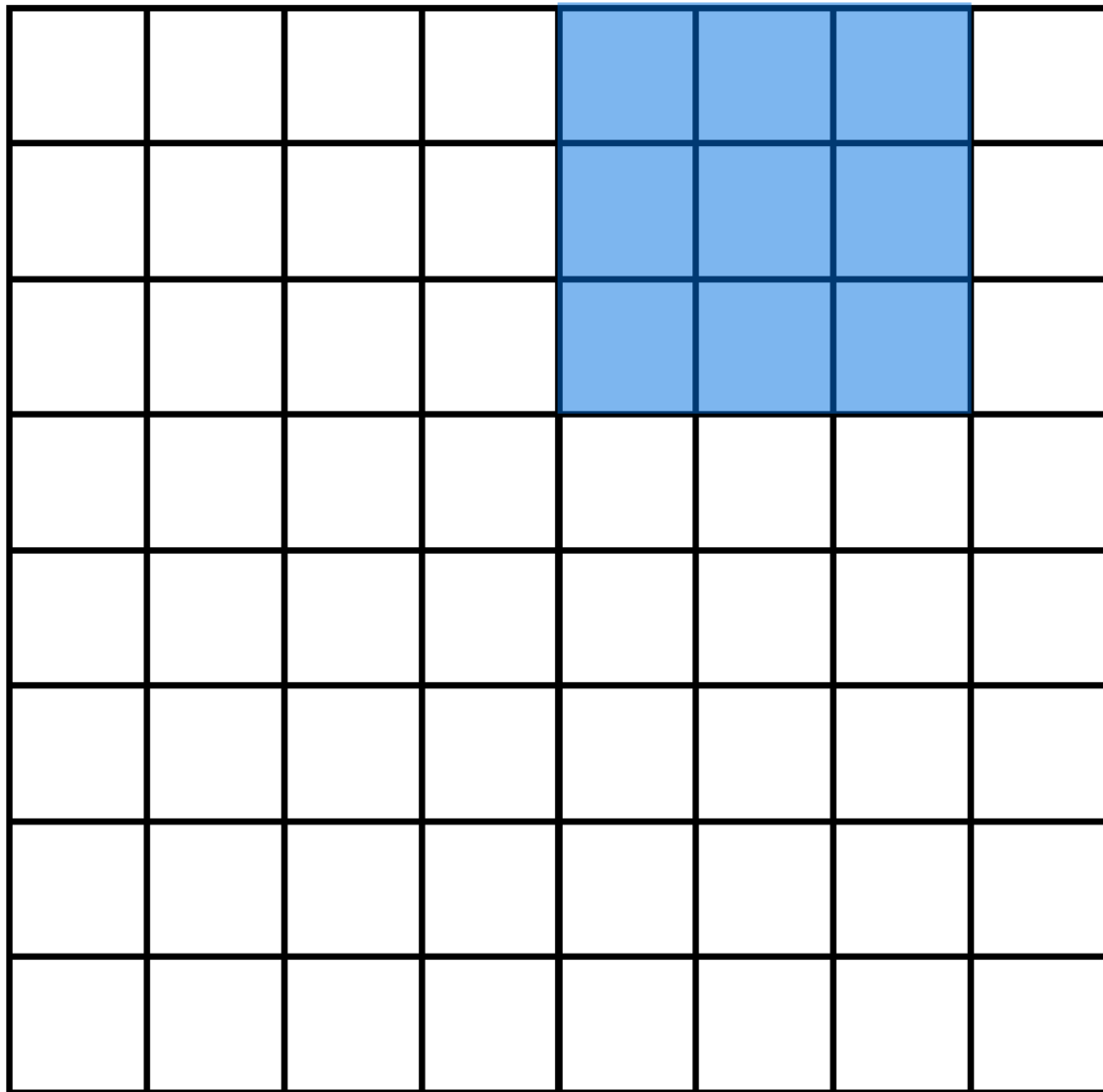
Input



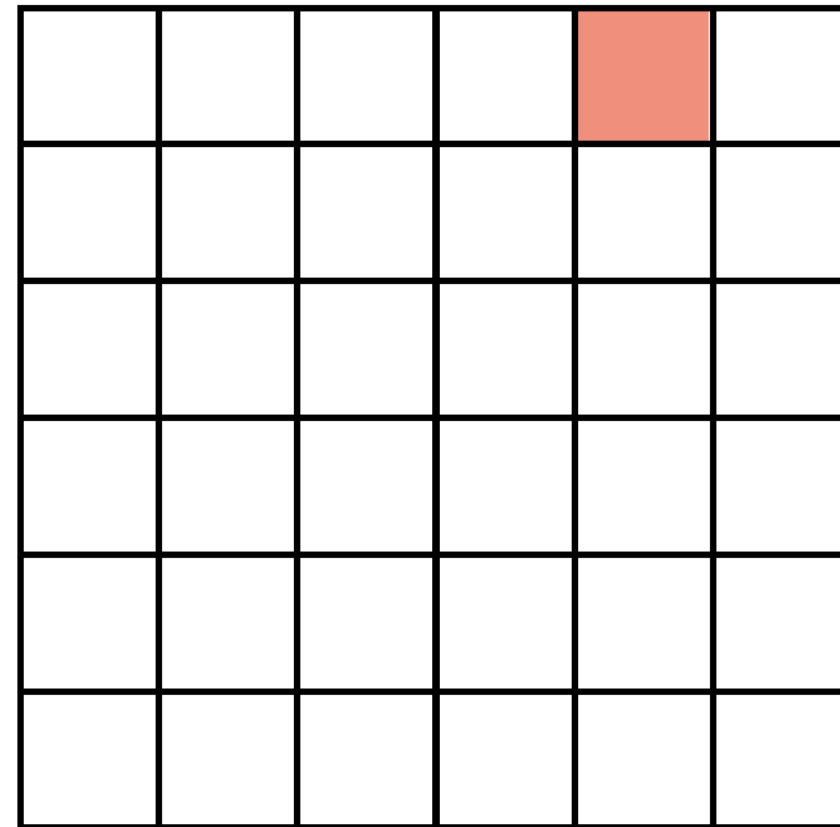
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



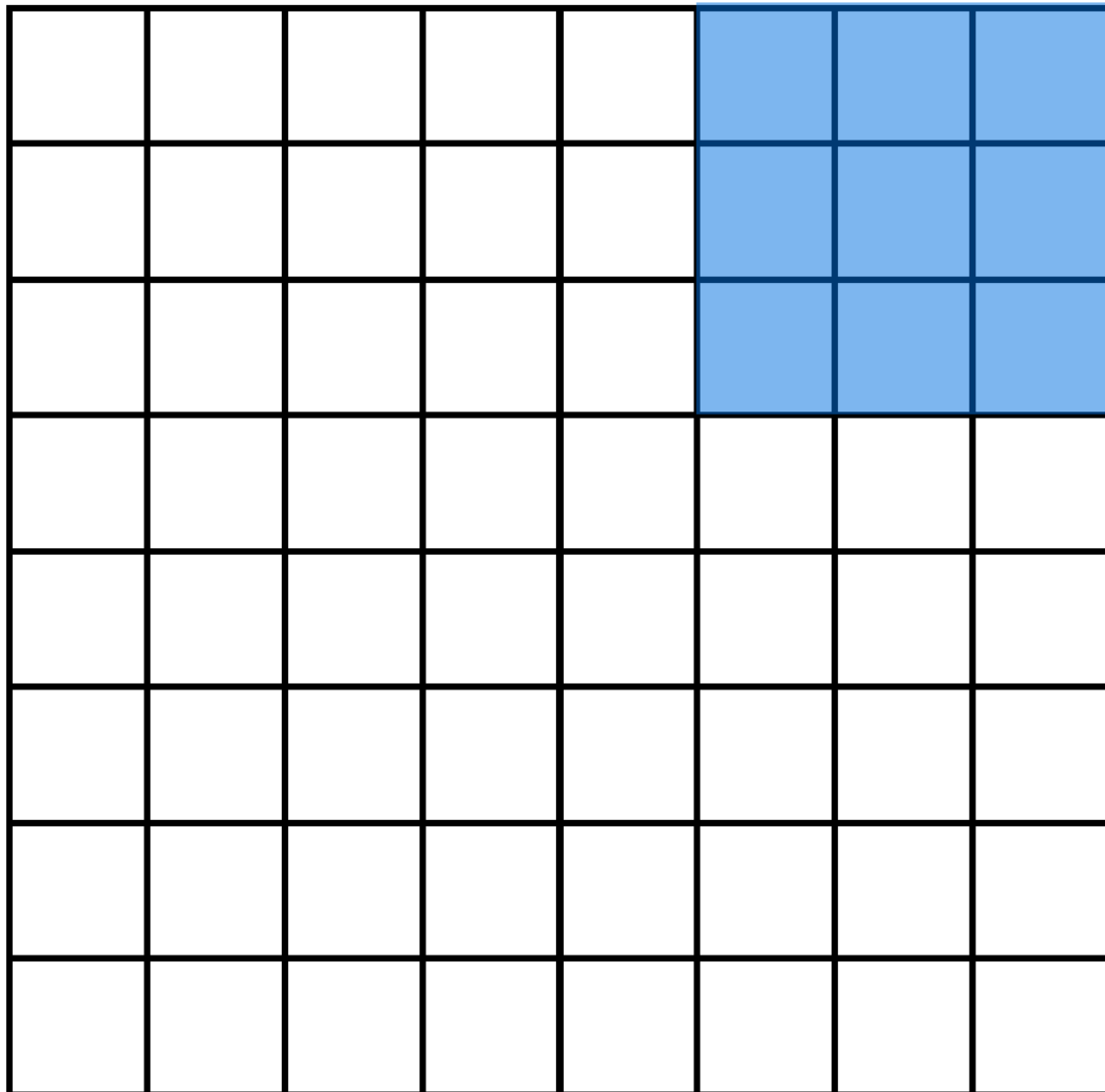
Input



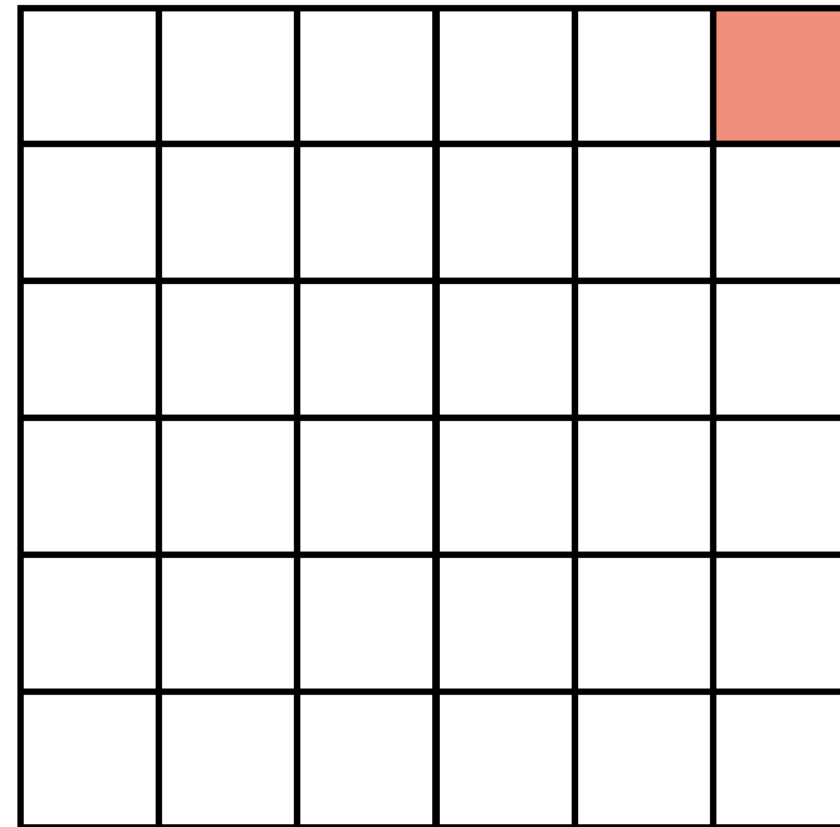
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



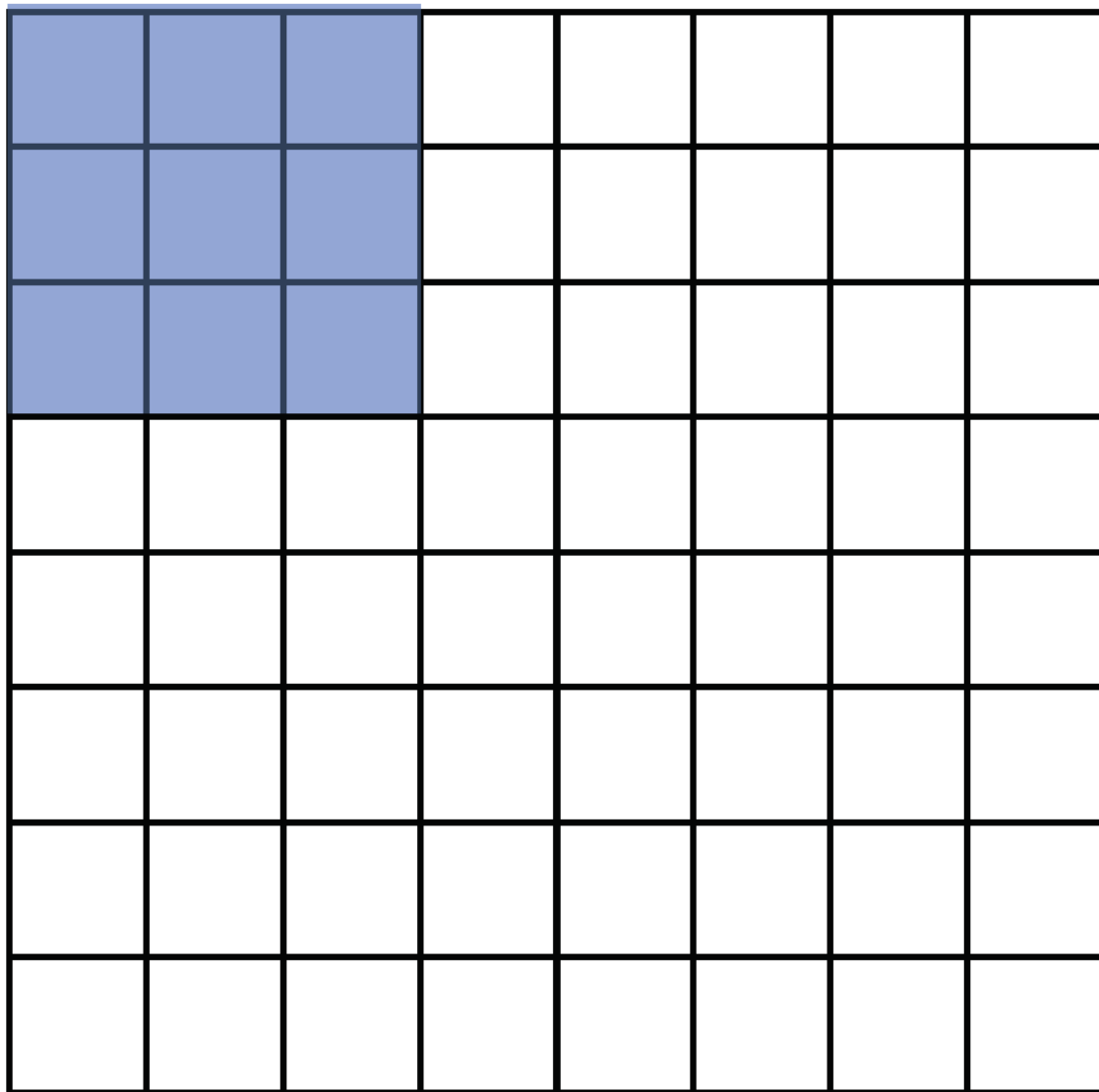
Input



Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



Input

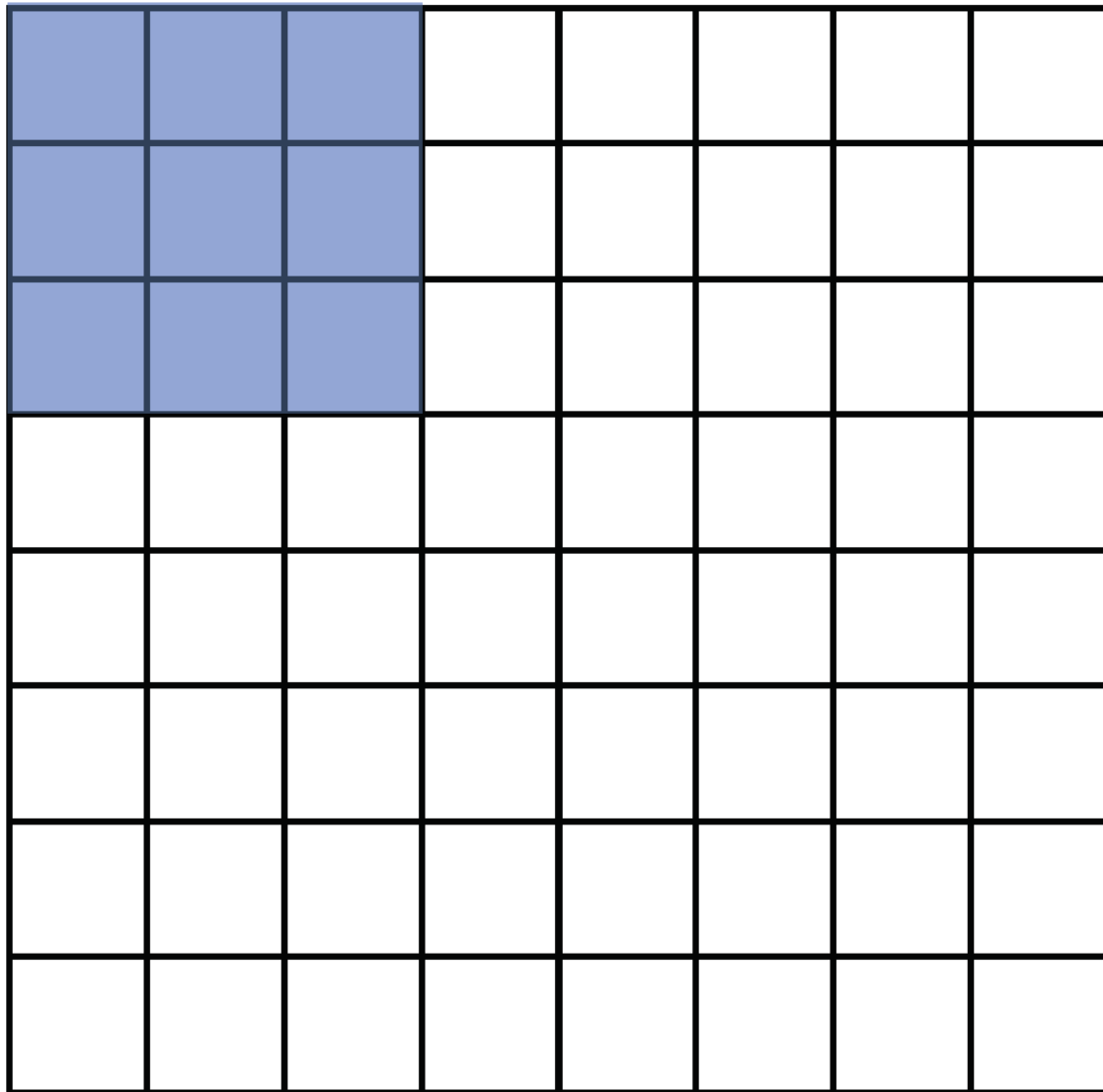
Recall that at each position, we are doing a **3D** sum:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

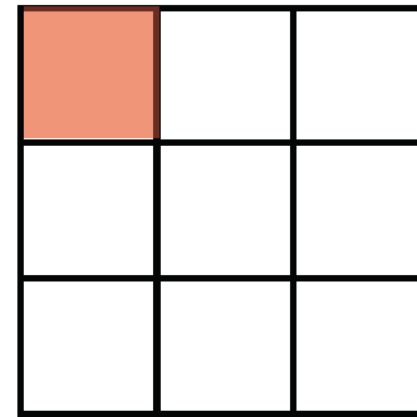
(channel, row, column)

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



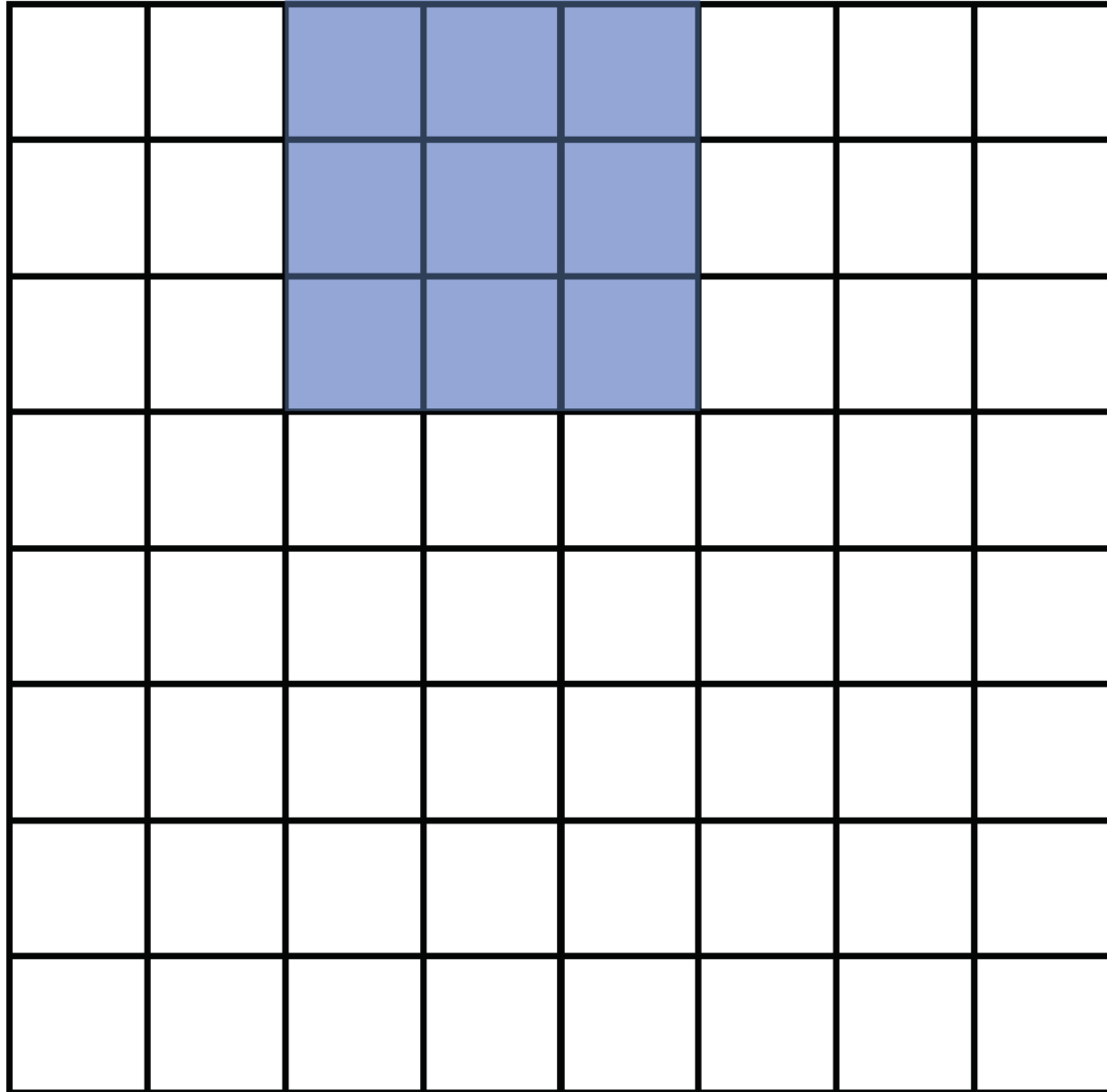
Input



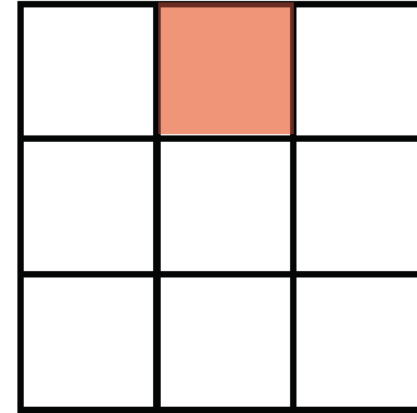
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



