CSCI 699: Robotic Perception

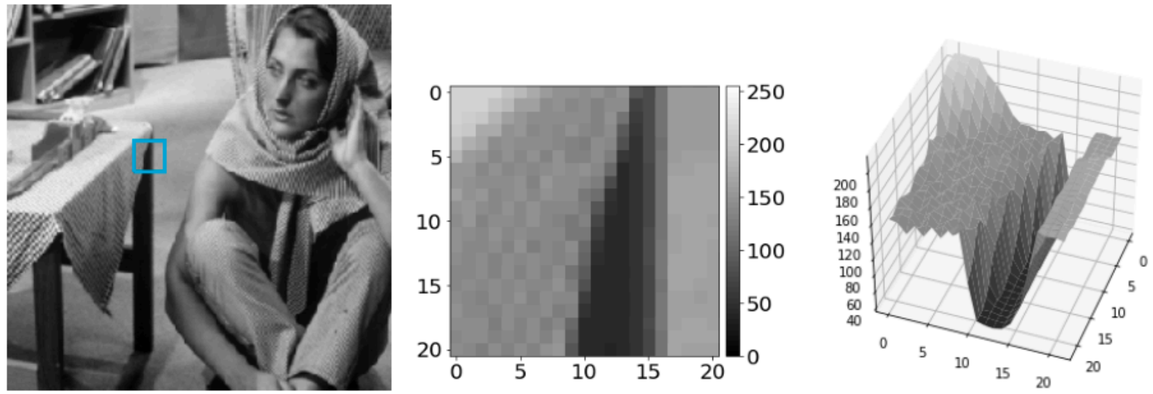
Yue Wang

Sep 3rd, 2025

Image processing

Image as functions

- An image is a grid of integer values.
- An image can be seen as a representation of a function
- $(x, y) \rightarrow I(x, y)$



- For a color image, we have vector valued $\mathrm{I}(x, y) = (R(x, y), G(x, y), B(x, y))$

Problem: denoising

Original

5% corrupted

35% corrupted

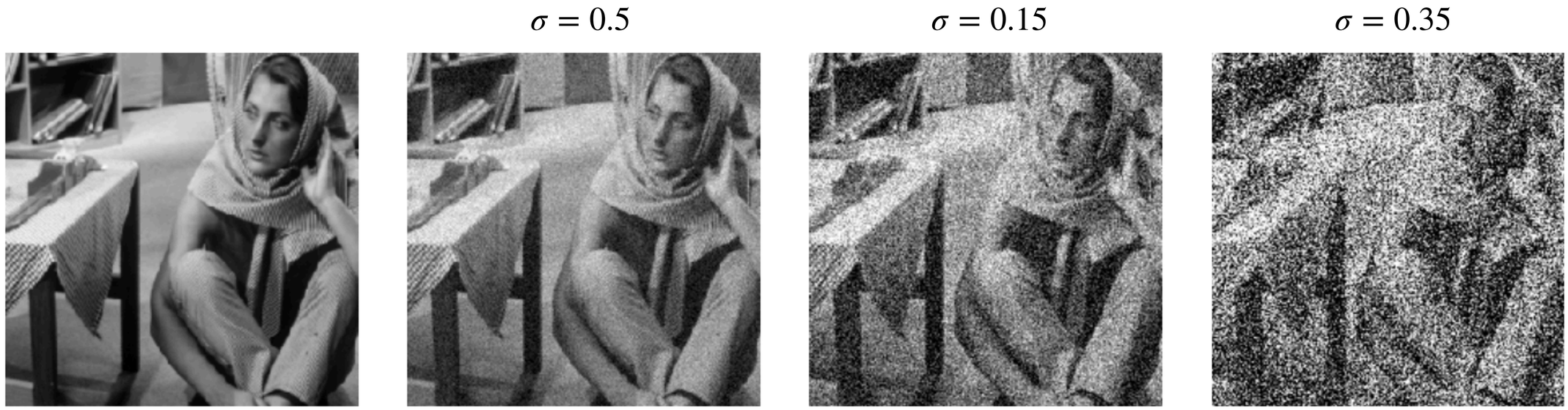
- Salt and pepper noise: randomly selected pixels get very high (salt) or very low (pepper) values

- Goal of denoising: recover original intensity values

Gaussian noise

- Additive noise model:

- $\widetilde{I}(x,y) = I(x,y) + \nu(x,y)\ v \sim \mathrm{N}(0,\sigma)$



Where does noise come from?

- Insufficient number of photons hitting the sensor
- Electronic noise (charge leakage, etc.)
- Analog-digital conversion
- Image processing (e.g., demosaicing)

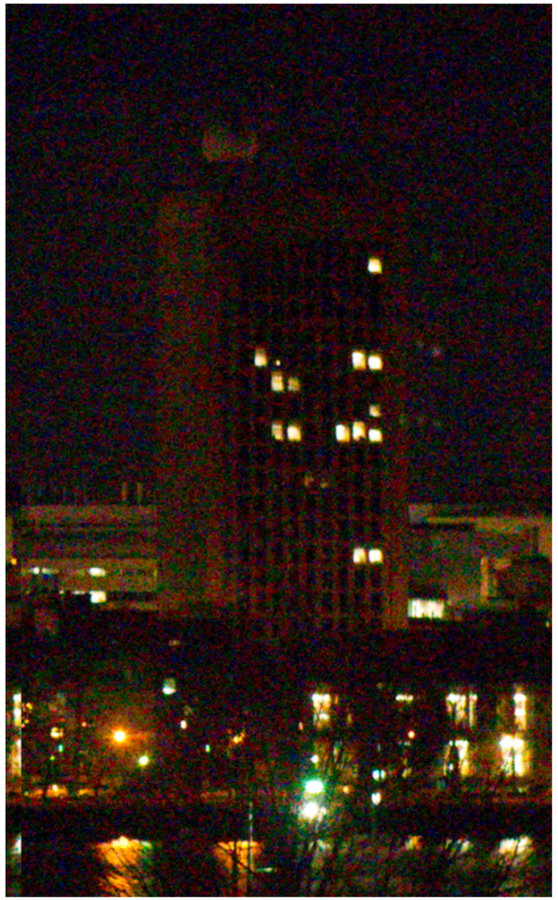


Image credits: Fredo Durand

How to denoise an image?

- Best way: take multiple pictures and average

- Can we do with a single image?

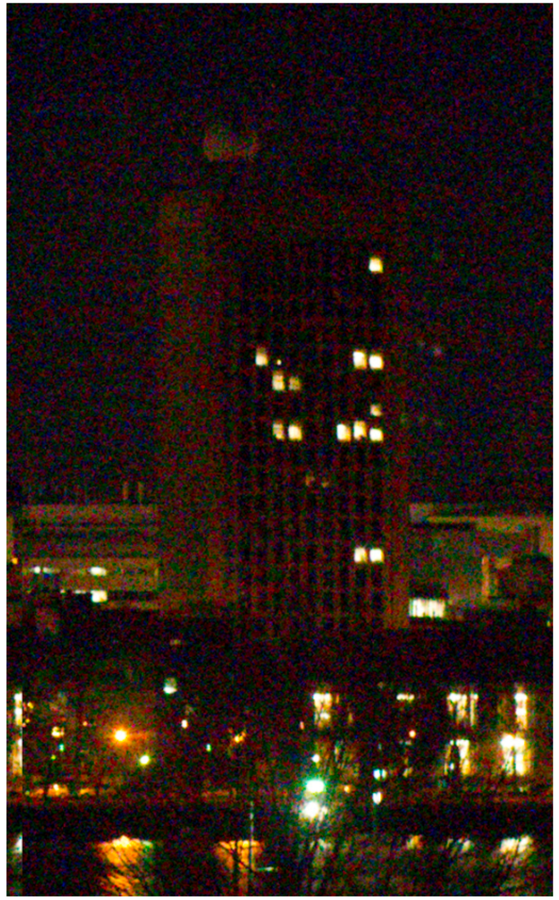
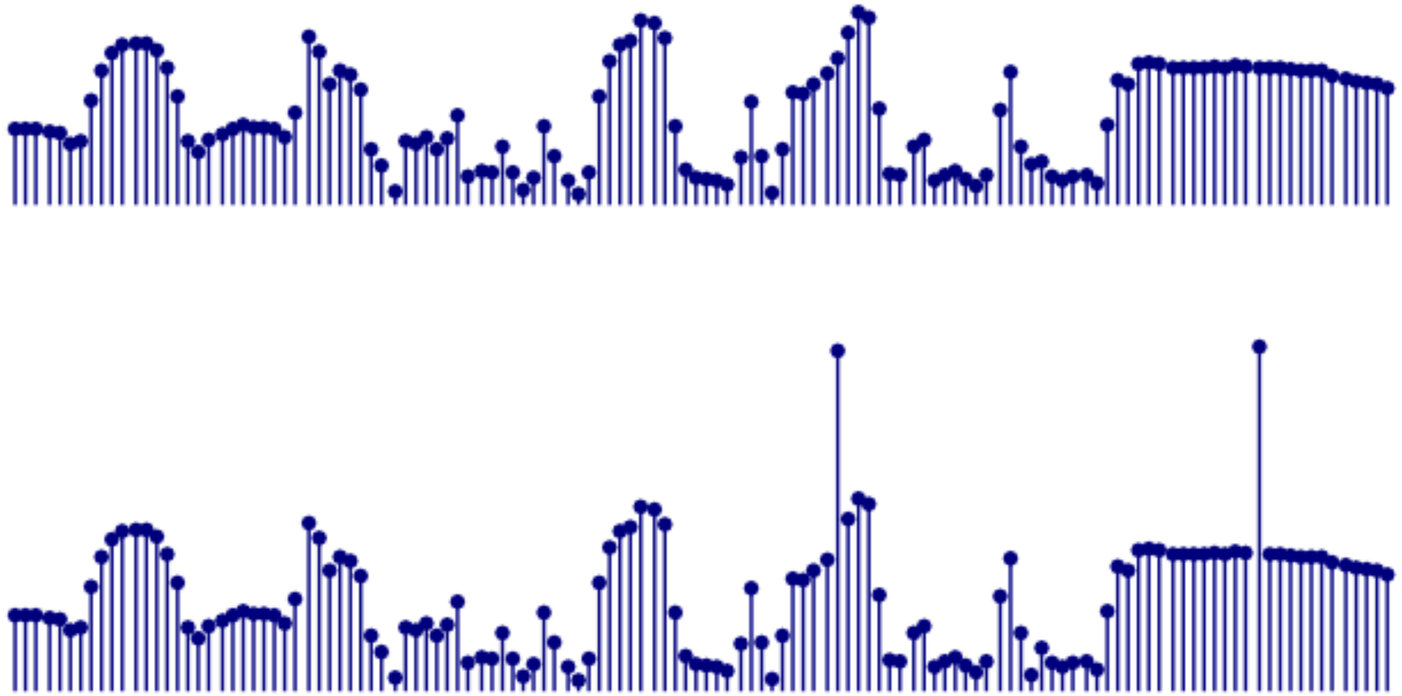
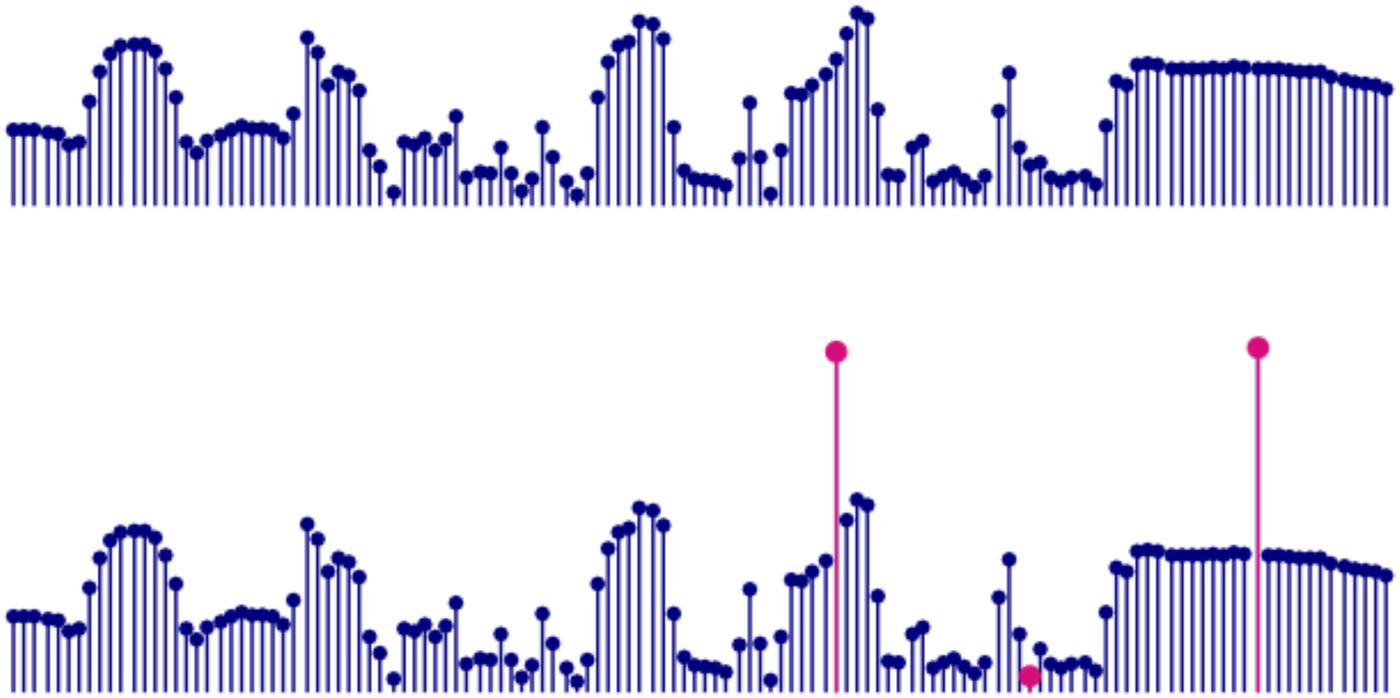


Image credits: Fredo Durand

Salt and pepper in 1D



Salt and pepper in 1D



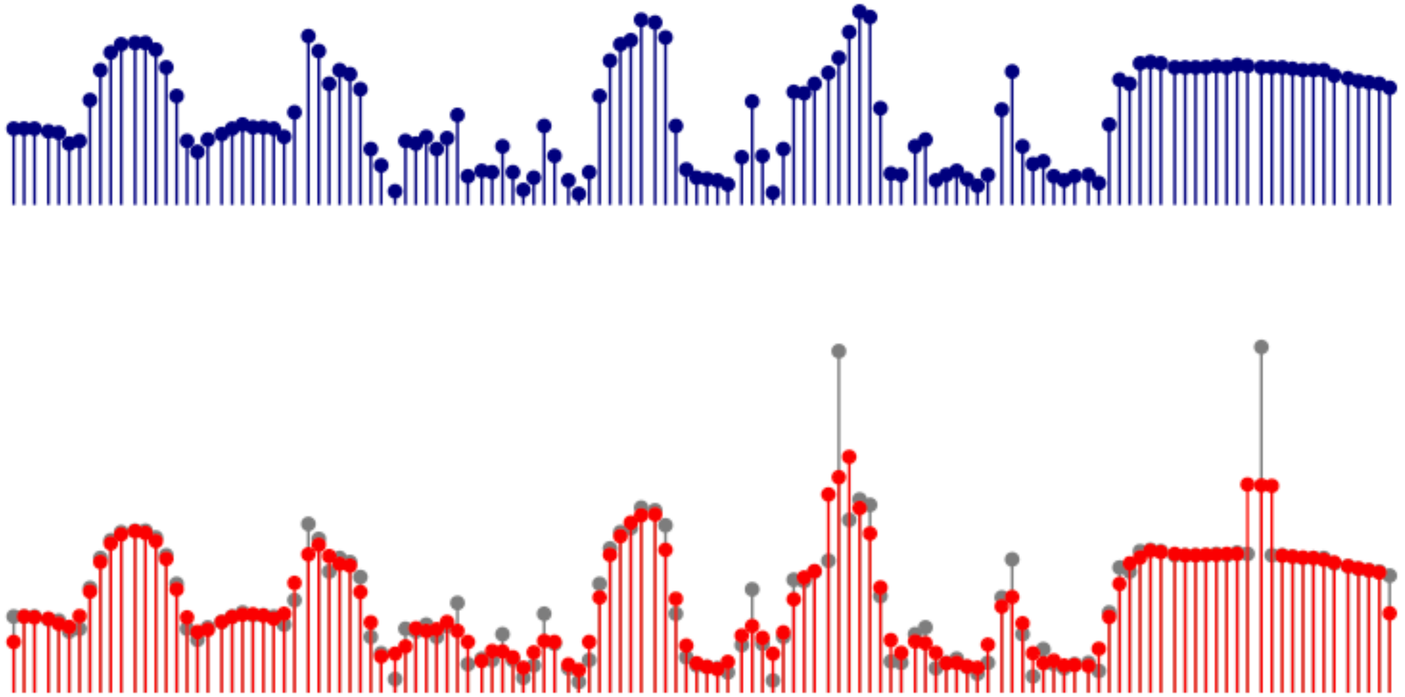
- Ideas about denoising?

- First attempt: noisy pixels are outliers; “smooth out” the noise by averaging

Salt and pepper in 1D

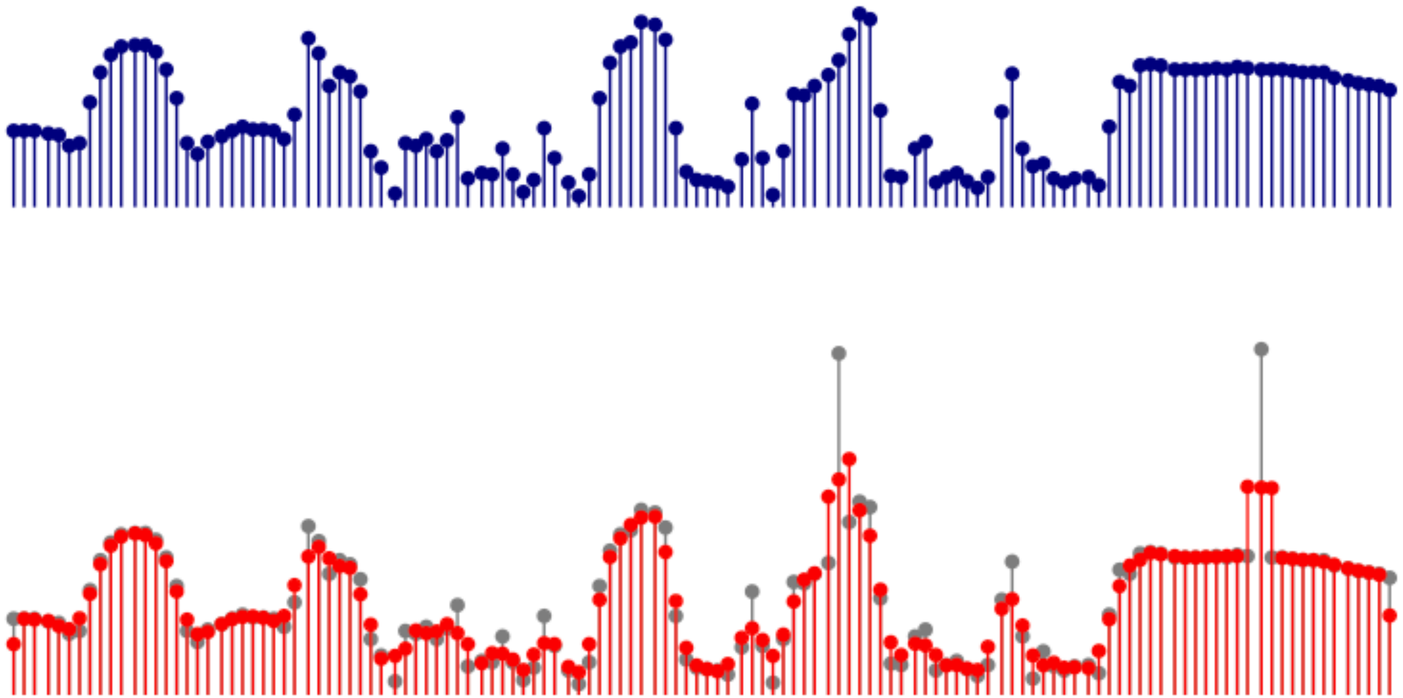
- Smoothing by moving average: replace each value with the average over the neighborhood.

- N=3: take neighbor on the right, neighbor no the left, own value, and average the three values



Salt and pepper in 1D

- Mathematically, for odd N:
- $$\hat{X}(i) = \frac{1}{N} \sum_{j=-(N-1)/2}^{(N-1)/2} X(i-j)$$



Smoothing in 2D

5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	50	5	5
5	5	5	5	5	5	5	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	0	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5

Smoothing in 2D

5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	50	5	5
5	5	5	5	5	5	5	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	0	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5
50	50	50	50	50	50	50	5	5	5

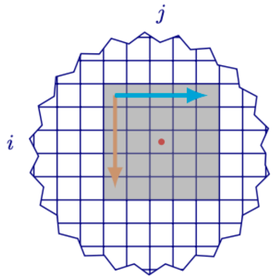
		5							
			44						
					50				

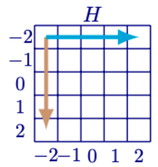
Mean (AKA boxcar) filter

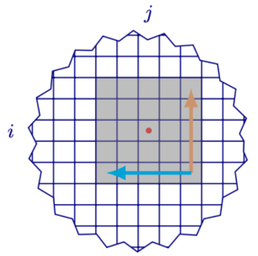
- We can represent the averaging operation with: for N=2k+1
- $$\hat{I}(i,j) = \frac{1}{Z} \sum_{q=i-k}^{i+k} \sum_{r=j-k}^{j+k} I(q,r)$$
- For mean (average) filtering: $Z = N^2$
- General formulation: given a 2D signal $I(i,j)$ and a filter matrix H (also called filter mask/kernel) define $G = \mathrm{H} \otimes I$ by:
- $$G(i,j) = \frac{1}{Z} \sum_{q=-(N-1)/2}^{(N-1)/2} \sum_{r=-(N-1)/2}^{(N-1)/2} I(i+q, j+r) \cdot H(q,r)$$
- This is called cross-correlation

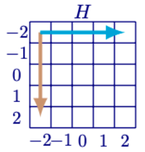
Correlation vs Convolution

- 2D cross-correlation of image $I(x, y)$ with $(2k + 1) \times (2k + 1)$ filter H
- $$(I \otimes H)(i,j) = \sum_{q=-k}^{k} \sum_{r=-k}^{k} I(x+q, y+r) \cdot H(q,r)$$
- 2D convolution:
- $$(I \otimes H)(i,j) = \sum_{q=-k}^{k} \sum_{r=-k}^{k} I(x-q, y-r) \cdot H(q,r)$$

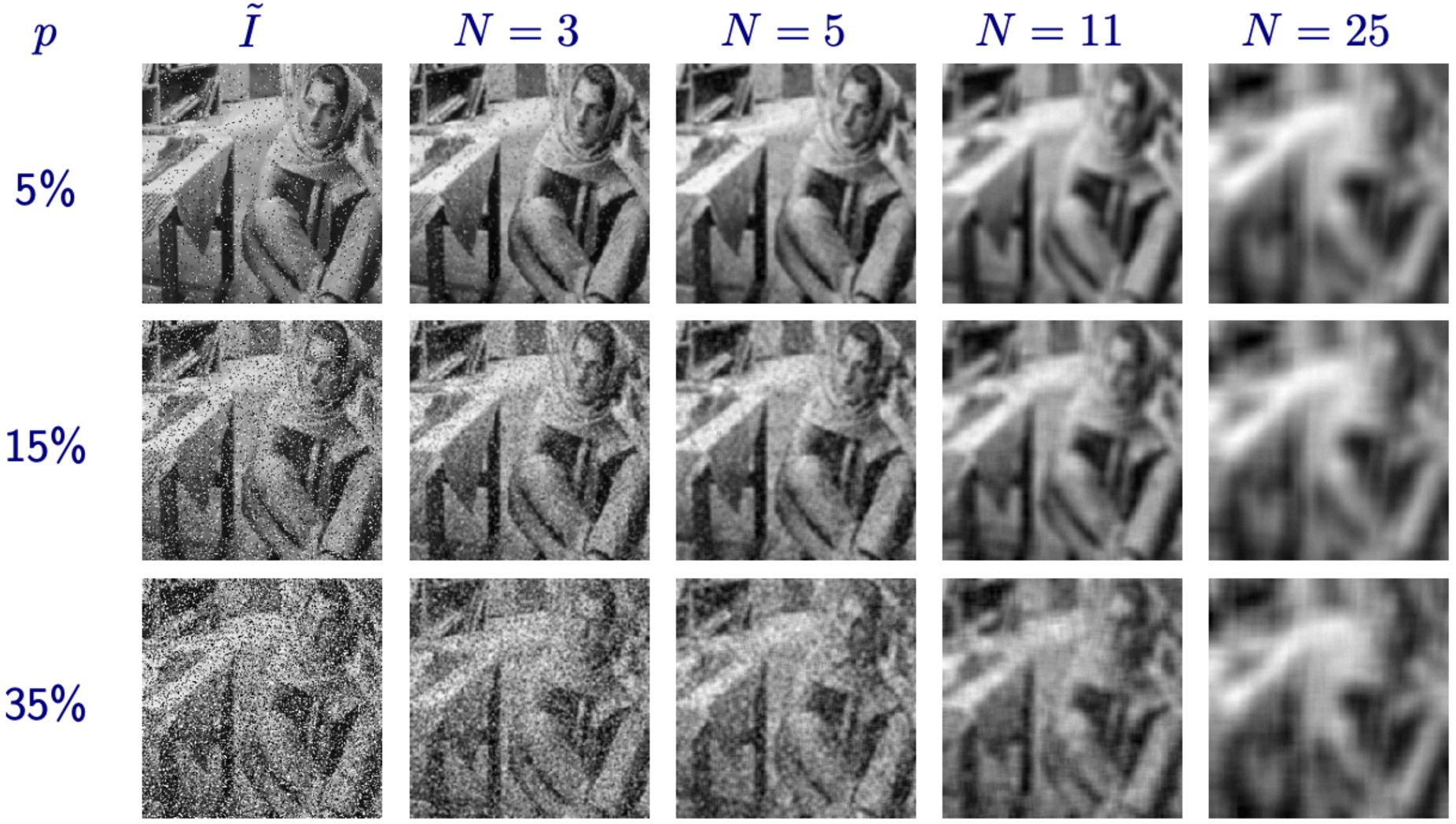








Denoising by mean filter



Smoothing by mean filter

- N=1

Smoothing by mean filter

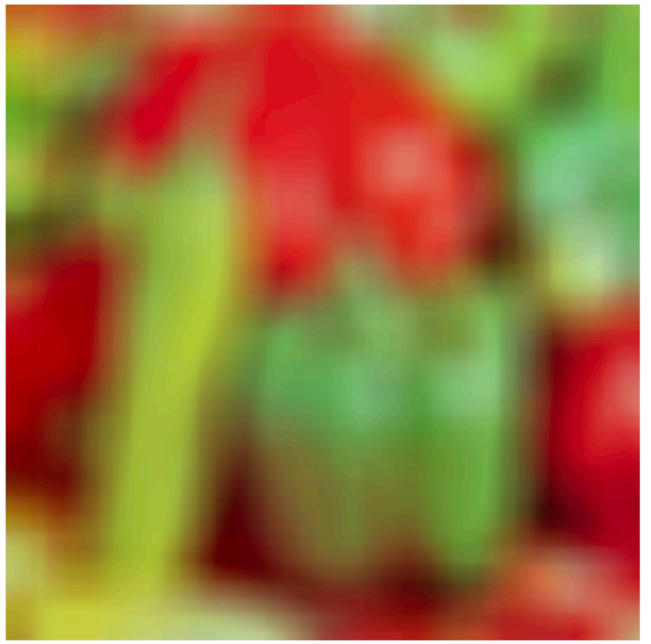
- N=3

Smoothing by mean filter

- N=11

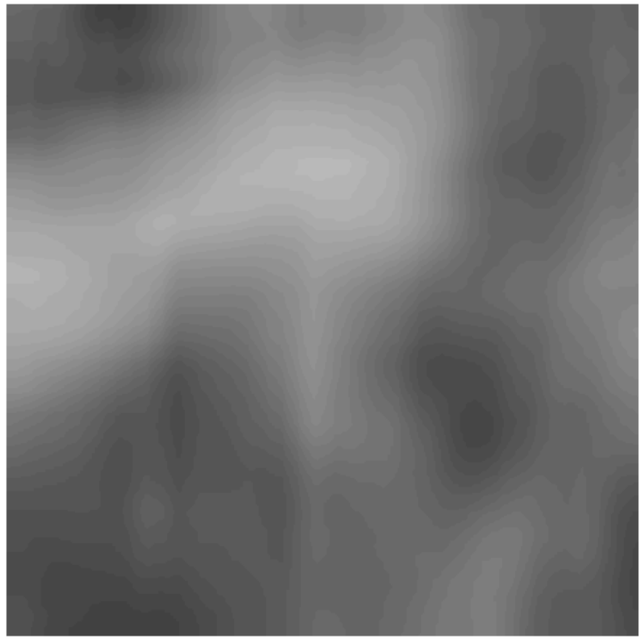
Smoothing by mean filter

- N=25



Smoothing by mean filter

- N=55



Smoothing by mean filter

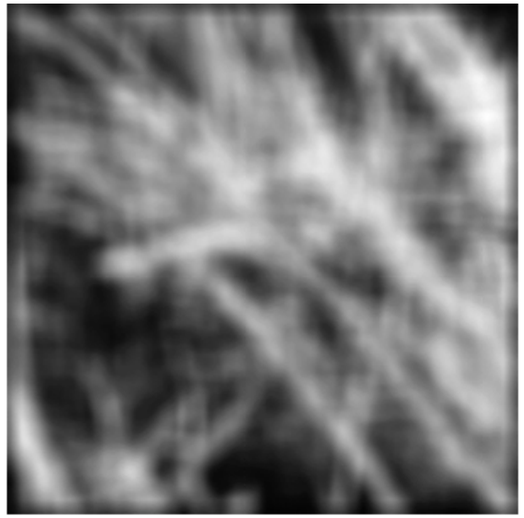
- N=175

Smoothing by mean filter

- N=512

Box filter and ringing

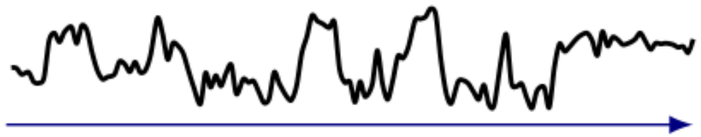
- What's wrong with this image?