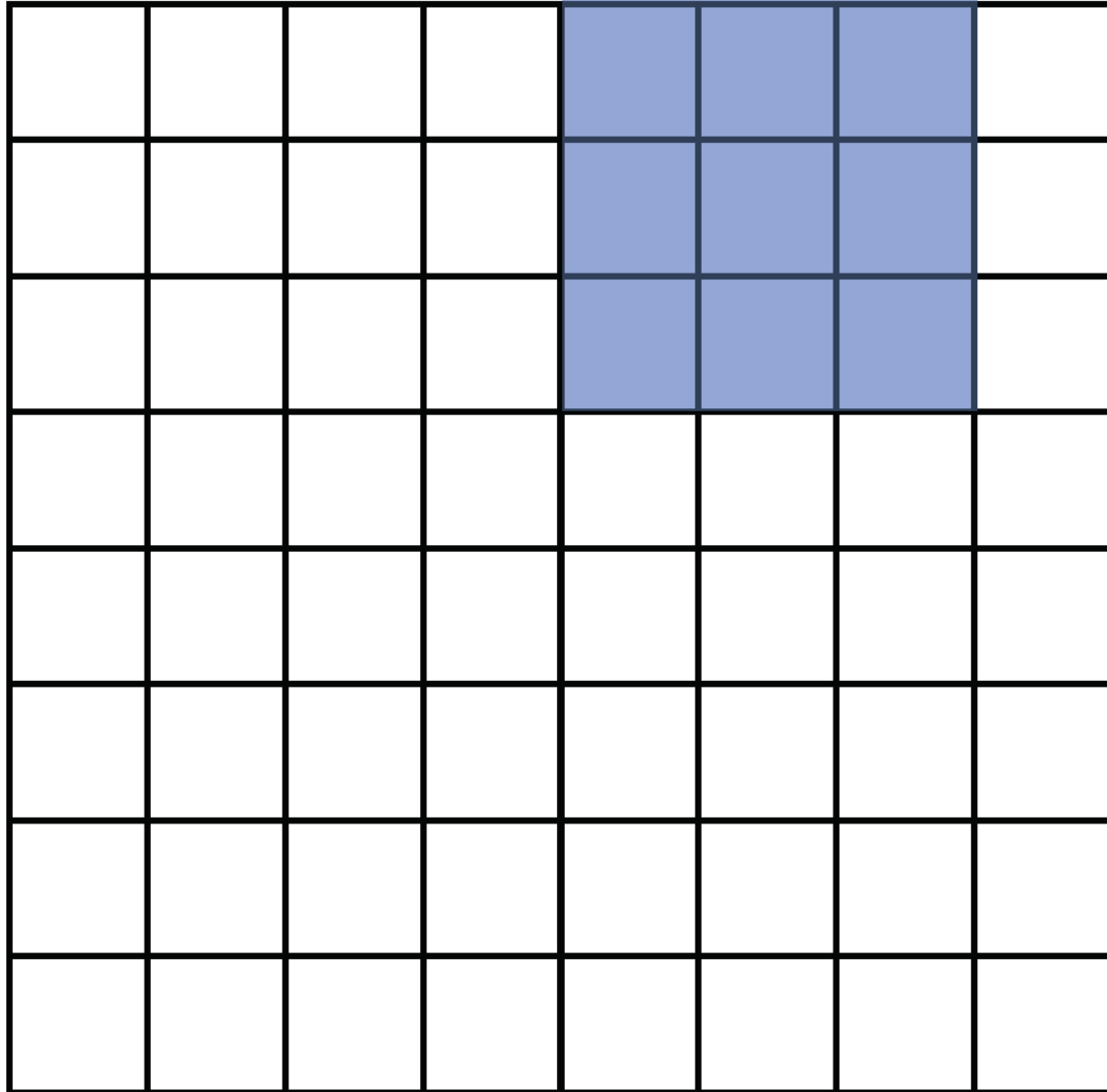
Input



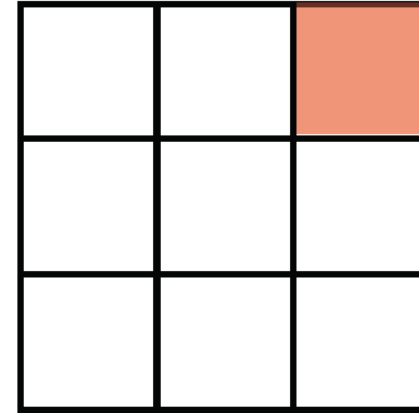
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



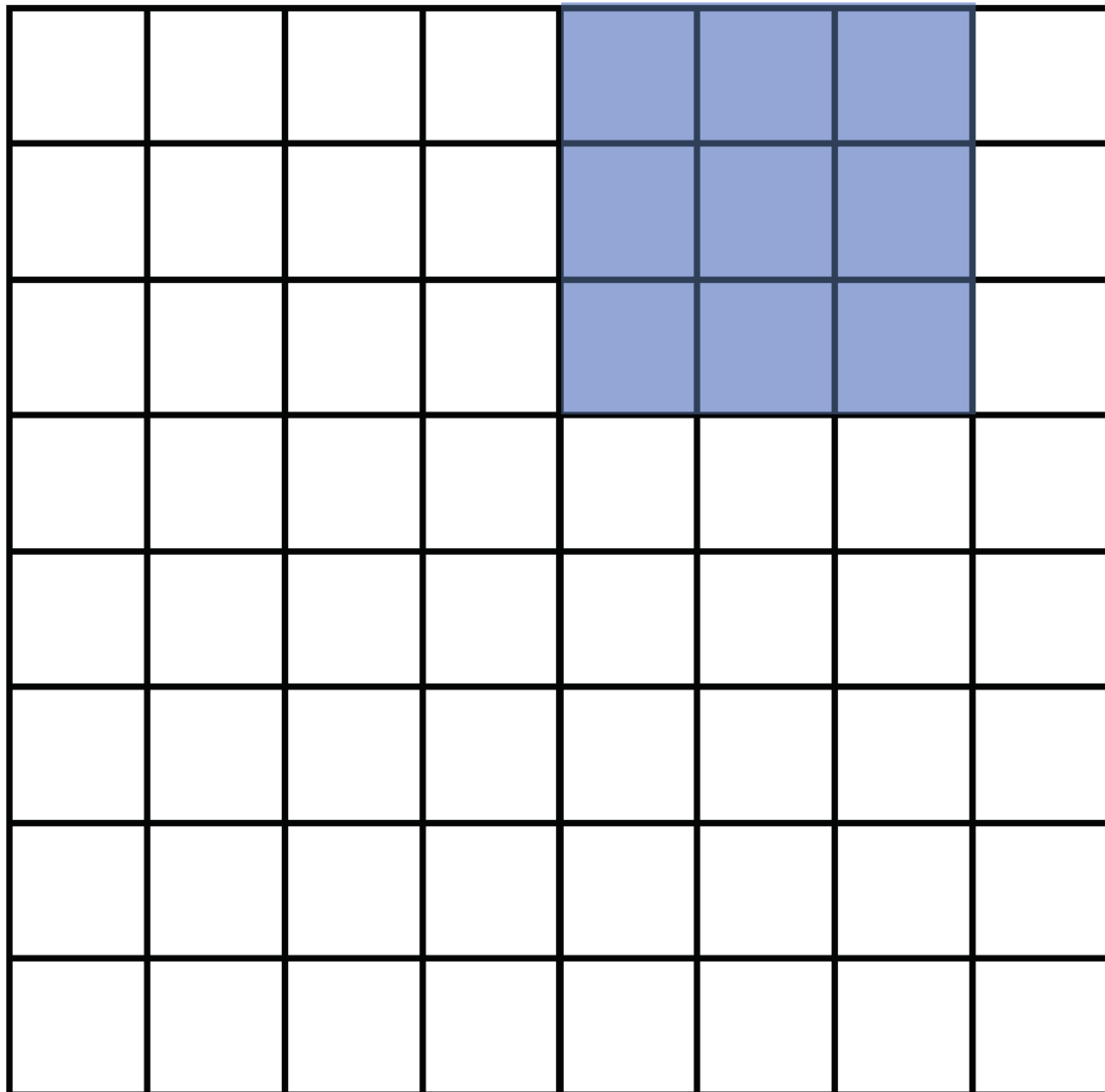
Input



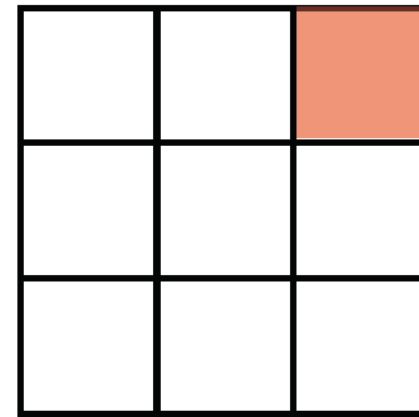
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



Input



Output

- Notice that with certain strides, we may not be able to cover all of the input
- The output is also half the size of the input

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1**, **stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1**, **stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1**, **stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1**, **stride = 2**

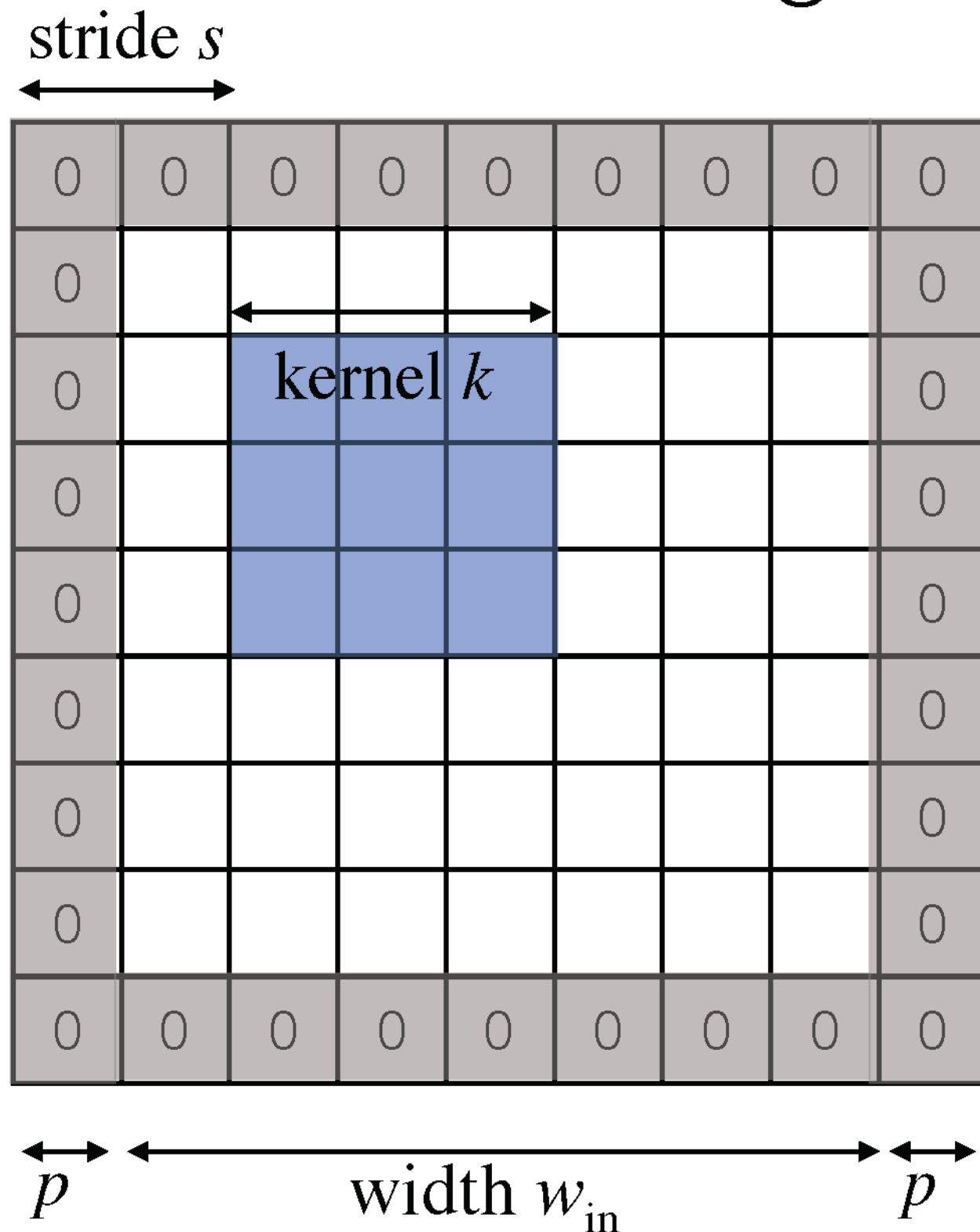
0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution:

How big is the output?

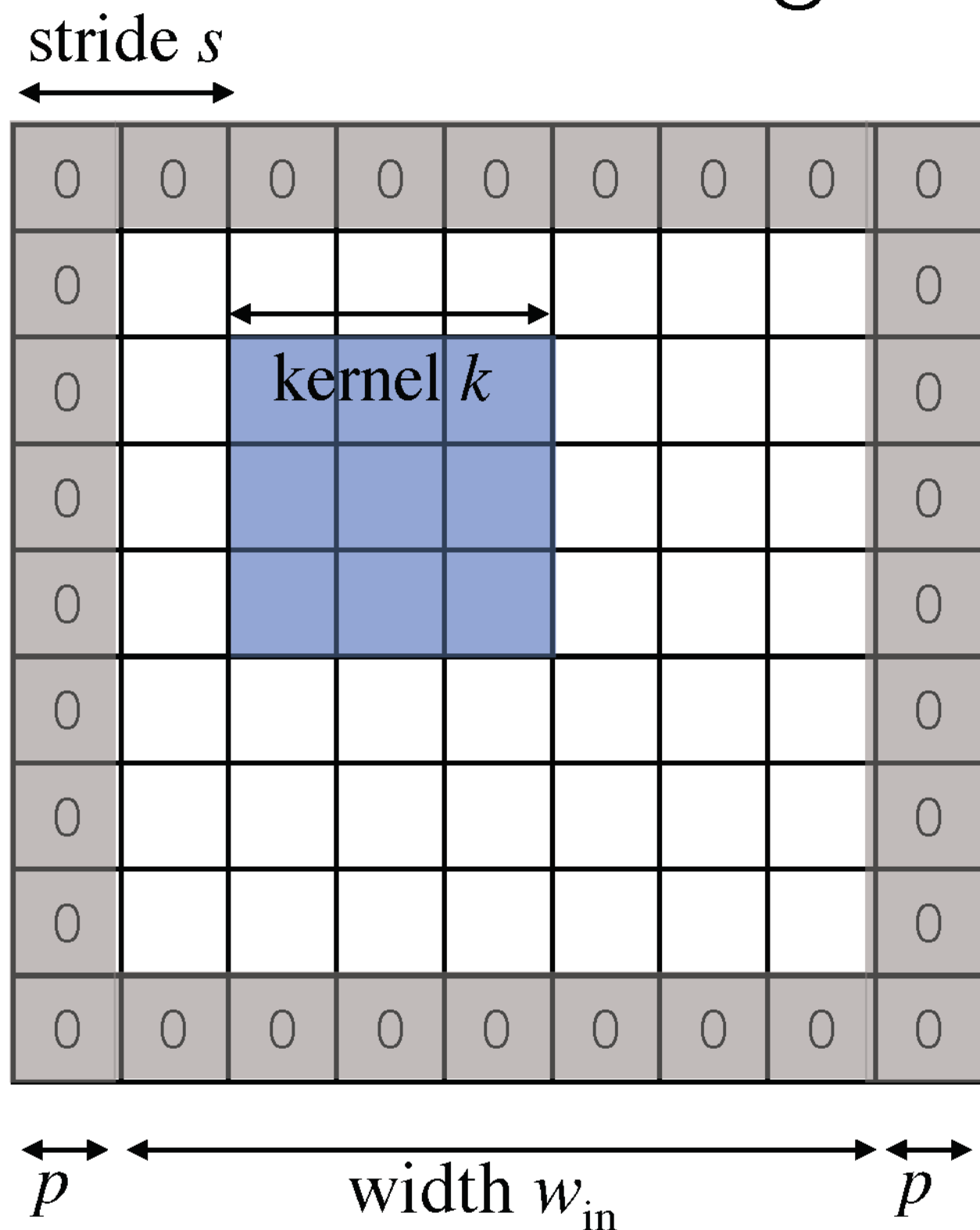


In general, the output has size:

$$w_{\text{out}} = \left\lfloor \frac{w_{\text{in}} + 2p - k}{s} \right\rfloor + 1$$

Convolution:

How big is the output?



Example: $k=3$, $s=1$, $p=1$

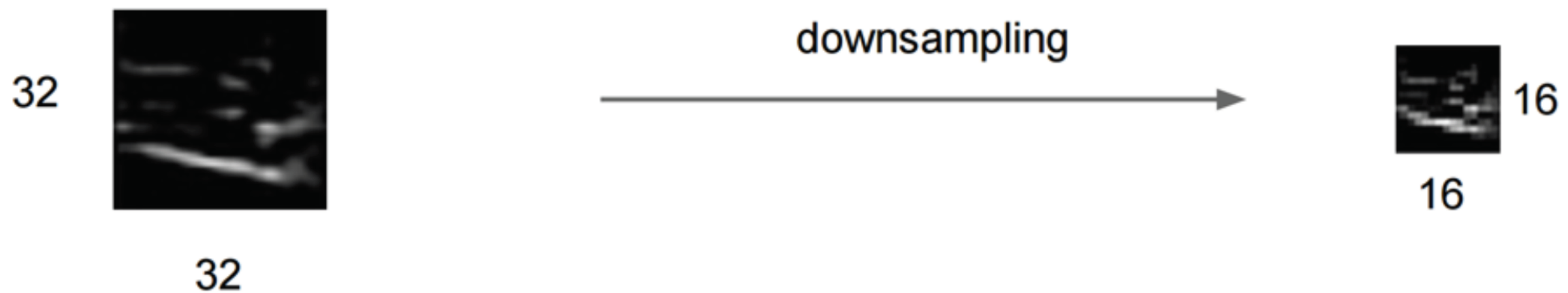
$$\begin{aligned} w_{\text{out}} &= \left\lfloor \frac{w_{\text{in}} + 2p - k}{s} \right\rfloor + 1 \\ &= \left\lfloor \frac{w_{\text{in}} + 2 - 3}{1} \right\rfloor + 1 \\ &= w_{\text{in}} \end{aligned}$$

VGGNet [Simonyan 2014]
uses filters of this shape

Pooling

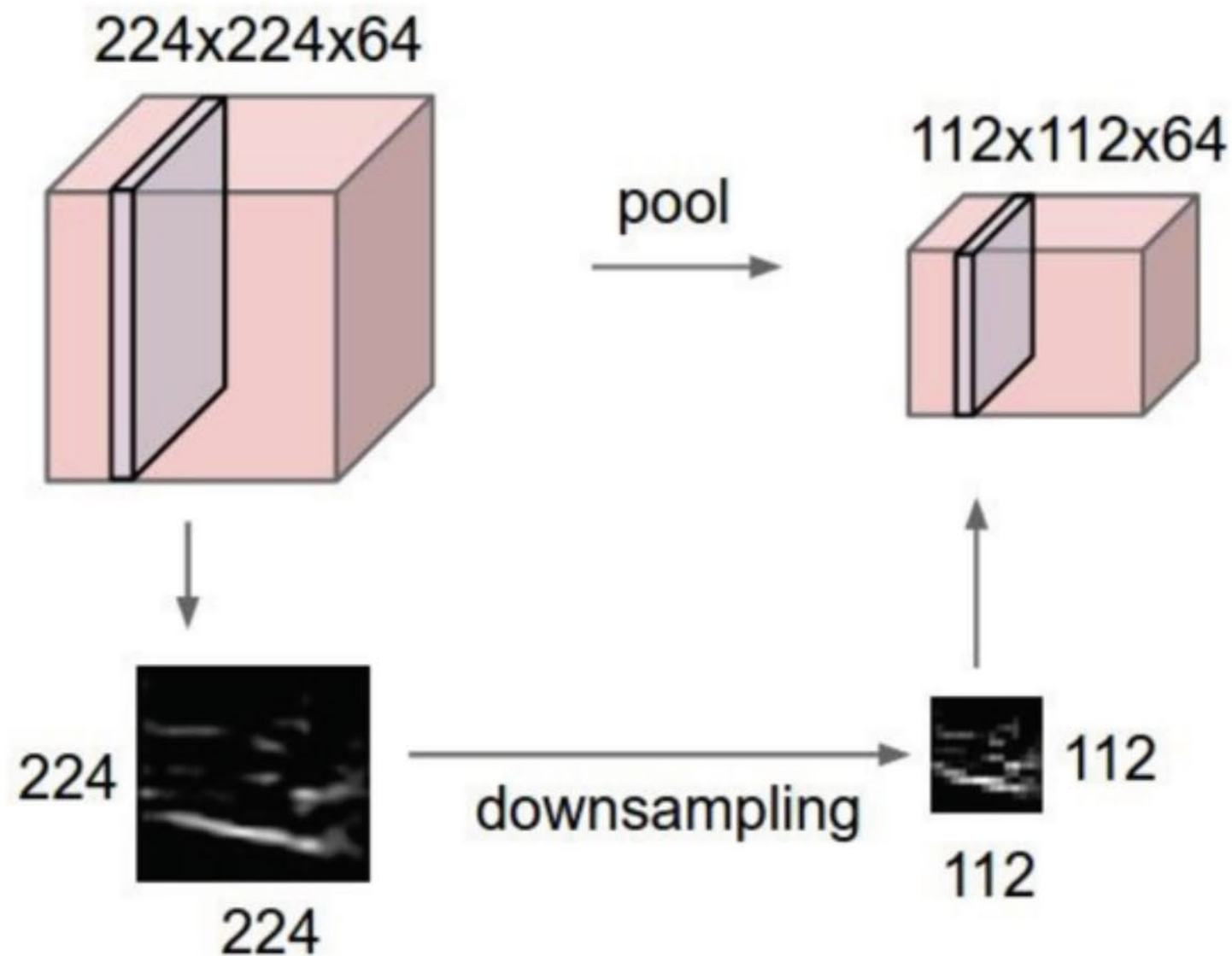
For most ConvNets, **convolution** is often followed by **pooling**:

- Creates a smaller representation while retaining the most important information
- The “max” operation is the most common
- Why might “avg” be a poor choice?

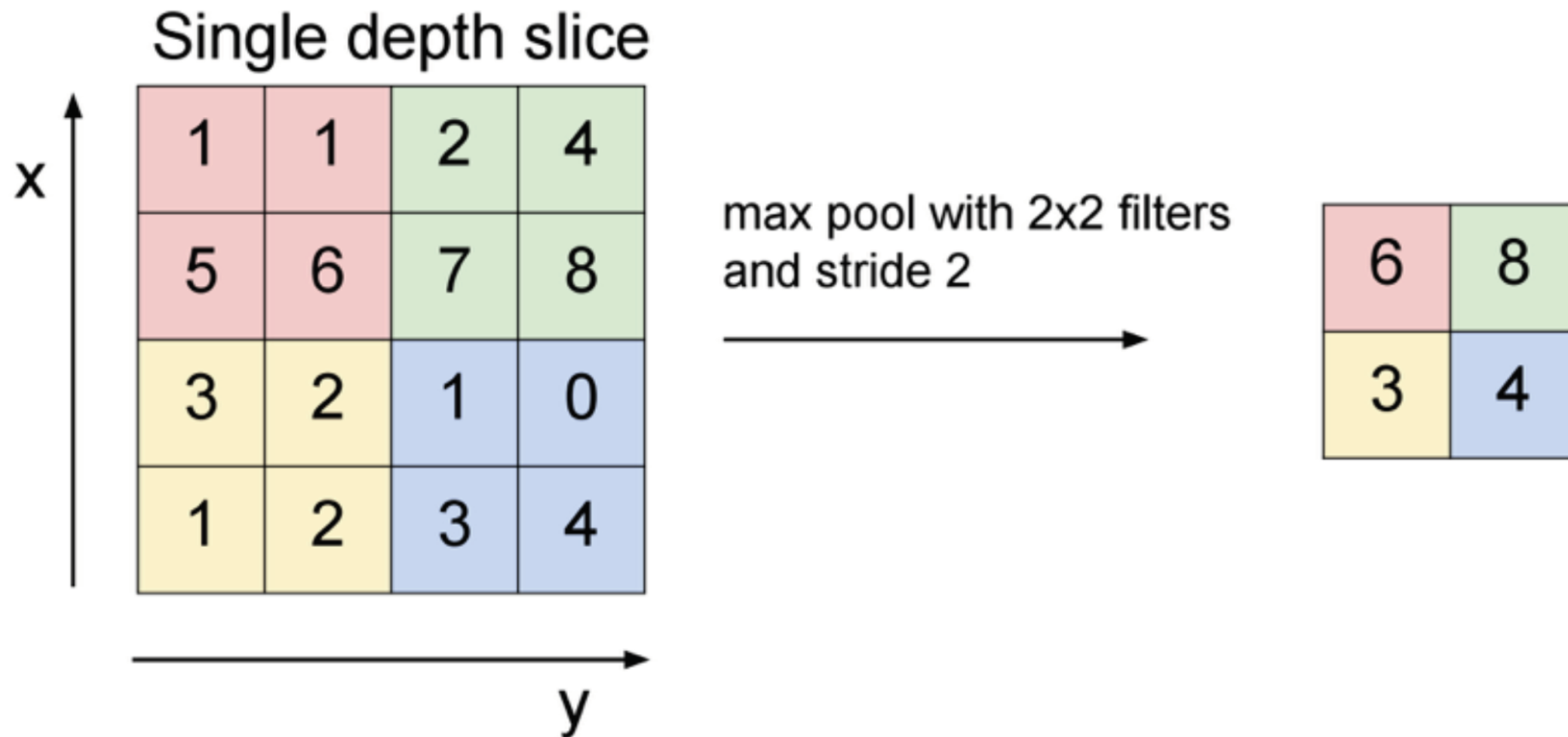


Pooling

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



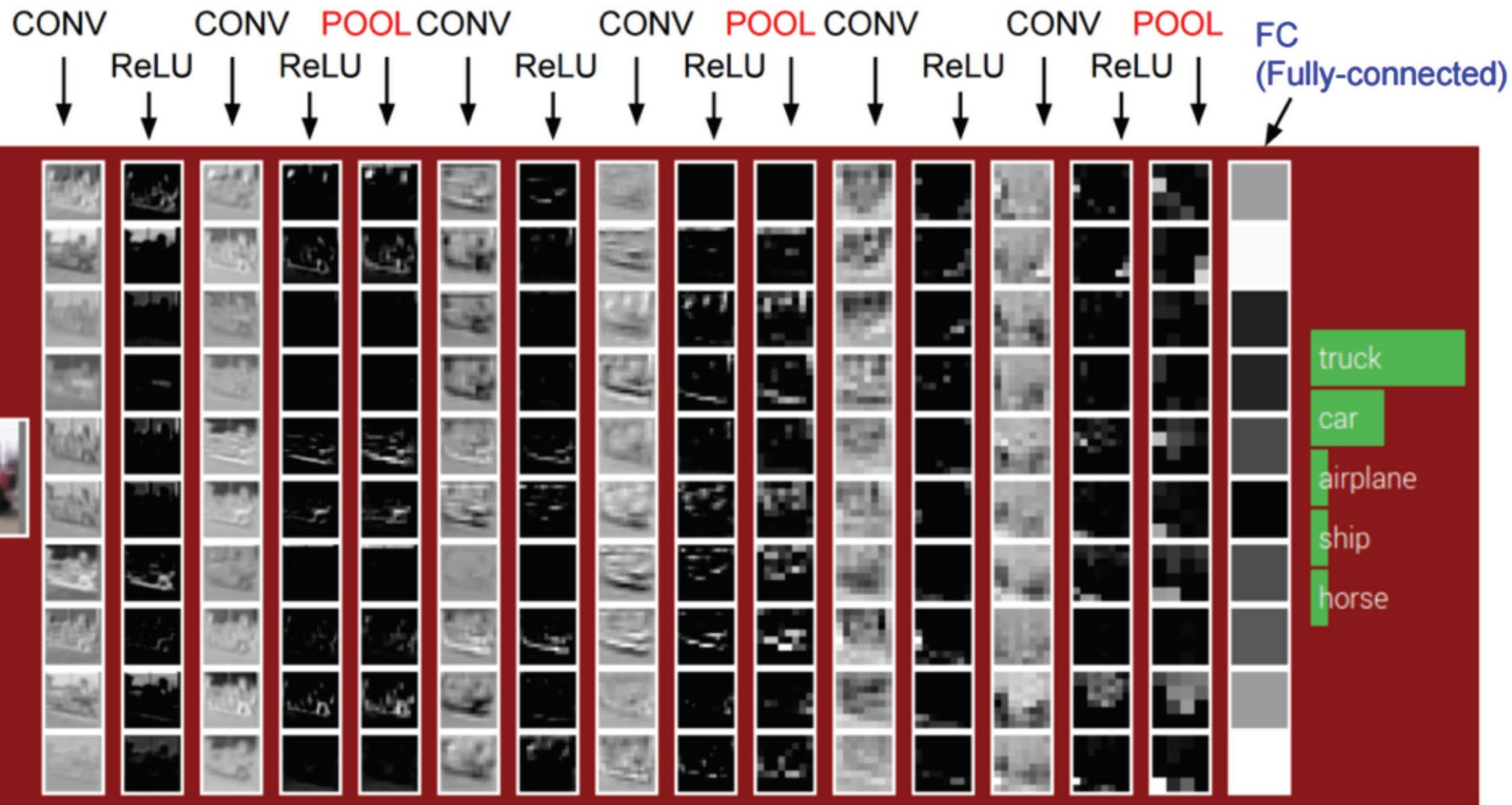
Max Pooling



What's the backprop rule for max pooling?

- In the forward pass, store the index that took the max
- The backprop gradient is the input gradient at that index

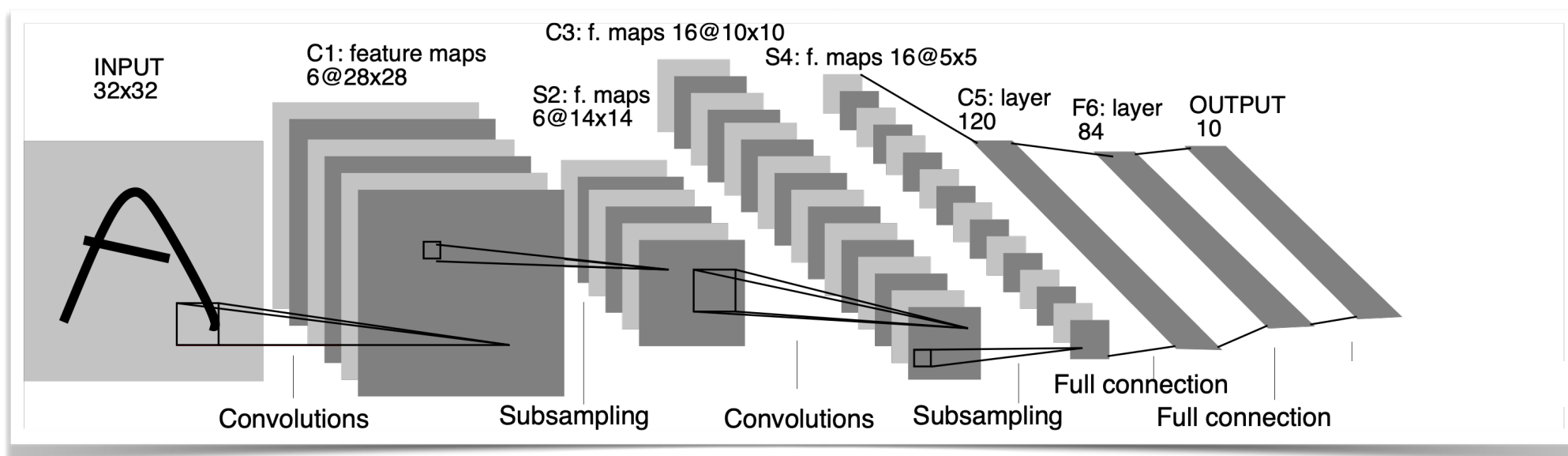
Example ConvNet



10x3x3 conv filters, stride 1, pad 1
2x2 pool filters, stride 2

Figure: Andrej Karpathy

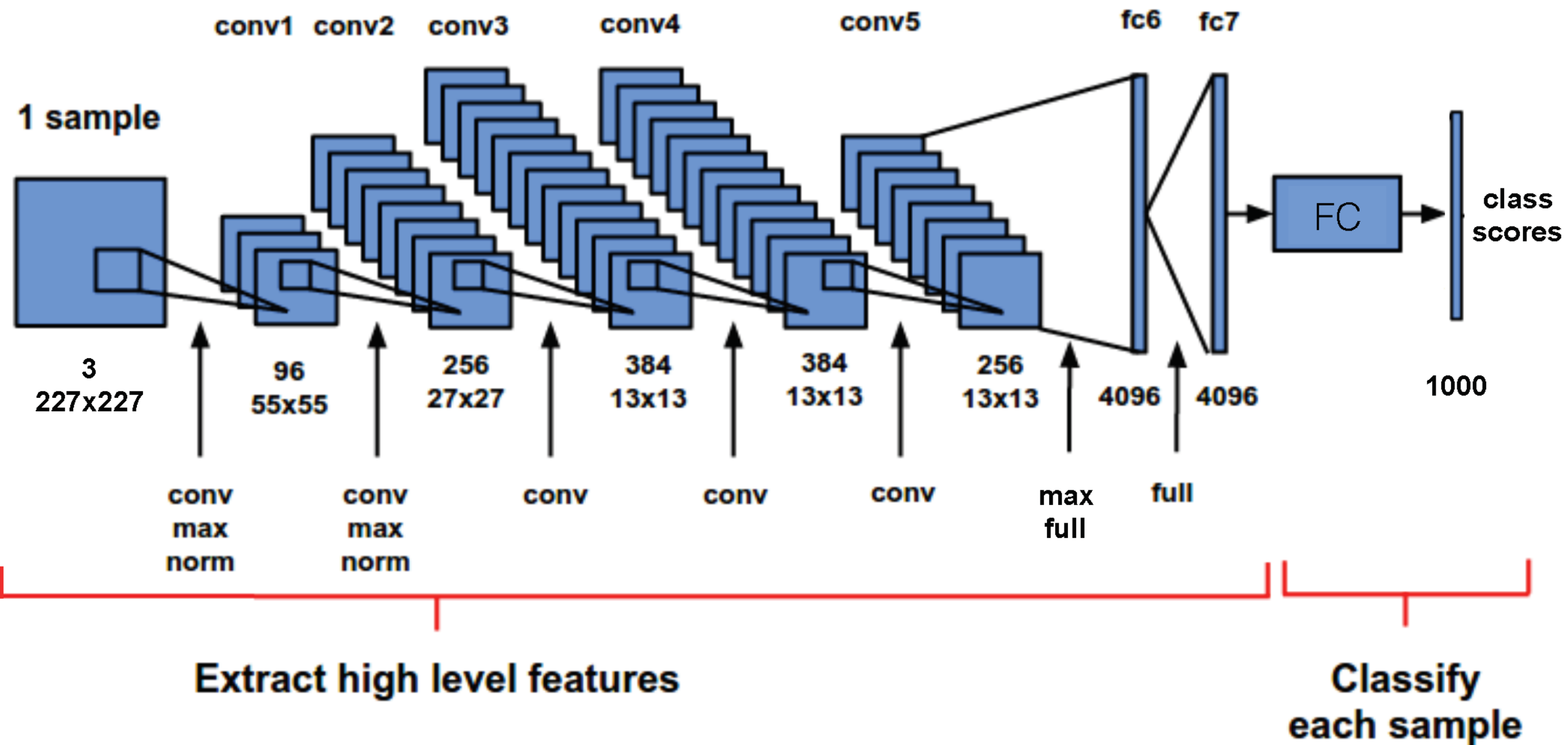
LeNet (98)



Alternate valid-convolutional layers (obtained by a filter bank and a sigmoid/tanh nonlinearity) with subsampling

“Fully-connected” layer is a bit of a misnomer (we’ll revisit)...

Example: AlexNet [Krizhevsky 2012]



"max": max pooling

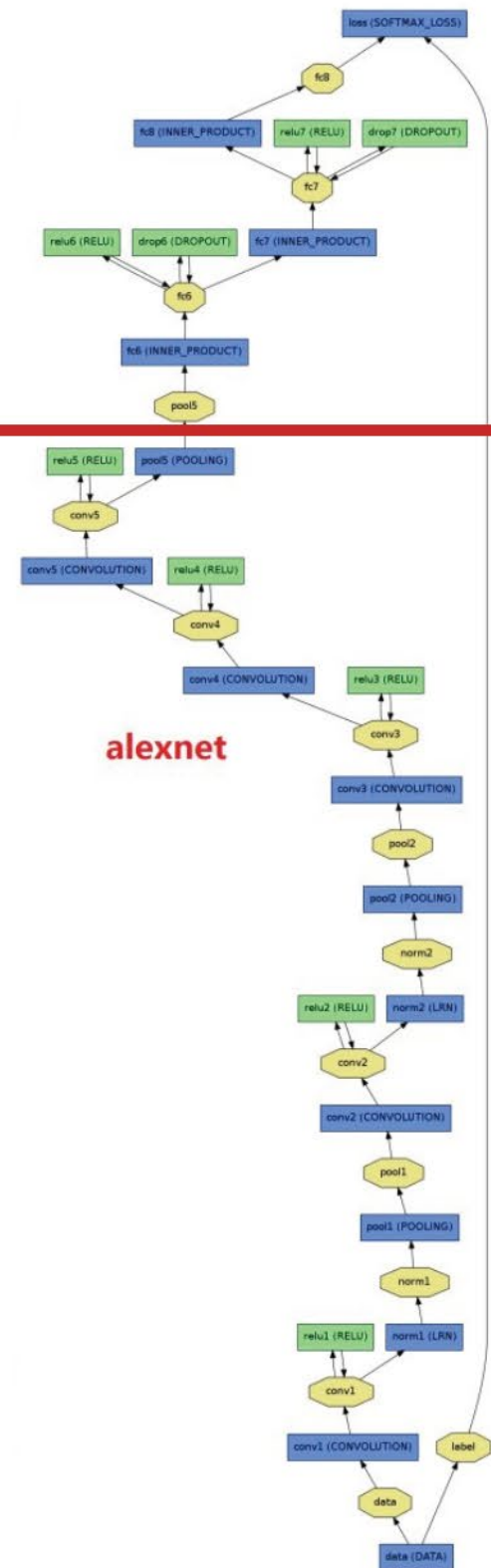
"norm": local response normalization

"full": fully connected

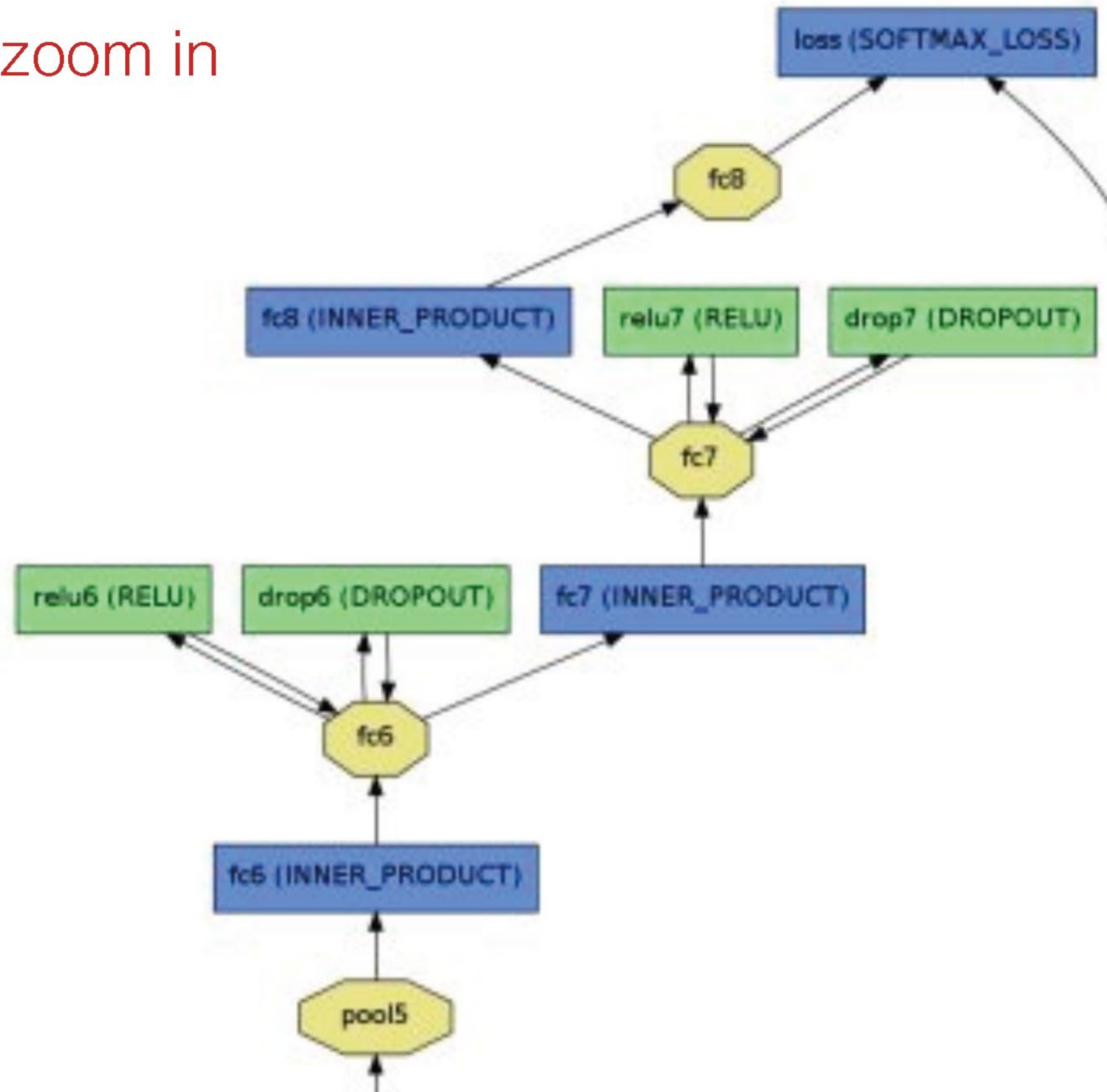
Figure: [Karnowski 2015] (with corrections)

Example: AlexNet [Krizhevsky 2012]

zoom in



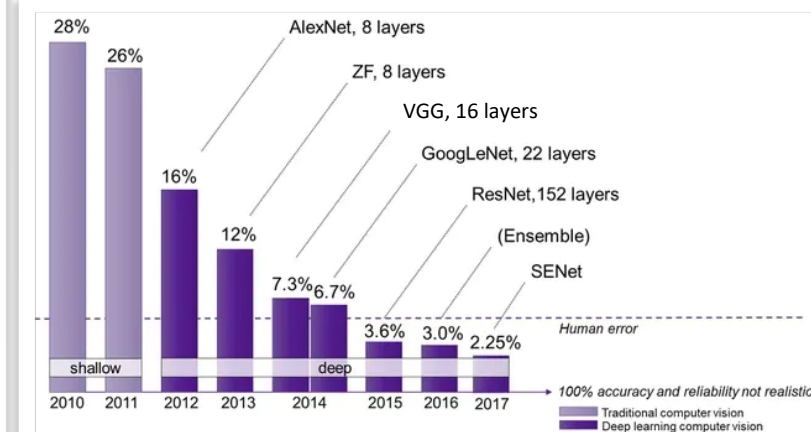
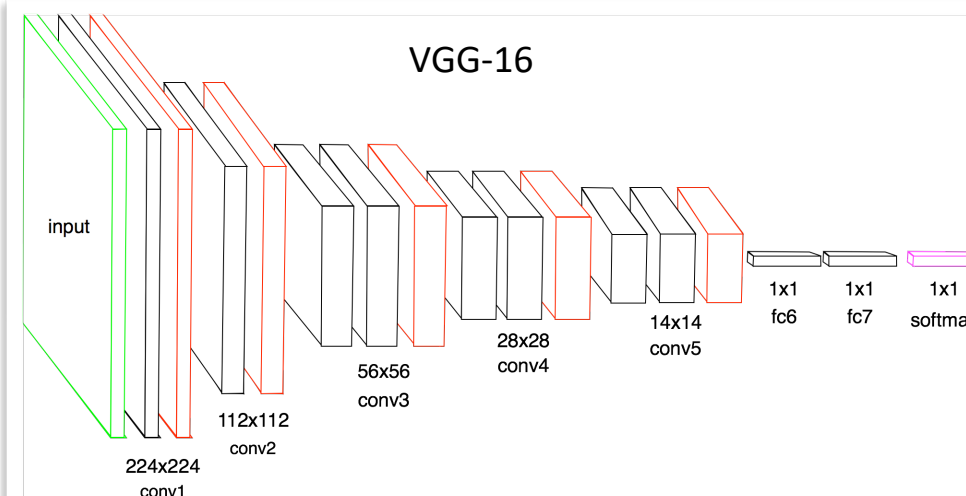
alexnet



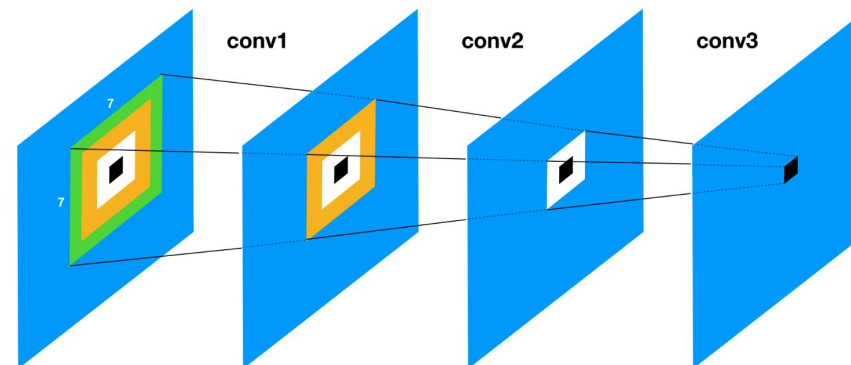
VGG networks

[era of “regular” small convolutions]

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					



Benefits of small convolutions



Decrease # of parameters

One $C_{in} \times 7 \times 7$ convolution has $49C_{in}$ parameters

Three-layers of $C_{in} \times 3 \times 3$ convolutions have $27C_{in}$ parameters

Add more nonlinearities

Two linear functions

$$b \cdot (a \cdot x)$$

Same as just one linear function

Two non-linear functions

$$F(b \cdot F(a \cdot x))$$

Can capture more complex decision boundary

Modular network design

```
class VGG16(nn.Module):
    def __init__(self):
        super(VGG16, self).__init__()
        # Initialize convolutional blocks using a helper function
        self.features = self._make_layers([64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'])
        self.flatten = nn.Flatten() # Flatten layer to convert 3D feature maps to 1D feature vectors
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, 1000) # Assuming 1000 classes for classification
        )

    def _make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1), nn.ReLU(inplace=True)]
                in_channels = x
        return nn.Sequential(*layers) # Instantiate Sequential nn.Model by passing in the unpacked list of layers

    def forward(self, x):
        # Assume input is [batch_size 3 224 224]
        x = self.features(x) # Apply convolutional blocks
        x = self.flatten(x) # Flatten the output for the classifier
        x = self.classifier(x) # Apply the classifier
        return x
```

Since filters are always 3x3, define network VGG-16 with conv layers that depend on # of channels (e.g., 256) and max-pooling layers “M”
Can explore architecture variants with a simple **config** change

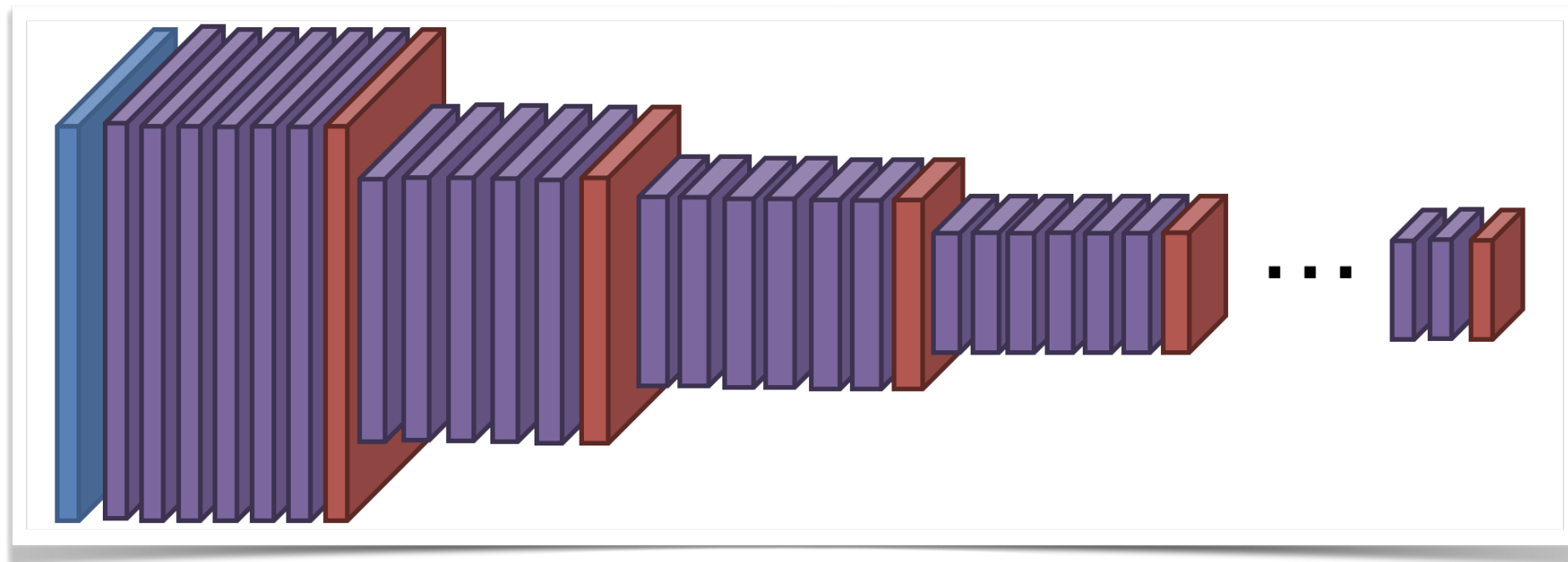
Training Deeper Networks

Every backpropagation step multiplies the gradient by the local gradient

$$1 * d * d * d \dots * d = d^{n-1}$$

If $d \ll 1$, n is big $\rightarrow$ Vanishing Gradients (more common)