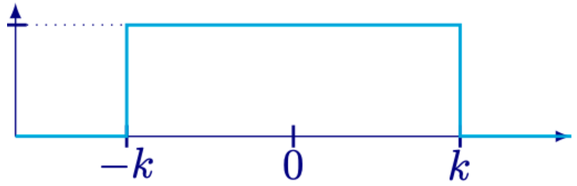


- It's blurred but not like the blur by camera defocus
- The edge artifacts are known as *ringing*.
- Roughly speaking: it's because of the corners of the box; can we eliminate them?

Gaussian filter in 1D

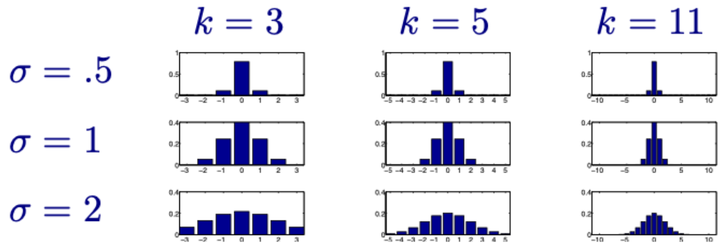
- Mean filtering: we convolve signals with a box filter





- Idea: instead, convolve with a filter shaped like a Gaussian

- Actual digit filter is an approximation of the Gaussian function $p(x; 0,\sigma)$, sampled at integer coordinates $x = -k, \ldots, k$.

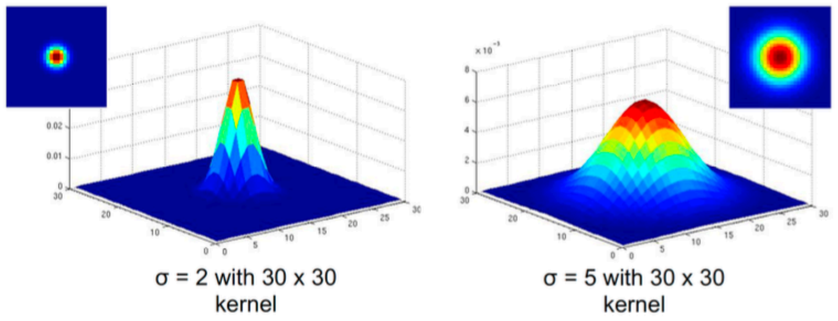


Gaussian filter in 1D

- Rotationally symmetrical 2D Gaussian with zero mean and unit variance:

$$H(i,j) = \frac{1}{2\pi\sigma^2} exp(-\frac{i^2 + j^2}{2\sigma^2})$$

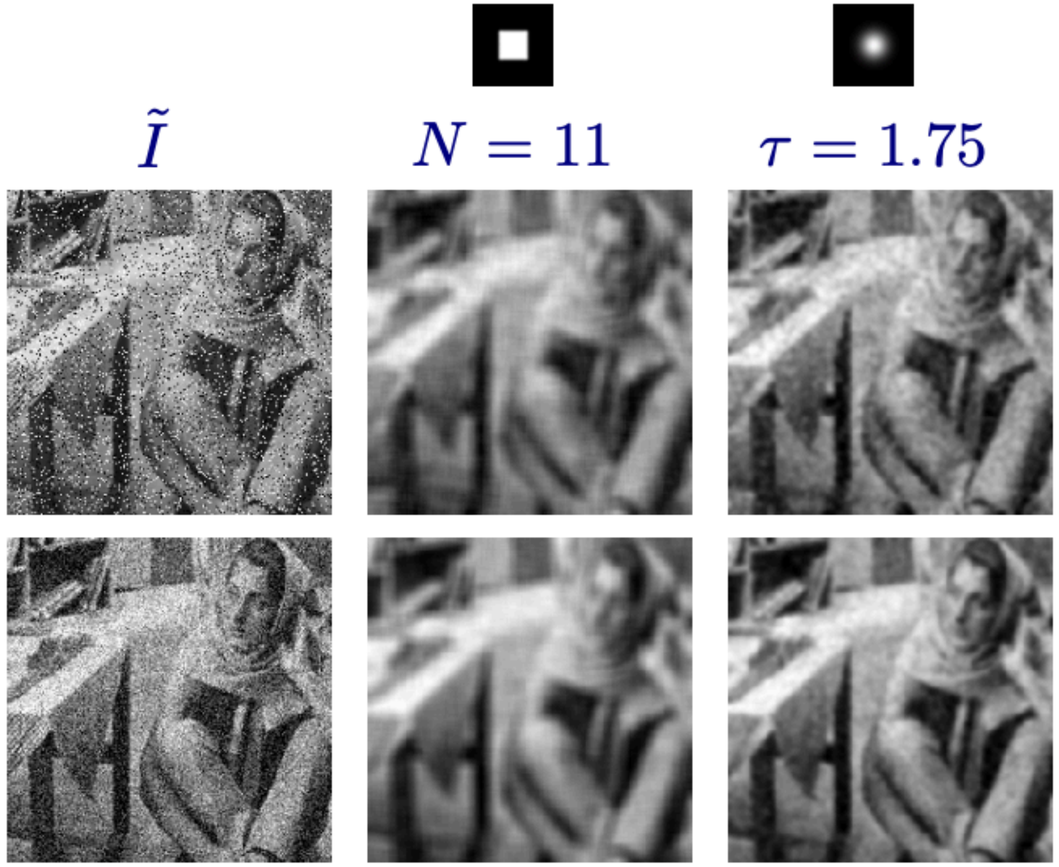
- In practice we use sampled and approximated discrete filter



- Good practice: truncate at 3 standard deviations from center; make sure to normalize (sum to one)

Denoising by Gaussian filtering

- Comparison, with approximately same window sizes:

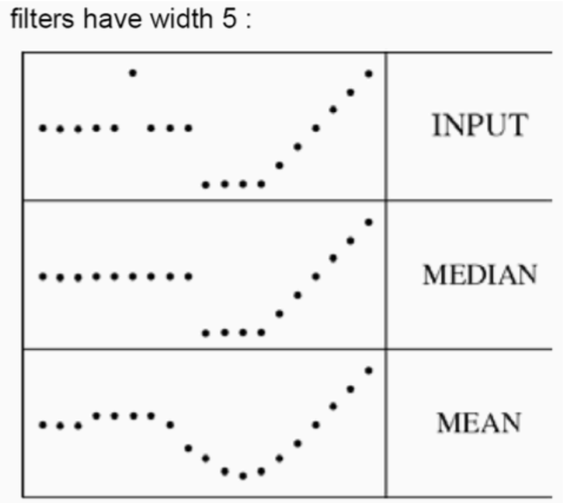


Blurring with Gaussian filter

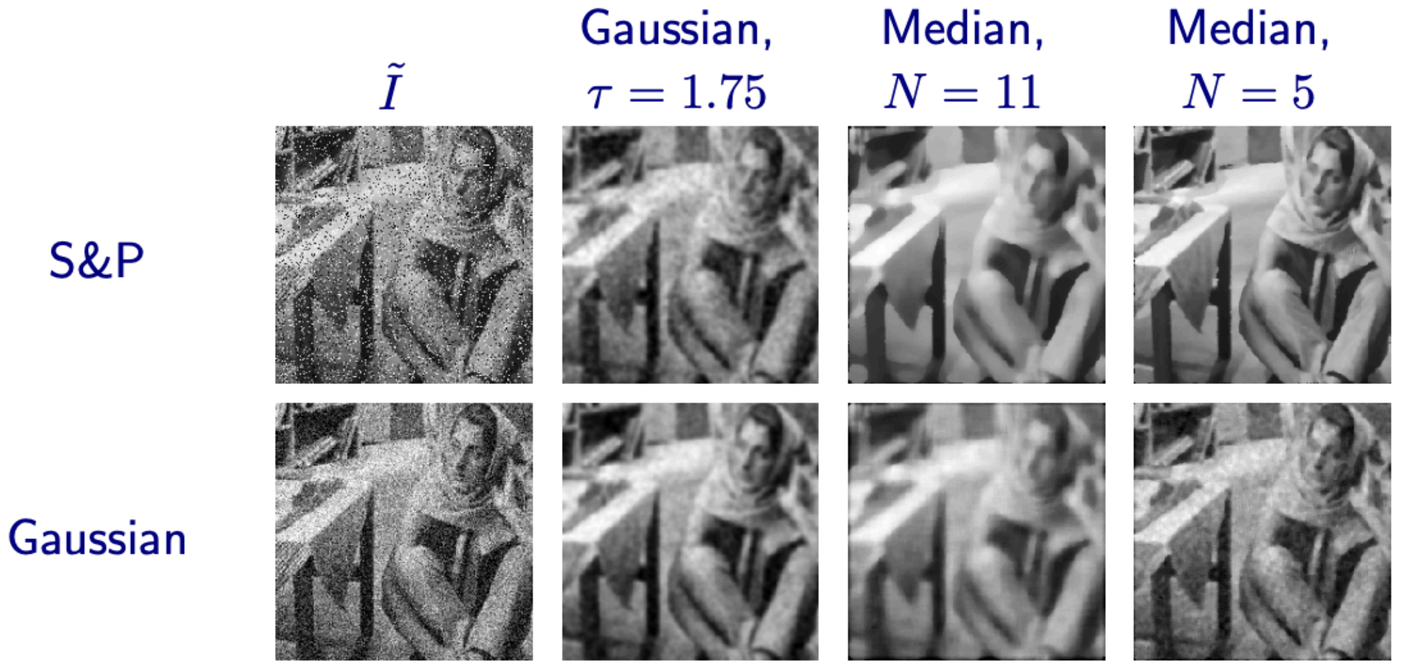
- Ringing mostly gone, but still have a tradeoff: can remove noise but overblur the image

Mean vs median

- Compute the mean: sum, divide by N^2
- Computing the median: sort the values, take the value in the middle
- For every pixel, replace the value with the median over a $N \times N$ window centered on that pixel
- Why is this better? Less sensitive to outliers!



Denoising by median filter



- Can we write median filter as a convolution?

Linear shift-invariant operators

- Consider two properties of an operation F :
  - Linearity: $F(ax + by) = aF(x) + bF(y)$
  - Shift-invariance: for a translation (shift) $T, F(T(x)) = T(F(x))$
- Theory: F can be expressed as a convolution (cross-correlation) if and only if it is linear and shift invariant
- Key (for modern hardware): correlation can be written as a linear operation on the image
- Is median filter linear?

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 2 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Global filtering?

- Intuition: linear filters (box, Gaussian) are non-adaptive
- Median filter is adaptive (depends on the signal), but still local (only depends on the area around the pixel in question)
- Idea: adapt the filtering to the global content on the image

$$y \to \hat{x} : \hat{x}_i = \frac{\sum_j w_{i,j} y_j}{\sum_j w_{i,j}}$$

Bilateral filter: definition

- $\hat{x}_i = \dfrac{\sum_j w_{i,j} y_j}{\sum_j w_{i,j}} \qquad w_{i,j} = \phi_{\text{position}}(s_i, s_j) \cdot \phi_{\text{color}}(y_i, y_j)$

- ϕ_{position}: nearby pixels affect more than faraway ones;

- ϕ_{color}: pixels with similar color/intensity affect more than the more different ones

Denoising by bilateral filter

- $\phi_{\text{color}}(y_i, y_j) = \exp(-||y_i - y_j||/2\tau^2_{\text{color}})$

(a) Noisy image $\sigma = 10$ (b) Bilateral filter $\tau_{\text{color}} = 5$ (c) $\tau_{\text{color}} = 20$

(d) $\tau_{\text{color}} = 40$ (e) $\tau_{\text{color}} = 100$ (f) $\tau_{\text{color}} = 200$

Image credits: C. Deledalle

Denoising by bilateral filter

- Preserves textures much better!
- Remaining problems: still some noise, and some over-smoothing

Image patches

- A common tool: extra sub-images from image
- Usually rectangular; typical sizes are 10-20 pixels across
- Larger patches: more information, harder to match/model
- Smaller patches: less information, easier to handle

Non-local images: idea

- $\hat{x}_i = \dfrac{\sum_j \in \mathcal{N}(i) w_{i,j} y_j}{\sum_j \in \mathcal{N}(i) w_{i,j}} \qquad w_{i,j} = \phi(||\mathcal{P}_i y - \mathcal{P}_j y||)$

- $\mathcal{N}(i)$: neighborhood around s_i (e.g., 21 pixels)

- $\mathcal{P}_i y$: patch (e.g., 7x7 pixels or 15x15) around s_i in y

Image credits: C. Deledalle

Non-local means weights

- How can we measure $||\mathscr{P}_i y - \mathscr{P}_j y||$? Typically, SSD (sum of squared pixel differences, after aligning the patches)

- For color images: measure SSD in a color space

Non-local means operation

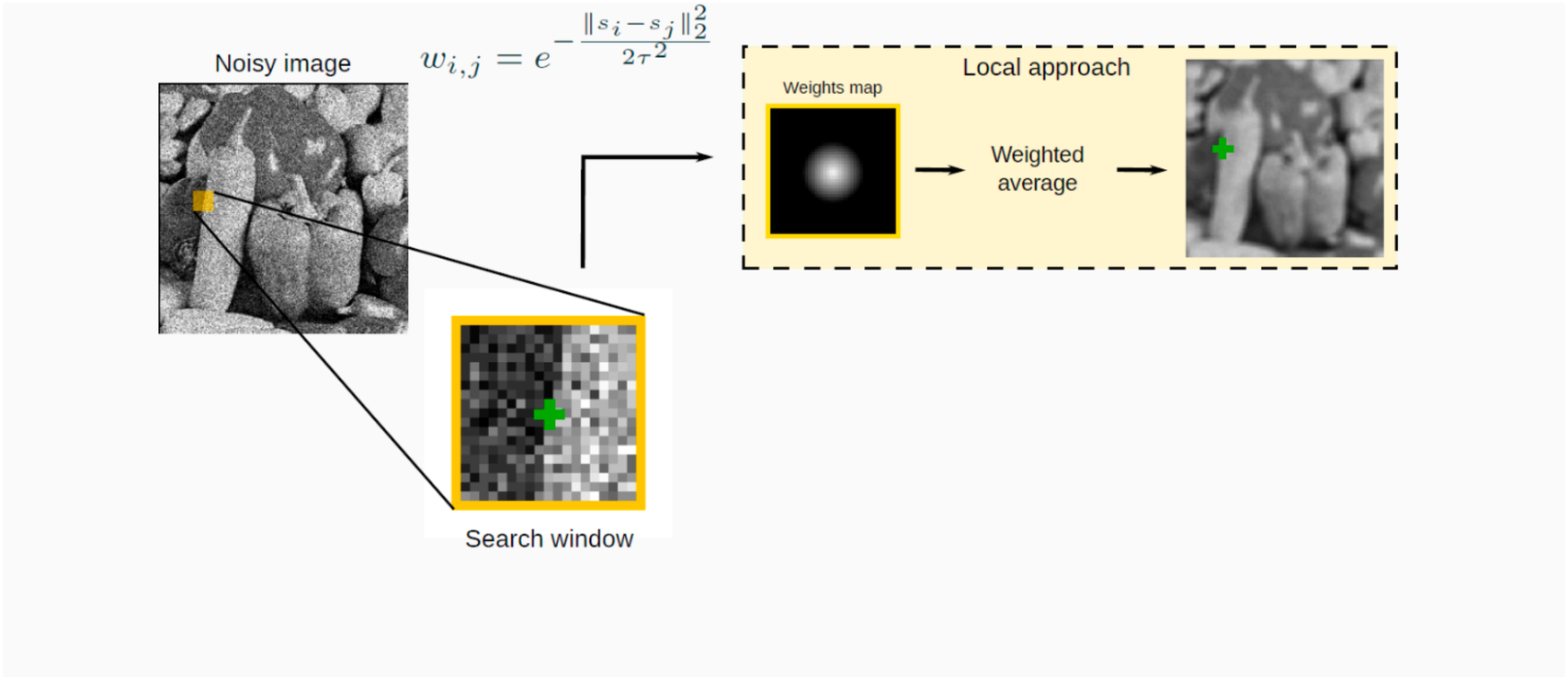


Image credits: C. Deledalle

Non-local means operation

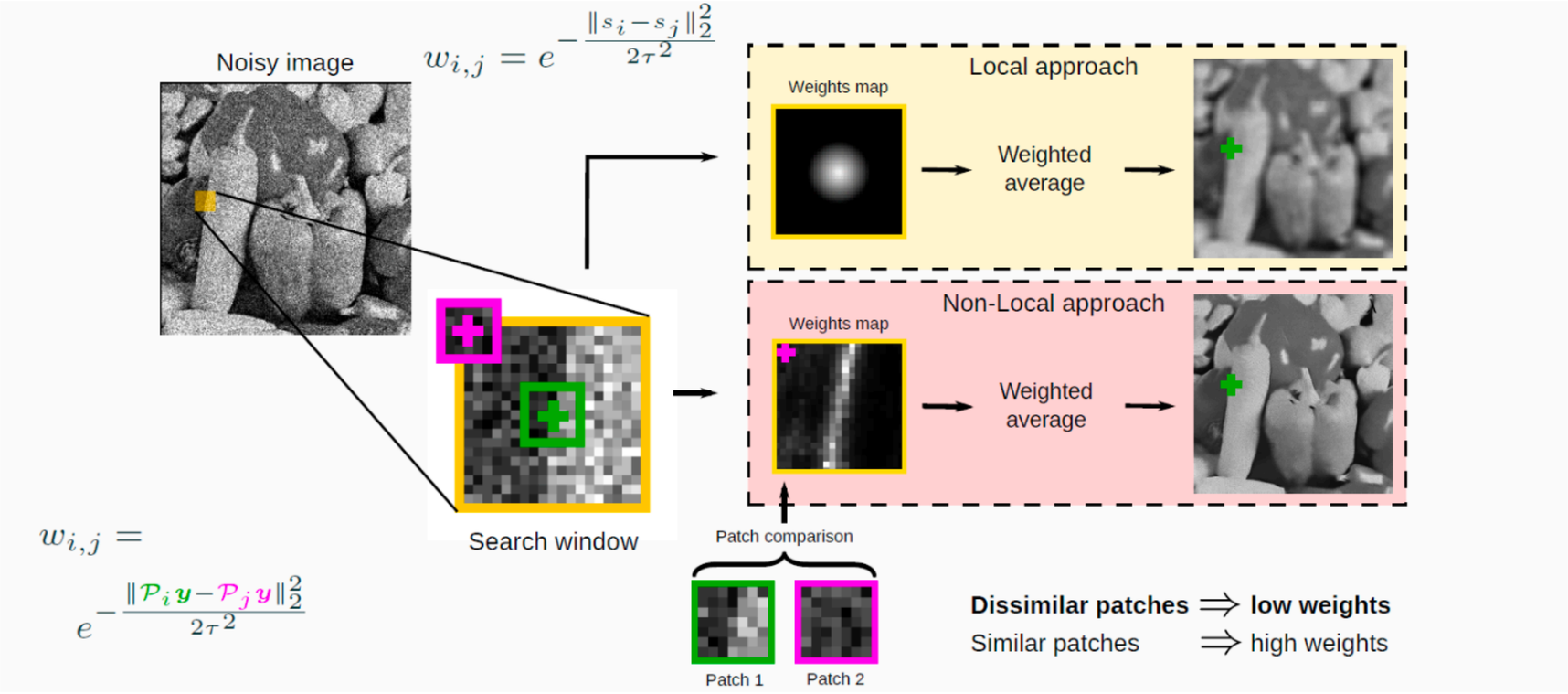
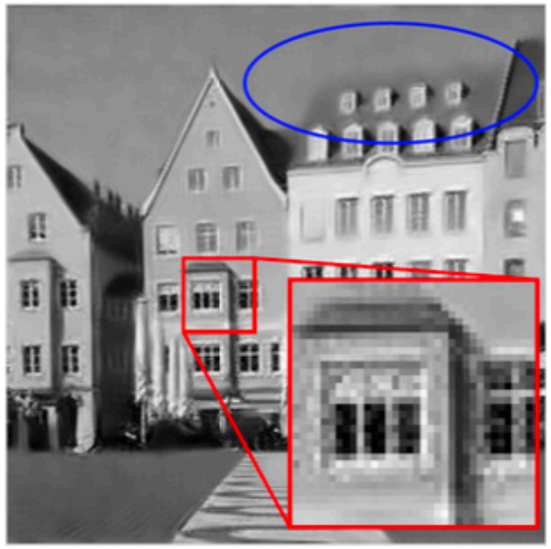


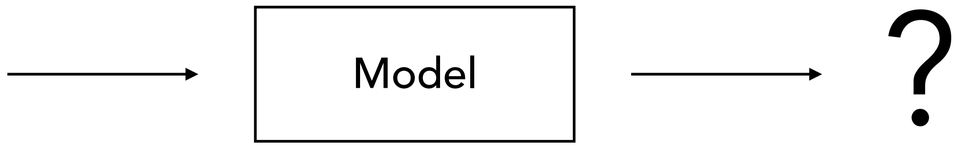
Image credits: C. Deledalle

Non-local mean results



- Preserves edges and textures
- Remaining problems: doesn’t deal well with rare patches; blurs areas with weak signal

Image classification




```
def classify_image(image):
  # Some magic here?
  return class_label
```

Image credits: Stanford CS231N

Training

( , dog) → Model

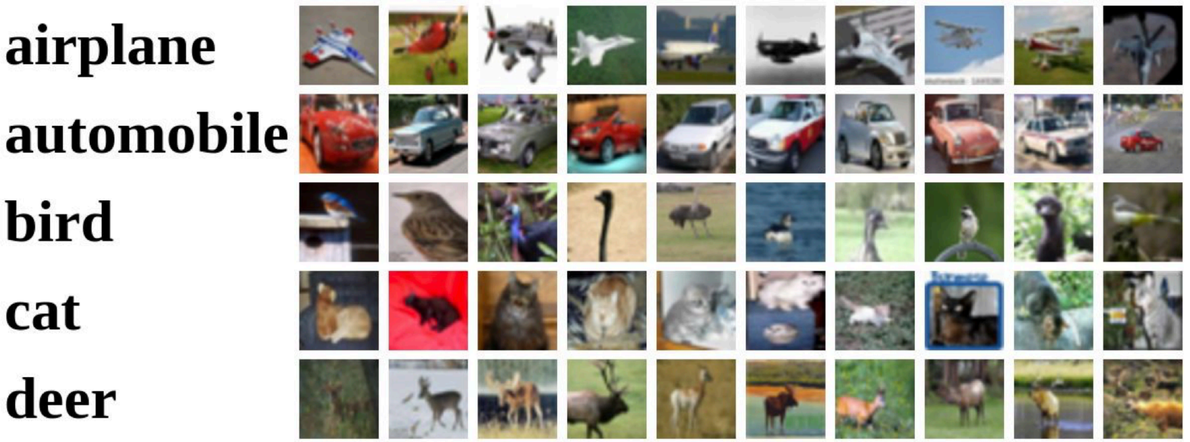
Training

( , labels) ⟶ Model

Testing

Data-driven approach to image classification

- Collect a dataset of images and labels
- Design machine learning algorithms to consume these images and labels and build a image classifier
- Evaluate the classifier on new images