If $d \gg 1$, n is big $\rightarrow$ Exploding Gradients



Implies that it is harder to train lower layers

Deep Residual Learning for Image Recognition

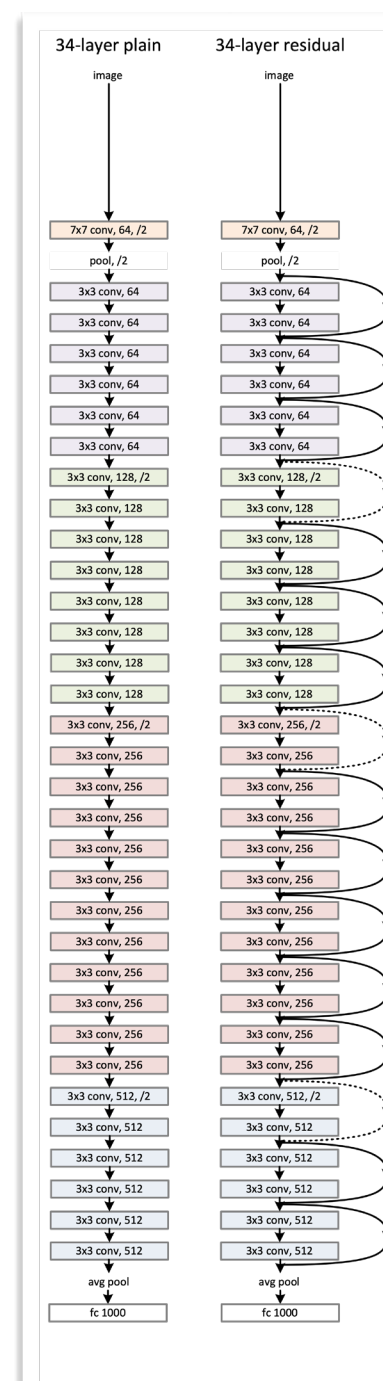
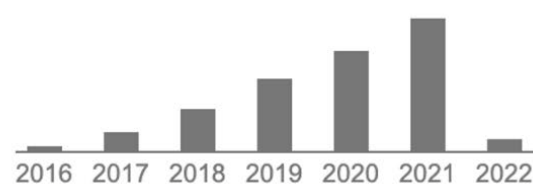
Authors Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Publication date 2016

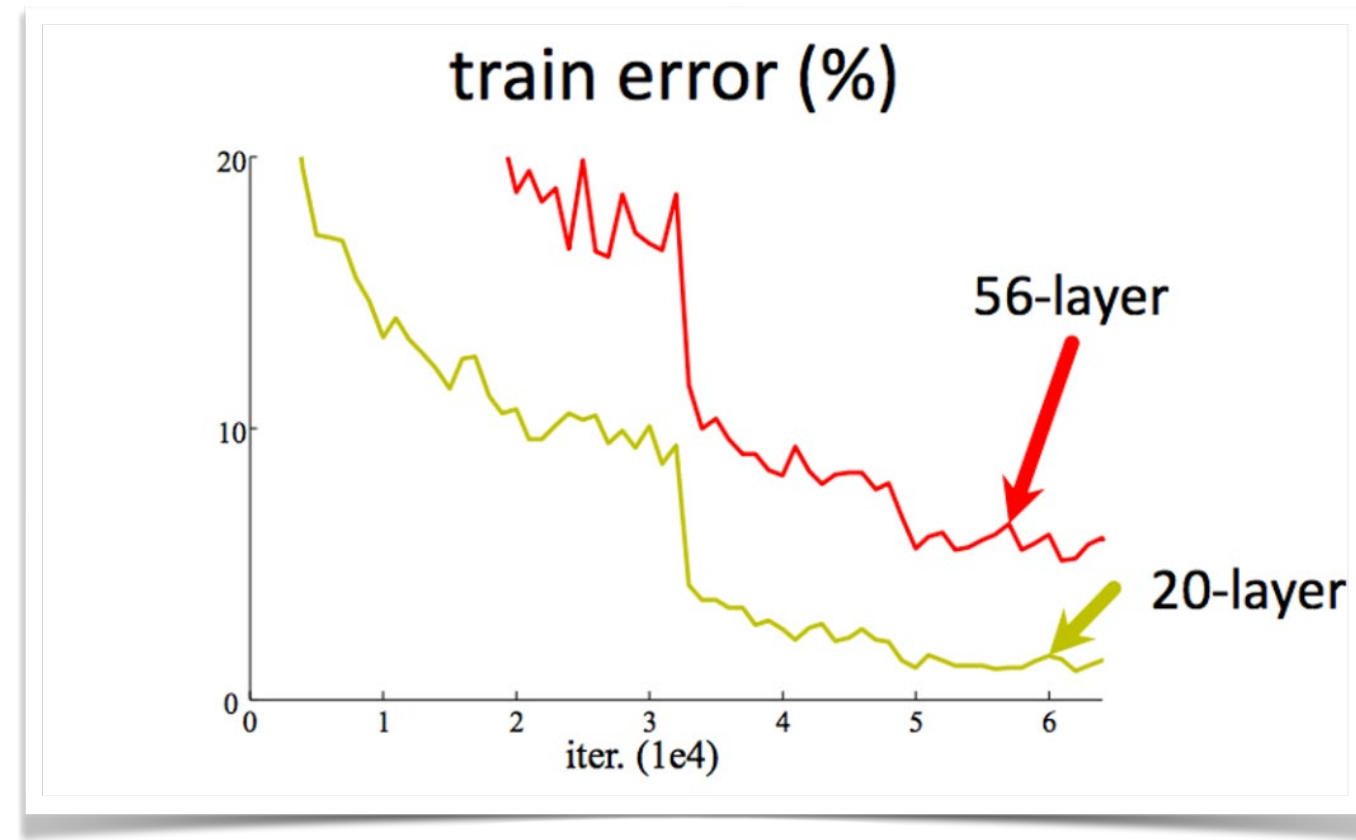
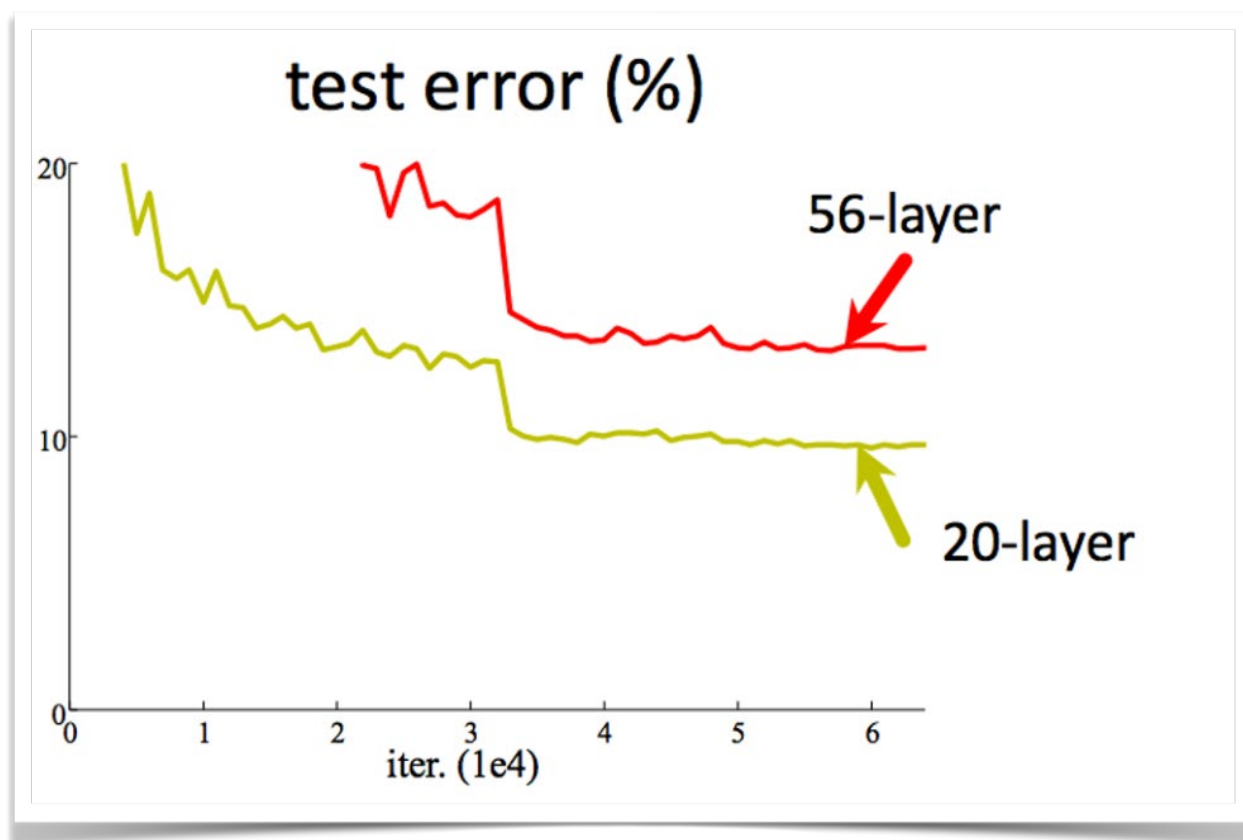
Conference Computer Vision and Pattern Recognition (CVPR), 2016

Description Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers---8x deeper than VGG nets but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers. The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions, where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

Total citations Cited by 106761



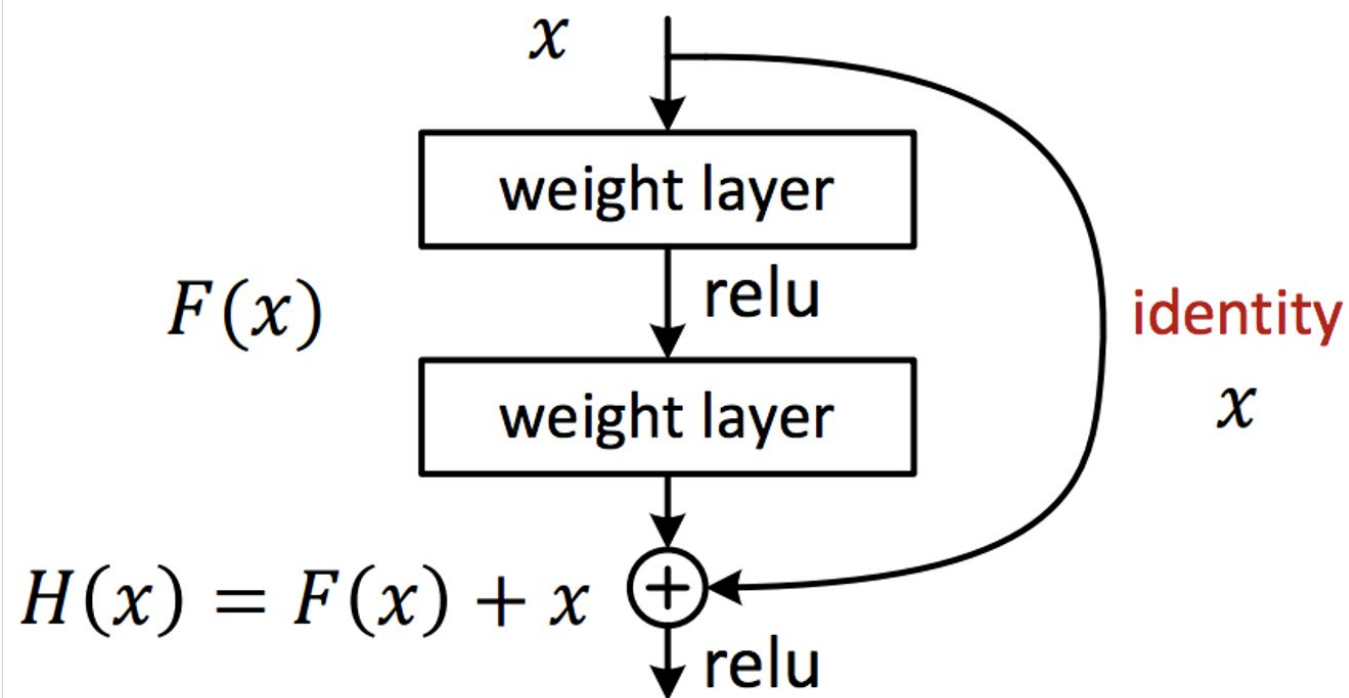
ResNet: Scaling to 100+ Layers



- Continuing to stack 3x3 convolutions over more layers does not help; 56-layer net has higher test error than 20-layer net. Why? Overfitting?
- No... training error is also worse. Implies that optimization (and *not* generalization) is the culprit

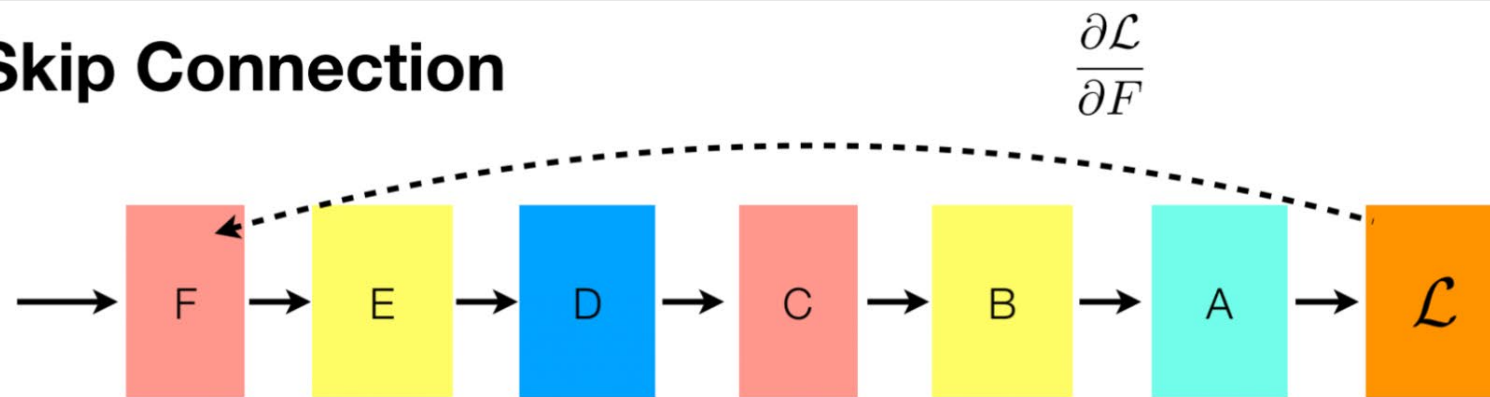
Deep Residual Learning

- **Residual** net



- If x is optimal, we can set weight layers to 0. It's easy to "ignore" extra model capacity.
- If x is close to optimal, it's easier to find small weights.
- Identity serves as a "gradient highway" to preserve strong signal from output to earlier layers during back propagation.

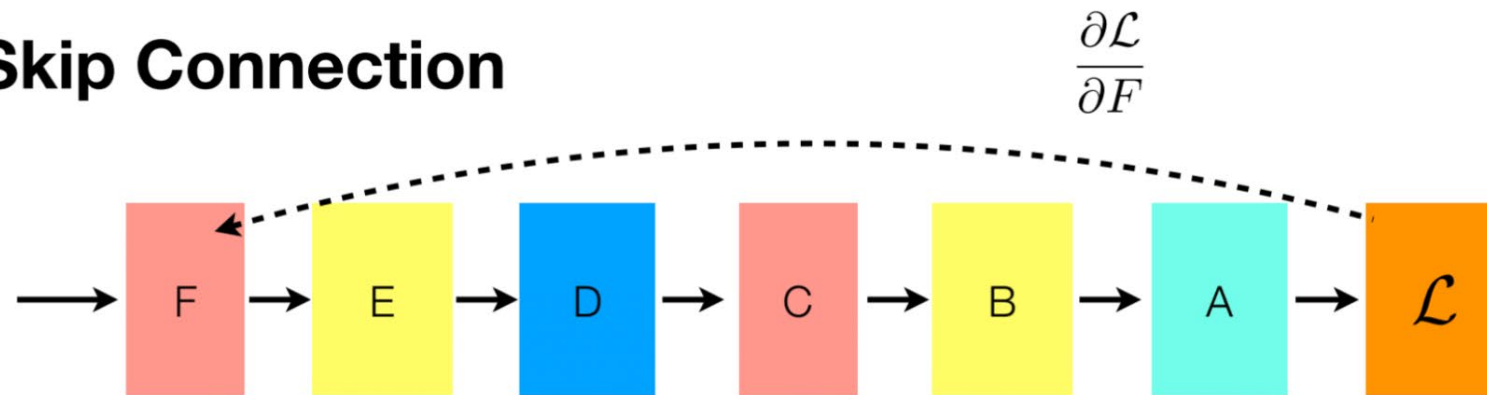
No Skip Connection



$$\frac{\partial \mathcal{L}}{\partial F} = \frac{\partial \mathcal{L}}{\partial A} \frac{\partial A}{\partial B} \frac{\partial B}{\partial C} \frac{\partial C}{\partial D} \frac{\partial D}{\partial E} \frac{\partial E}{\partial F}$$

Goes to zero

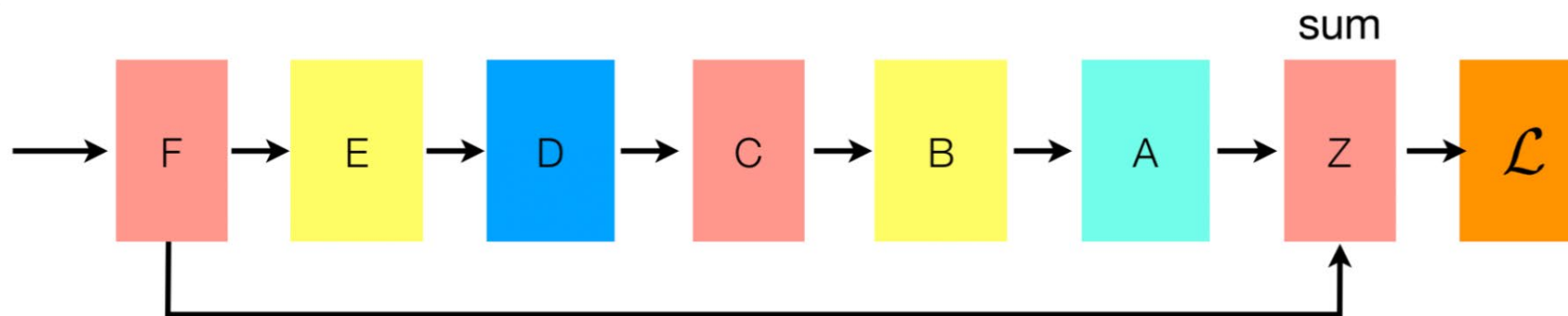
No Skip Connection



$$\frac{\partial \mathcal{L}}{\partial F} = \frac{\partial \mathcal{L}}{\partial A} \frac{\partial A}{\partial B} \frac{\partial B}{\partial C} \frac{\partial C}{\partial D} \frac{\partial D}{\partial E} \frac{\partial E}{\partial F}$$

Goes to zero

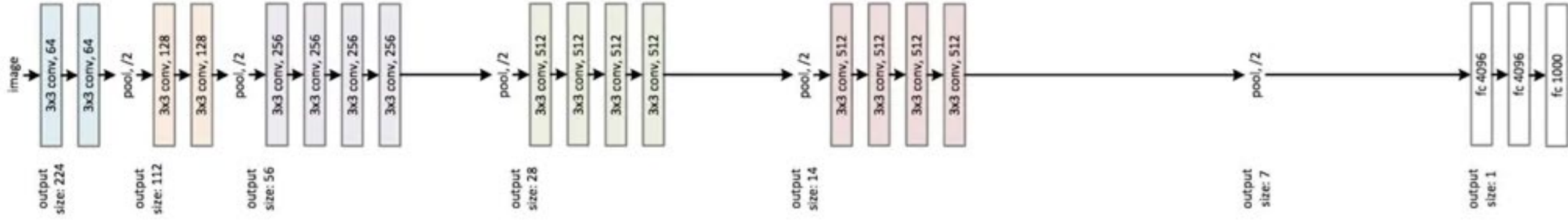
Skip Connection “gradient highways”



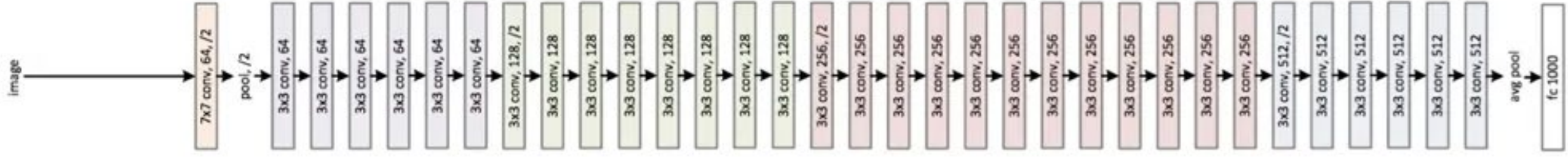
$$\frac{\partial \mathcal{L}}{\partial F} = \frac{\partial \mathcal{L}}{\partial Z} \left(\frac{\partial A}{\partial B} \frac{\partial B}{\partial C} \frac{\partial C}{\partial D} \frac{\partial D}{\partial E} \frac{\partial E}{\partial F} + 1 \right)$$

Doesn't go to zero

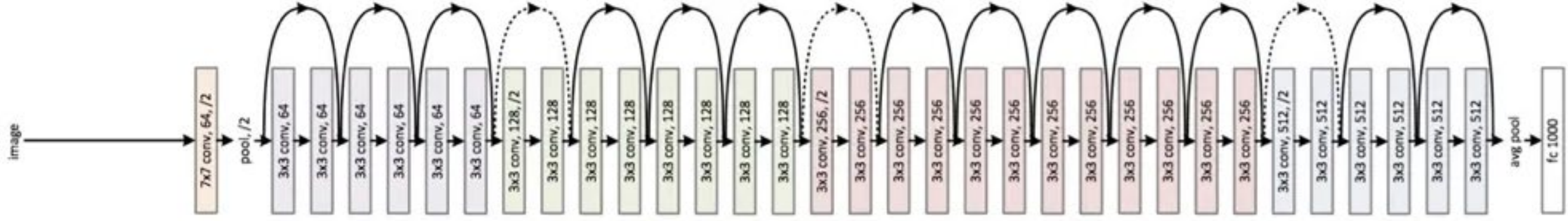
VGG-19



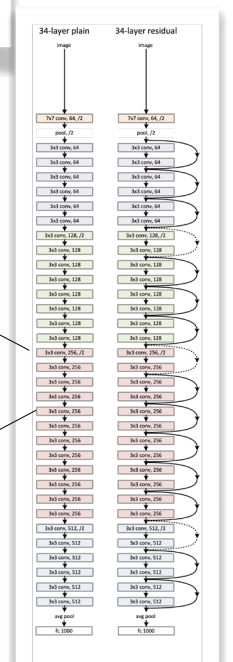
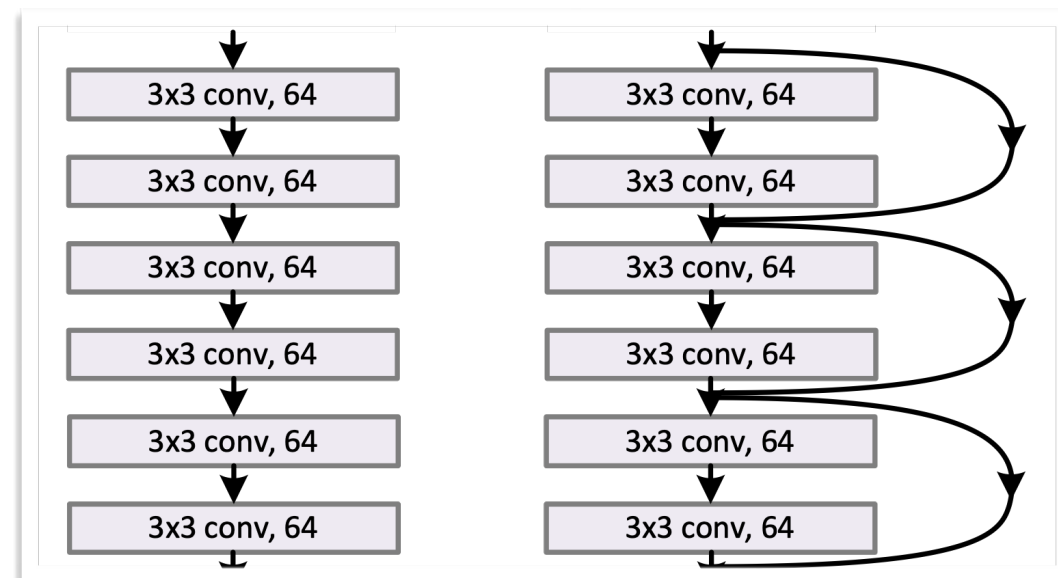
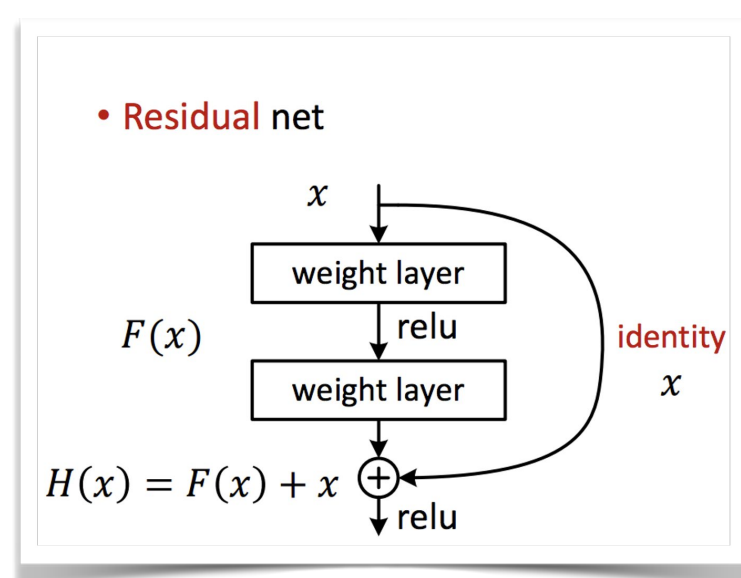
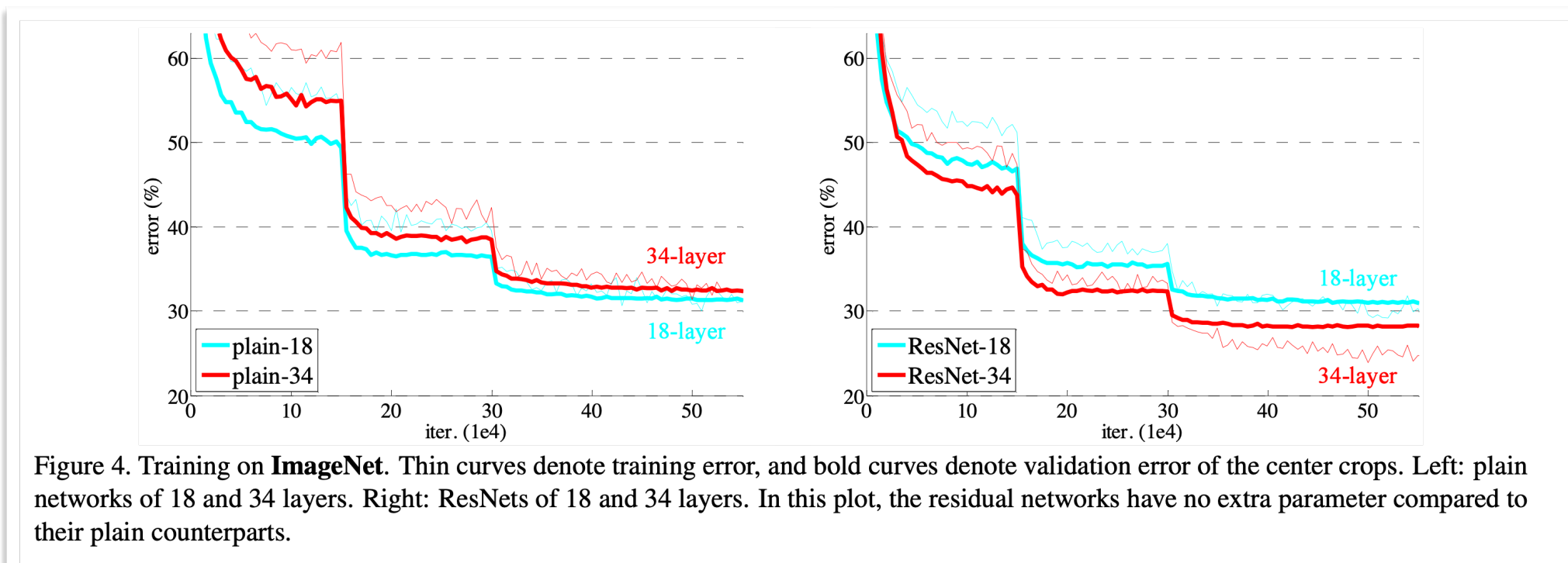
34-layer plain



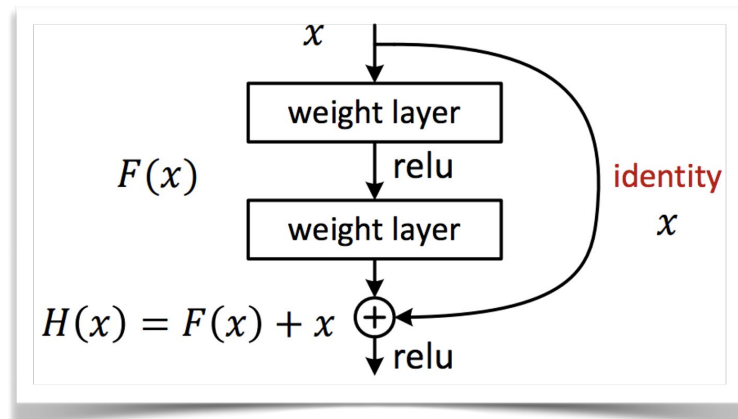
34-layer residual



Intuition: residual connections bias deep models to behave like shallow models during learning



ResNetBlock



```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        # First conv may have a stride, second must have stride of 1
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            # If needed, use 1x1 filter to map in_channels to out_channels
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = self.shortcut(x) # Shortcut connection
        out = self.conv1(x)        # Apply Conv1
        out = self.bn1(out)         # Apply BatchNorm
        out = F.relu(out)          # Apply ReLU
        out = self.conv2(out)       # Apply Conv2
        out = self.bn2(out)         # Apply BatchNorm

        out += identity            # Element-wise addition with shortcut
        out = F.relu(out)          # Apply ReLU
        return out
```

Repeat ResNetBlocks

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=1000):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Residual layers
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)

        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, block, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)        # Initial conv: [batch, 3, 224, 224] -> [batch, 64, 112, 112]
        x = self.bn1(x)          # Initial batch norm
        x = F.relu(x)            # Initial ReLU
        x = self.pool1(x)        # Initial pooling: [batch, 64, 112, 112] -> [batch, 64, 56, 56]

        x = self.layer1(x)       # Layer 1: [batch, 64, 56, 56] -> [batch, 64, 56, 56]
        x = self.layer2(x)       # Layer 2: [batch, 64, 56, 56] -> [batch, 128, 28, 28]
        x = self.layer3(x)       # Layer 3: [batch, 128, 28, 28] -> [batch, 256, 14, 14]
        x = self.layer4(x)       # Layer 4: [batch, 256, 14, 14] -> [batch, 512, 7, 7]

        x = self.avg_pool(x)     # Adaptive average pooling: [batch, 512, 7, 7] -> [batch, 512, 1, 1]
        x = torch.flatten(x, 1)  # Flatten: [batch, 512, 1, 1] -> [batch, 512]
        x = self.fc(x)           # Fully connected: [batch, 512] -> [batch, num_classes]
        return x