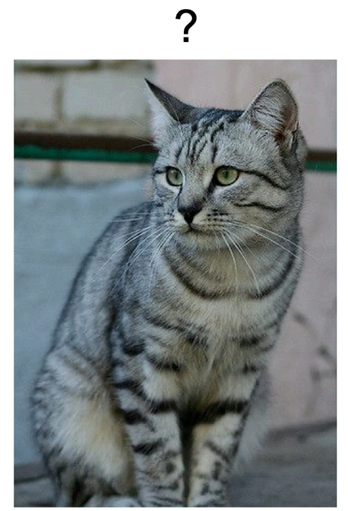

```
def train(images, labels):
  # Machine learning!
  return model
```

```
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

Slide credits: Stanford CS231N

K nearest neighbor classifier

Training data



Test data

How similar are these two images?

Nearest neighbor classifier

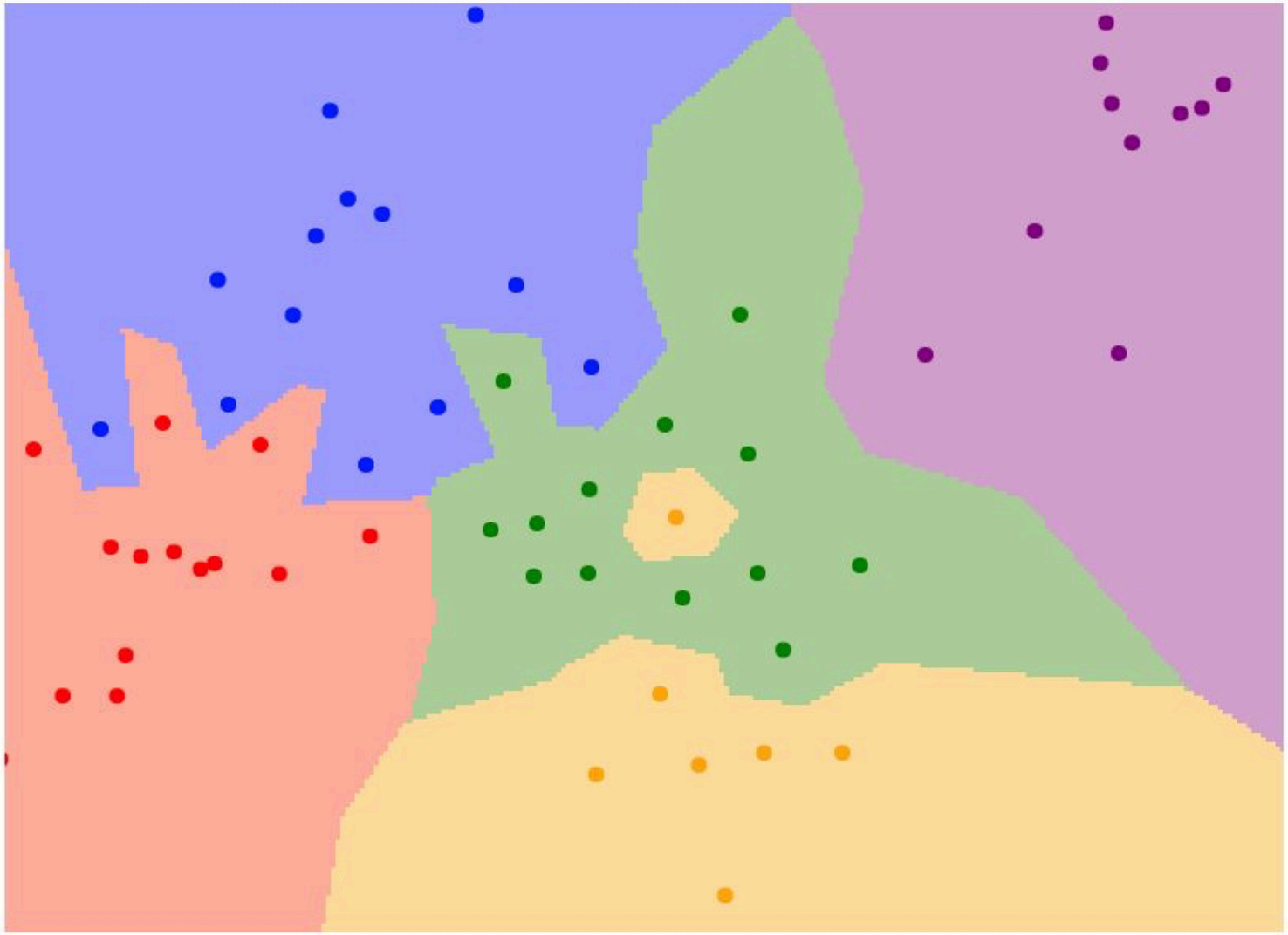
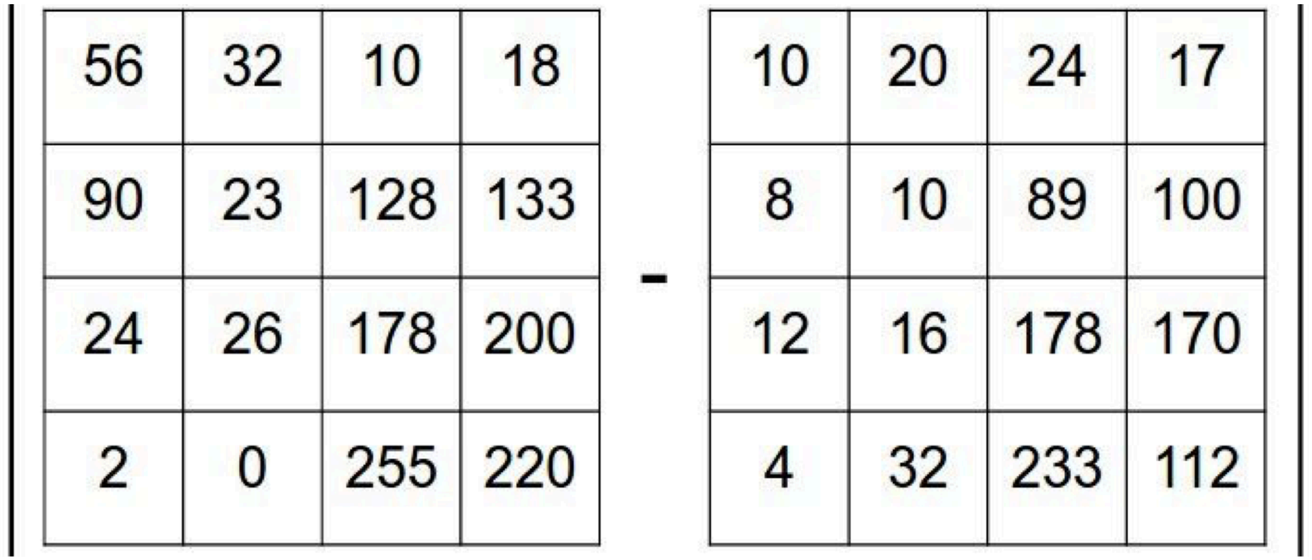


Image credits: Stanford CS231N

Distance metrics

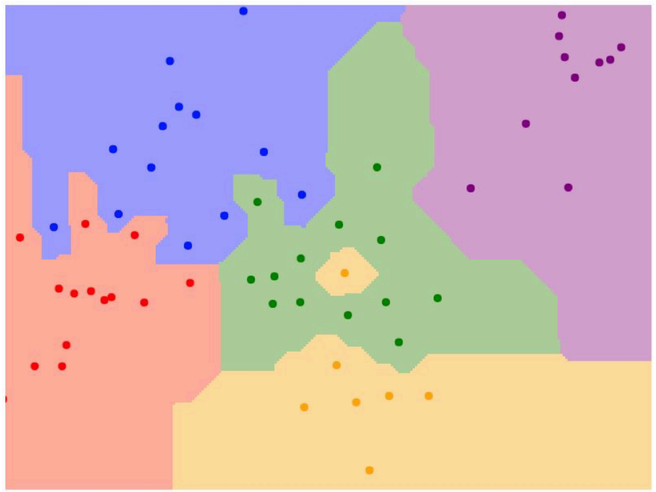


- L1 metric
  - $d(I_1, I_2) = \sum |(I_1(x, y) - I_2(x, y)|$
- L2 metric
  - $d(I_1, I_2) = \sqrt{\sum |(I_1(x, y) - I_2(x, y)|^2}$

Image credits: Stanford CS231N

Distance metrics

- L1 metric
  - $d(I_1, I_2) = \sum |(I_1(x, y) - I_2(x, y)|$



- L2 metric
  - $d(I_1, I_2) = \sqrt{\sum |(I_1(x, y) - I_2(x, y)|^2}$

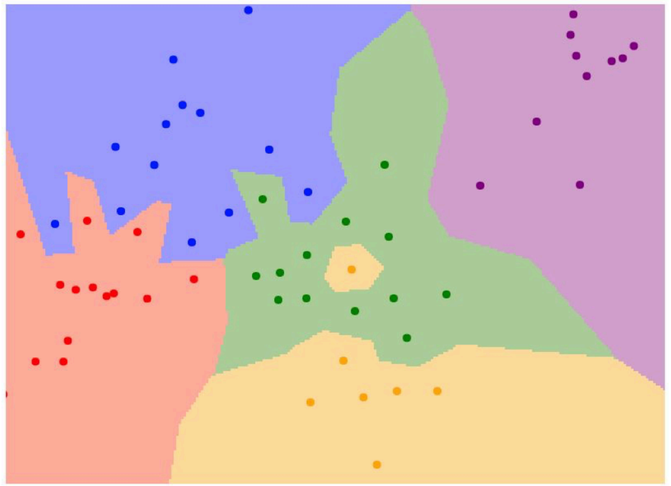


Image credits: Stanford CS231N

1 nearest neighbor

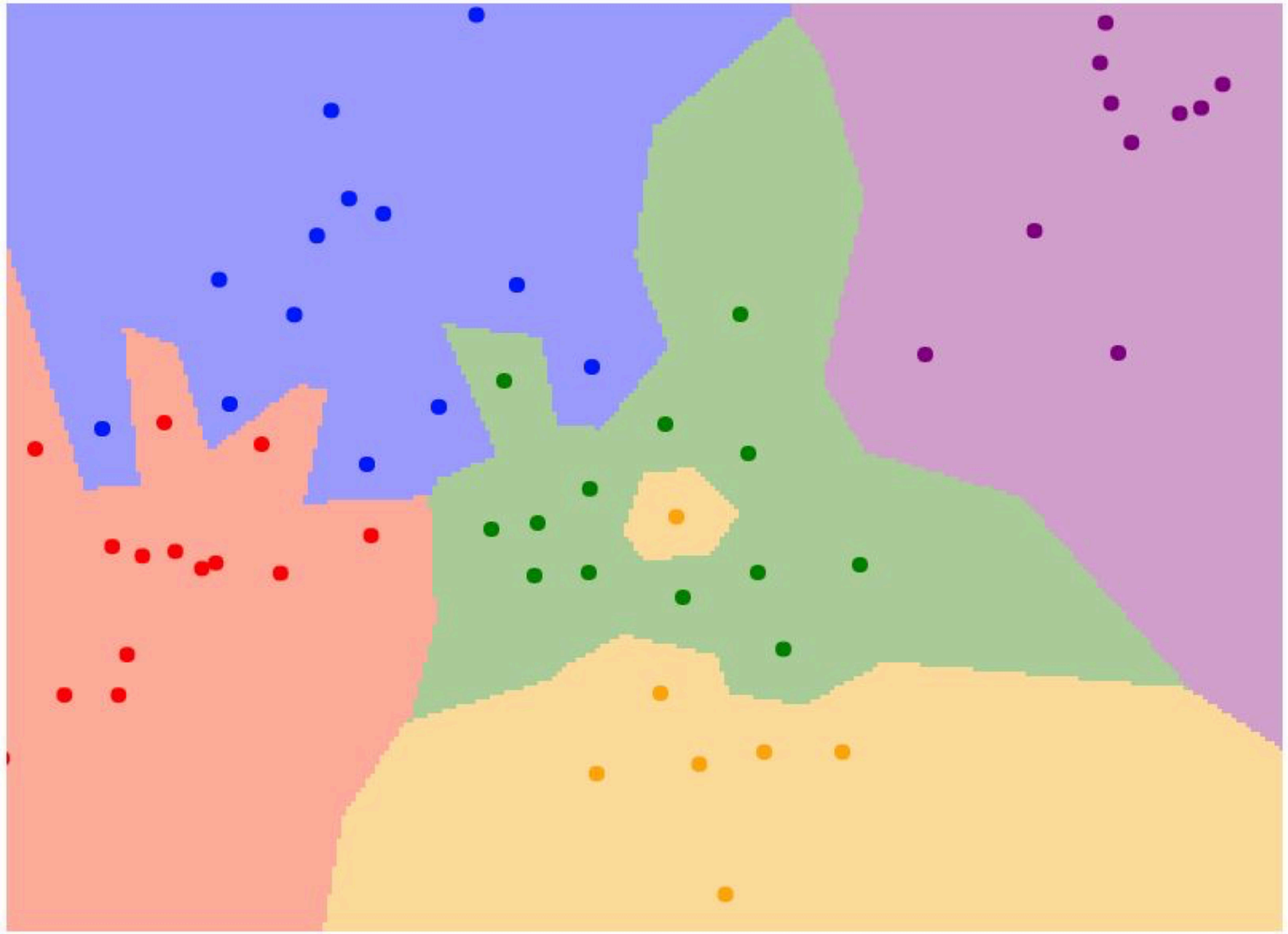


Image credits: Stanford CS231N

1 nearest neighbor

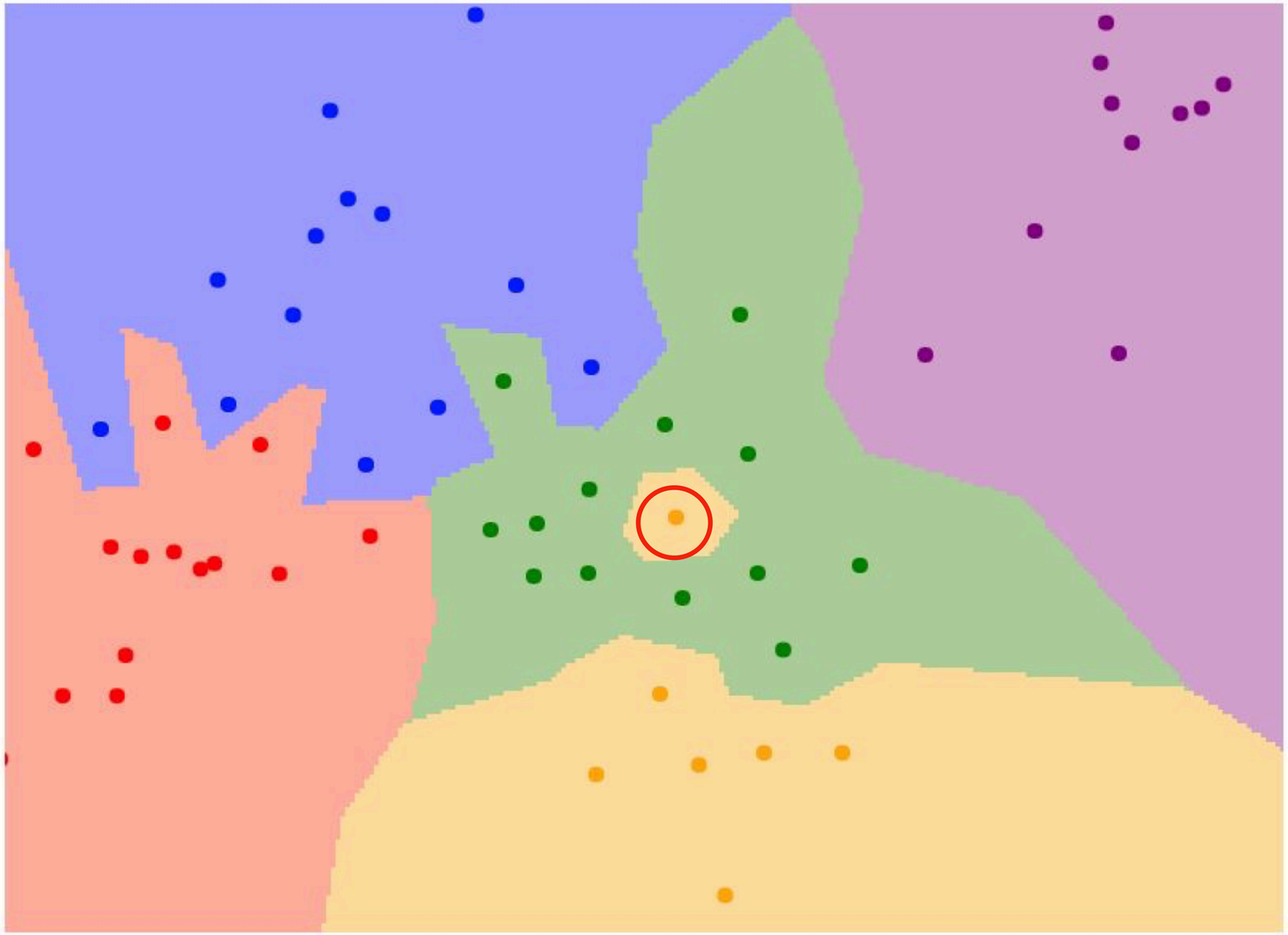
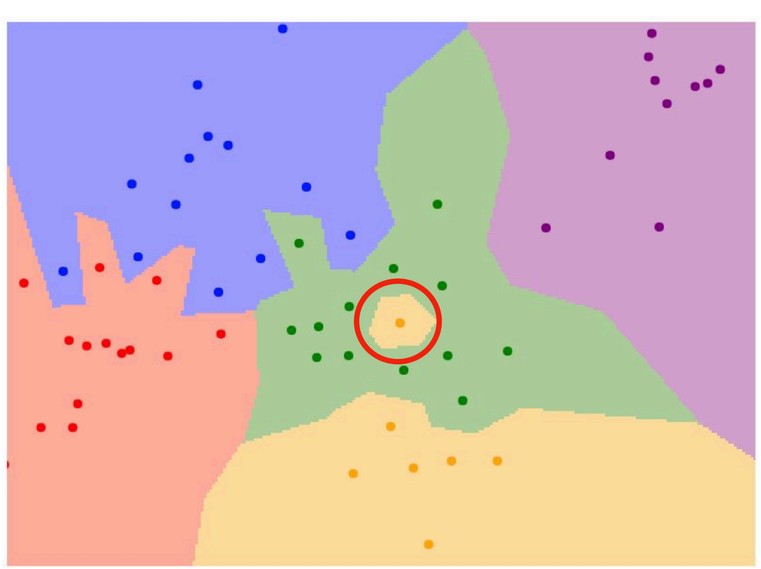


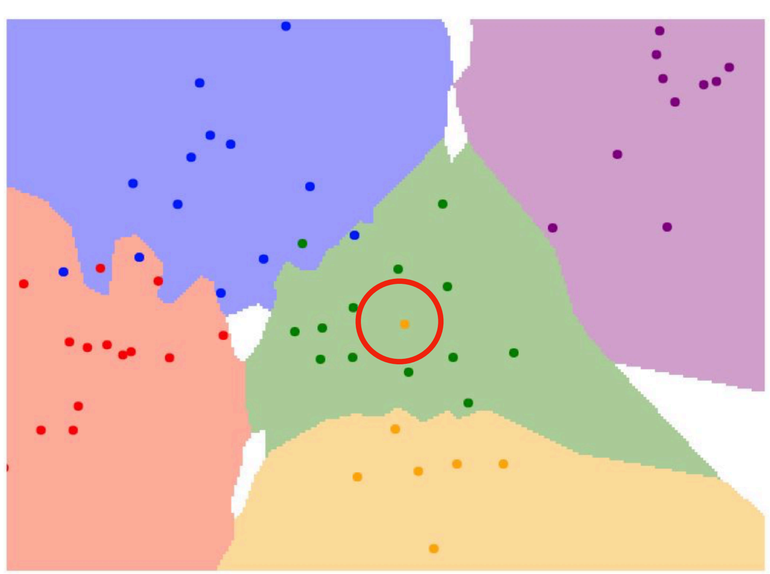
Image credits: Stanford CS231N

K nearest neighbors

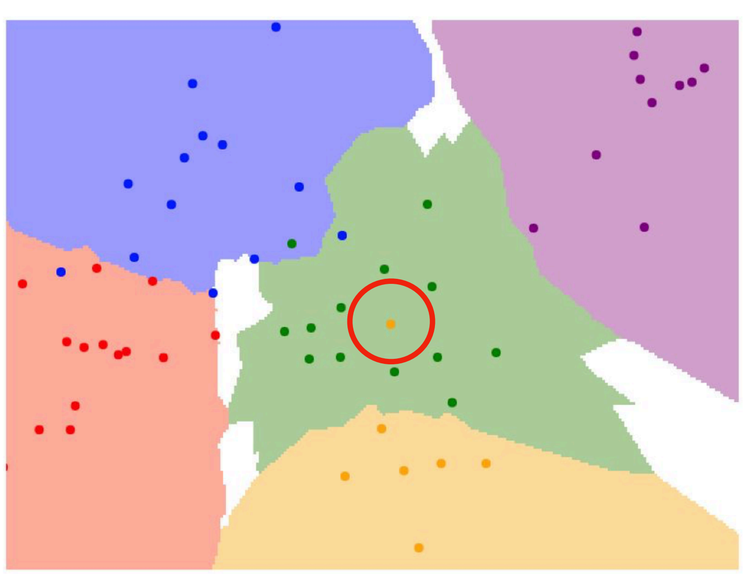
- Take majority vote from K nearest neighbors



K = 1



K = 3



K = 5

In reality…

- K-nearest-neighbor classifier with pixel distance is never used.
  - All these images on the right have the same distances to the original one.

Original

Occluded

Shifted (1 pixel)

Tinted

Feature descriptor

SIFT

(CNNs)

Transformers

Linear classifier

- From non-parametric approach (e.g. KNN) to parametric approach
  - Why?
  - Easy to work with — adjust values of parameters
  - More statistical power
  - Computationally faster
    - Think about KNN with 1K training examples, 1M training examples, 1B training examples

Linear classifier

x: 32x32x3=3072 numbers

f(x, w)

Parameters

10 numbers/
class scores

Linear classifier

x: 32x32x3=3072 numbers

f(x, W) → 10 numbers/ class scores

Parameters

$$\underset{10\times1}{f(x, W)} = \underset{10\times3072}{W}\underset{3072\times1}{x}$$

Linear classifier

x: 32x32x3=3072 numbers

f(x, W)

10 numbers/
class scores

Parameters

$$\underset{10\times1}{f(x, W)} = \underset{10\times3072}{W}\underset{3072\times1}{x} + \underset{10\times1}{b}$$

Simple example

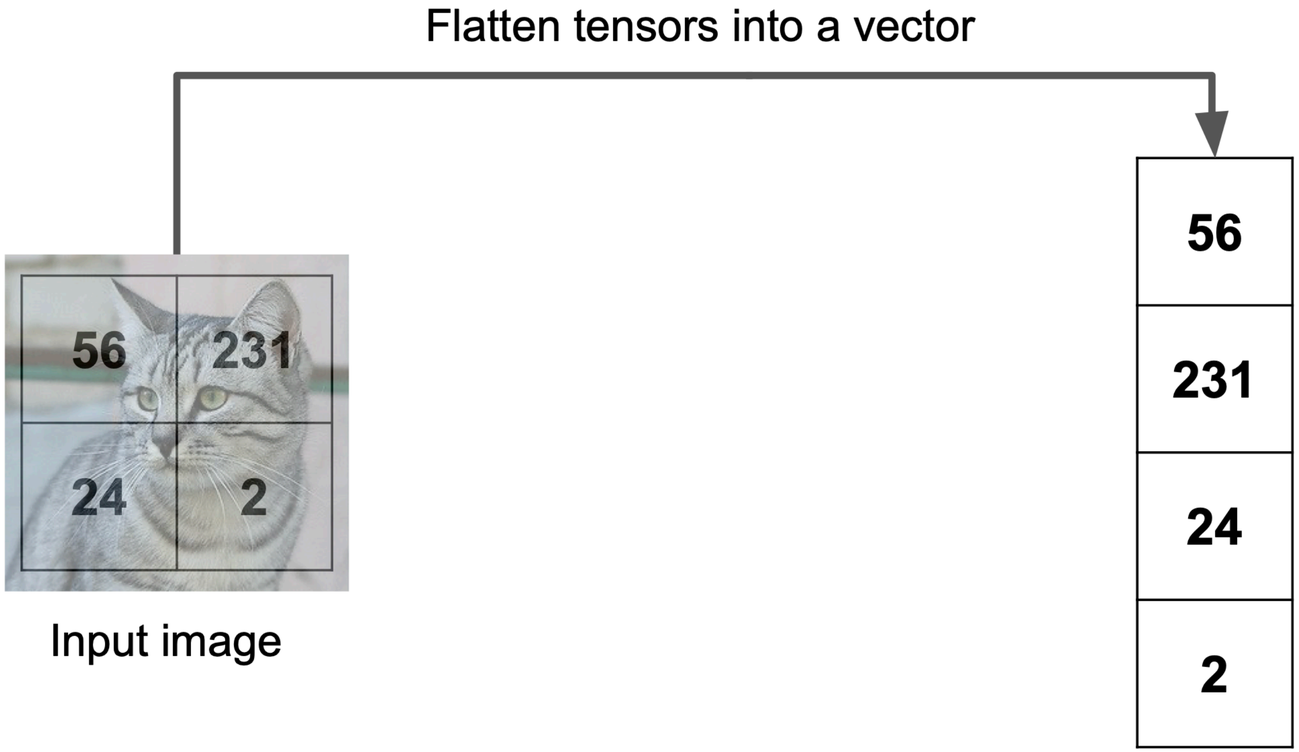


Image credits: Stanford CS231N

Simple example

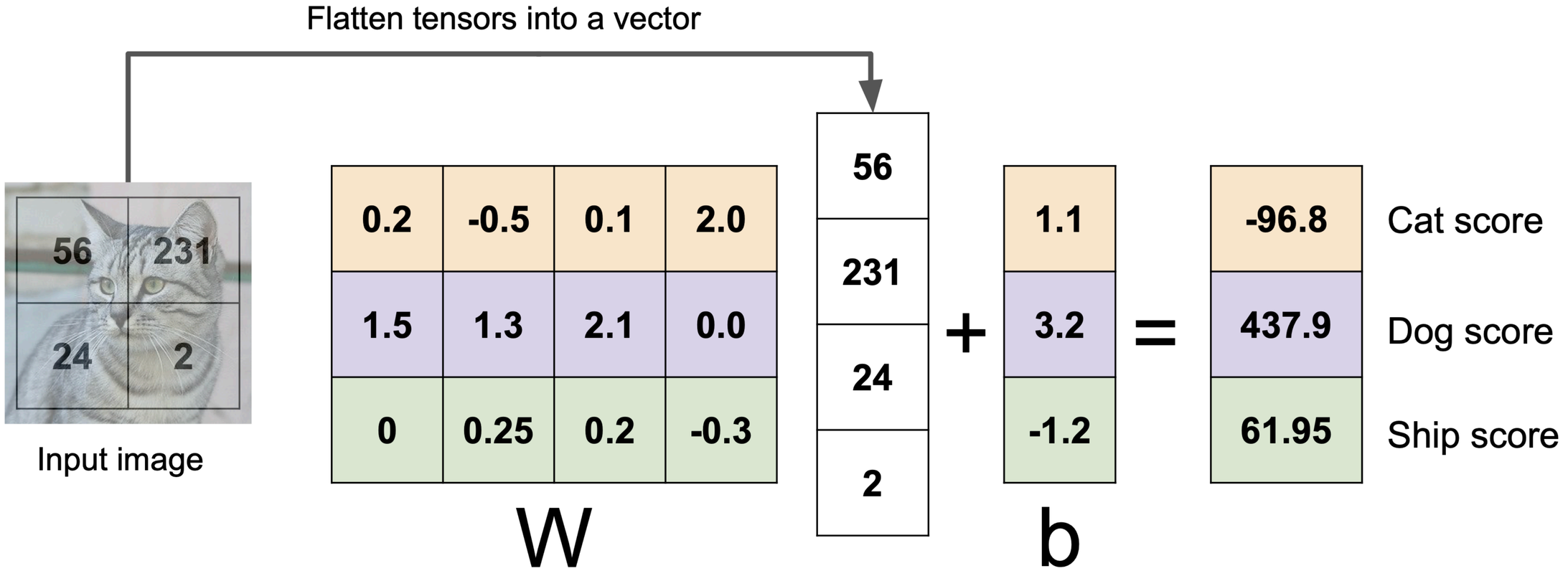


Image credits: Stanford CS231N

Simple example: interpretation

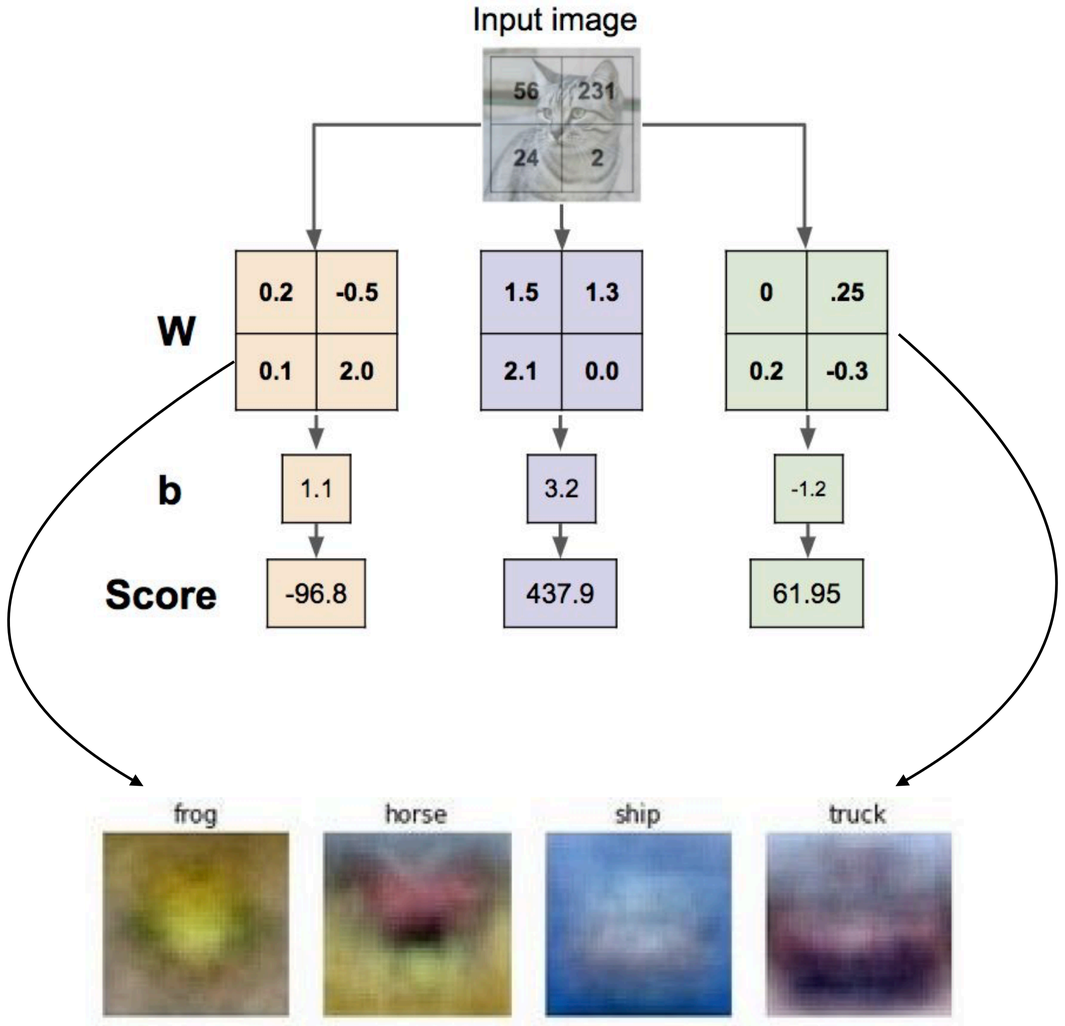


Image credits: Stanford CS231N

Simple example: interpretation

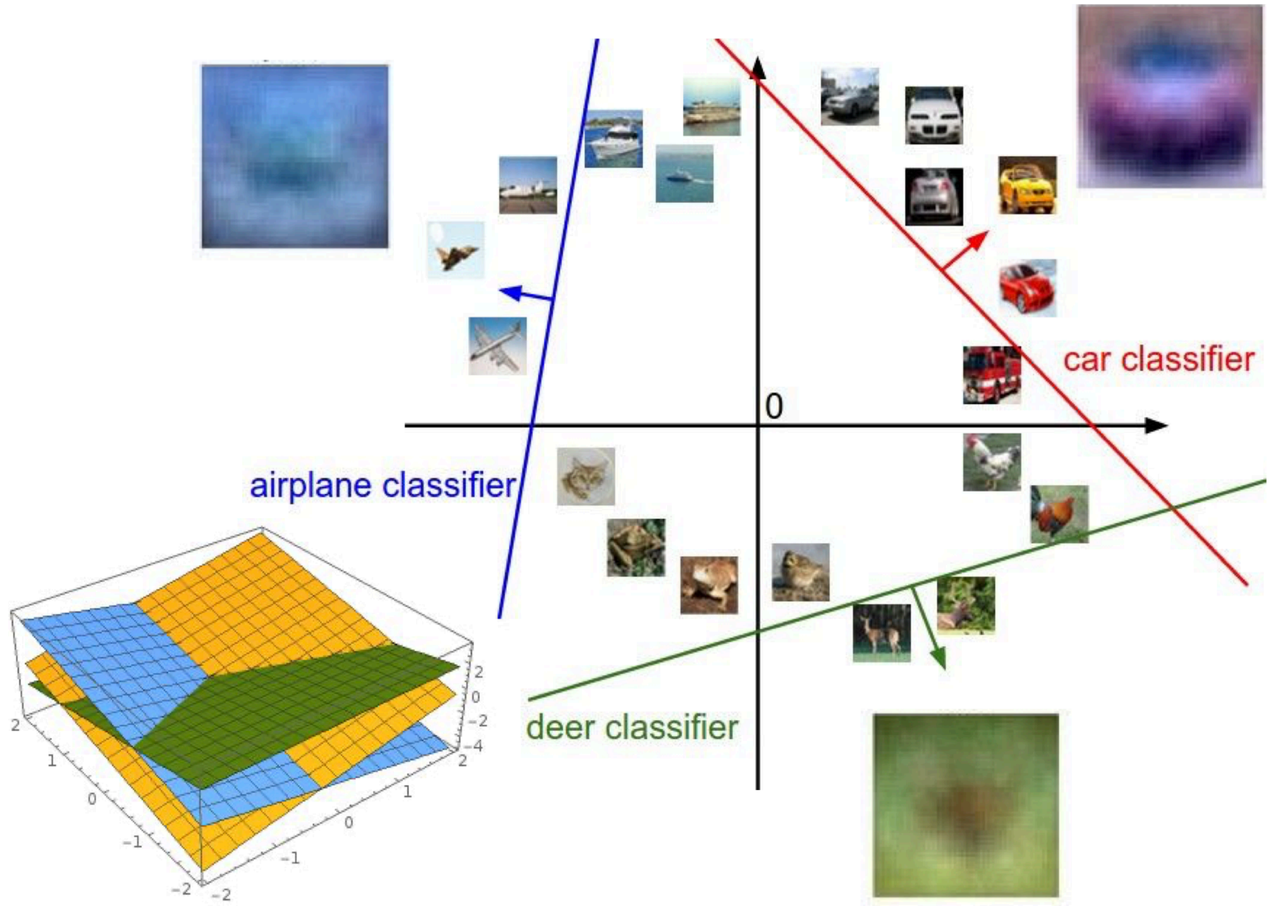


Image credits: Stanford CS231N

How to train this classifier?

- Find a way to penalize model's prediction against ground-truth labels.
  - This is usually called loss function.