# Define the ResNet18 model
def ResNet18():
    return ResNet(BasicBlock, [2, 2, 2, 2], num_classes=1000) # 2 blocks in each of the 4 layers
```

Training ConvNets

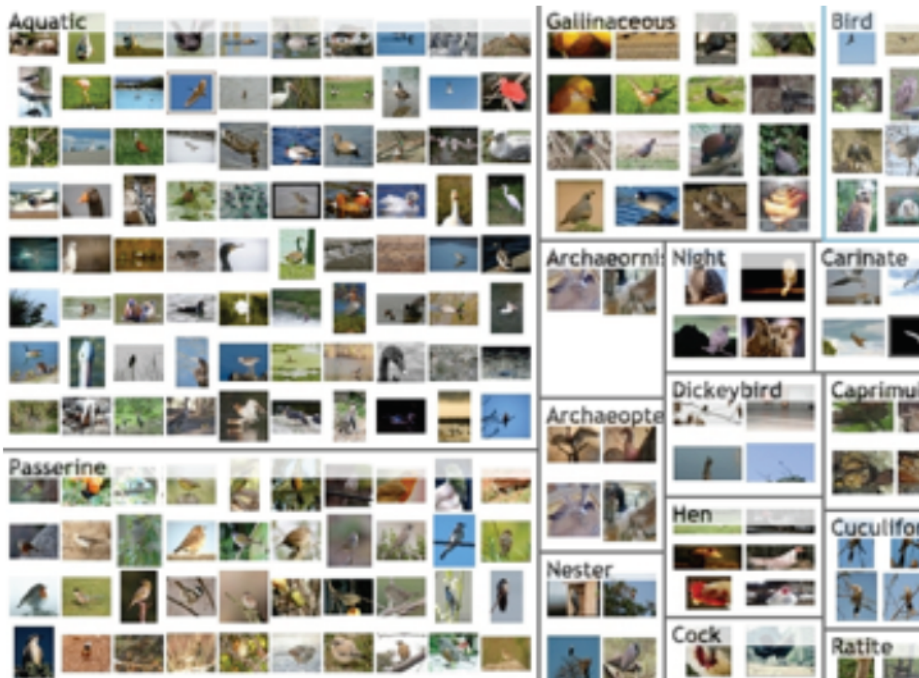
How do you actually train these things?

Roughly speaking:

Gather
labeled data

Find a ConvNet
architecture

Minimize
the loss



Training a convolutional neural network

- Split and preprocess your data
- Choose your network architecture
- Initialize the weights
- Find a learning rate and regularization strength
- Minimize the loss and monitor progress
- Fiddle with knobs

Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
3. Backwards pass: compute gradient
4. Update all parameters

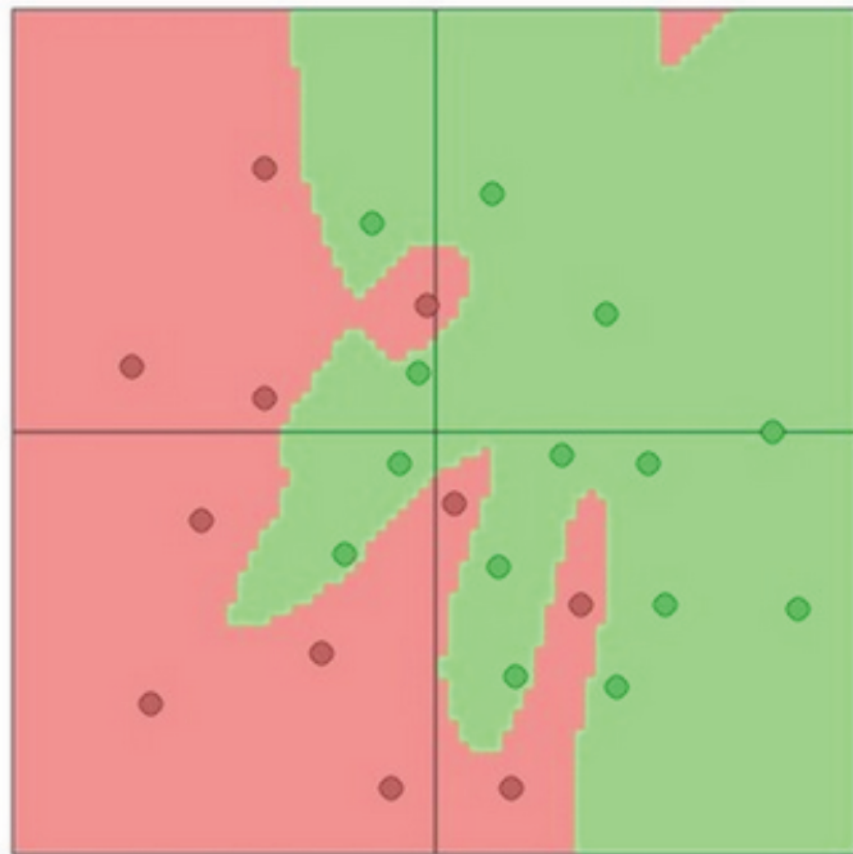
Note: usually called “stochastic gradient descent” even though SGD has a batch size of 1

Regularization

Regularization reduces overfitting:

$$L = L_{\text{data}} + L_{\text{reg}} \qquad L_{\text{reg}} = \lambda \frac{1}{2} \|w\|_2^2$$

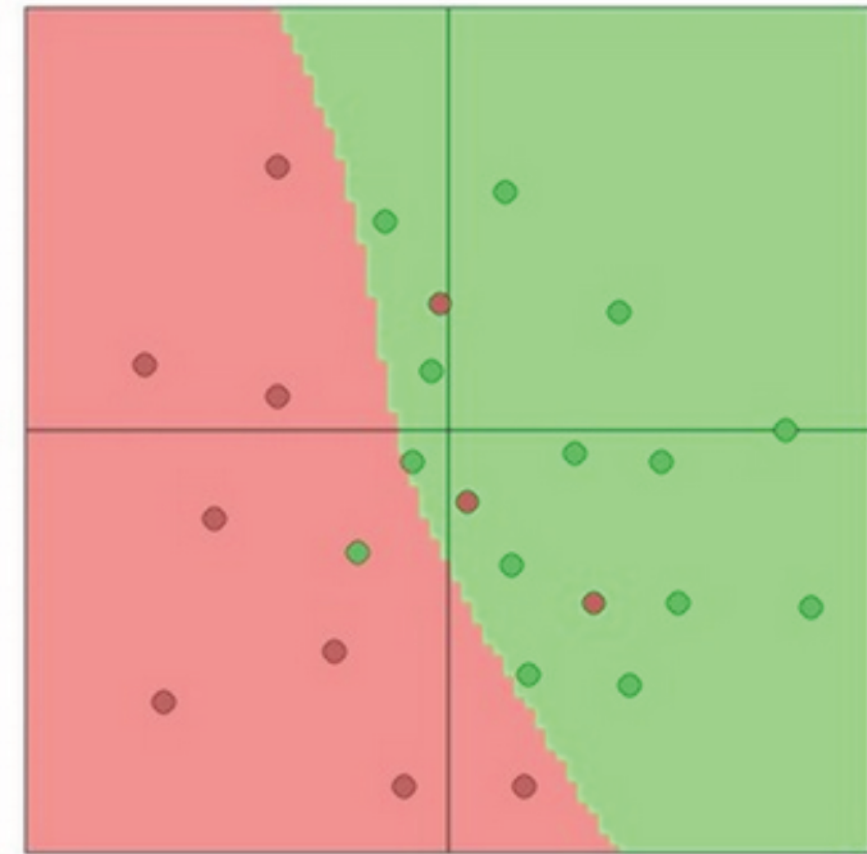
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$

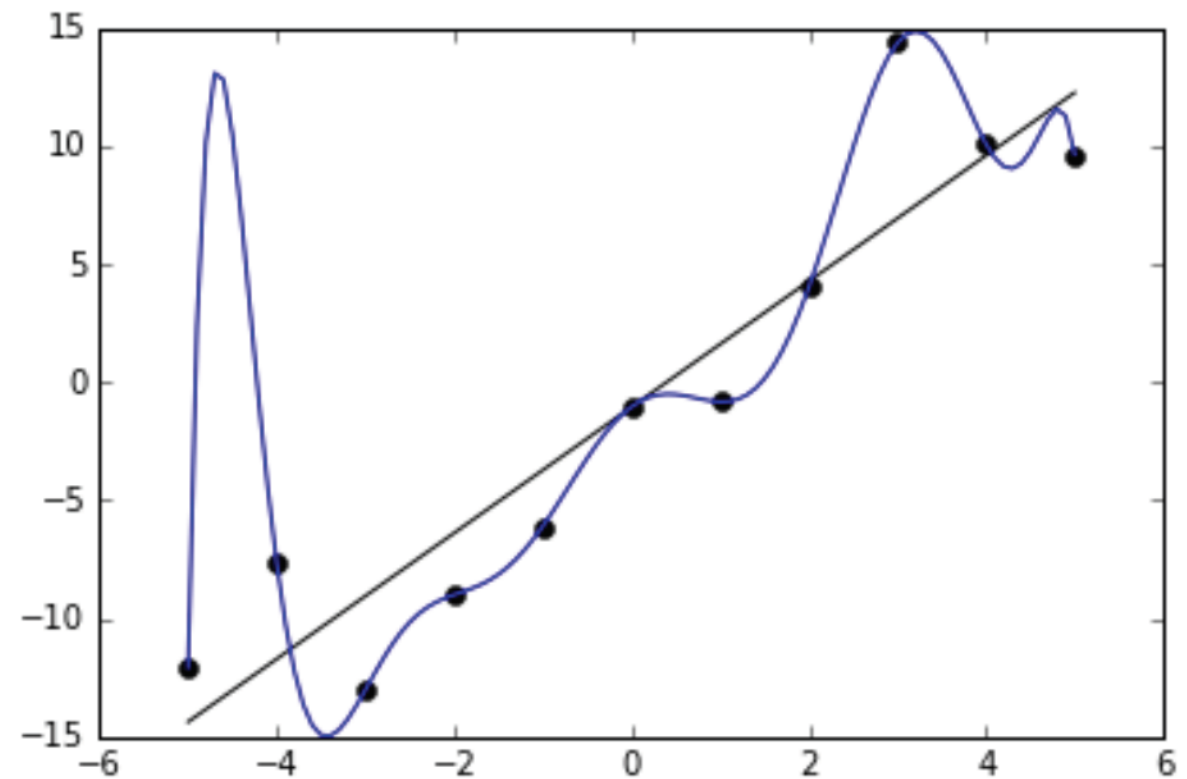


Overfitting

Overfitting: modeling noise in the training set instead of the “true” underlying relationship

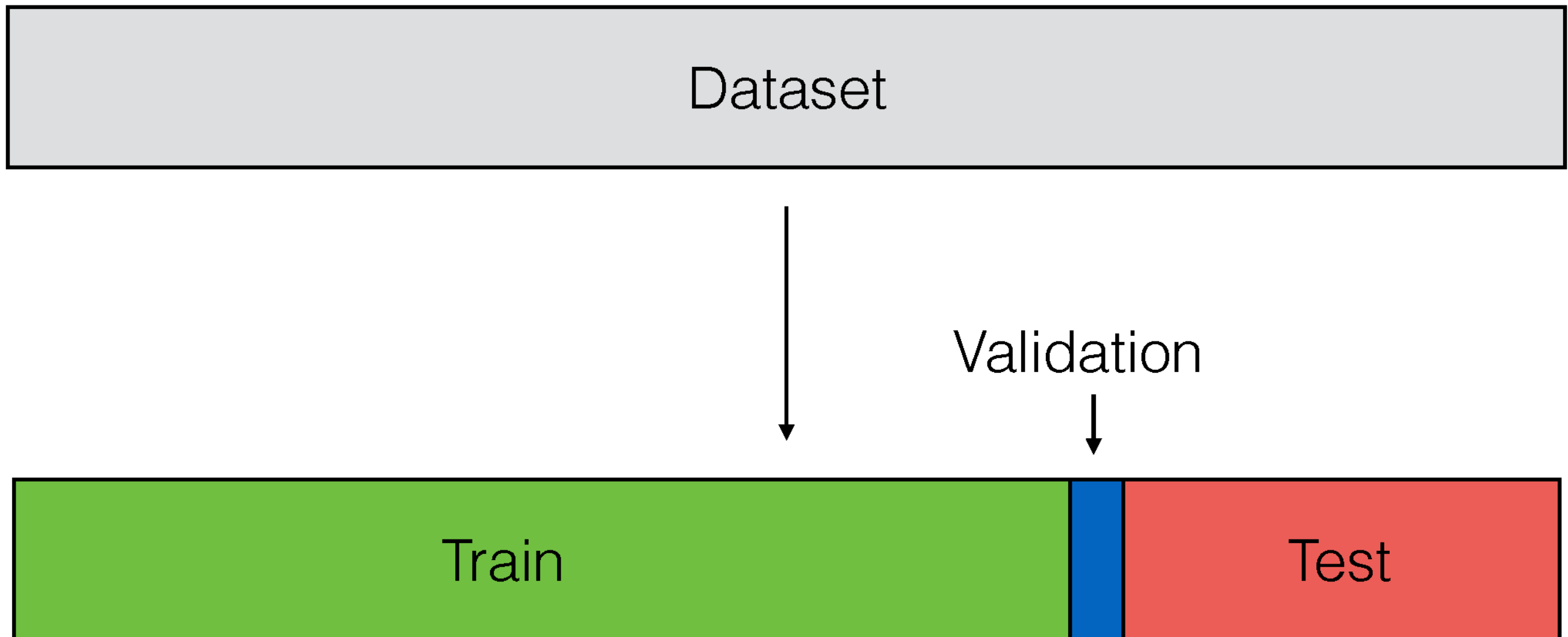
Underfitting: insufficiently modeling the relationship in the training set

General rule: models that are “bigger” or have more capacity are more likely to overfit



(0) Dataset split

Split your data into “train”, “validation”, and “test”:



(0) Dataset split



Train: gradient descent and fine-tuning of parameters

Validation: determining hyper-parameters (learning rate, regularization strength, etc) and picking an architecture

Test: estimate real-world performance
(e.g. accuracy = fraction correctly classified)

(0) Dataset split



Be careful with false discovery:

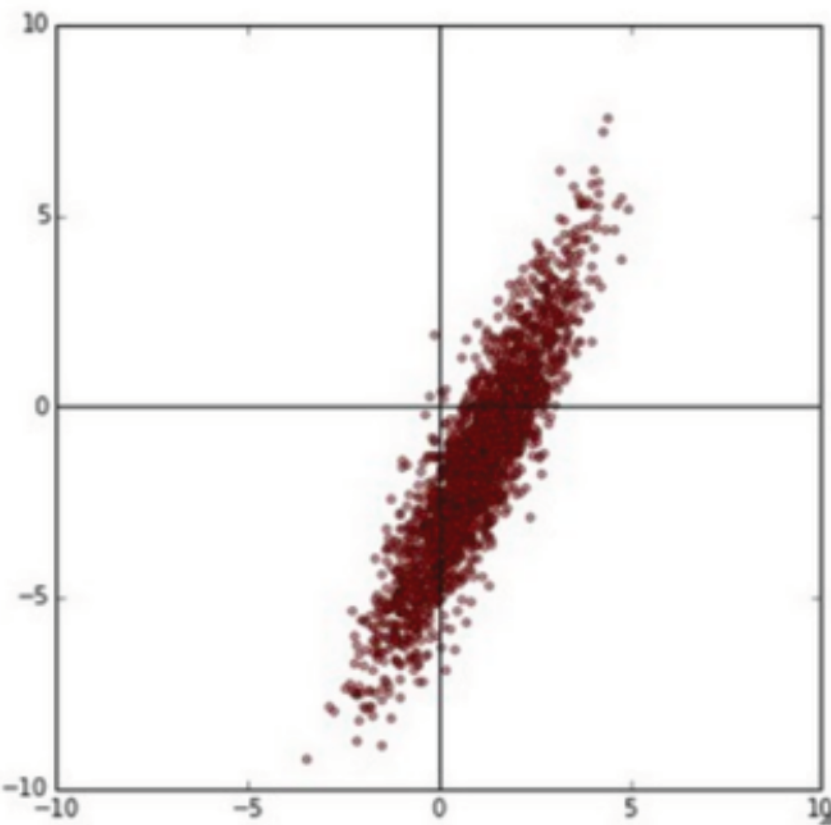
To avoid false discovery, once we have used a test set once, we should *not use it again* (but nobody follows this rule, since it's expensive to collect datasets)

Instead, try and avoid looking at the test score until the end

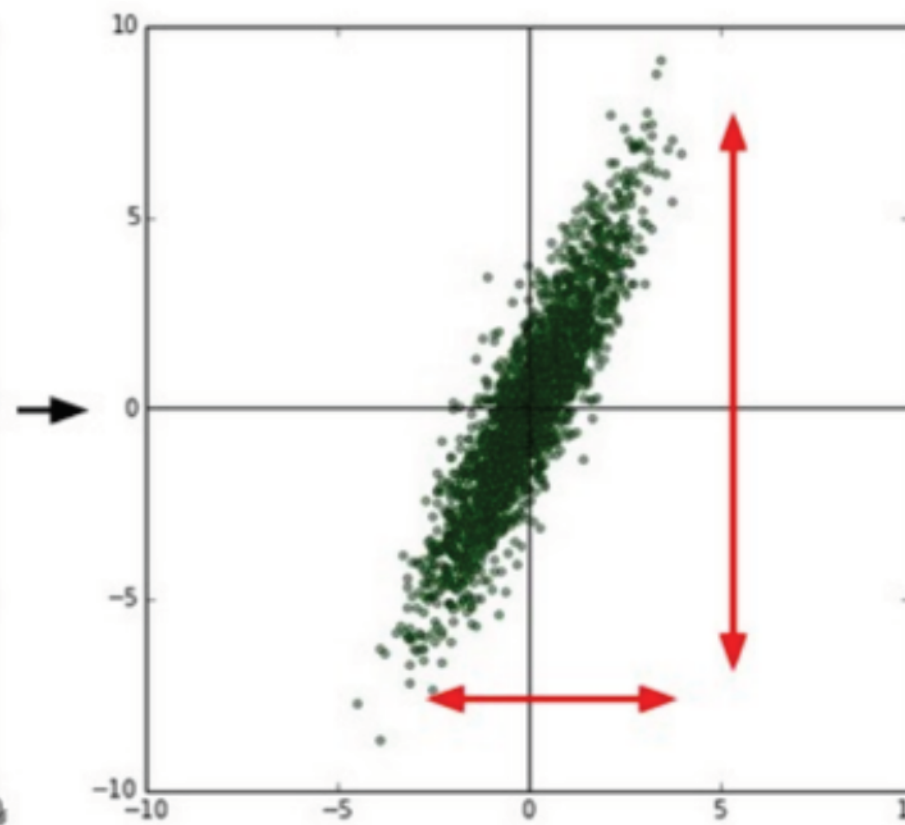
(1) Data preprocessing

Preprocess the data so that learning is better conditioned:

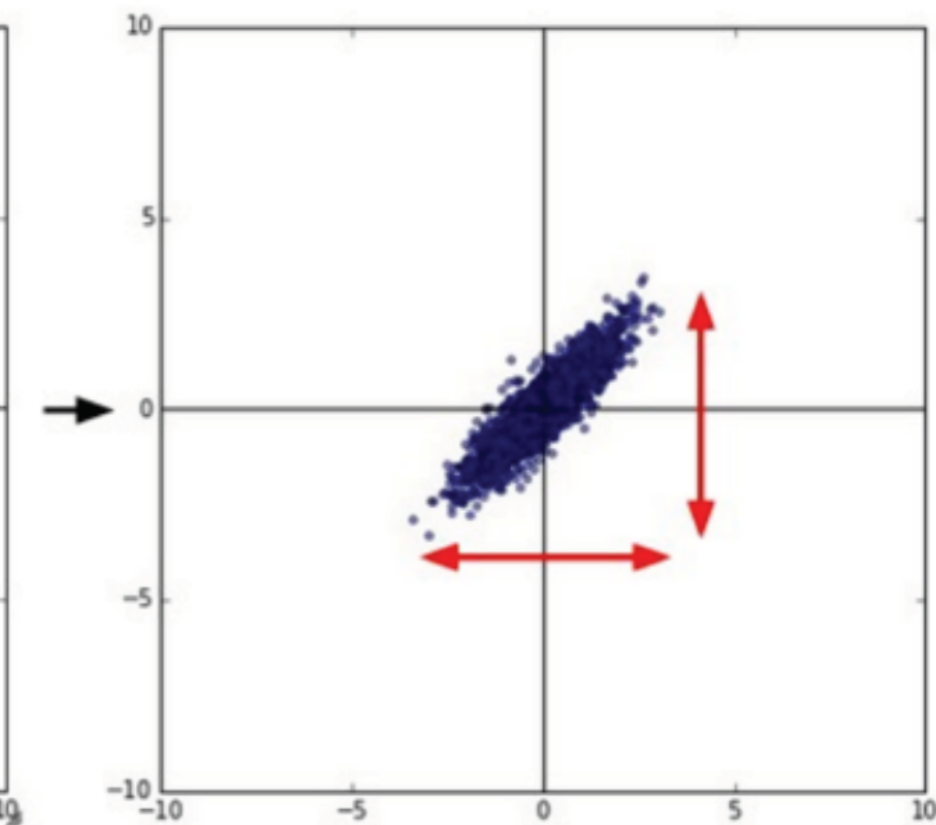
original data



zero-centered data



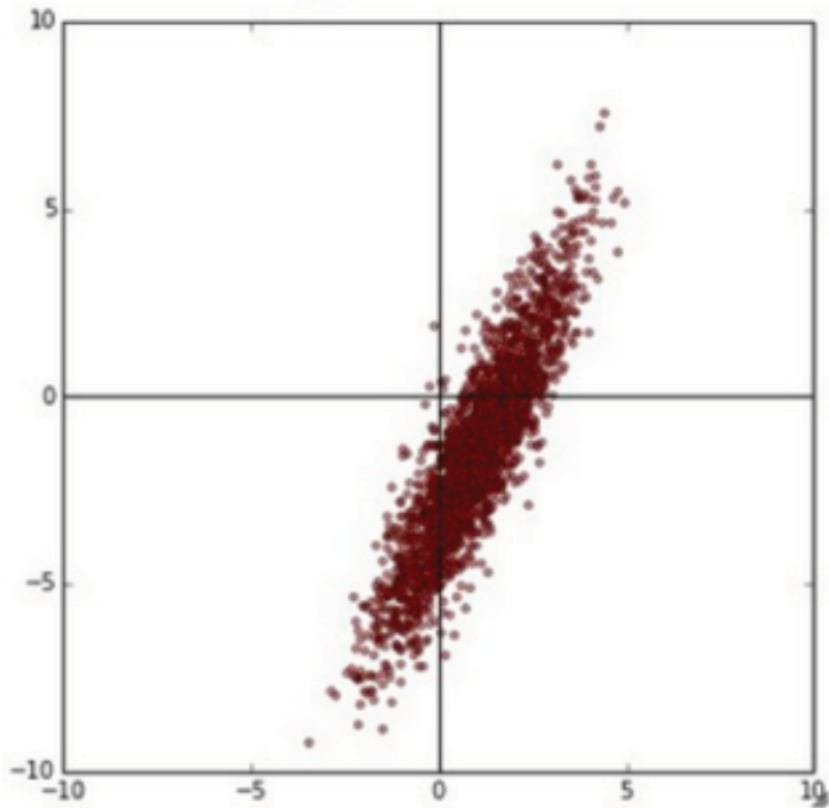
normalized data



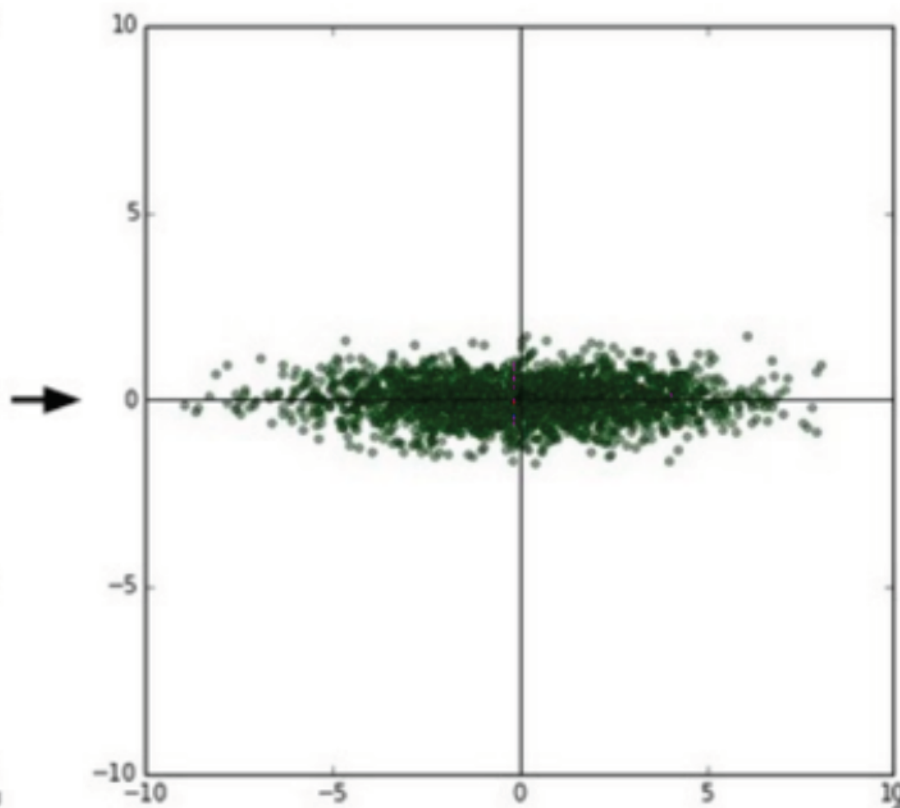
(1) Data preprocessing

In practice, you may also see **PCA** and **Whitening** of the data:

original data

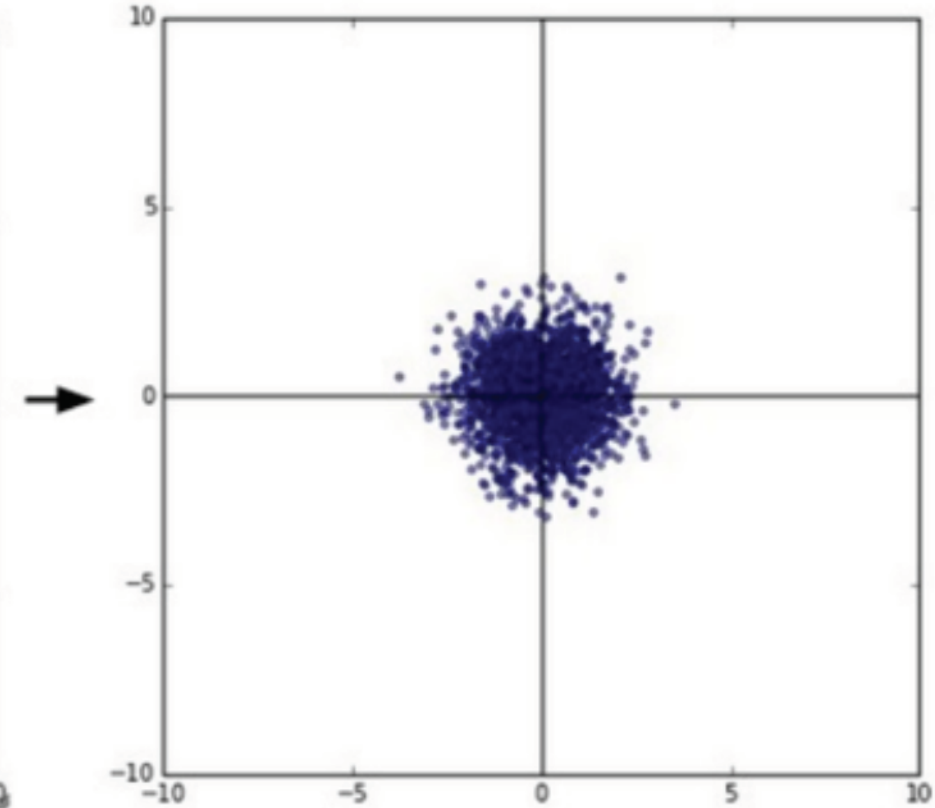


decorrelated data



(data has diagonal covariance matrix)

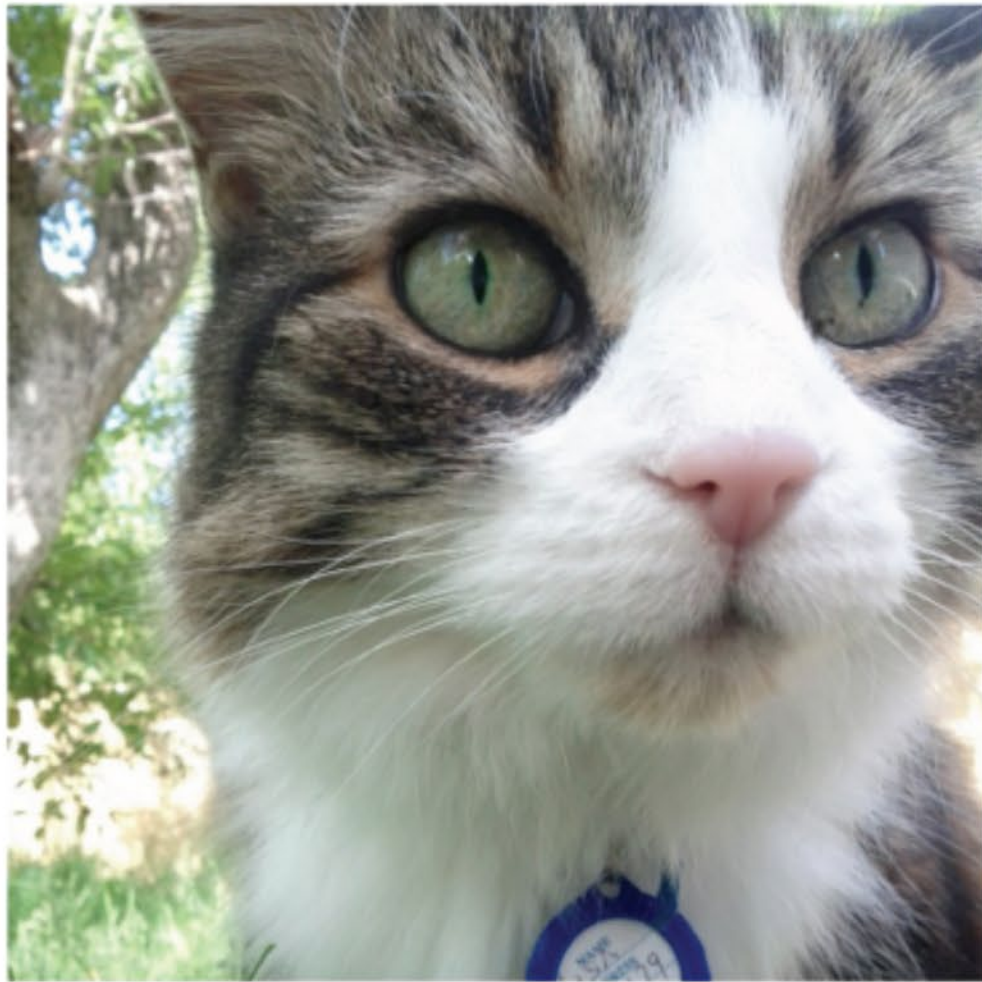
whitened data



(covariance matrix is the identity matrix)

(1) Data preprocessing

For ConvNets, typically only the mean is subtracted.



An input image (256x256)



Minus sign



The mean input image

A per-channel mean also works (one value per R,G,B).

(1) Data preprocessing

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical ConvNets (e.g. [Krizhevsky 2012]) overfit the data.



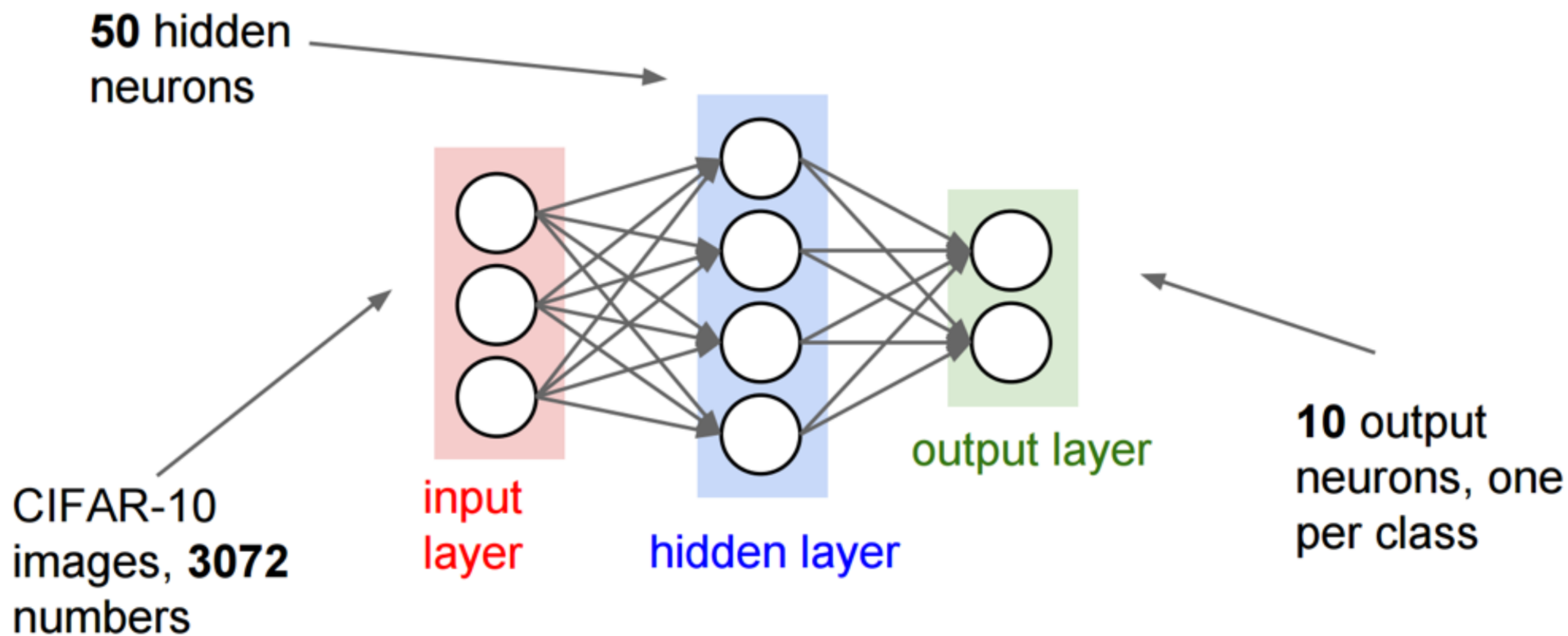
E.g. 224x224 patches
extracted from 256x256 images

Randomly reflect horizontally

Perform the augmentation live
during training

(2) Choose your architecture

Toy example: one hidden layer of size 50



(3) Initialize your weights

Set the weights to small random numbers:

```
W = np.random.randn(D, H) * 0.001
```

(matrix of small random numbers drawn from a Gaussian distribution)

(the magnitude is important and this is not optimal — more on this later)

Set the bias to zero (or small nonzero):

```
b = np.zeros(H)
```

(3) Check that the loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 0.0) # disable regularization  
print loss
```



returns the loss and the
gradient for all parameters

(3) Check that the loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 1e3) # crank up regularization  
print loss
```

 loss went up, good. (sanity check)

(4) Overfit a small portion of the data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```



Details:

‘sgd’: vanilla gradient descent (no momentum etc)

learning_rate_decay = 1: constant learning rate

sample_batches = False (full gradient descent, no batches)

epochs = 200: number of passes through the data

(4) Overfit a small portion of the data

100% accuracy on the training set (good)

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03  
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03  
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03  
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03  
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03  
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03  
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03  
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03  
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03  
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03  
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03  
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03  
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03  
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03  
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03  
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03  
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03  
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03  
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03  
Finished epoch 20 / 200: cost 1.305760, train: 0.650000, val 0.650000, lr 1.000000e-03  
  
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03  
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03  
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03  
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03  
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03  
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03  
finished optimization. best validation accuracy: 1.000000
```


(4) Find a learning rate

Let's start with small regularization and find the learning rate that makes the loss decrease:

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```



(4) Find a learning rate

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Epoch	Cost	Train Accuracy	Val Accuracy	Learning Rate
1 / 10	2.302576	0.080000	0.103000	1.000000e-06
2 / 10	2.302582	0.121000	0.124000	1.000000e-06
3 / 10	2.302558	0.119000	0.138000	1.000000e-06
4 / 10	2.302519	0.127000	0.151000	1.000000e-06
5 / 10	2.302517	0.158000	0.171000	1.000000e-06
6 / 10	2.302518	0.179000	0.172000	1.000000e-06
7 / 10	2.302466	0.180000	0.176000	1.000000e-06
8 / 10	2.302452	0.175000	0.185000	1.000000e-06
9 / 10	2.302459	0.206000	0.192000	1.000000e-06
10 / 10	2.302420	0.190000	0.192000	1.000000e-06

finished optimization. best validation accuracy: 0.192000

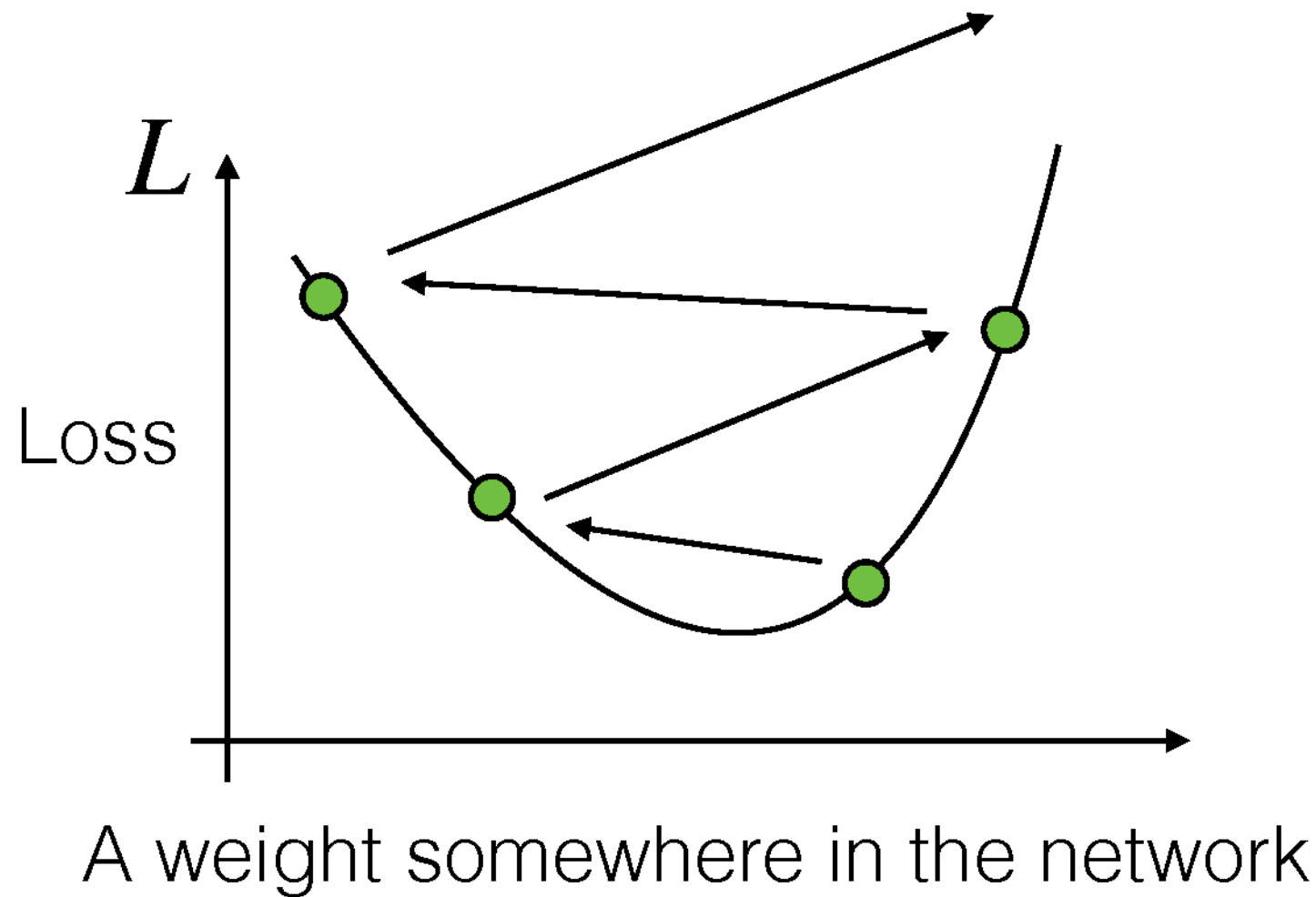
Loss barely changes

Why is the accuracy 20%?

(learning rate is too low or regularization too high)

(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?



(4) Find a learning rate

Coarse to fine search

First stage: only a few epochs (passes through the data) to get a rough idea

Second stage: longer running time, finer search

Tip: if $\text{loss} > 3 * \text{original loss}$, quit early
(learning rate too high)

(4) Find a learning rate

Normally, you don't have the budget for lots of cross-validation —> visualize as you go

Plot the loss

For very small learning rates, the loss decreases linearly and slowly

(Why linearly?)

Larger learning rates tend to look more exponential

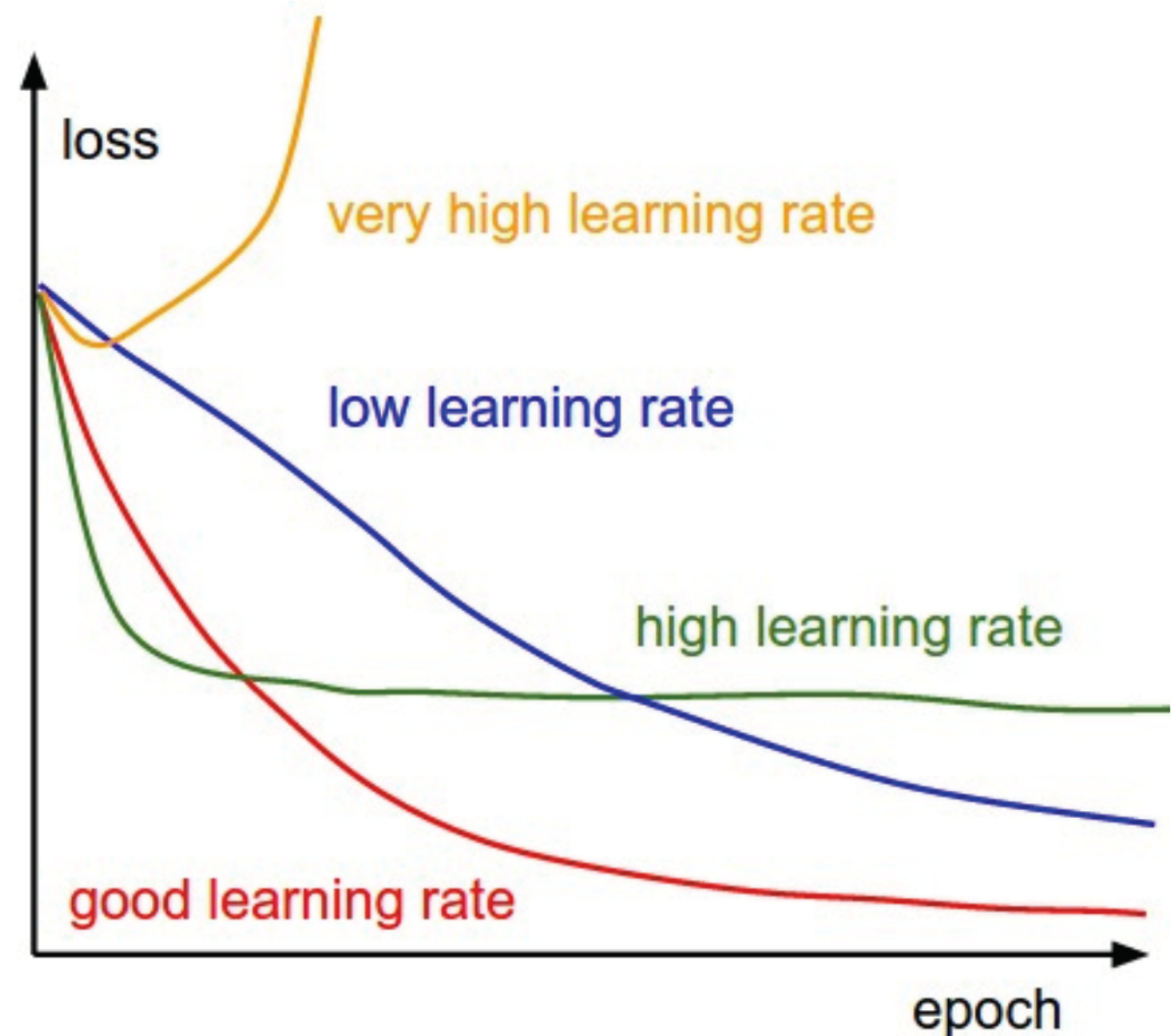


Figure: Andrej Karpathy

(4) Find a learning rate

Normally, you don't have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Why is it varying so rapidly?

The width of the curve is related to the batchsize — if too noisy, increase the batch size

Possibly too linear
(learning rate too small)

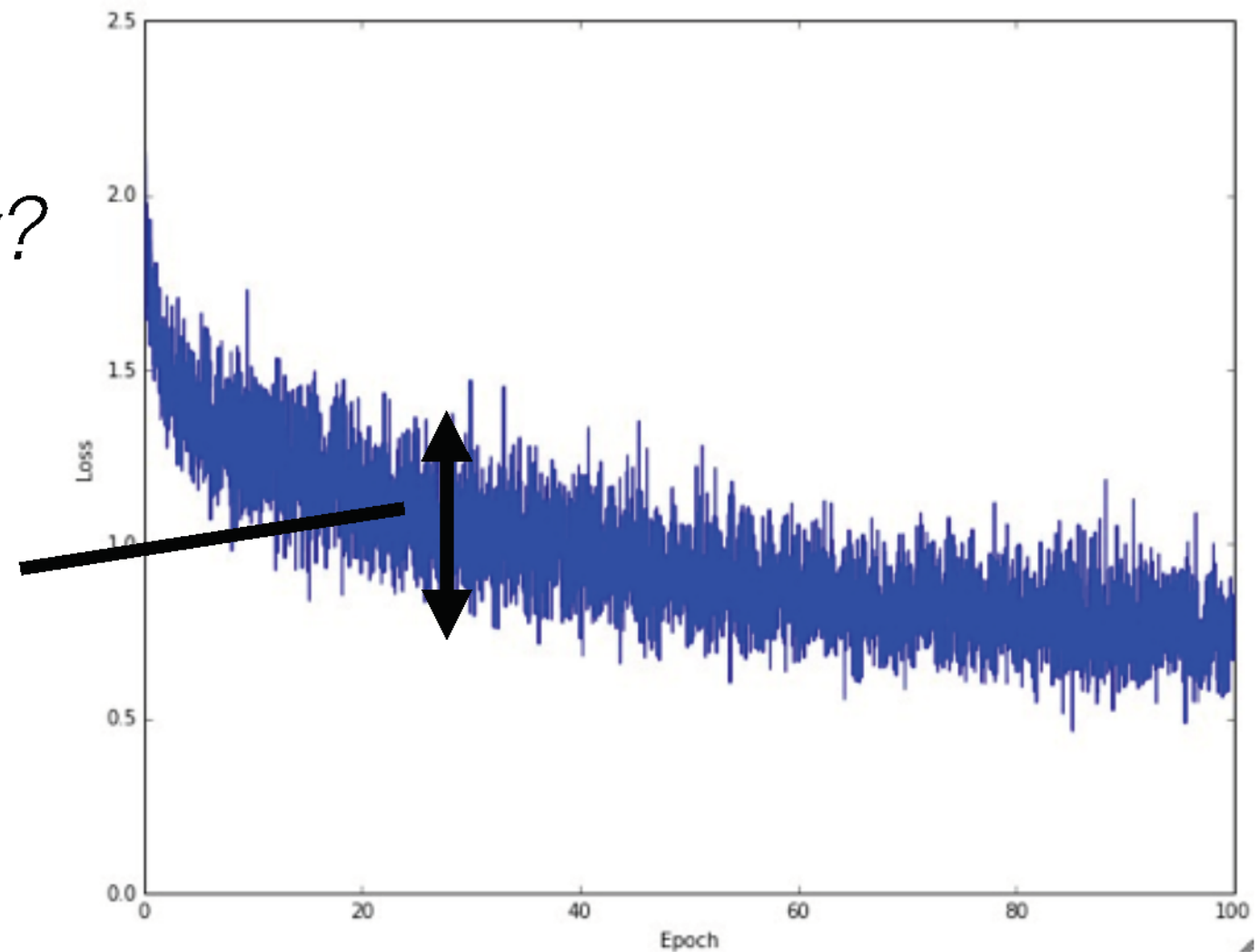
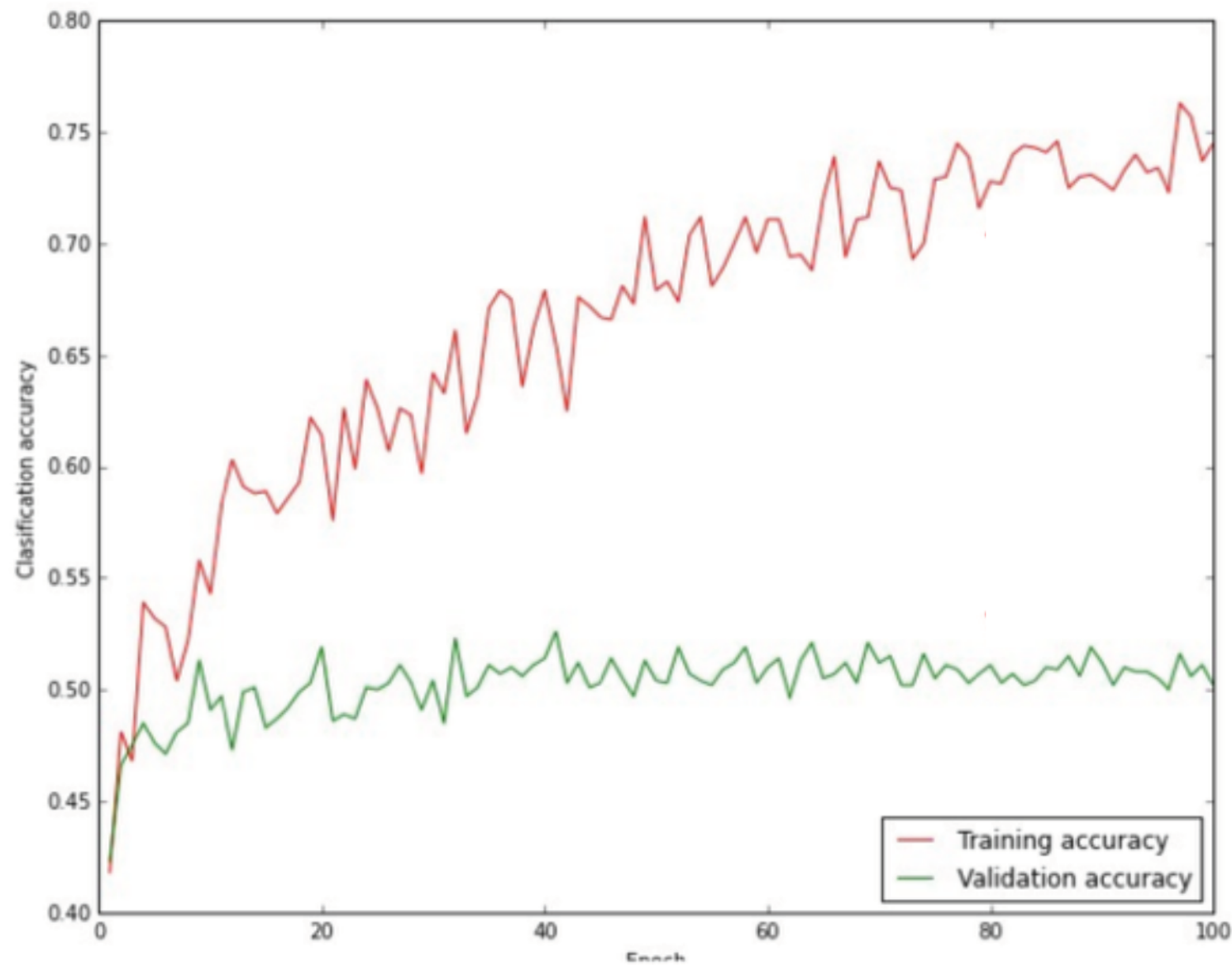


Figure: Andrej Karpathy

(4) Find a learning rate

Visualize the accuracy



Big gap: overfitting
(increase regularization)

No gap: underfitting
(increase model capacity,
make layers bigger
or decrease regularization)

(4) Find a learning rate

Visualize the weights

Noisy weights: possibly regularization not strong enough

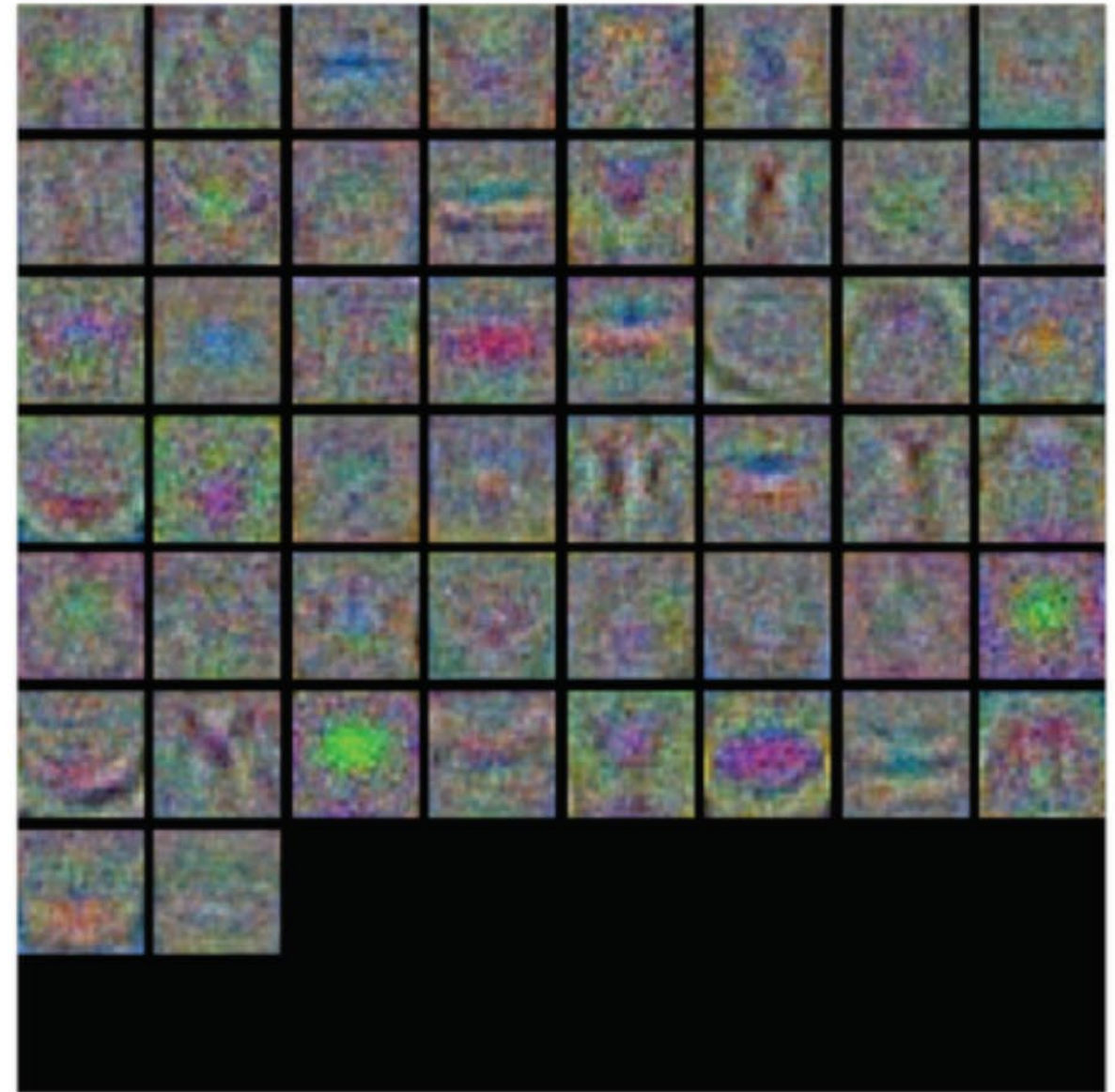
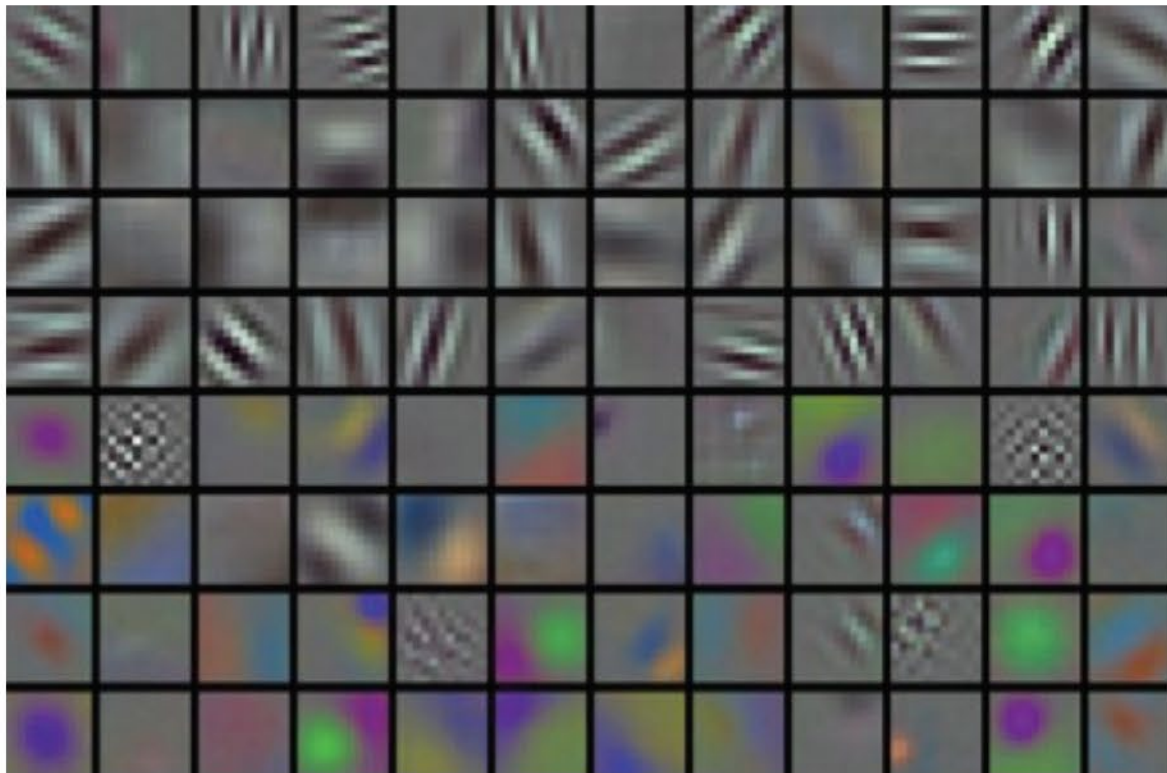


Figure: Andrej Karpathy

(4) Find a learning rate

Visualize the weights



Nice clean weights:
training is proceeding well

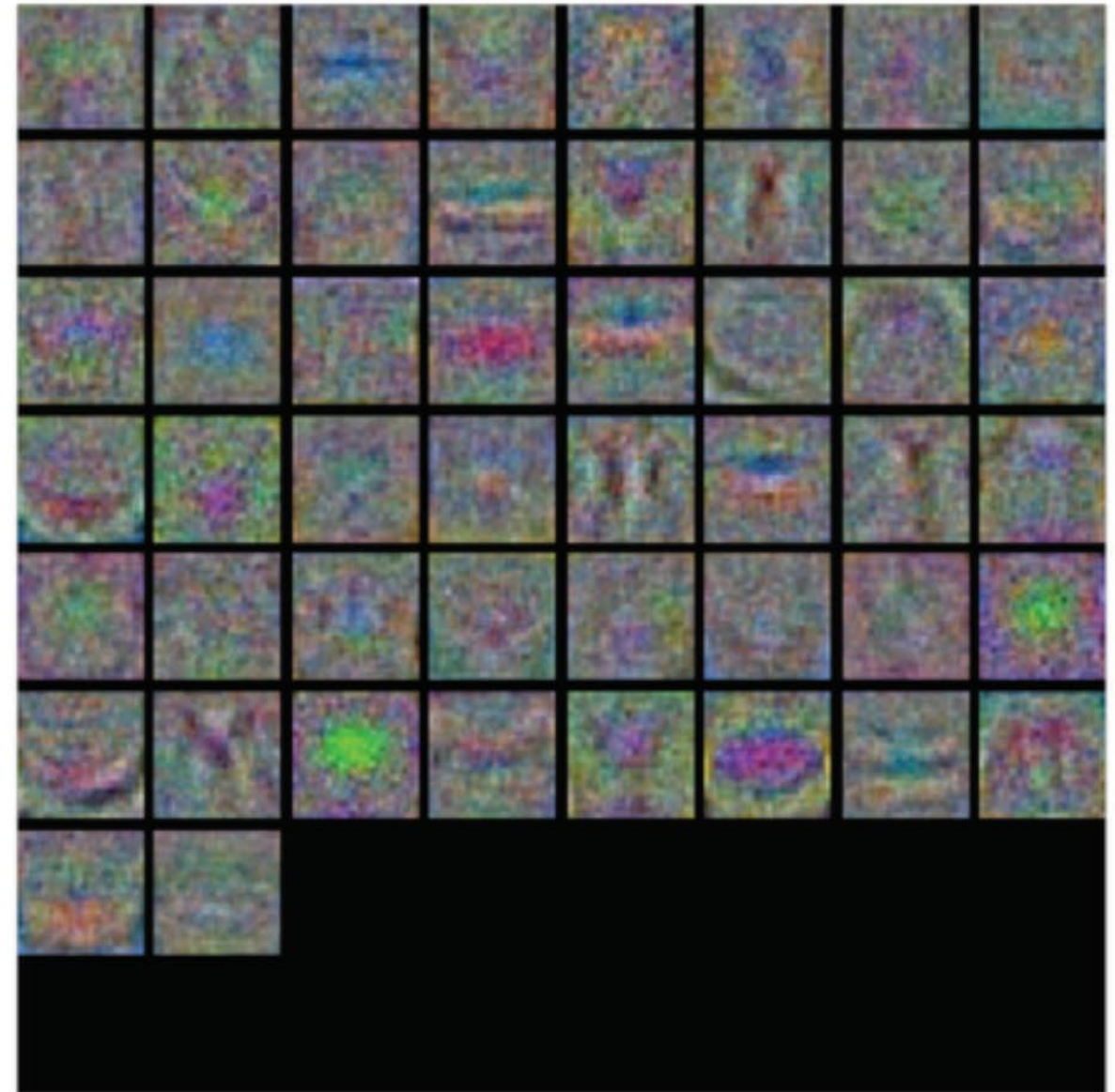


Figure: Alex Krizhevsky , Andrej Karpathy

Learning rate schedule

How do we change the learning rate over time?