- Optimize the parameters based on the penalty.
  - E.g. gradient descent

Loss function

- A loss function tells how good the classifier is.
- $\{(x_i, y_i)_{i=1}^N\}$ where x_i is image and y_i is a categorial label (integer).
- The total loss is the average of loss over examples
  - $L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$

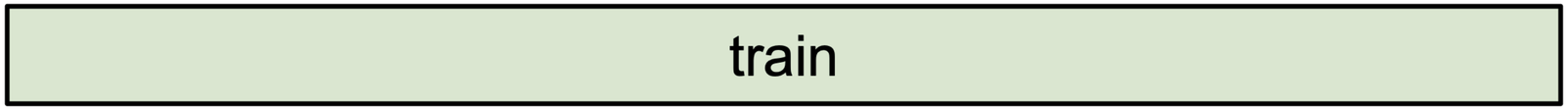
Hyper-parameters

- Parameters are not directly optimized by algorithms, instead choices about the algorithms themselves
- Choice of loss function: cross-entropy loss (Softmax) vs hinge loss (SVM)
- Number of neighbors in KNN
- How long you want to train your model
- ……

Image credits: Stanford CS231N

Hyper-parameters

- Choice 1: hyper-parameters that work best on the training data.



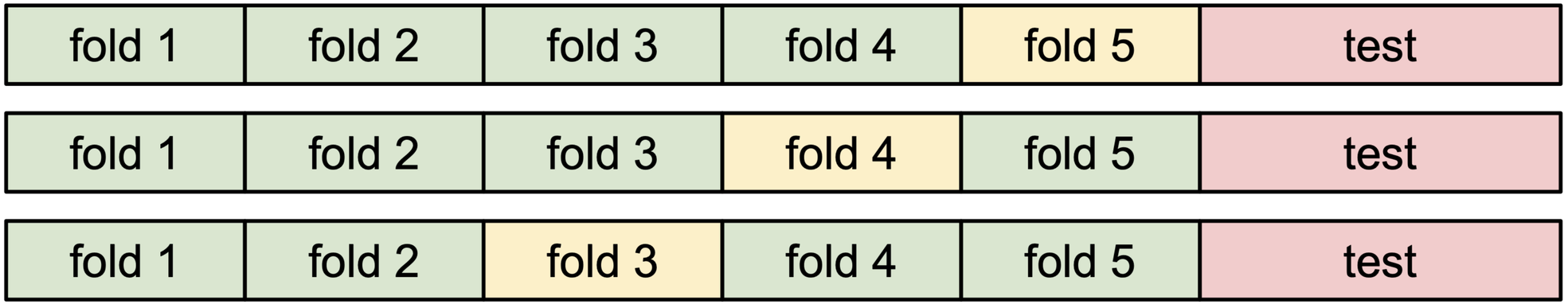
- Choice 2: hyper-parameters that work best o the test data.

- Choice 3: split train into train+validation, assume validation can well represent test data

Slides adapted from Stanford CS231N

Hyper-parameters

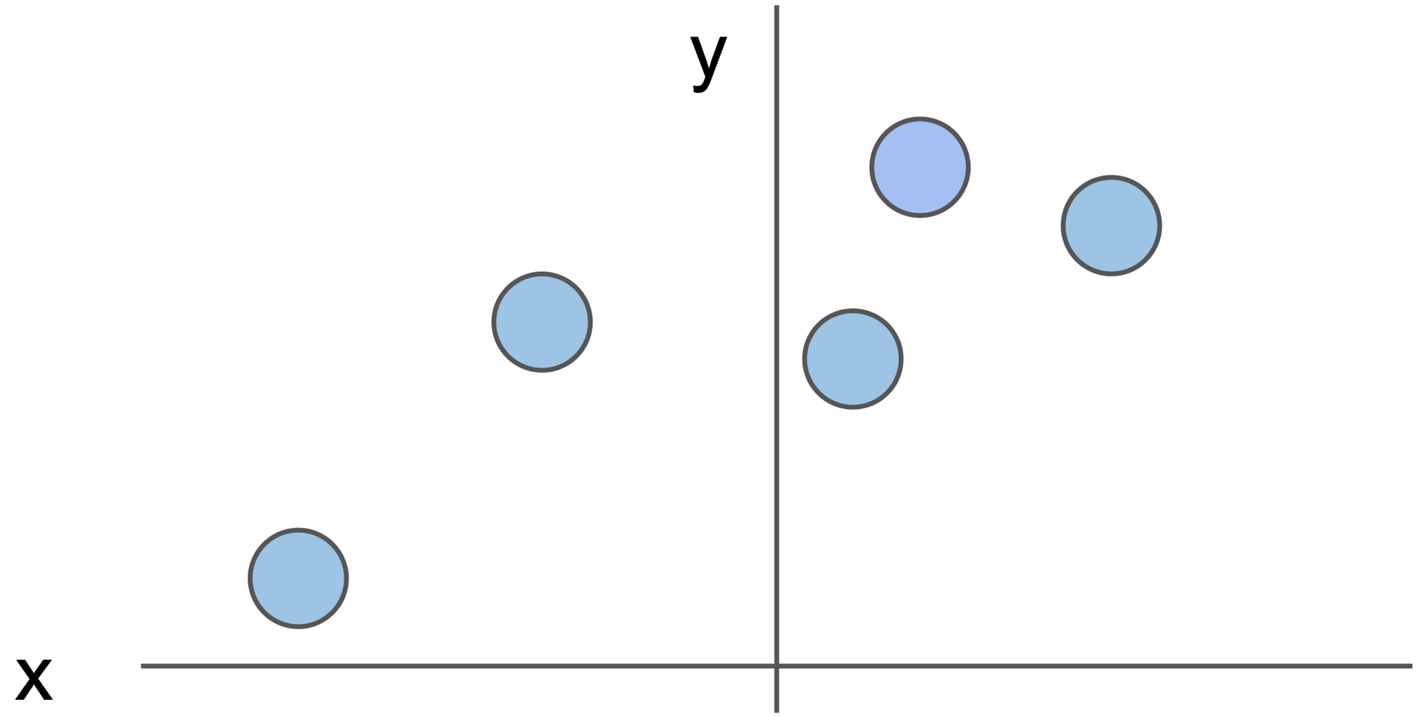
- Choice 4: cross-validation, split data into folds, try each fold as validation.
  - Expensive, not widely used in machine learning and computer vision.



Slides adapted from Stanford CS231N

Regularization

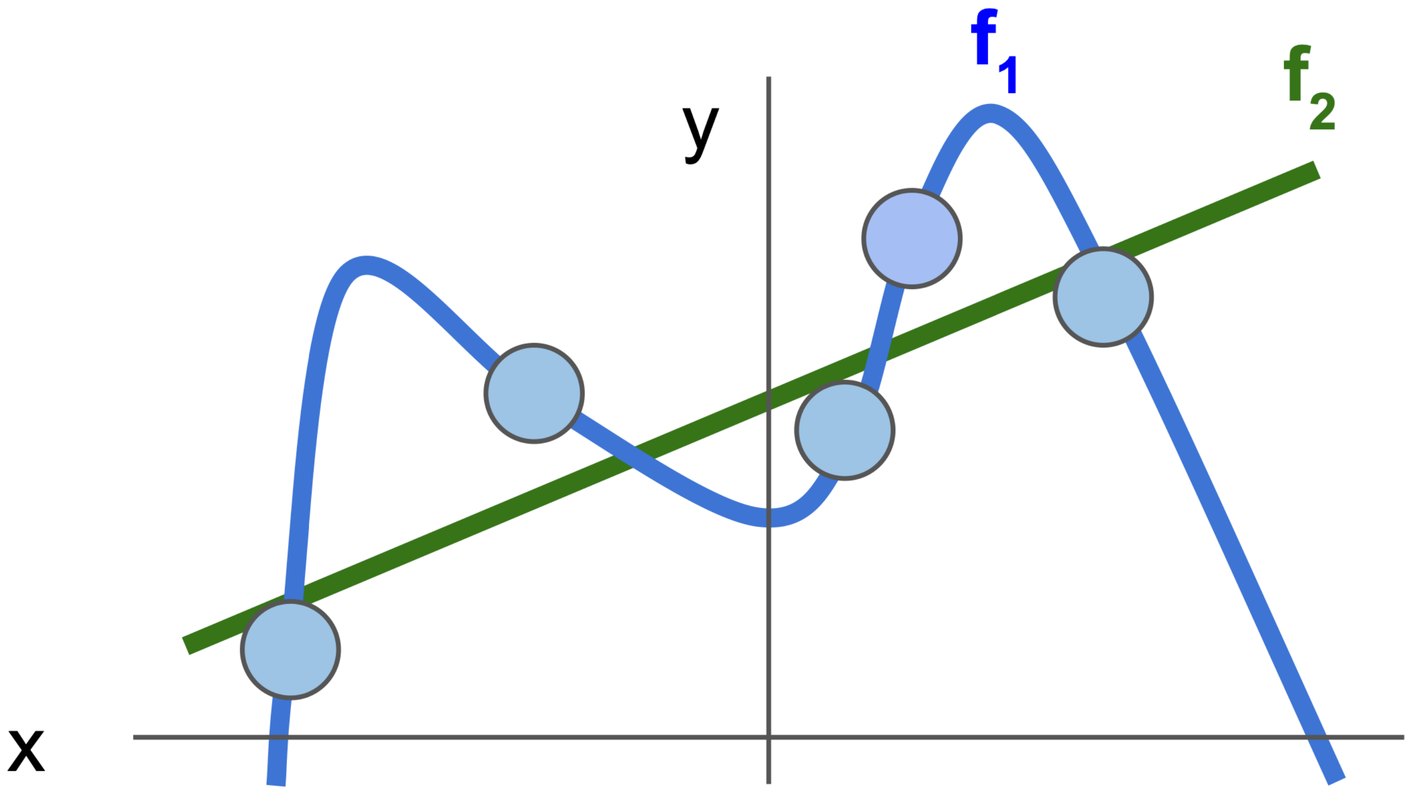
- Preventing models from being too complicated



Slides adapted from Stanford CS231N

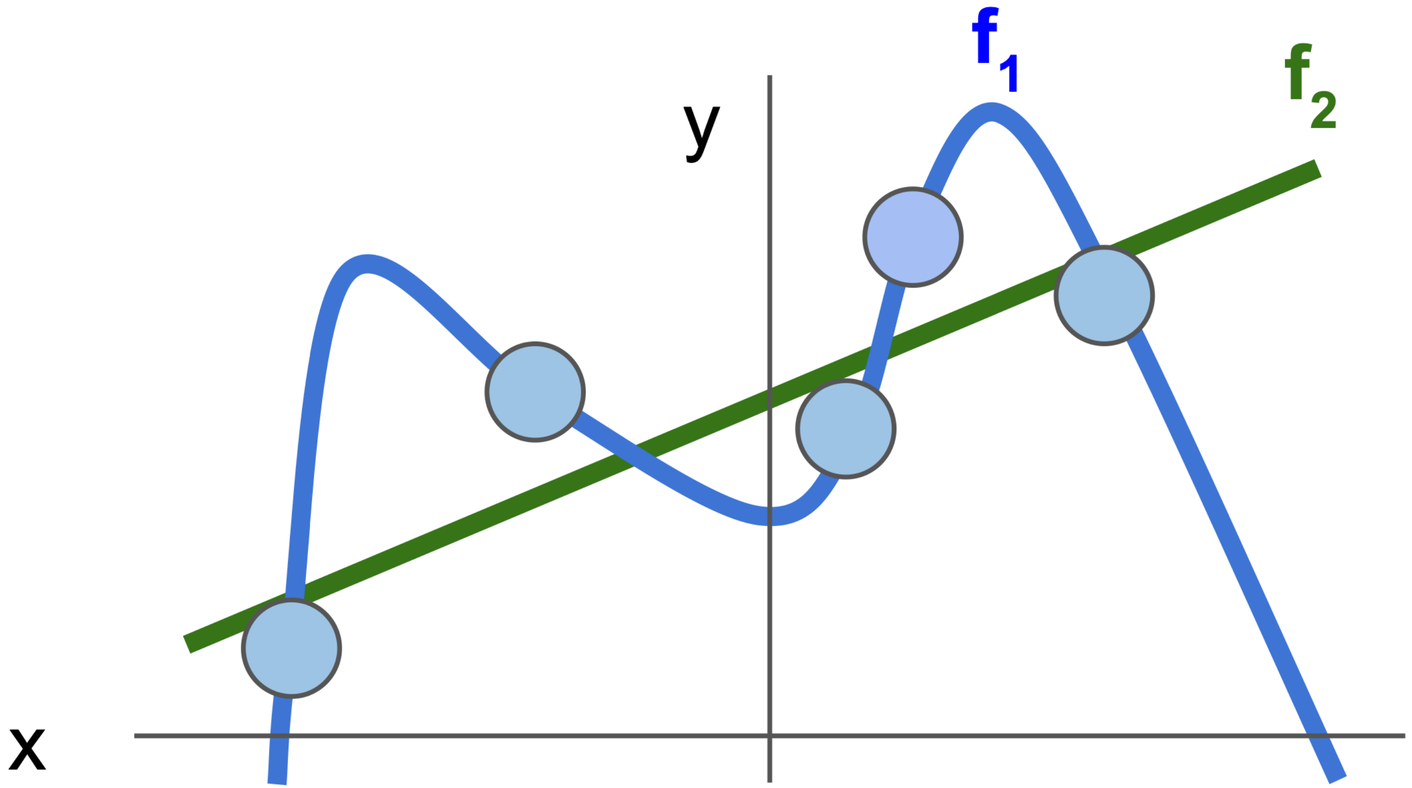
Regularization

- Preventing models from being too complicated



Slides adapted from Stanford CS231N

Regularization



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

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Slides adapted from Stanford CS231N

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 procedure[1].

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 vectors[2]. 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

$$x_j = \sum_i y_i w_{ji} \tag{1}$$

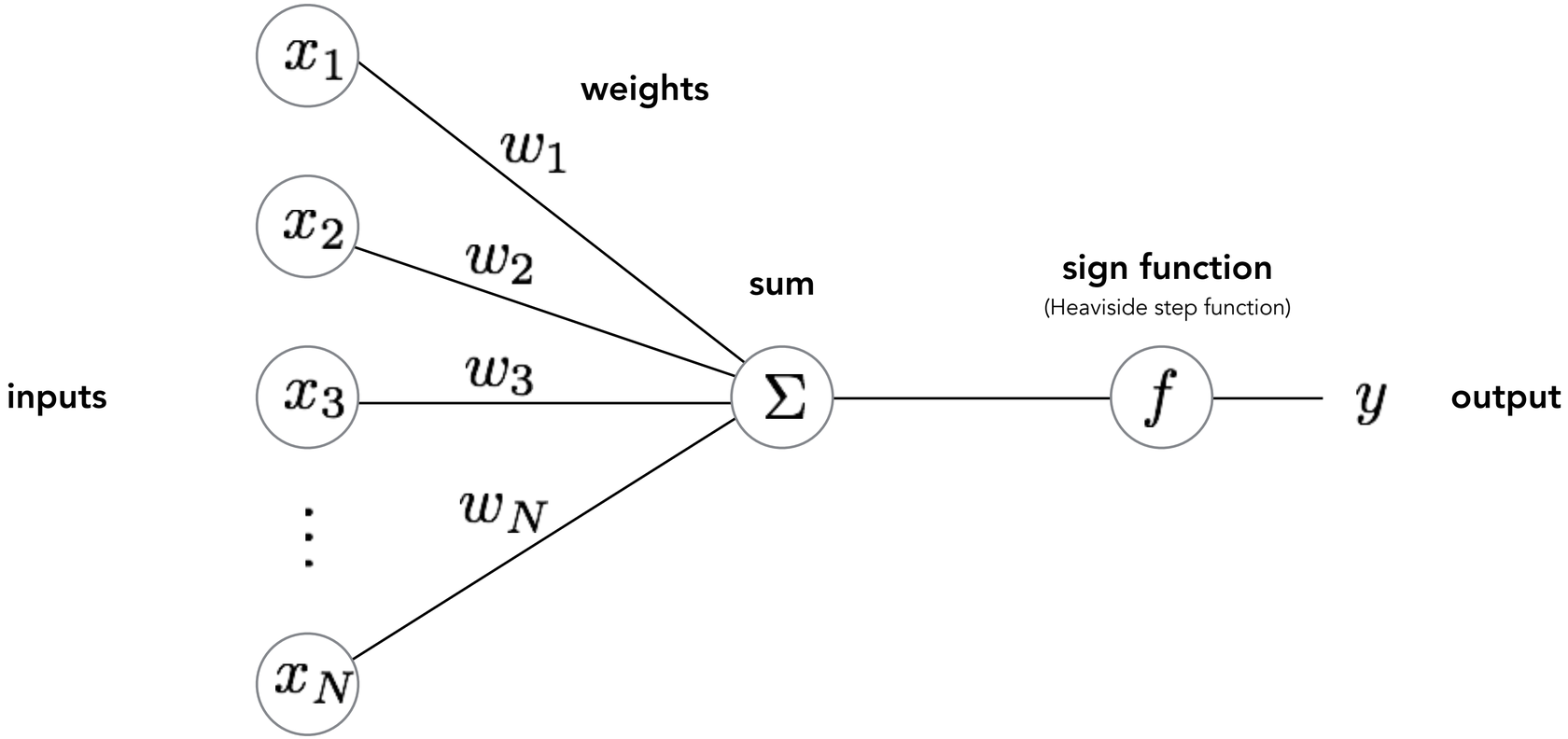
Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

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

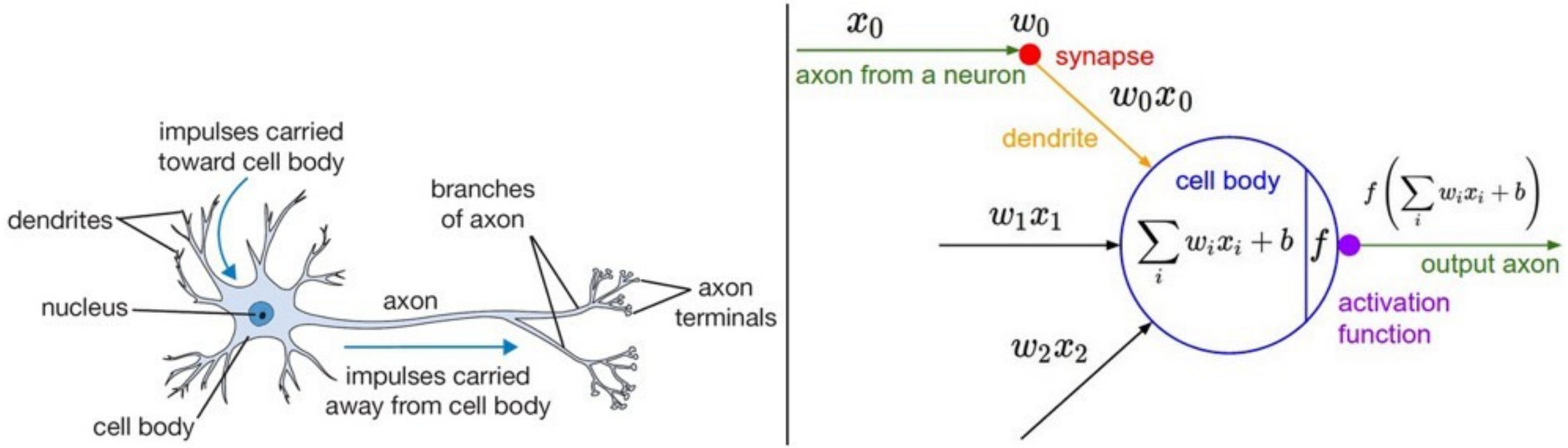
$$y_j = \frac{1}{1+e^{-x_j}} \tag{2}$$

† To whom correspondence should be addressed.

The Perceptron



Aside: Inspiration from Biology

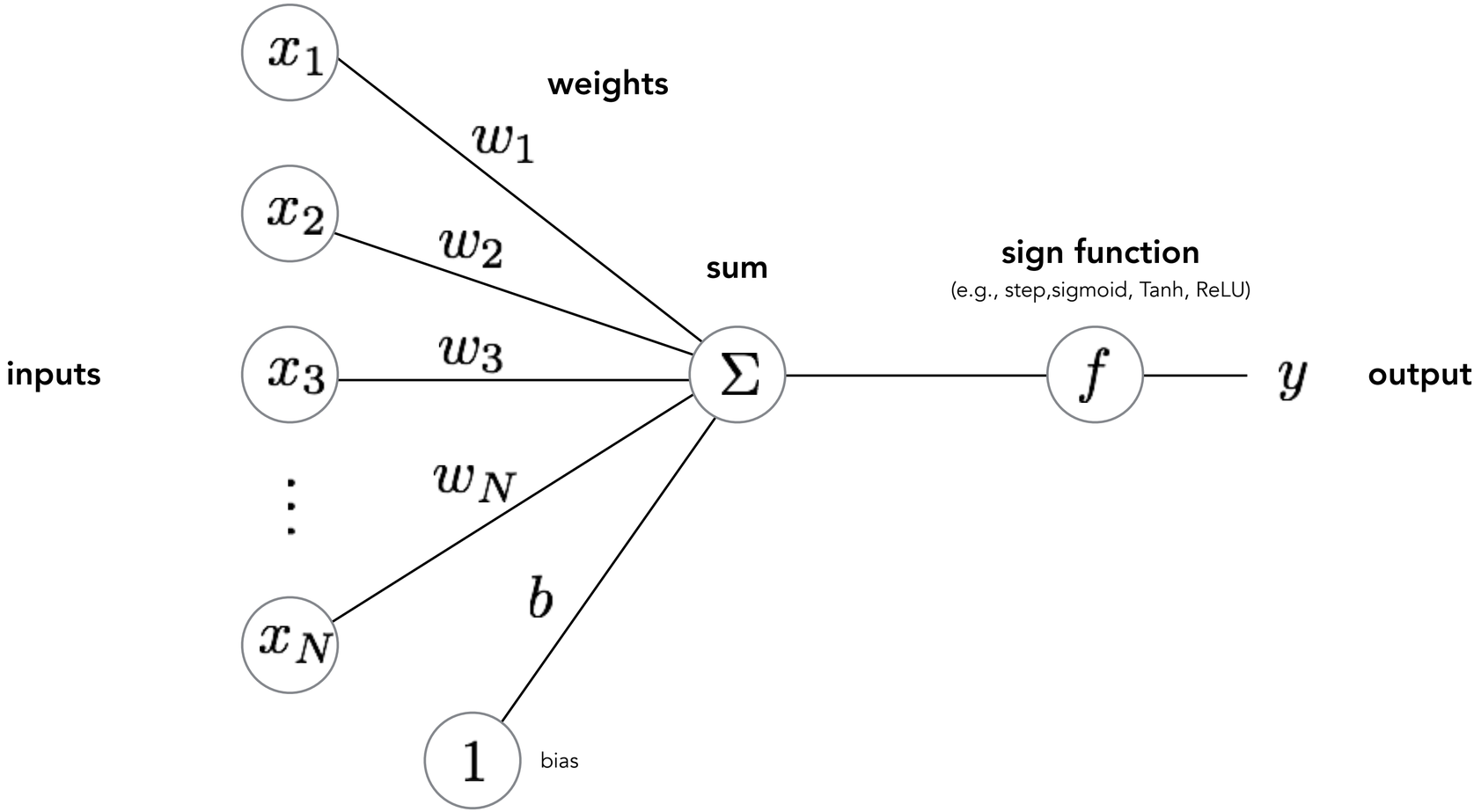


A cartoon drawing of a biological neuron (left) and its mathematical model (right).

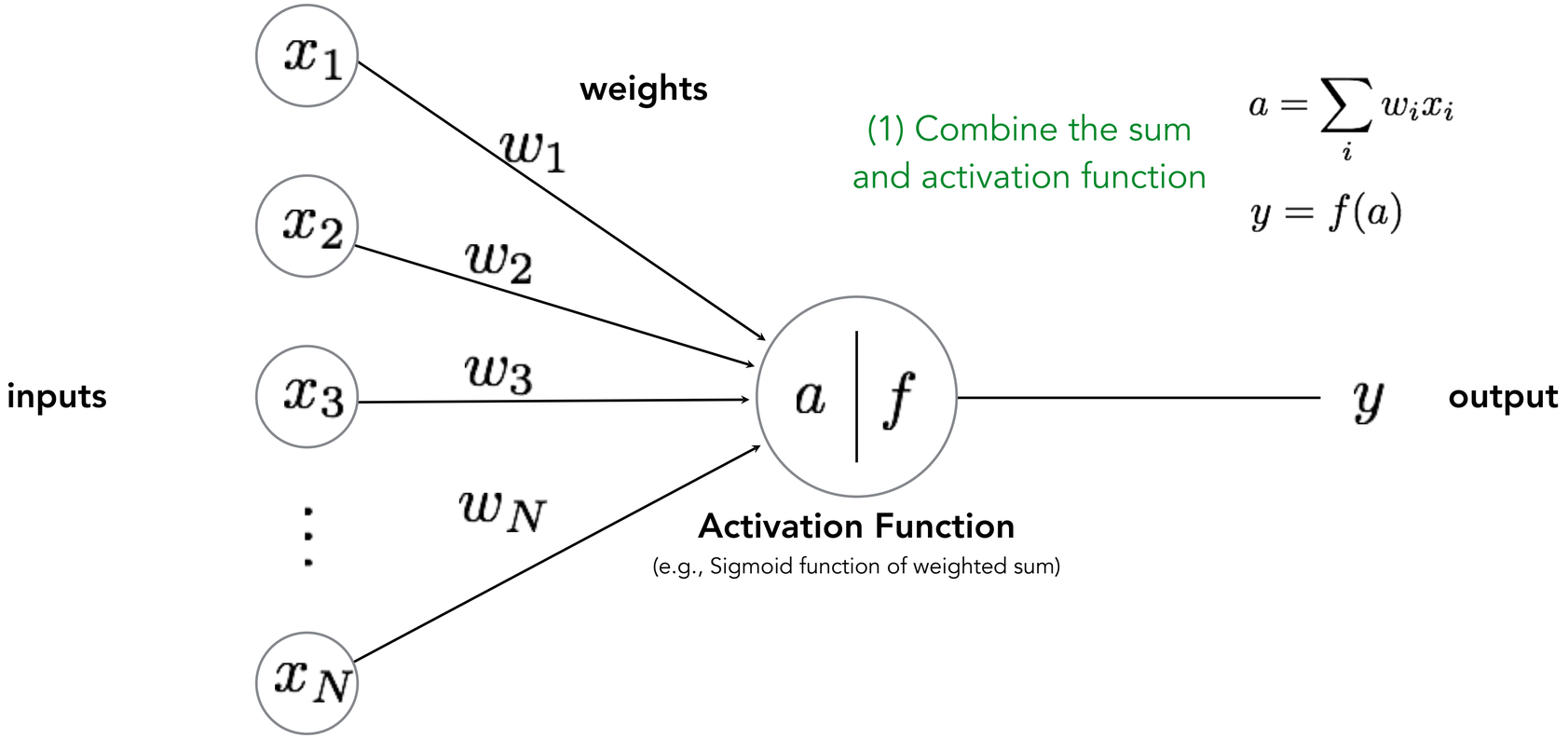
Neural nets/perceptrons are **loosely** inspired by biology.

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

The Perceptron



Another way to draw it…



(1) Combine the sum and activation function

$$a = \sum_i w_i x_i$$

$$y = f(a)$$

(2) suppress the bias term (less clutter)

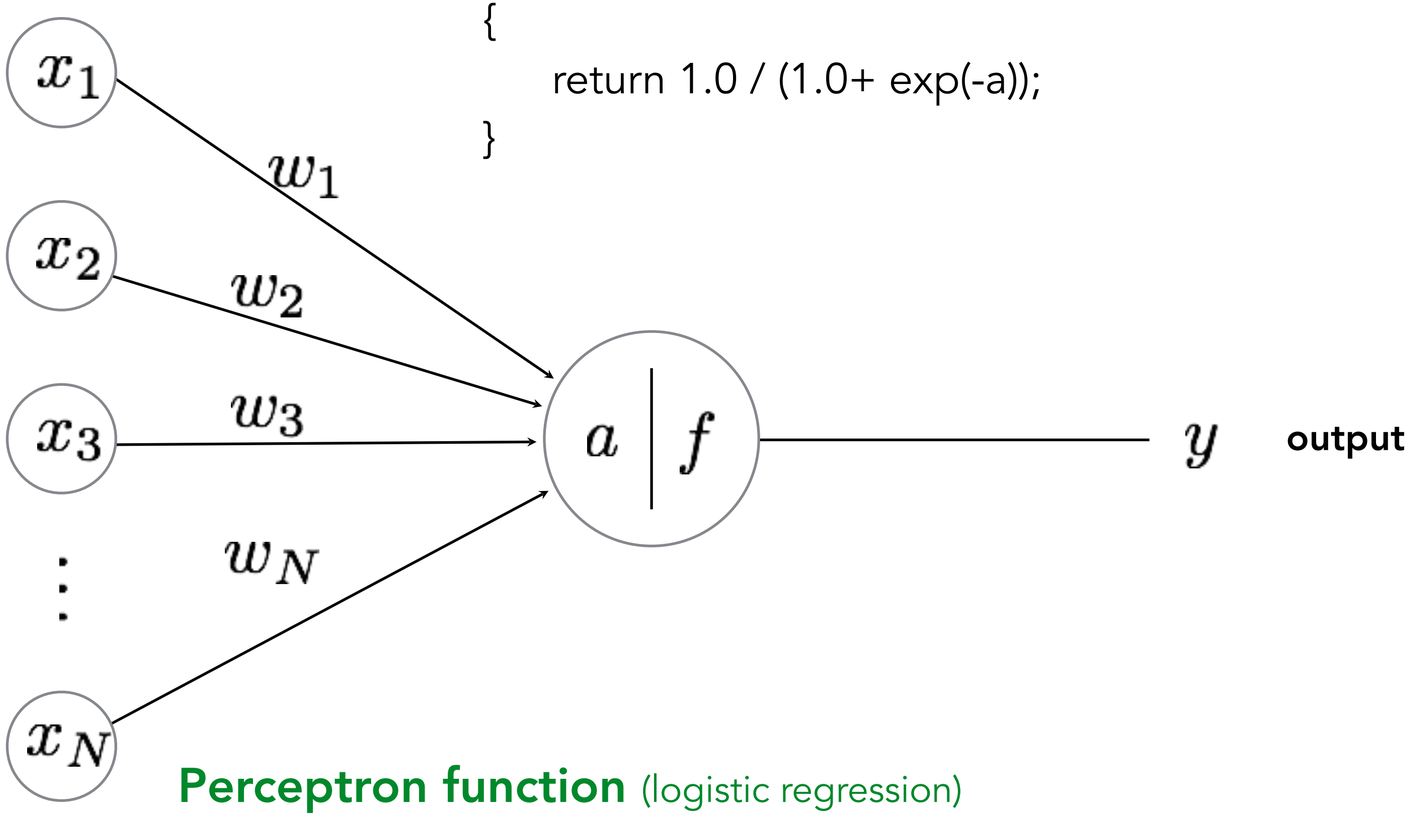
$$x_N = 1$$

$$w_N = b$$

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);
}
```

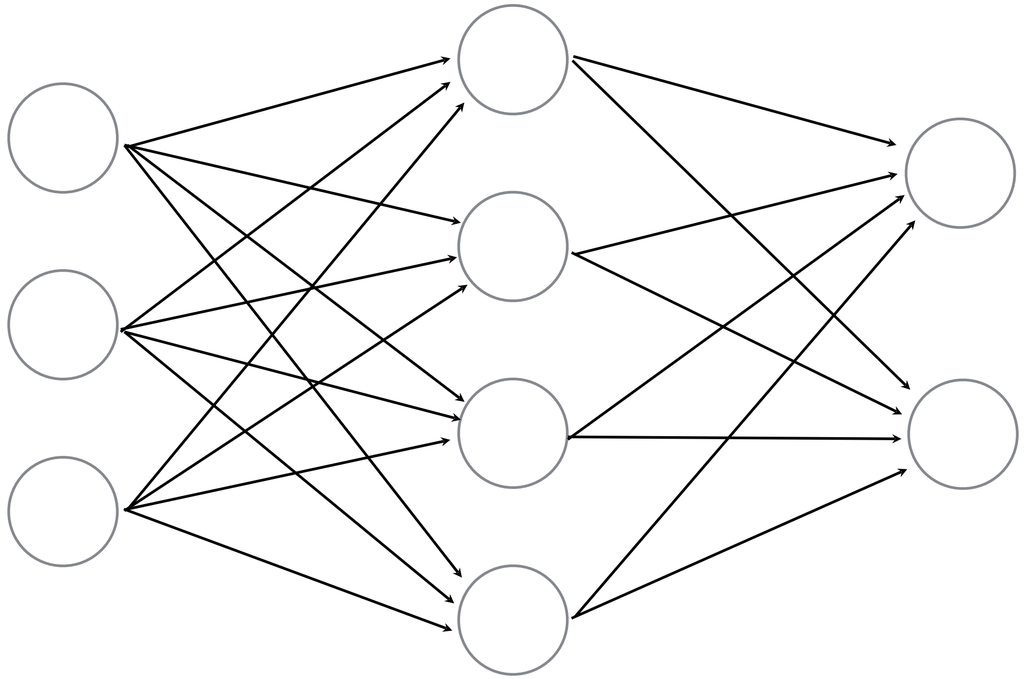
Neural networks

Connect a bunch of perceptrons together …

Connect a bunch of perceptrons together …

Neural Network

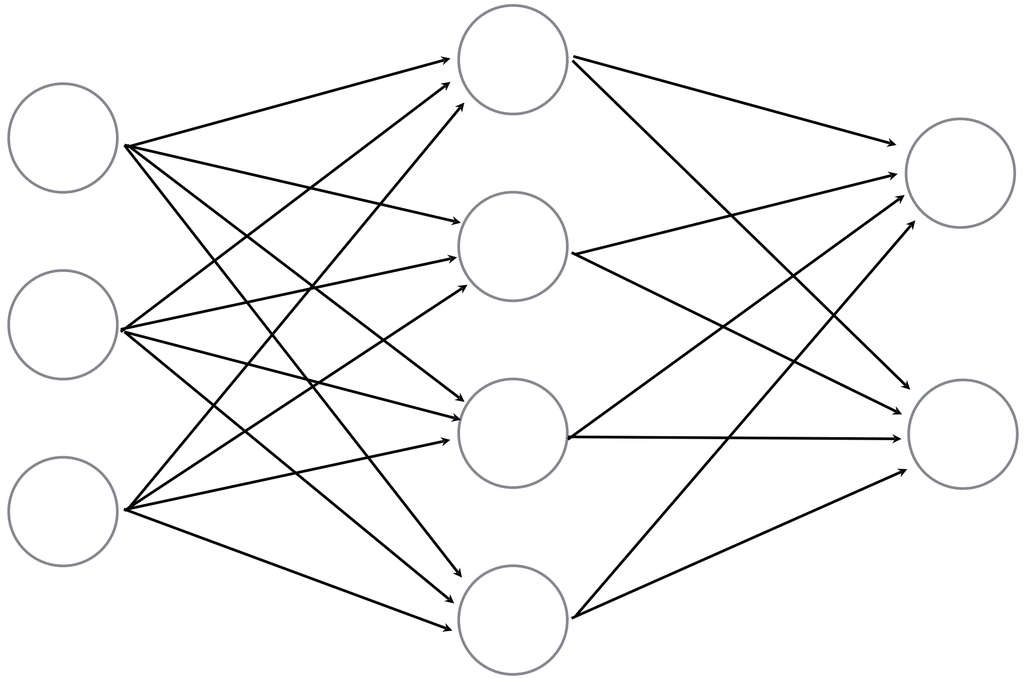
a collection of connected perceptrons



Connect a bunch of perceptrons together …

Neural Network

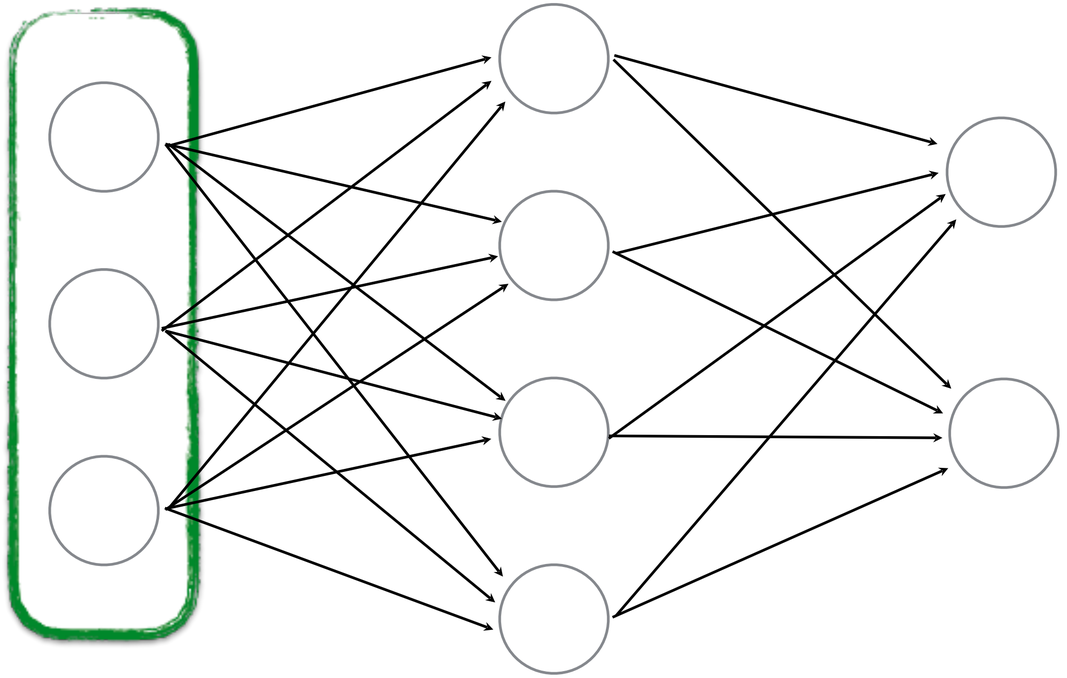
a collection of connected perceptrons



How many perceptrons in this neural network?

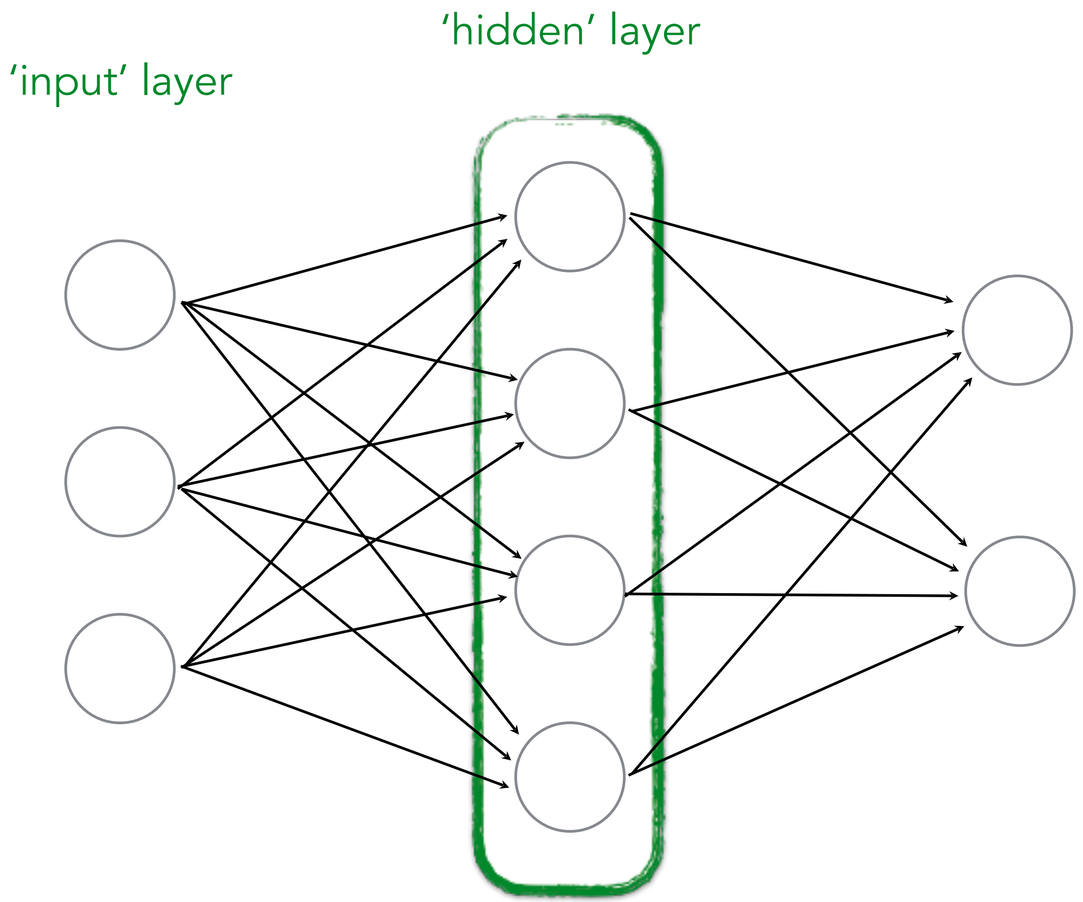
Some terminology…

‘input’ layer



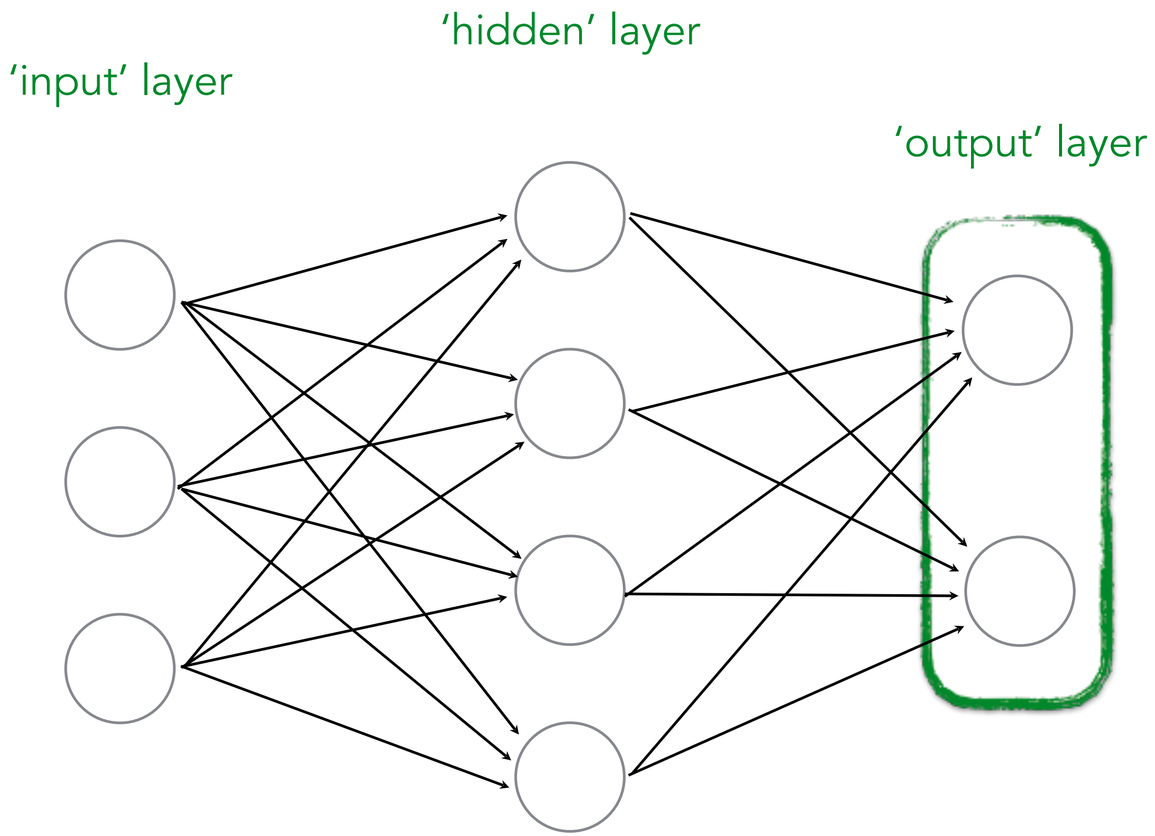
…also called a **Multi-layer Perceptron** (MLP)

Some terminology…



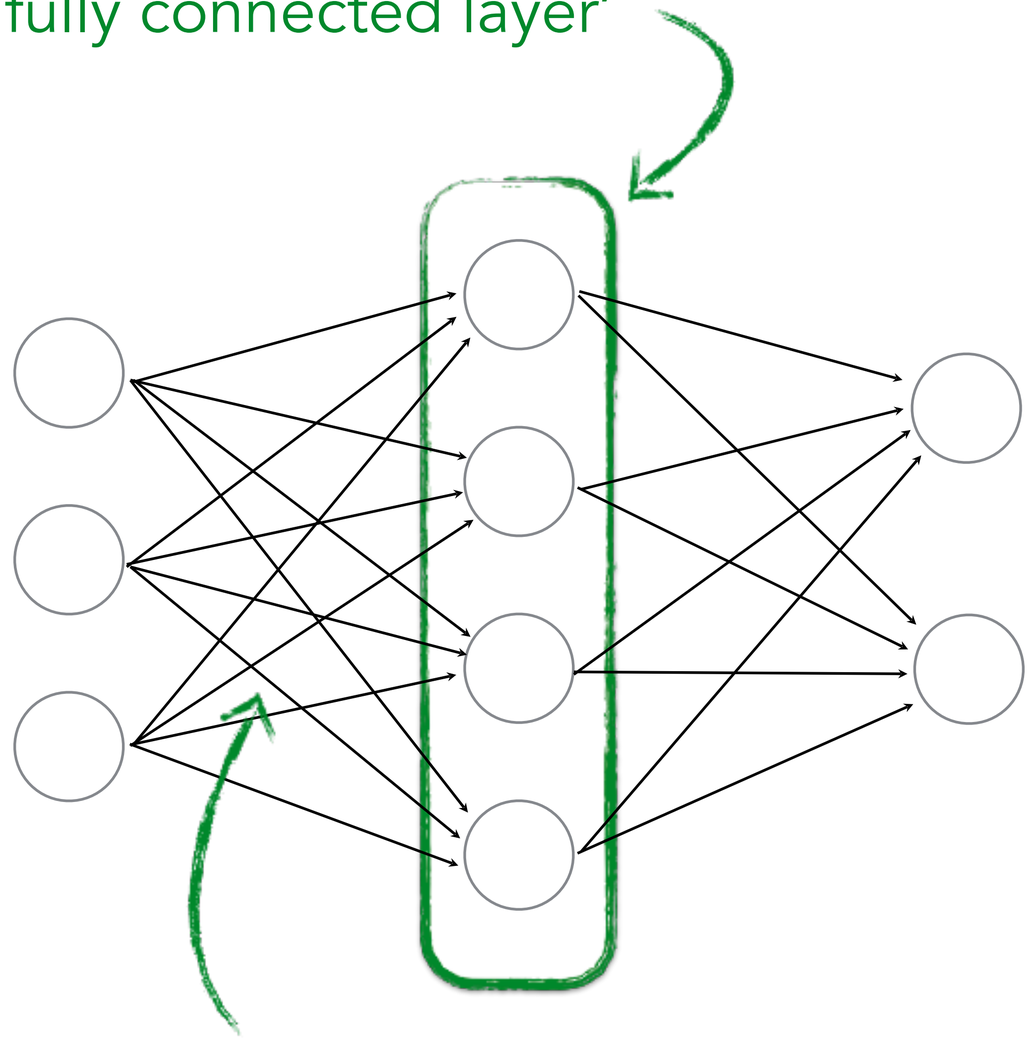
…also called a **Multi-layer Perceptron** (MLP)

Some terminology…



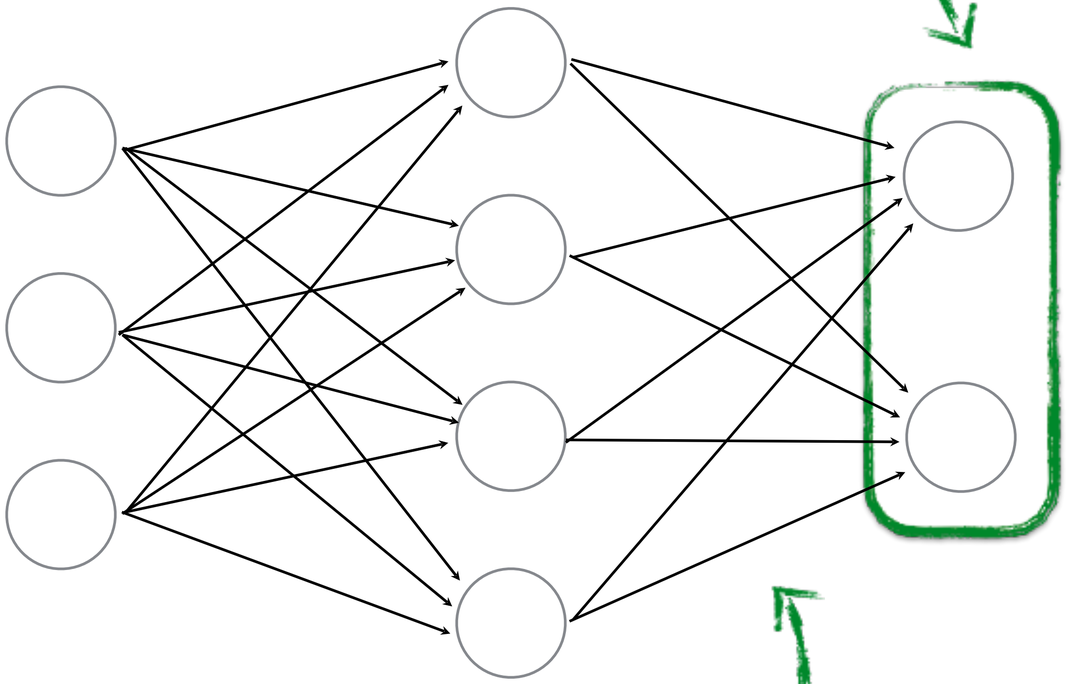
…also called a **Multi-layer Perceptron** (MLP)

this layer is a
'fully connected layer'



all pairwise neurons between layers are connected

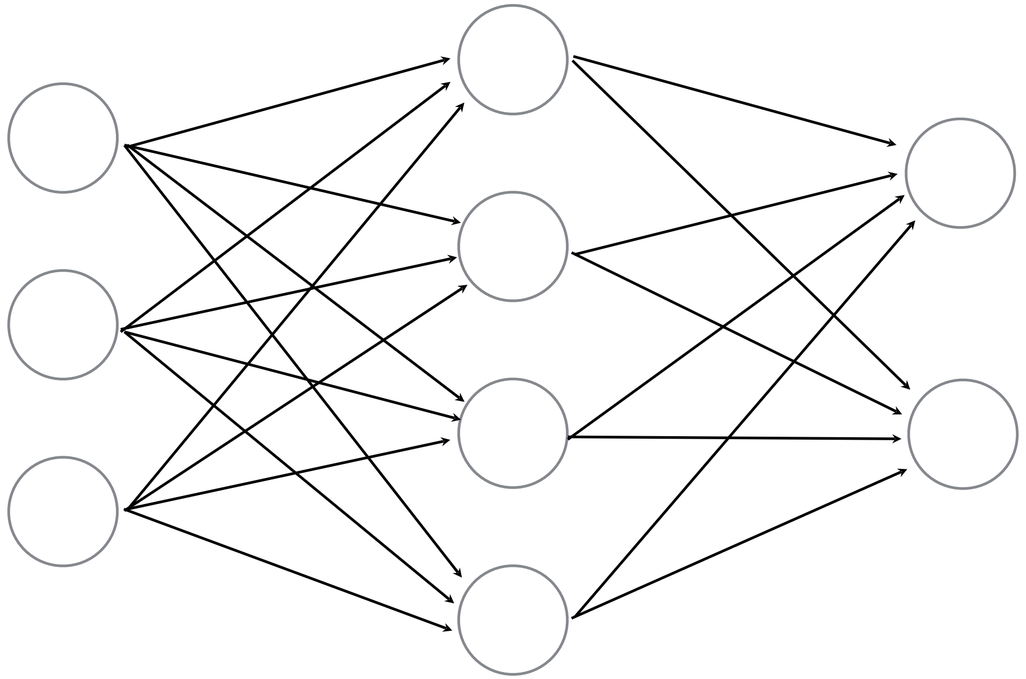
so is this



all pairwise neurons between layers are connected

How many neurons (perceptrons)?

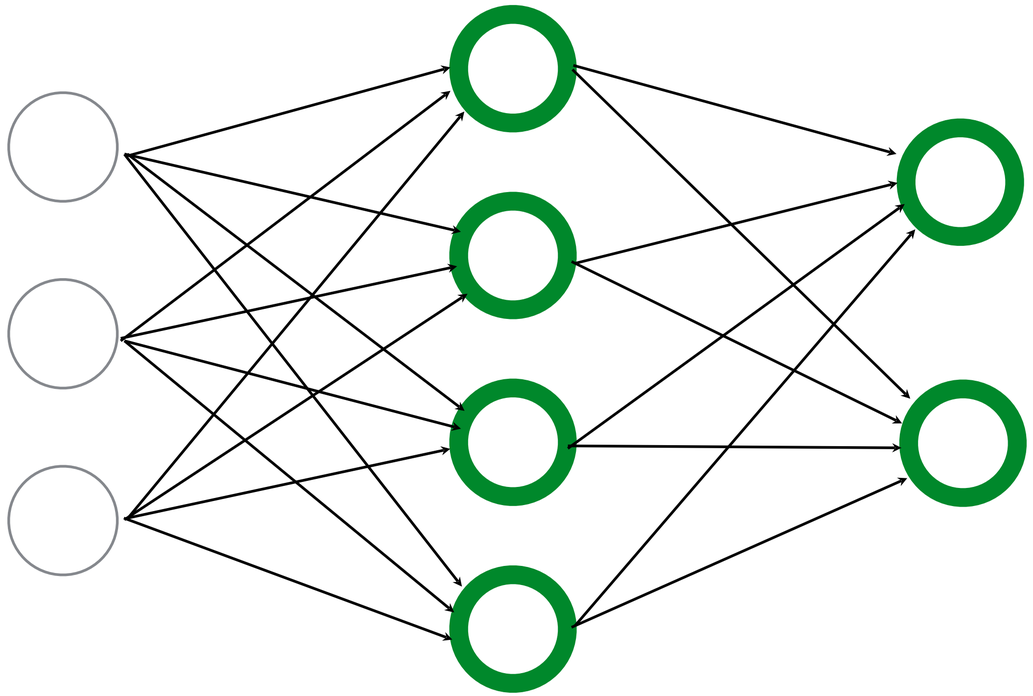
How many weights (edges)?



How many learnable parameters total?

How many neurons (perceptrons)? 4 + 2 = 6

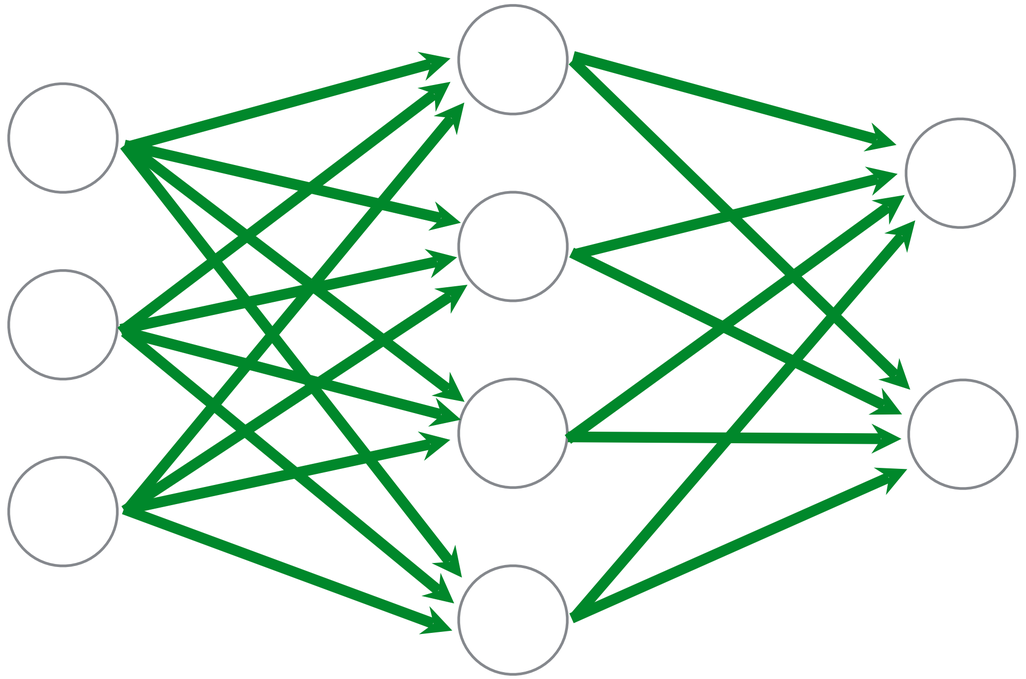
How many weights (edges)?



How many learnable parameters total?

How many neurons (perceptrons)? 4 + 2 = 6

How many weights (edges)? (3 x 4) + (4 x 2) = 20



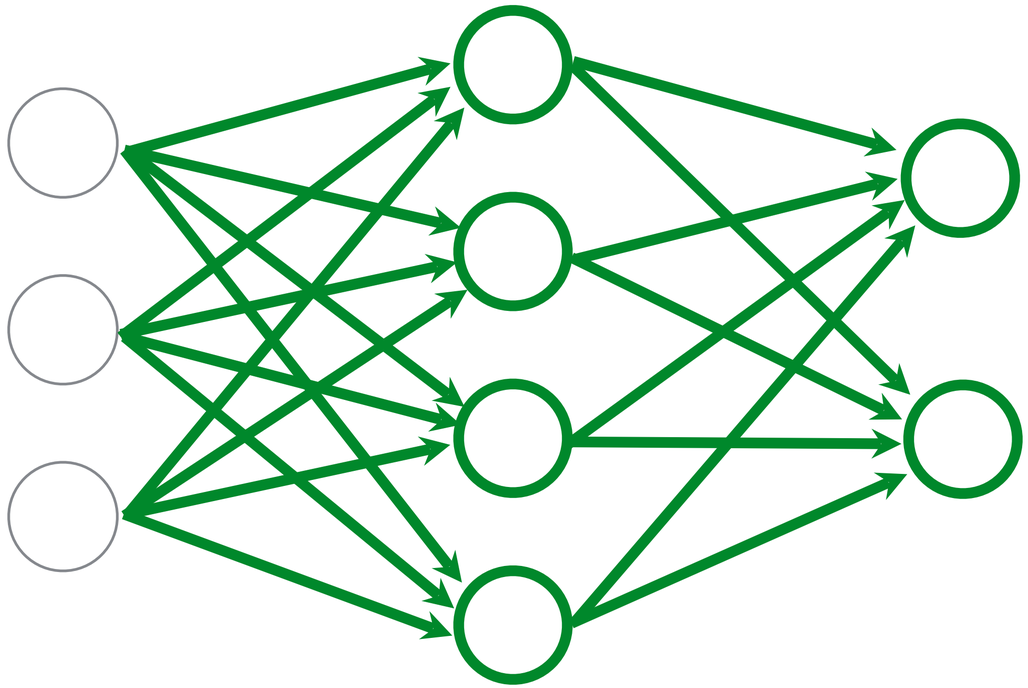
How many learnable parameters total?

How many neurons (perceptrons)?

4 + 2 = 6

How many weights (edges)?

(3 x 4) + (4 x 2) = 20

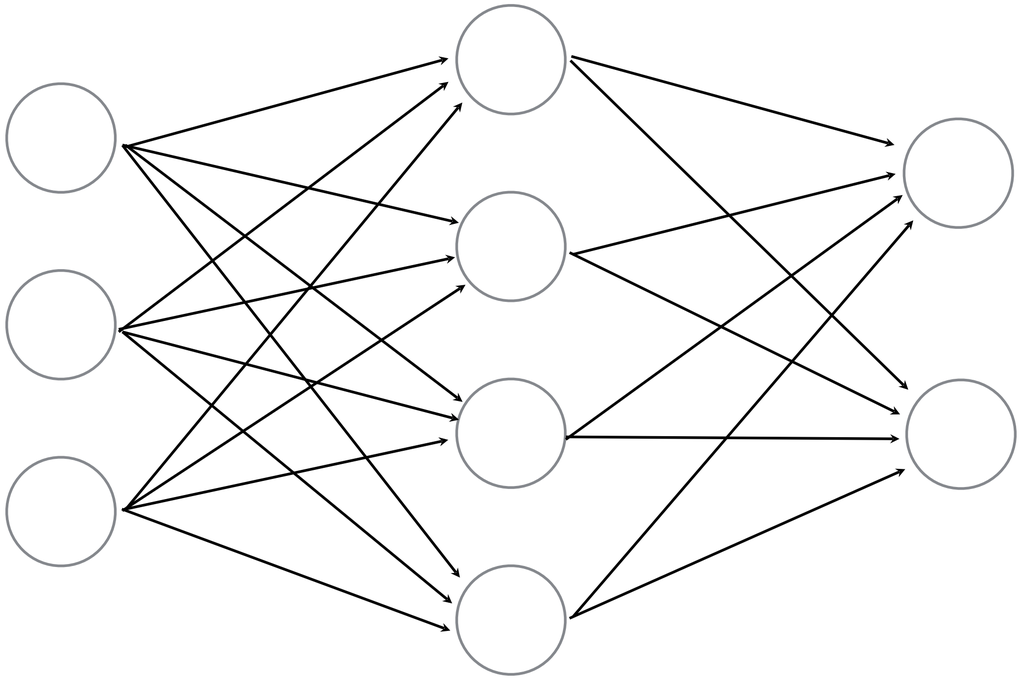


How many learnable parameters total?

20 + 4 + 2 = 26

bias terms

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



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

Before diving into gradient descent, we need to understand …

Loss Function

defines what is means to be

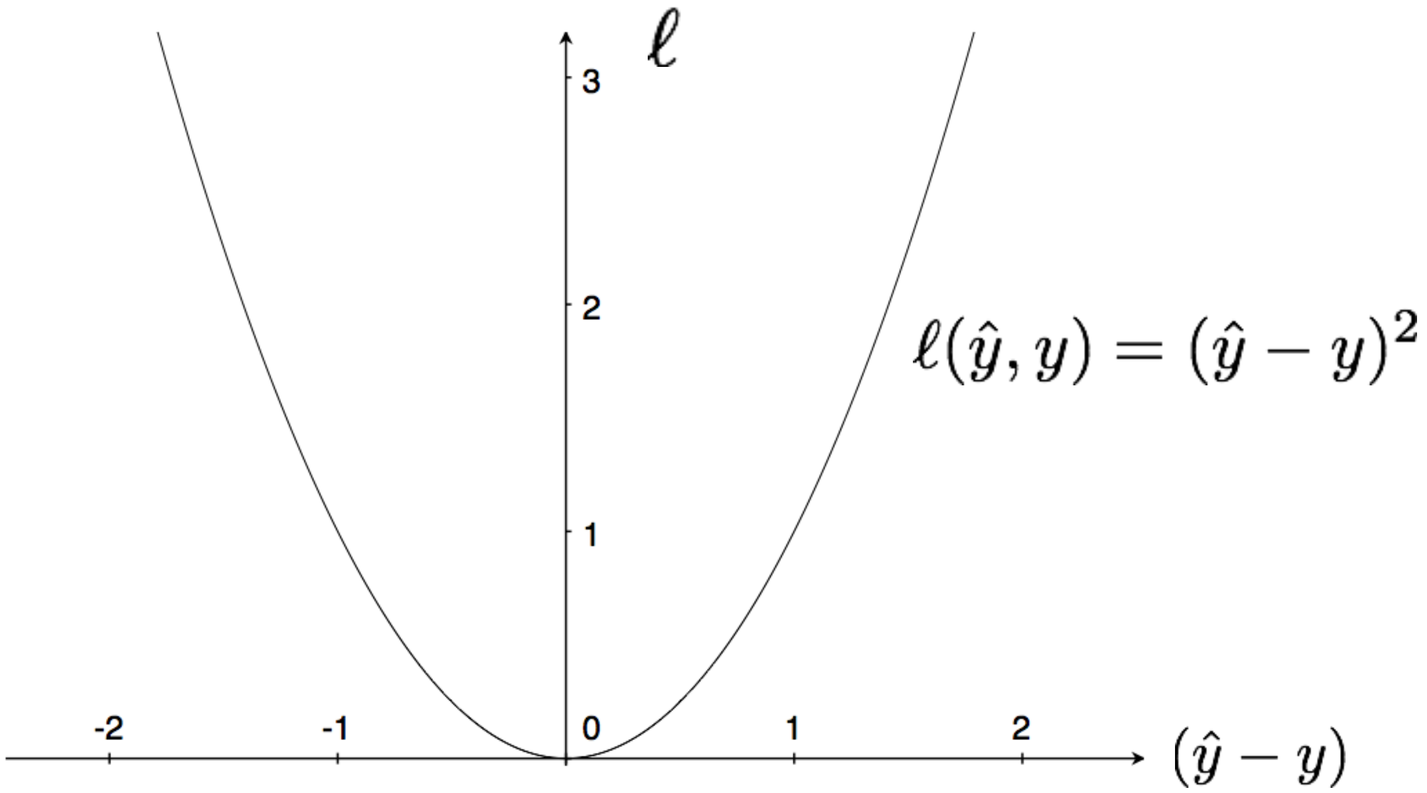
close to the true solution

YOU get to chose the loss function!

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

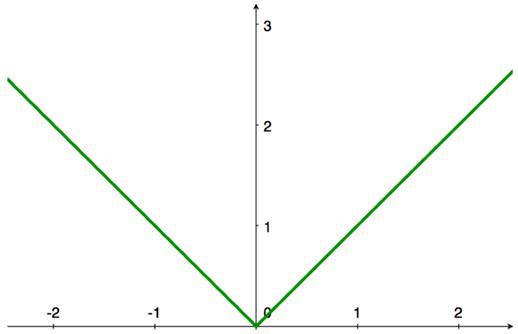
Squared Error (L2)

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



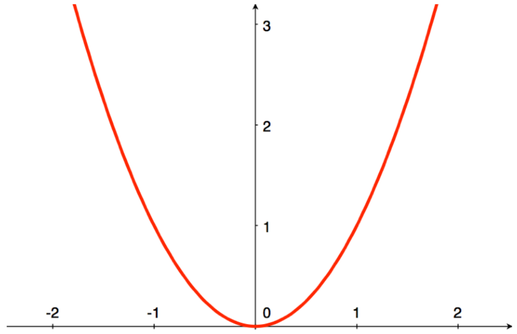
L1 Loss

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



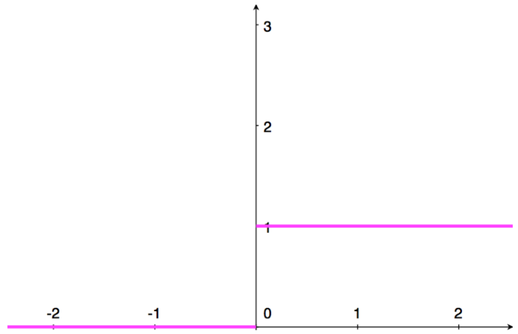
L2 Loss

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



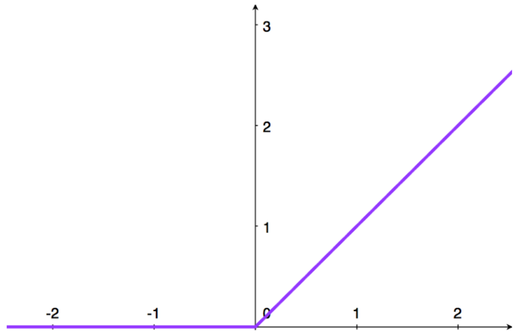
Zero-One Loss

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



Hinge Loss

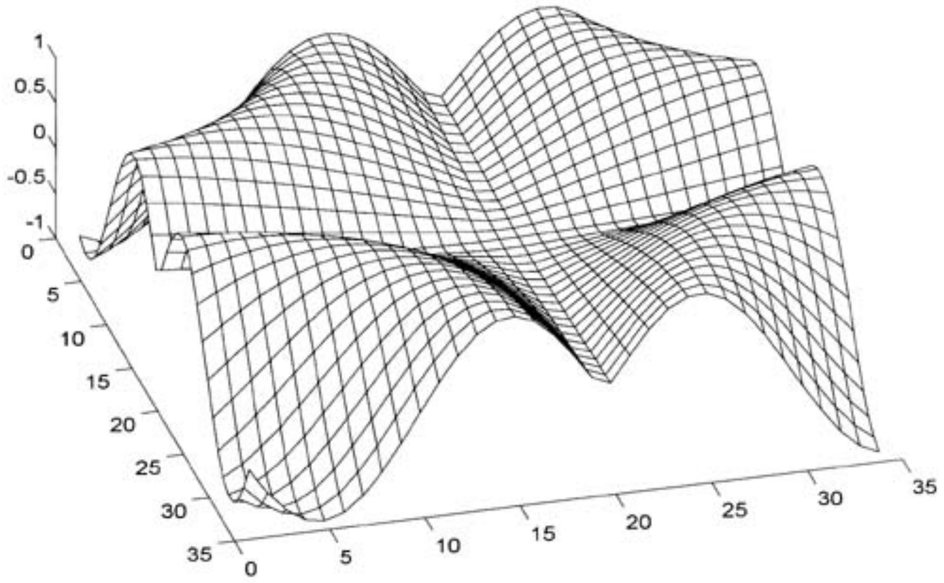
$$\ell(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$$



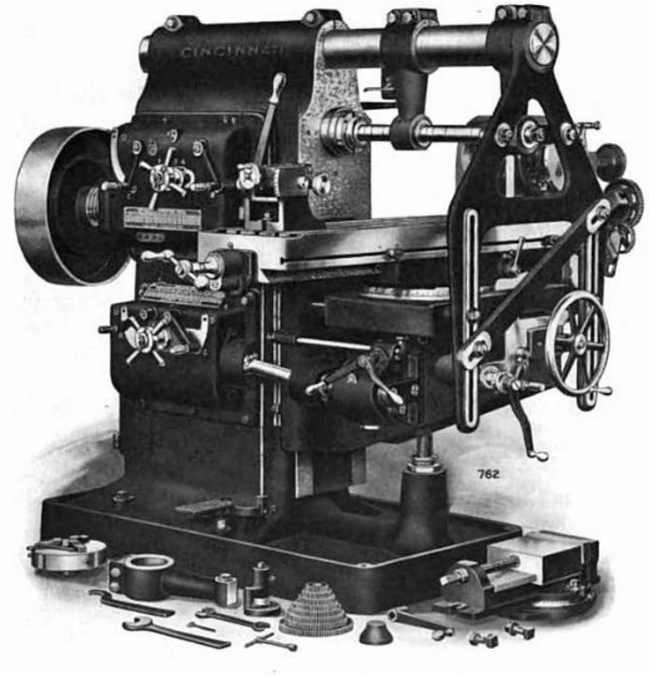
Gradient descent

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

Two ways to think about them:

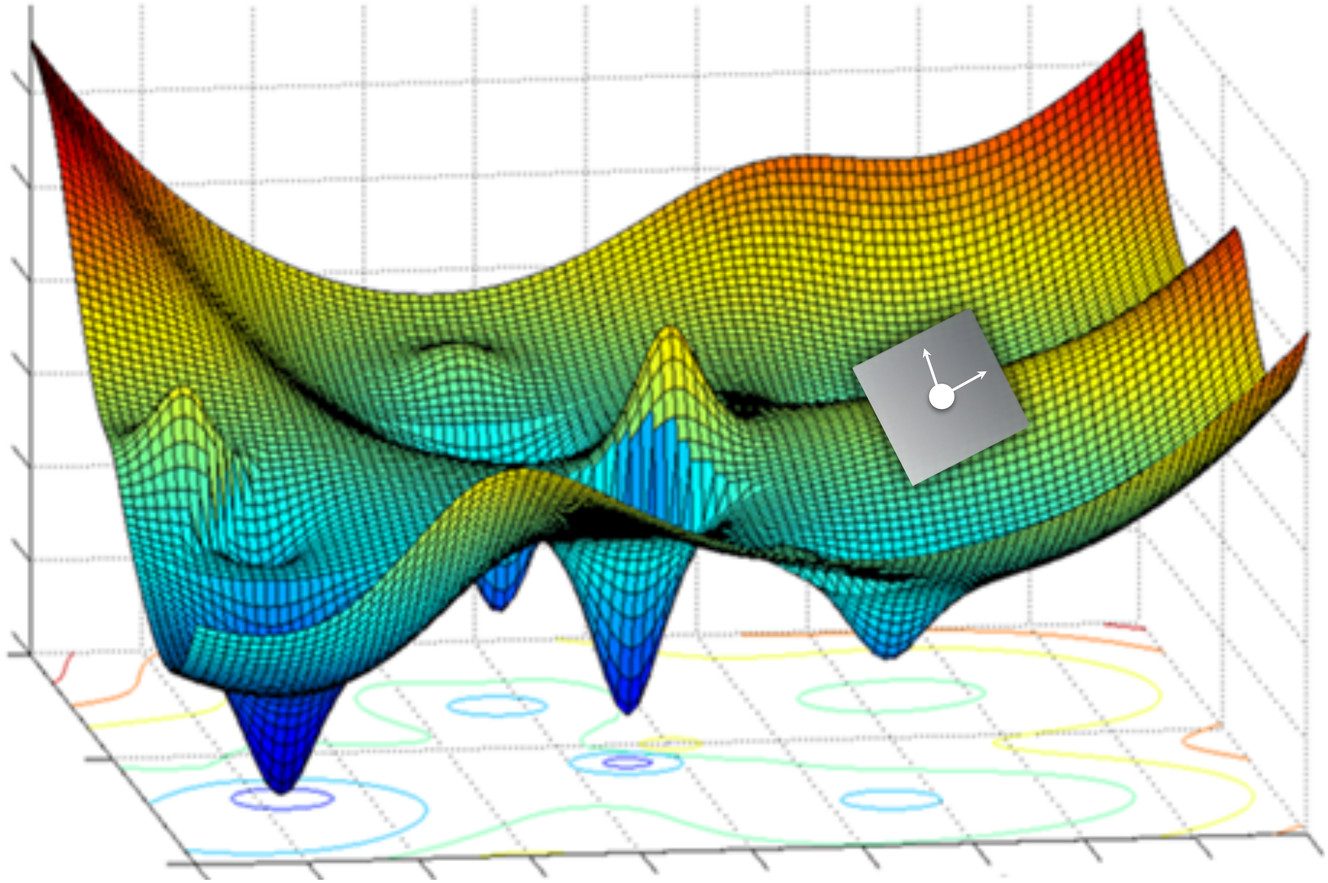


Slope of a function



Knobs on a machine

1. Slope of a function:



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

describes the slope around a point

2. Knobs on a machine:

input $\boldsymbol{x}$

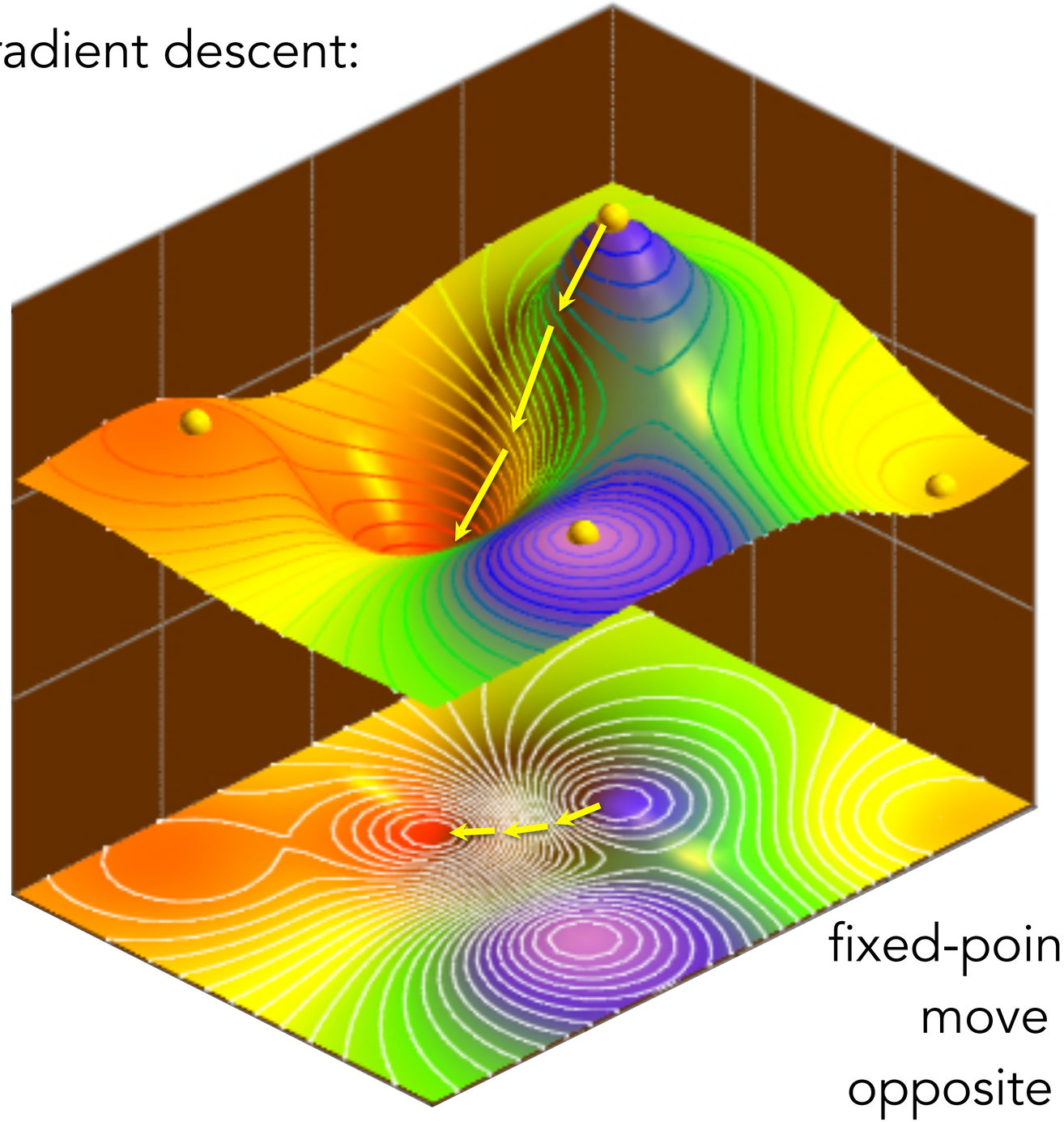
output $f(\boldsymbol{x}; \boldsymbol{w})$

describes how each ‘knob’ affects the output

$$\frac{\partial f(x)}{\partial w_1} \quad \frac{\partial f(x)}{\partial w_2} \quad \frac{\partial f(x)}{\partial w_3}$$

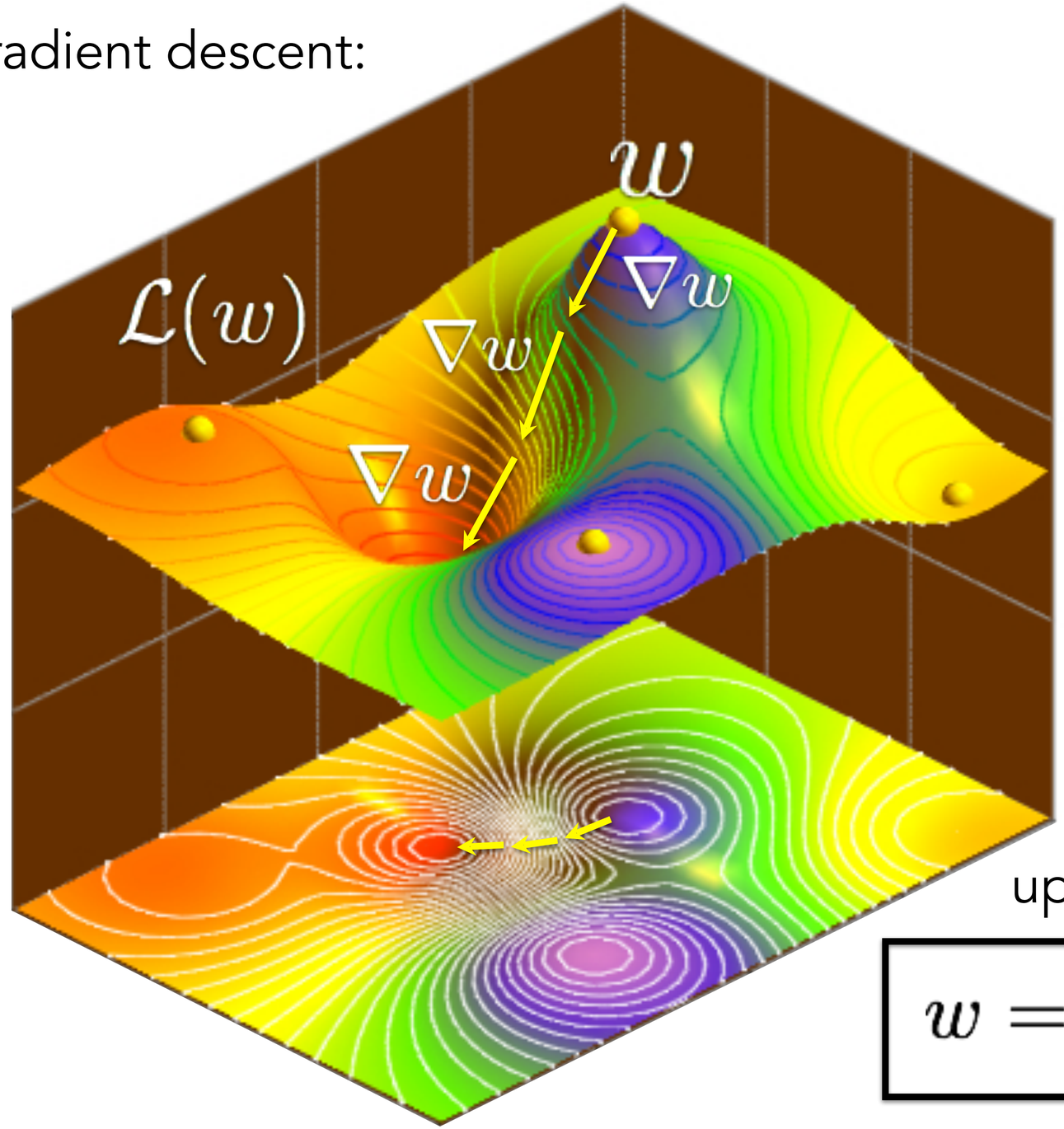
small change in parameter Δw_1 ➡ output will change by $\frac{\partial f(x)}{\partial w_1} \Delta w_1$

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!

Compute the derivative

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

$$= -(y-\hat{y})\frac{dwx}{dw}$$

$$= -(y-\hat{y})x = \nabla w$$ just shorthand

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

$$w = w - \nabla w$$ move in direction of negative gradient

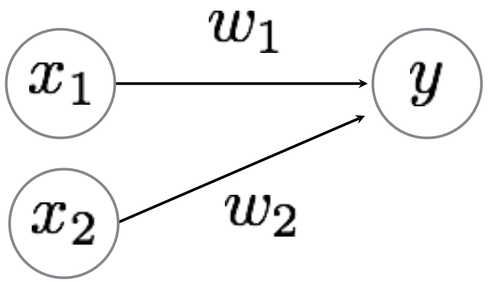
$$= w + (y-\hat{y})x$$

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 $w = w - \nabla w$

world's (second) smallest **perceptron**!



function of **two** parameters!

Derivative computation

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial}{\partial w_1}\left\{\frac{1}{2}(y-\hat{y})^2\right\}$$

$$= -(y-\hat{y})\frac{\partial \hat{y}}{\partial w_1}$$

$$= -(y-\hat{y})\frac{\partial \sum_i w_i x_i}{\partial w_1}$$

$$= -(y-\hat{y})\frac{\partial w_1 x_1}{\partial w_1}$$

$$= -(y-\hat{y})x_1 = \nabla w_1$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial}{\partial w_2}\left\{\frac{1}{2}(y-\hat{y})^2\right\}$$

$$= -(y-\hat{y})\frac{\partial \hat{y}}{\partial w_2}$$

$$= -(y-\hat{y})\frac{\partial \sum_i w_i x_i}{\partial w_1}$$

$$= -(y-\hat{y})\frac{\partial w_2 x_2}{\partial w_2}$$

$$= -(y-\hat{y})x_2 = \nabla w_2$$

Why do we have partial derivatives now?

Derivative computation

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial}{\partial w_1}\left\{\frac{1}{2}(y-\hat{y})^2\right\}$$

$$= -(y-\hat{y})\frac{\partial \hat{y}}{\partial w_1}$$

$$= -(y-\hat{y})\frac{\partial \sum_i w_i x_i}{\partial w_1}$$

$$= -(y-\hat{y})\frac{\partial w_1 x_1}{\partial w_1}$$

$$= -(y-\hat{y})x_1 = \nabla w_1$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial}{\partial w_2}\left\{\frac{1}{2}(y-\hat{y})^2\right\}$$

$$= -(y-\hat{y})\frac{\partial \hat{y}}{\partial w_2}$$

$$= -(y-\hat{y})\frac{\partial \sum_i w_i x_i}{\partial w_1}$$

$$= -(y-\hat{y})\frac{\partial w_2 x_2}{\partial w_2}$$

$$= -(y-\hat{y})x_2 = \nabla w_2$$

Gradient Update

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

$$= w_1 + \eta(y-\hat{y})x_1$$

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

$$= w_2 + \eta(y-\hat{y})x_2$$

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

two lines now

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

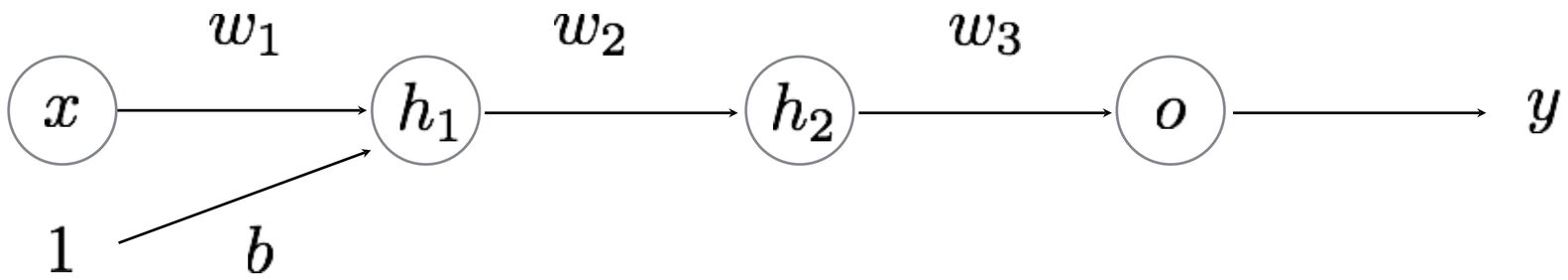
    b. Gradient update

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

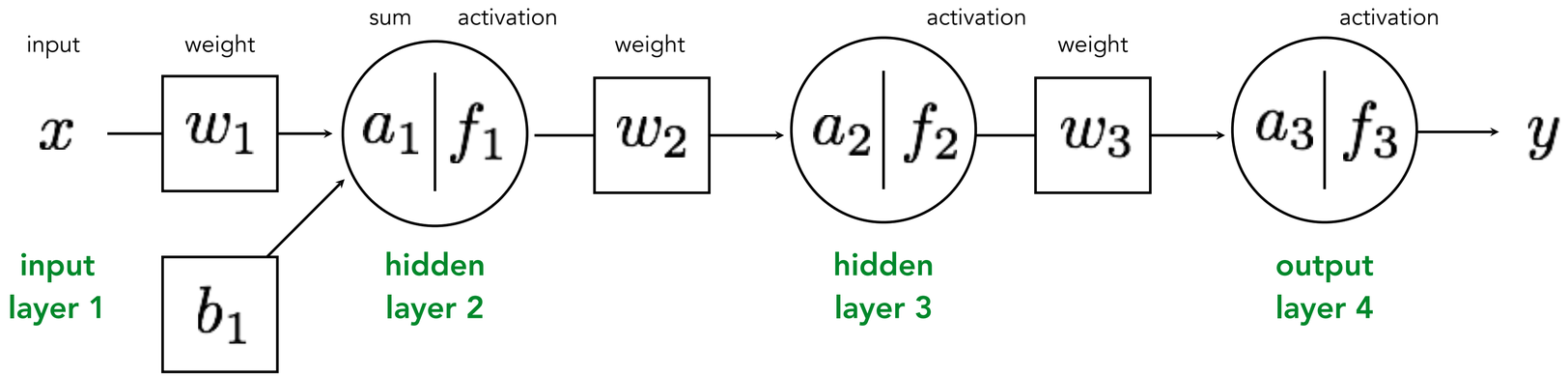
(adjustable step size)

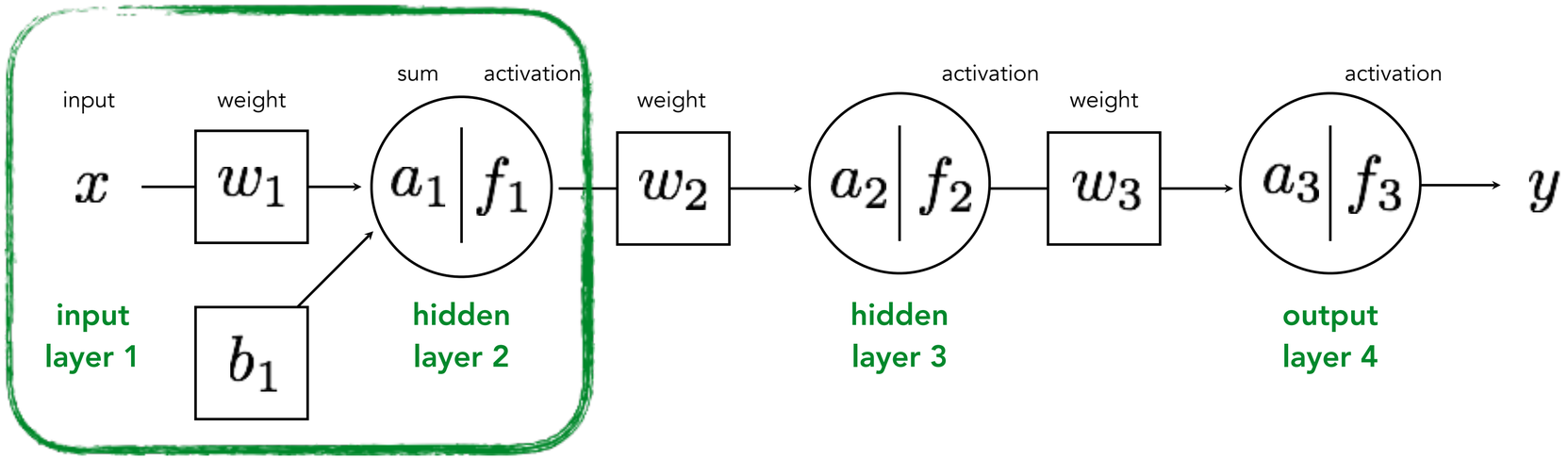
We haven’t seen a lot of ‘propagation’ yet
because our perceptrons only had one layer…

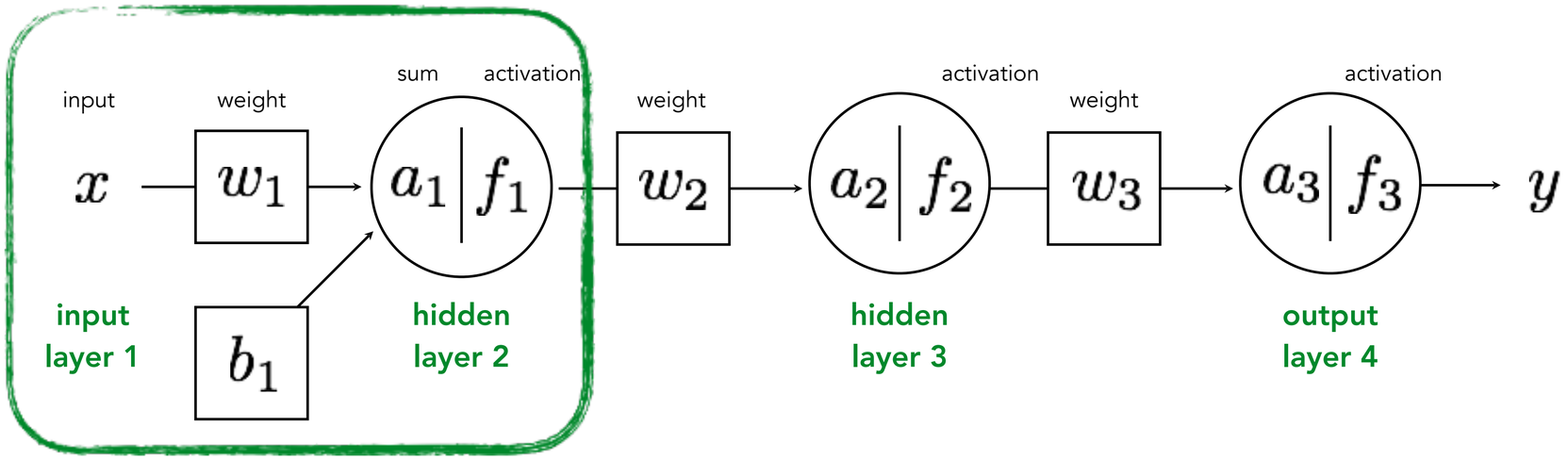
multi-layer perceptron



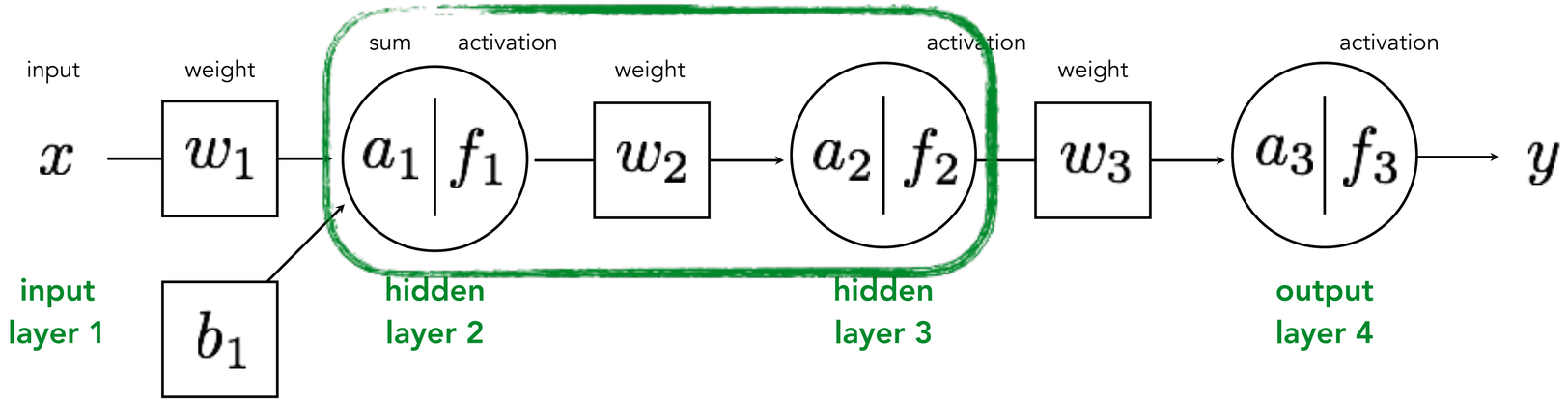
function of **FOUR** parameters and **FOUR** layers!


input
weight
sum
activation
weight
activation
weight
activation
x
w_1
a_1 f_1
w_2
a_2 f_2
w_3
a_3 f_3
y
b_1
input layer 1
hidden layer 2
hidden layer 3
output layer 4

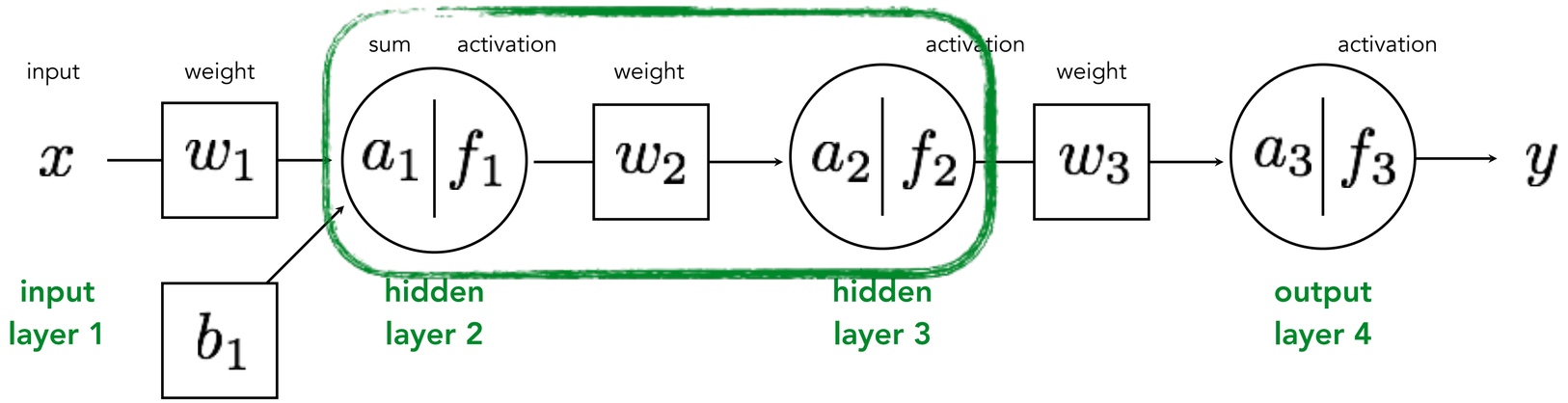

input
weight
sum
activation
weight
activation
weight
activation
x
w_1
a_1 f_1
w_2
a_2 f_2
w_3
a_3 f_3
y
b_1
input layer 1
hidden layer 2
hidden layer 3
output layer 4



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

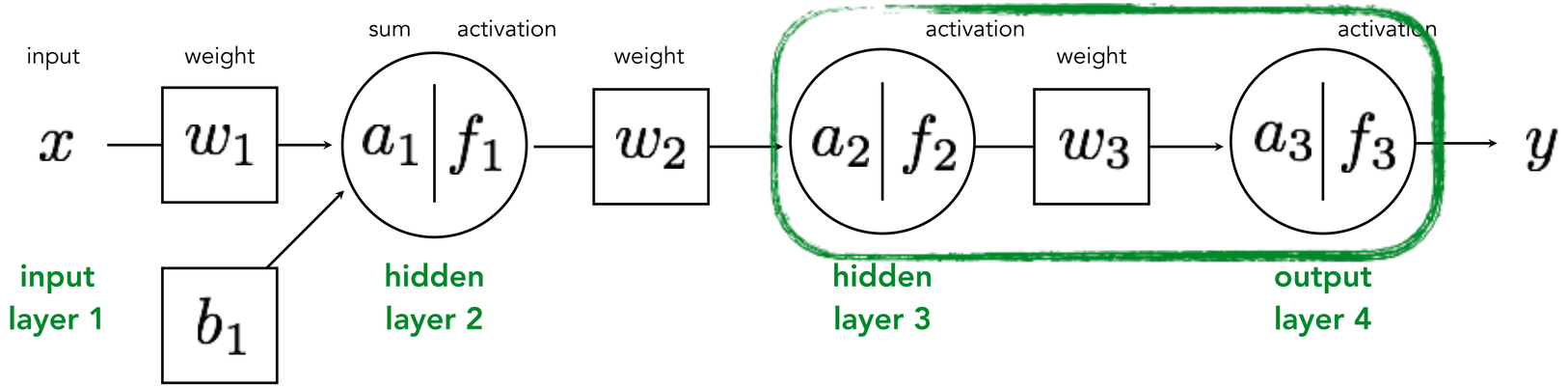


$$a_1 = w_1 \cdot x + b_1$$



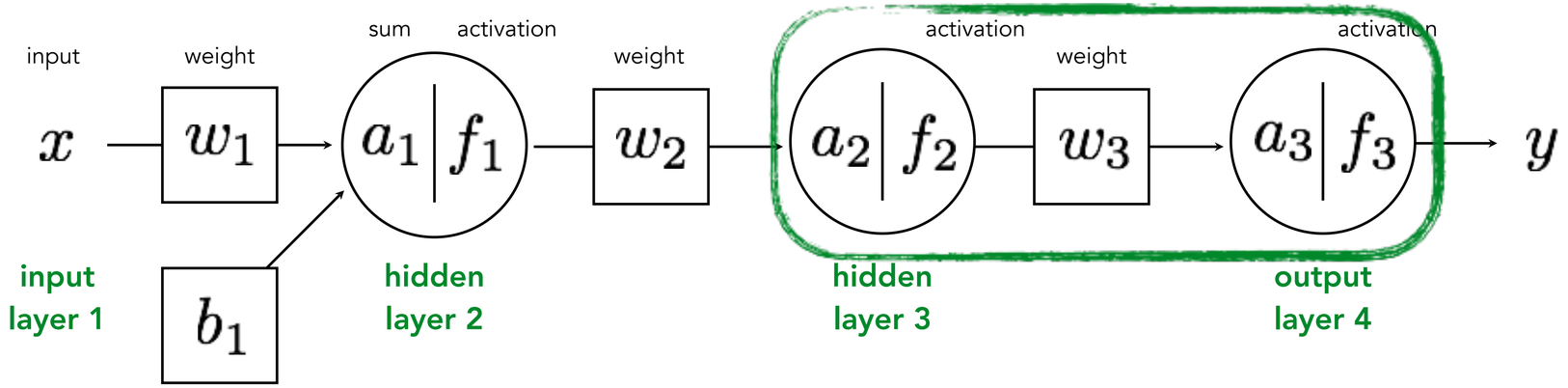
$$a_1 = w_1 \cdot x + b_1$$

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



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

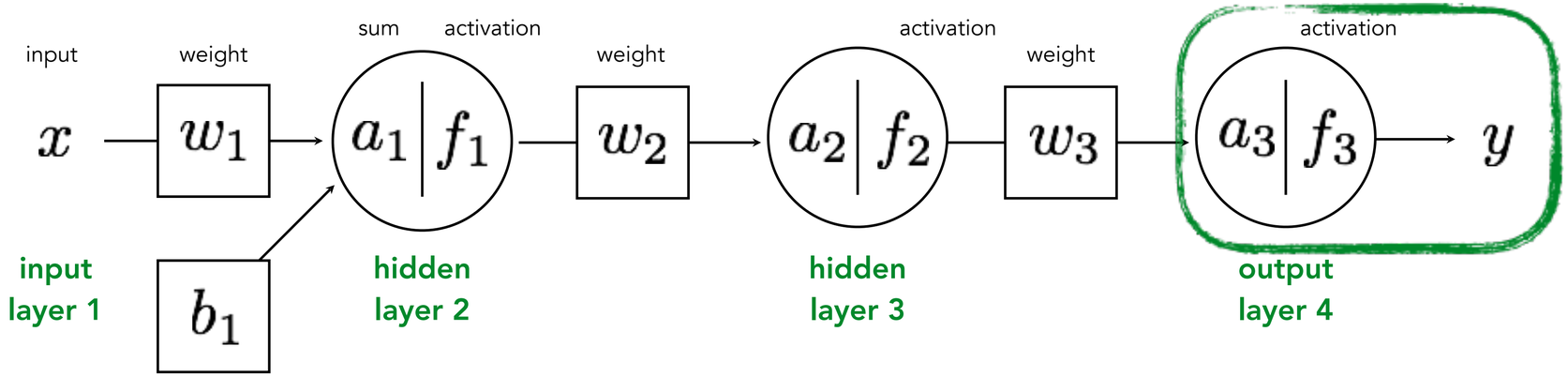
$$a_2 = w_2 \cdot f_1(w_1 \cdot x + b_1)$$



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

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

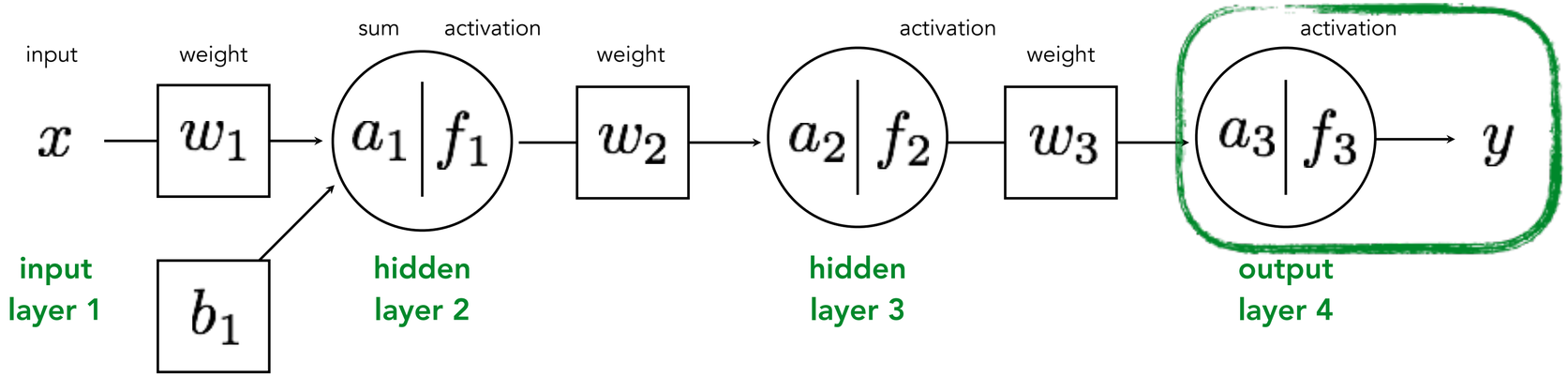
$$a_3 = w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1))$$



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

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

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



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

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

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

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

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

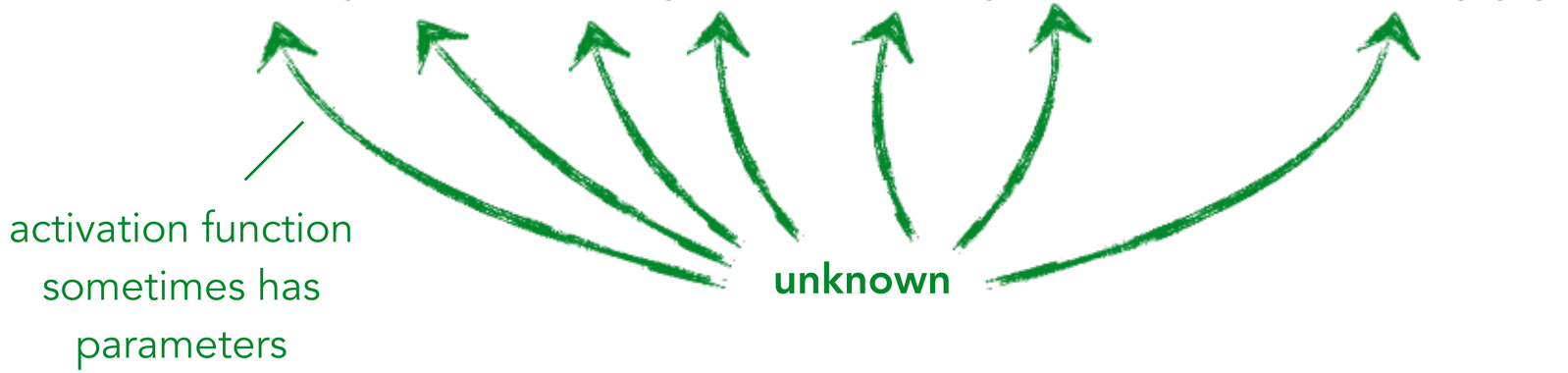
$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)))$$

We need to train the network:

What is known? What is unknown?

Entire network can be written out as a long equation

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



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 $\{x_i, y_i\}$

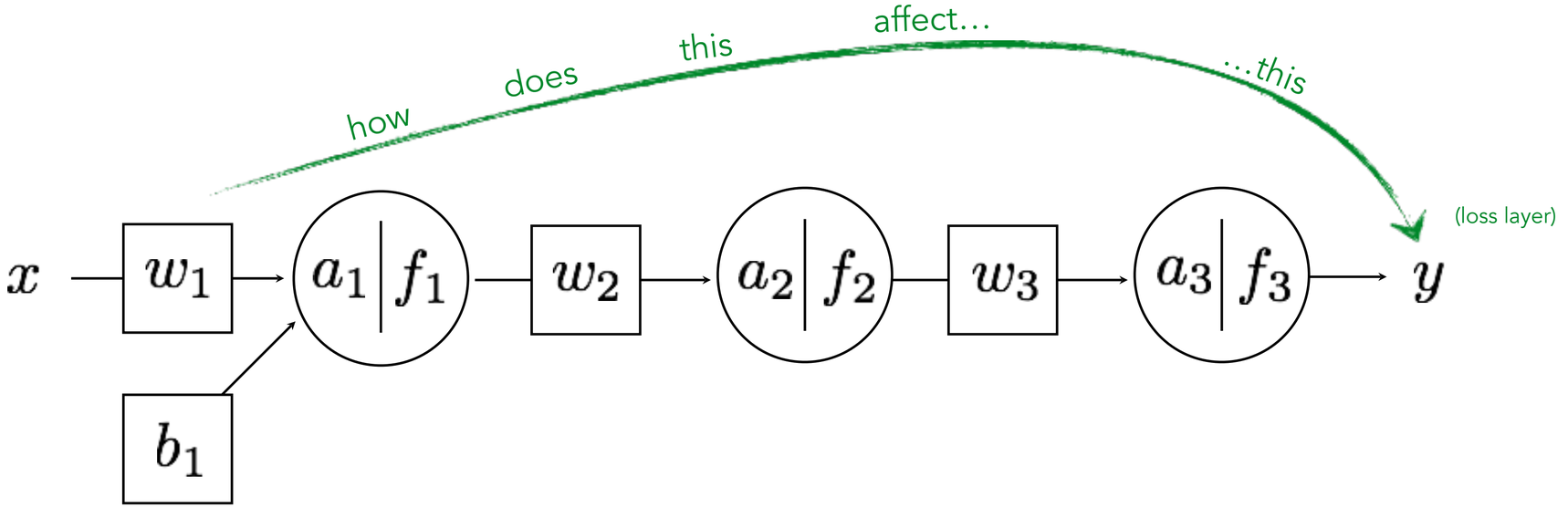
1. Predict
    a. Forward pass $\hat{y} = f_{\text{MLP}}(x_i; \theta)$
    b. Compute Loss
2. Update
    a. Back Propagation $\frac{\partial \mathcal{L}}{\partial \theta}$ vector of parameter partial derivatives
    b. Gradient update $\theta \leftarrow \theta - \eta \nabla \theta$ vector of parameter update equations

So we need to compute the partial derivatives

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

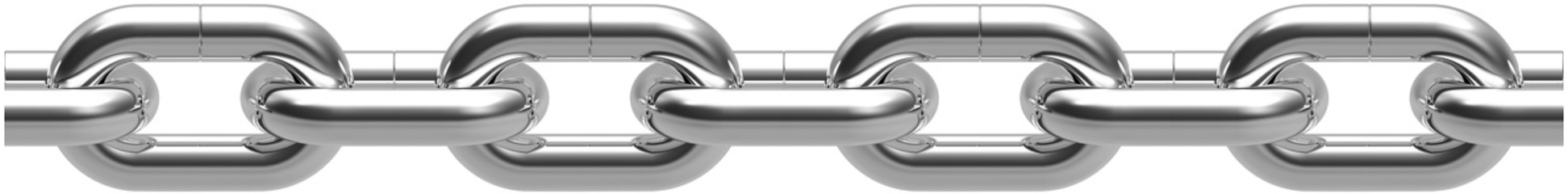
Remember,

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



So, how do you compute it?

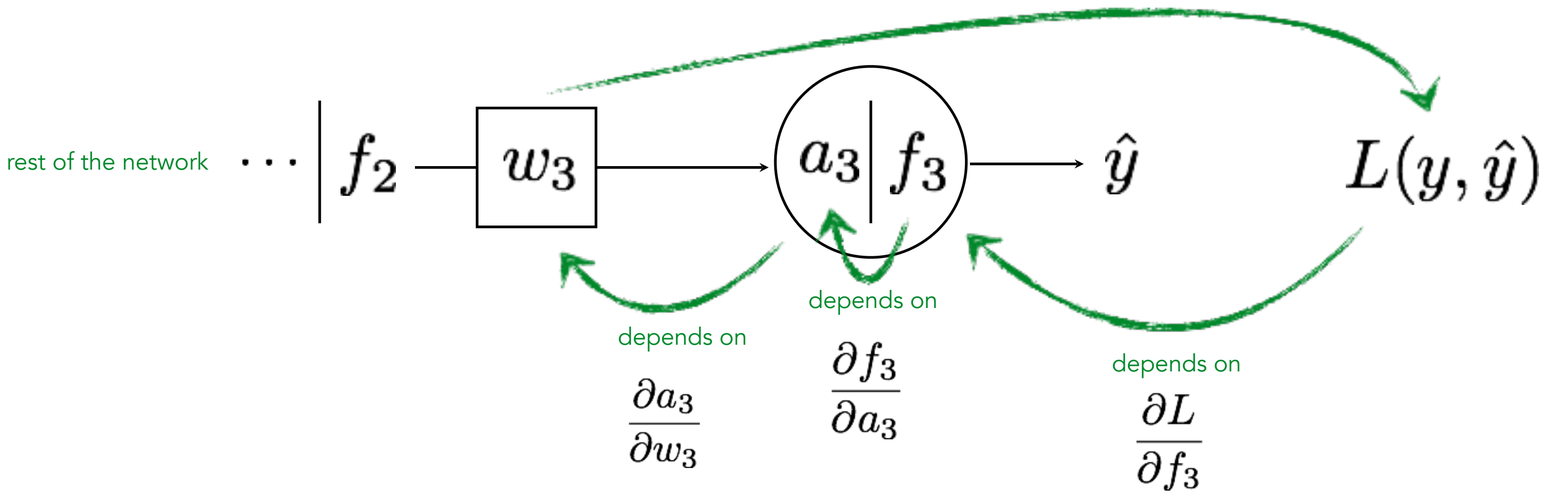
The Chain Rule

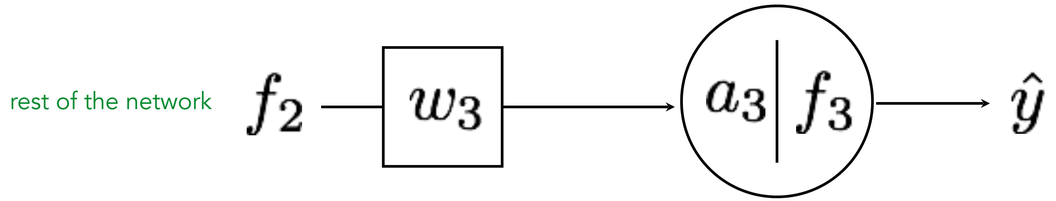


According to the chain rule…

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

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

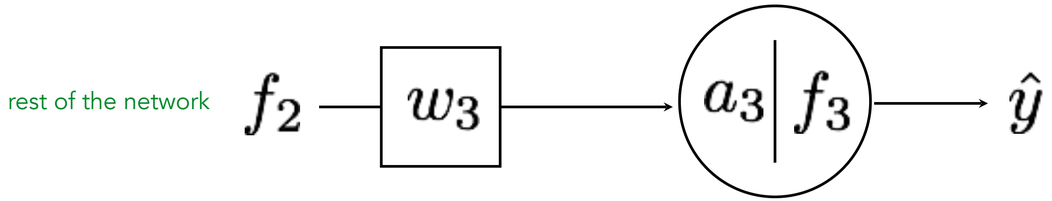




$$L(y, \hat{y})$$

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

Chain Rule!

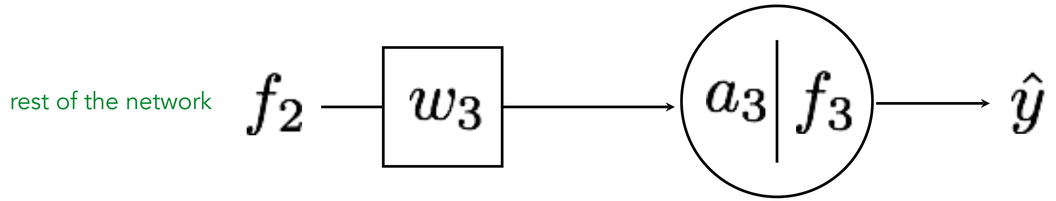


$$L(y, \hat{y})$$

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

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

Just the partial derivative of L2 loss



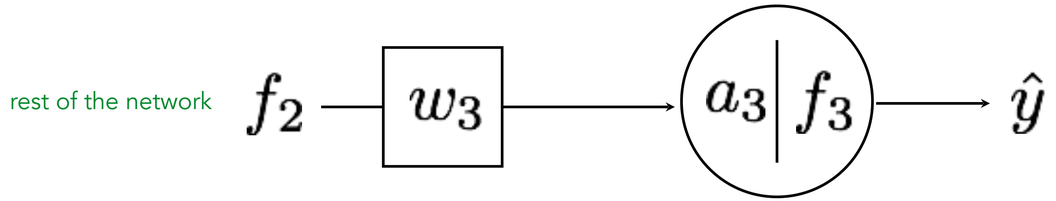
$$L(y, \hat{y})$$

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

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

Let's use a Sigmoid function

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



$$L(y, \hat{y})$$

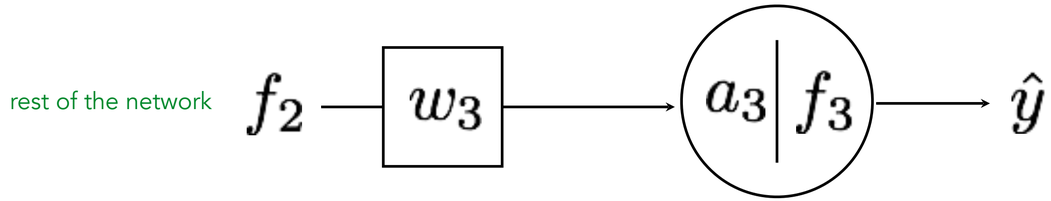
$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial w_3}$$

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

$$= -\eta(y - \hat{y}) f_3(1 - f_3)\frac{\partial a_3}{\partial w_3}$$

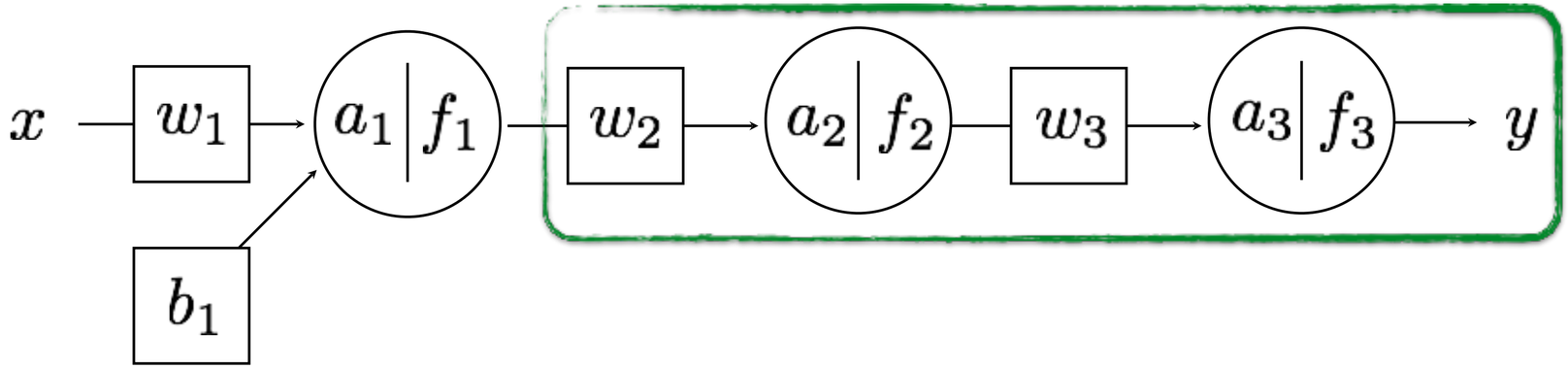
Let's use a Sigmoid function

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

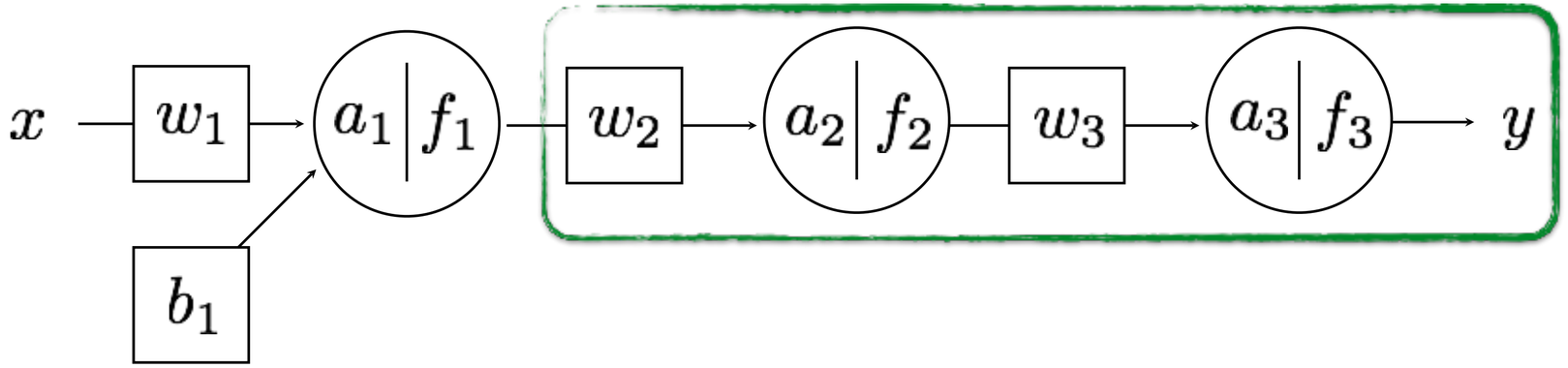


$$L(y, \hat{y})$$

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



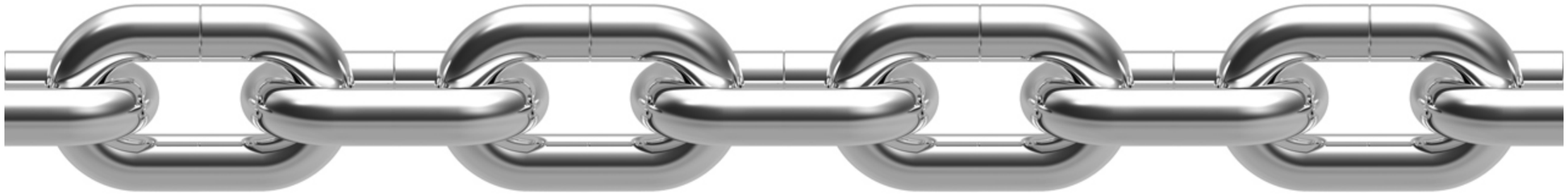
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial f_2}\frac{\partial f_2}{\partial a_2}\frac{\partial a_2}{\partial w_2}$$



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

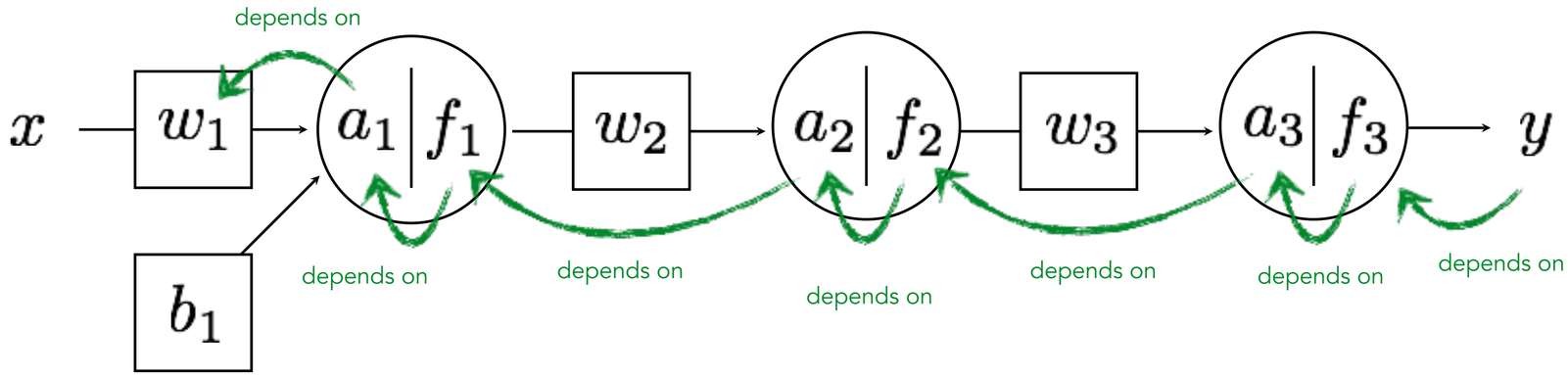
already computed.
re-use (propagate)!

The Chain Rule



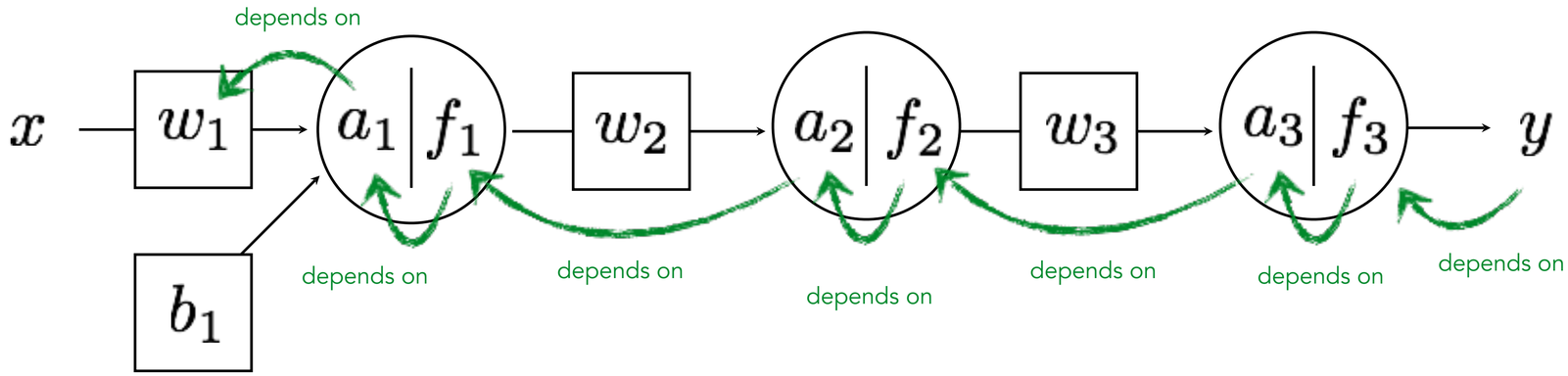
a.k.a. backpropagation

The chain rule says…



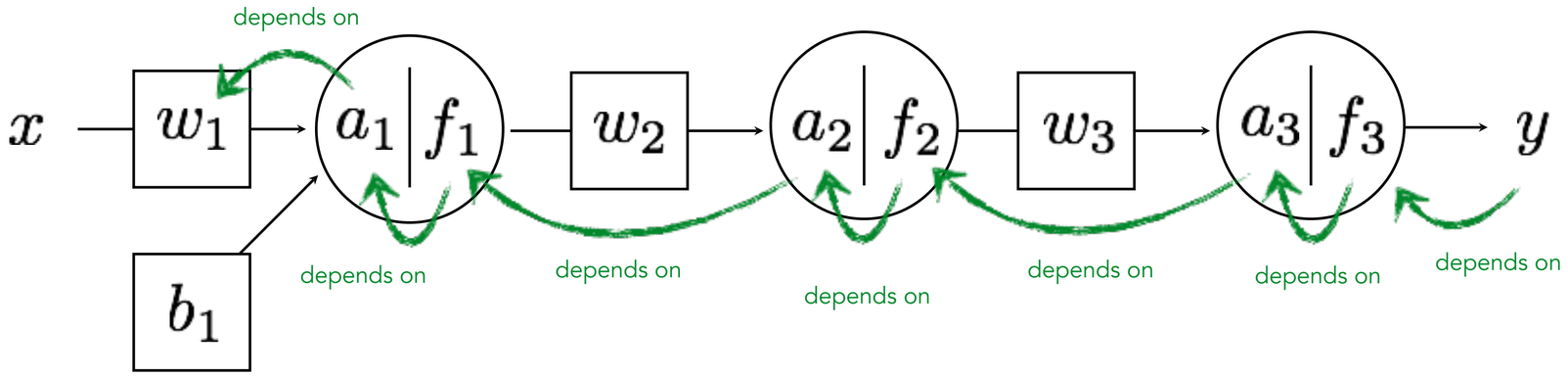
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial f_2}\frac{\partial f_2}{\partial a_2}\frac{\partial a_2}{\partial f_1}\frac{\partial f_1}{\partial a_1}\frac{\partial a_1}{\partial w_1}$$

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)!

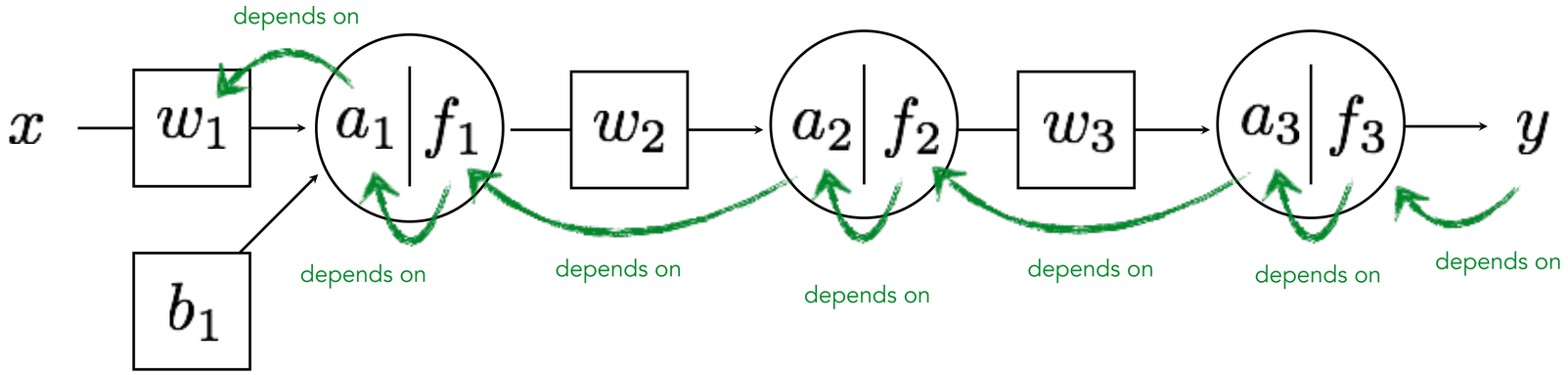


$$\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial w_3}$$

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

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

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

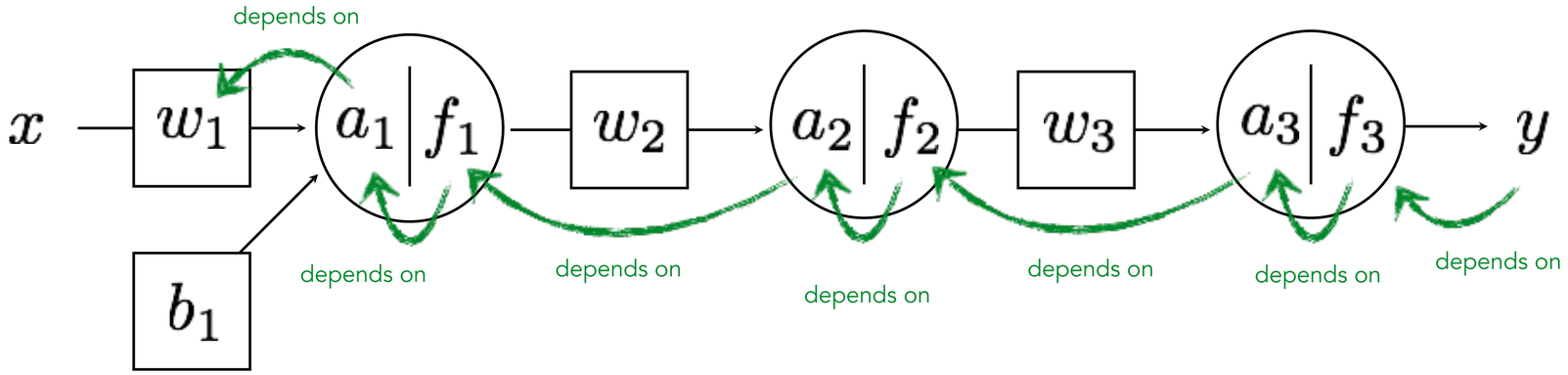


$$\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial w_3}$$

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

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

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



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

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

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

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

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

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

    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?

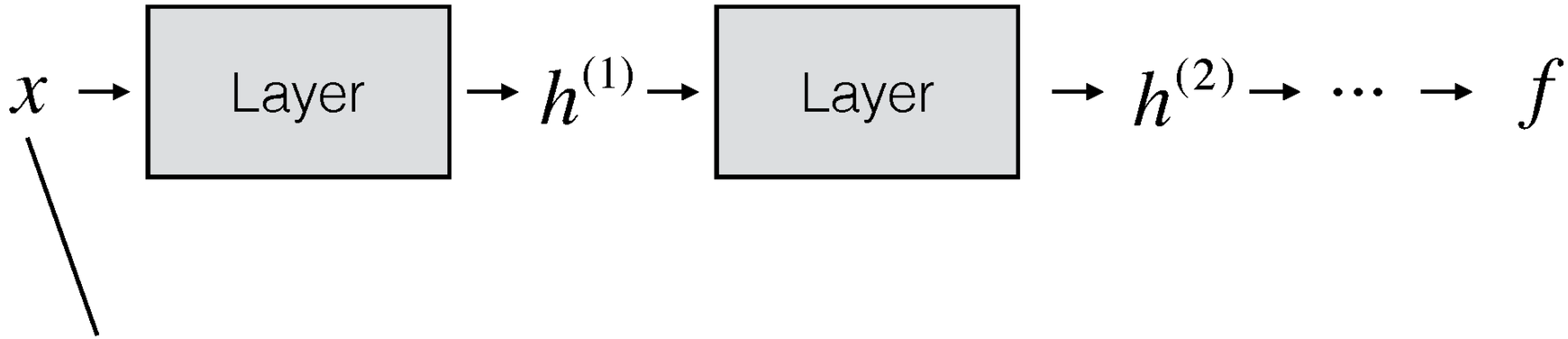
- It's very expensive when N is large (big data).

Do I lose anything by using stochastic GD?

- Same convergence guarantees and complexity!
- Better generalization.

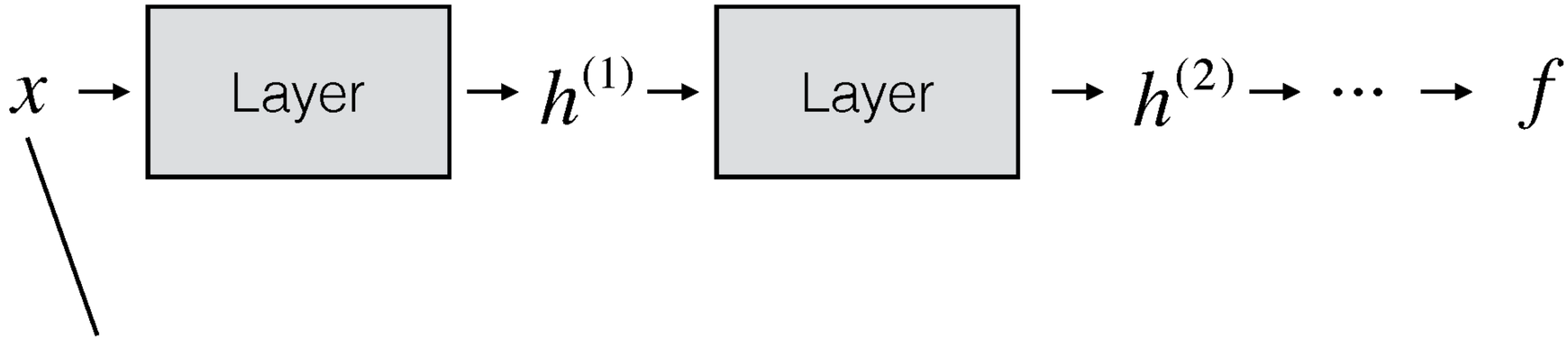
ConvNets

What shape should the activations have?



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

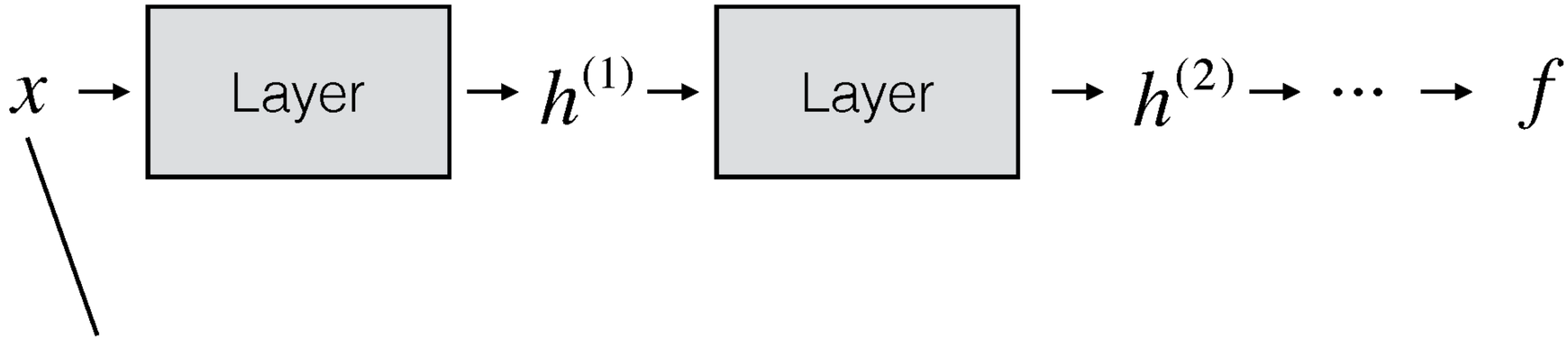
What shape should the activations have?



- 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

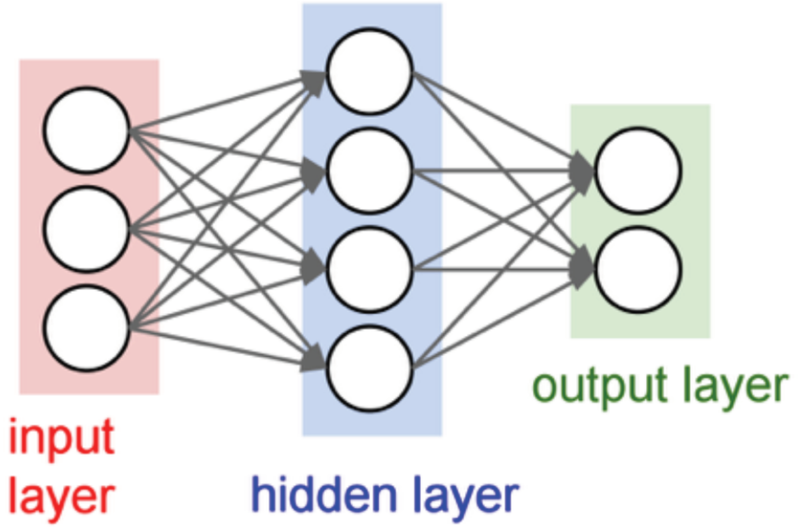
- What about keeping everything in 3D?

ConvNets

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

3D Activations

before:



(1D vectors)

Figure: Andrej Karpathy

3D Activations