Various choices:

- Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])
- Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])
- Scale by $\sqrt{1-t/\text{max_t}}$ (used by BVLC to re-implement GoogLeNet)
- Scale by $1/t$
- Scale by $\exp(-t)$

Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, ...)
- Loss function (softmax, SVM, ...)
- Weight initialization

Neural network
parameters



(Recall) Regularization reduces overfitting

$$L = L_{\text{data}} + L_{\text{reg}} \qquad L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$

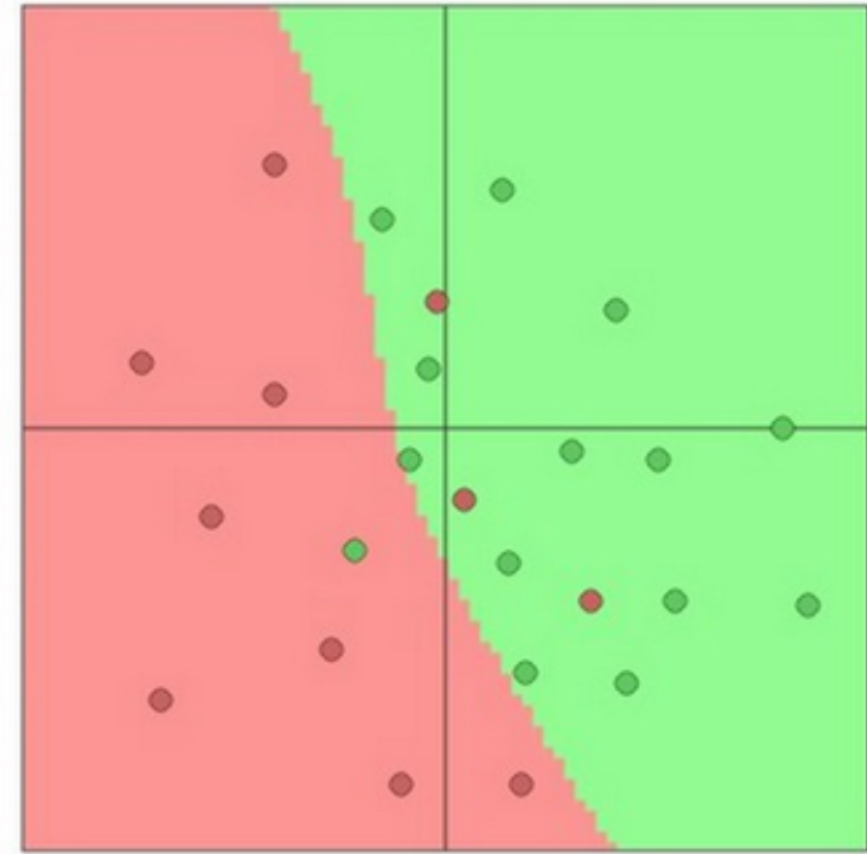
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$



Example Regularizers

L2 regularization

$$L_{\text{reg}} = \lambda \frac{1}{2} ||W||_2^2$$

(L2 regularization encourages small weights)

L1 regularization

$$L_{\text{reg}} = \lambda ||W||_1 = \lambda \sum_{ij} |w_{ij}|$$

(L1 regularization encourages sparse weights:
weights are encouraged to reduce to exactly zero)

“Elastic net”

$$L_{\text{reg}} = \lambda_1 ||W||_1 + \lambda_2 ||W||_2^2$$

(combine L1 and L2 regularization)

Max norm

Clamp weights to some max norm

$$||W||_2^2 \leq c$$

“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

$$L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2 \longrightarrow \frac{\partial L}{\partial W} = \lambda W$$

Gradient descent step:

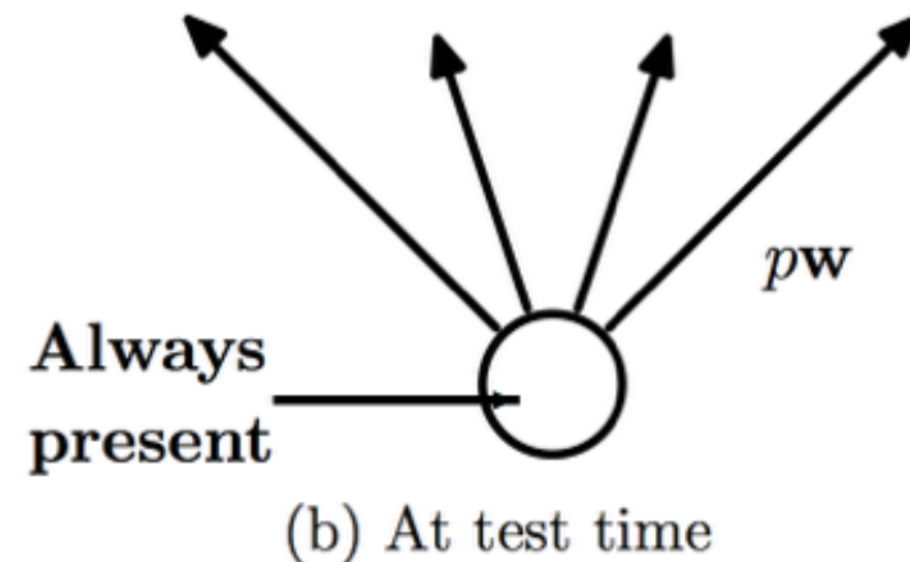
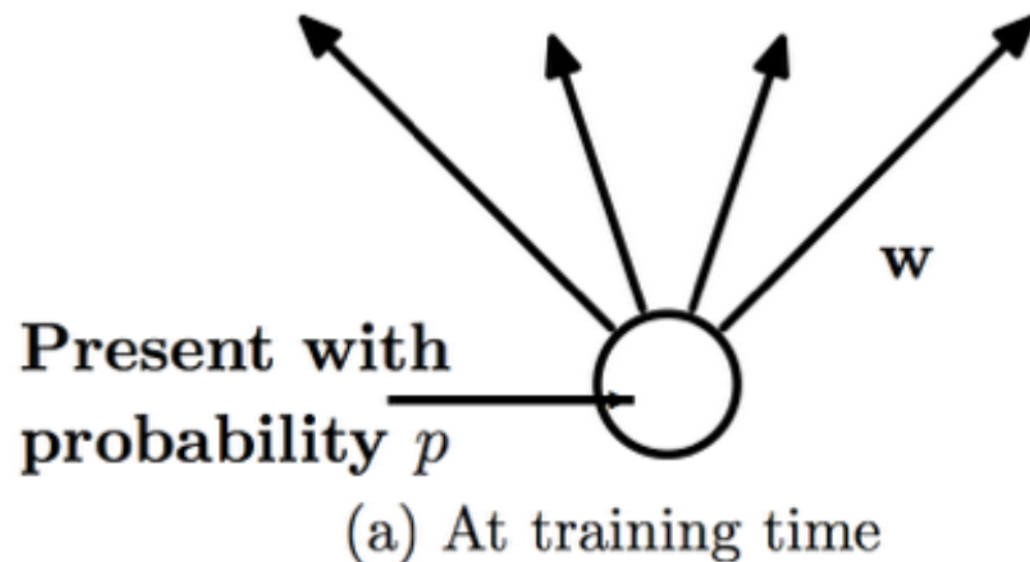
$$W \leftarrow W - \alpha \lambda W - \frac{\partial L_{\text{data}}}{\partial W}$$

Weight decay: $\alpha \lambda$ (weights always decay by this amount)

Note: biases are sometimes excluded from regularization

Dropout

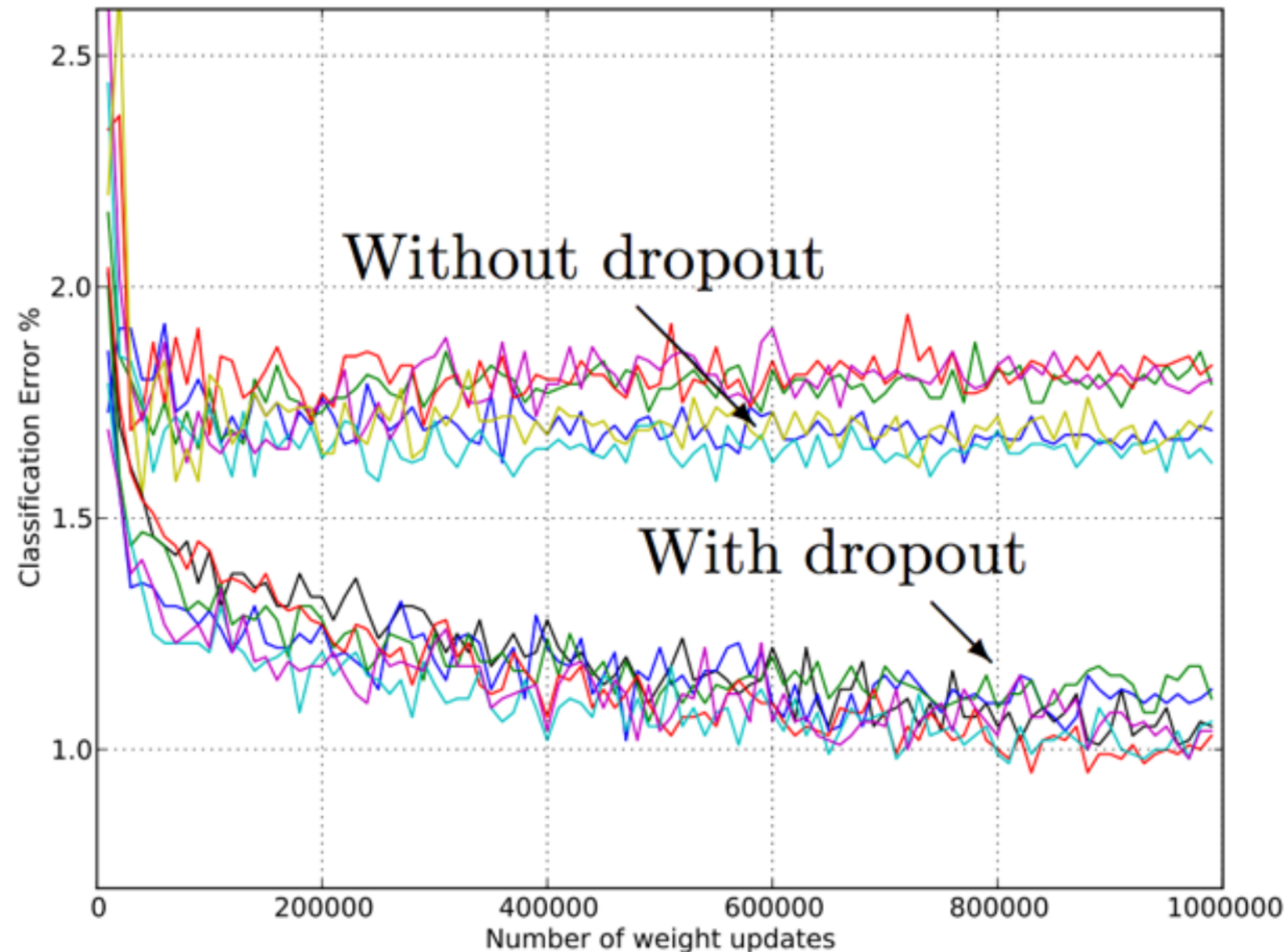
Simple but powerful technique to reduce overfitting:



[Srivasta et al, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, JMLR 2014]

Dropout

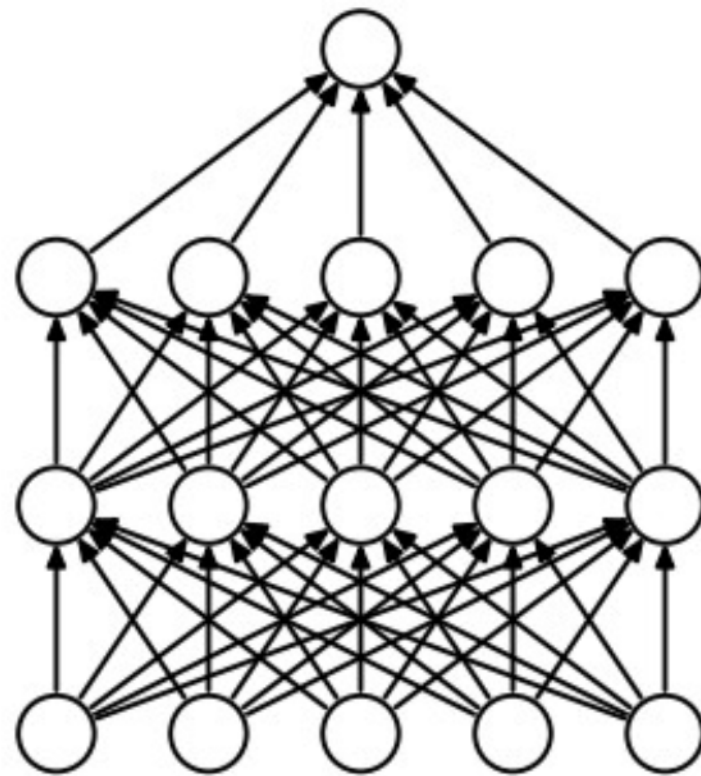
Simple but powerful technique to reduce overfitting:



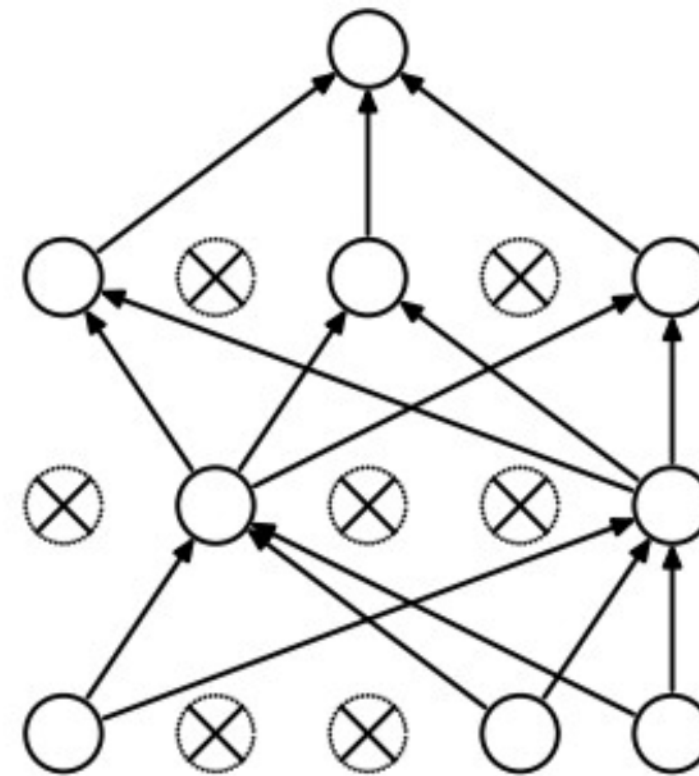
[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

Simple but powerful technique to reduce overfitting:



(a) Standard Neural Net



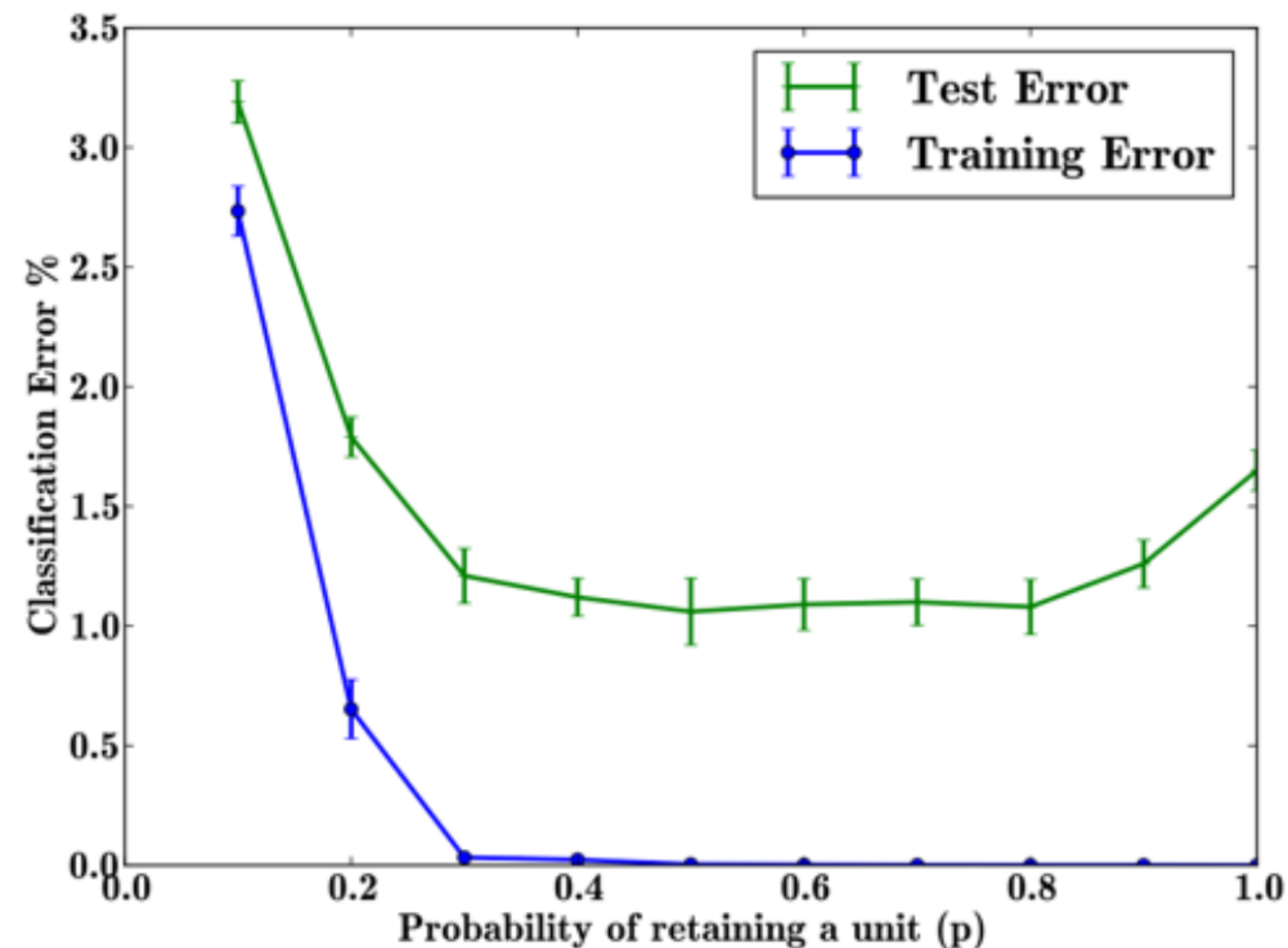
(b) After applying dropout.

Note: Dropout can be interpreted as an approximation to taking the geometric mean of an ensemble of exponentially many models

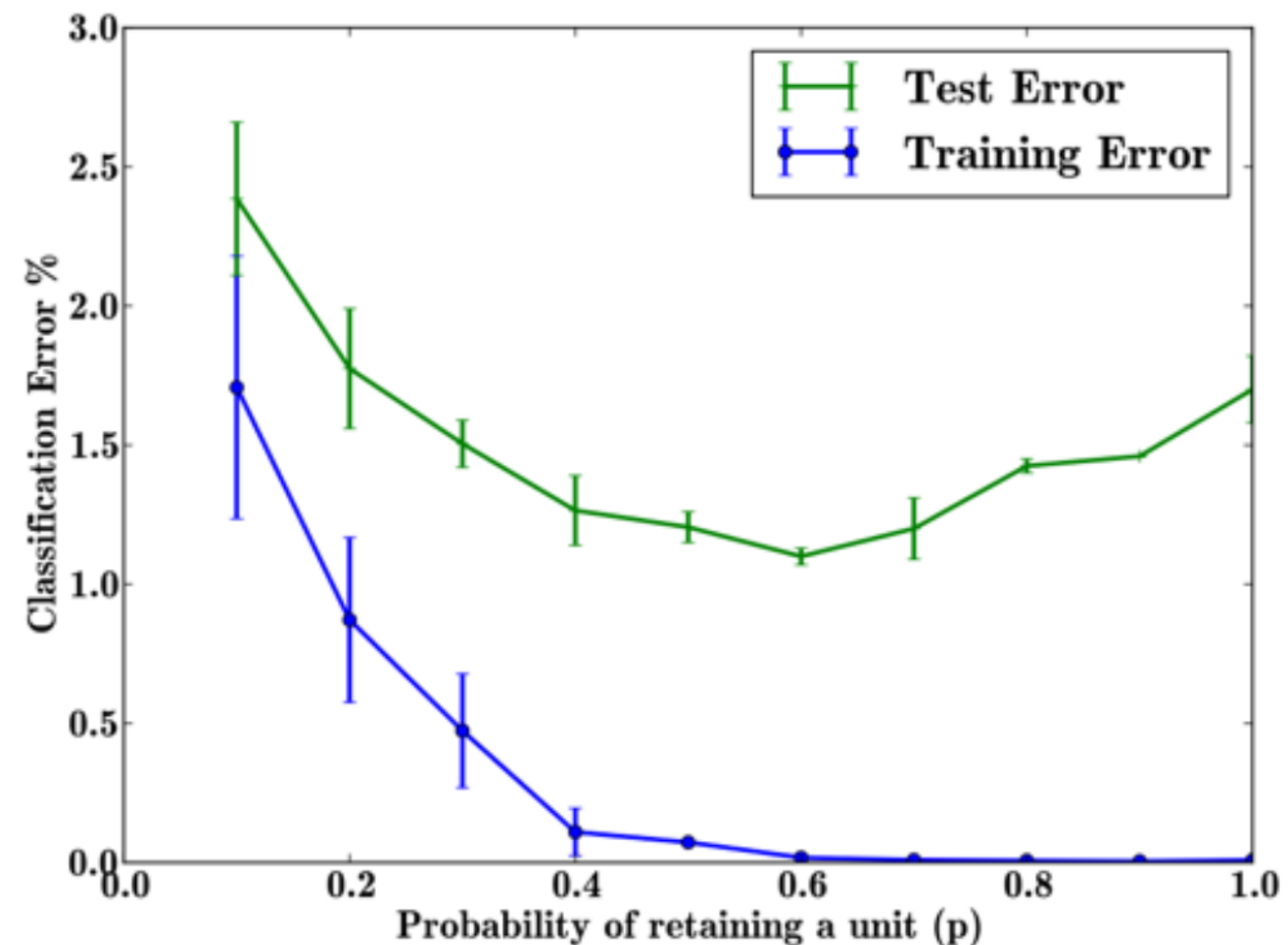
[Srivasta et al, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, JMLR 2014]

Dropout

How much dropout? Around $p = 0.5$



(a) Keeping n fixed.



(b) Keeping pn fixed.

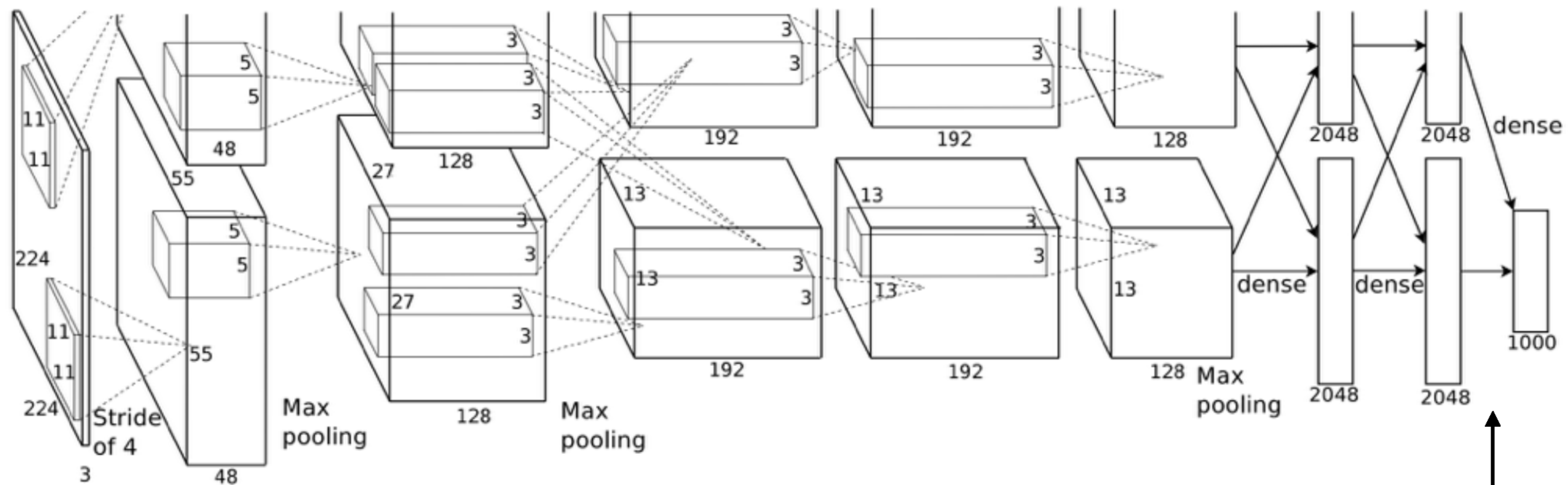
[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

Case study: [Krizhevsky 2012]

“Without dropout, our network exhibits substantial overfitting.”

Dropout here



But not here — why?

[Krizhevsky et al, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012]

Dropout

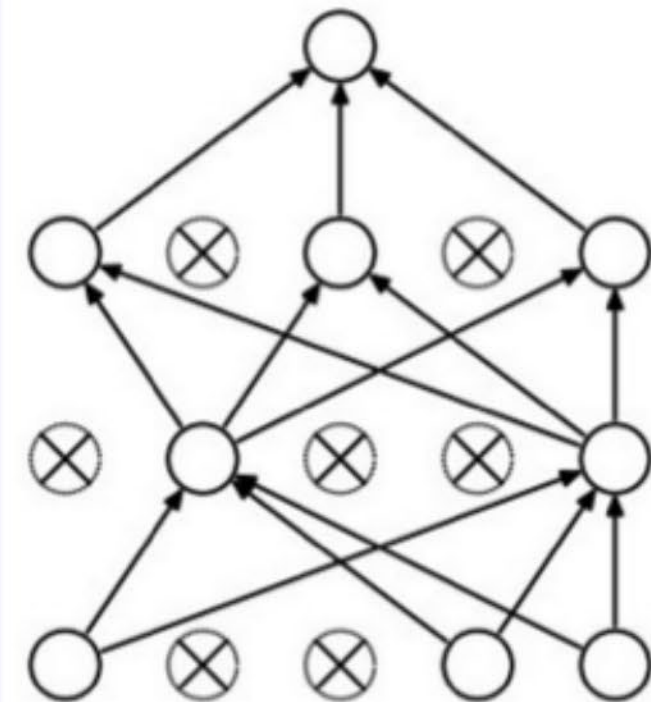
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



(note, here X is a single input)

Dropout

Test time: scale the activations

Expected value of a neuron h with dropout:

$$E[h] = ph + (1 - p)0 = ph$$

```
def predict(X):  
    # ensembled forward pass  
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations  
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations  
    out = np.dot(W3, H2) + b3
```

We want to keep the same expected value

Summary

- Preprocess the data (subtract mean, sub-crops)
- Initialize weights carefully
- Use Dropout
- Use SGD + Momentum
- Fine-tune from ImageNet
- Babysit the network as it trains

References

Basic reading: No standard textbooks yet! Some good resources:

- <https://sites.google.com/site/deeplearningsummerschool/>
- <http://www.deeplearningbook.org/>
- <http://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf>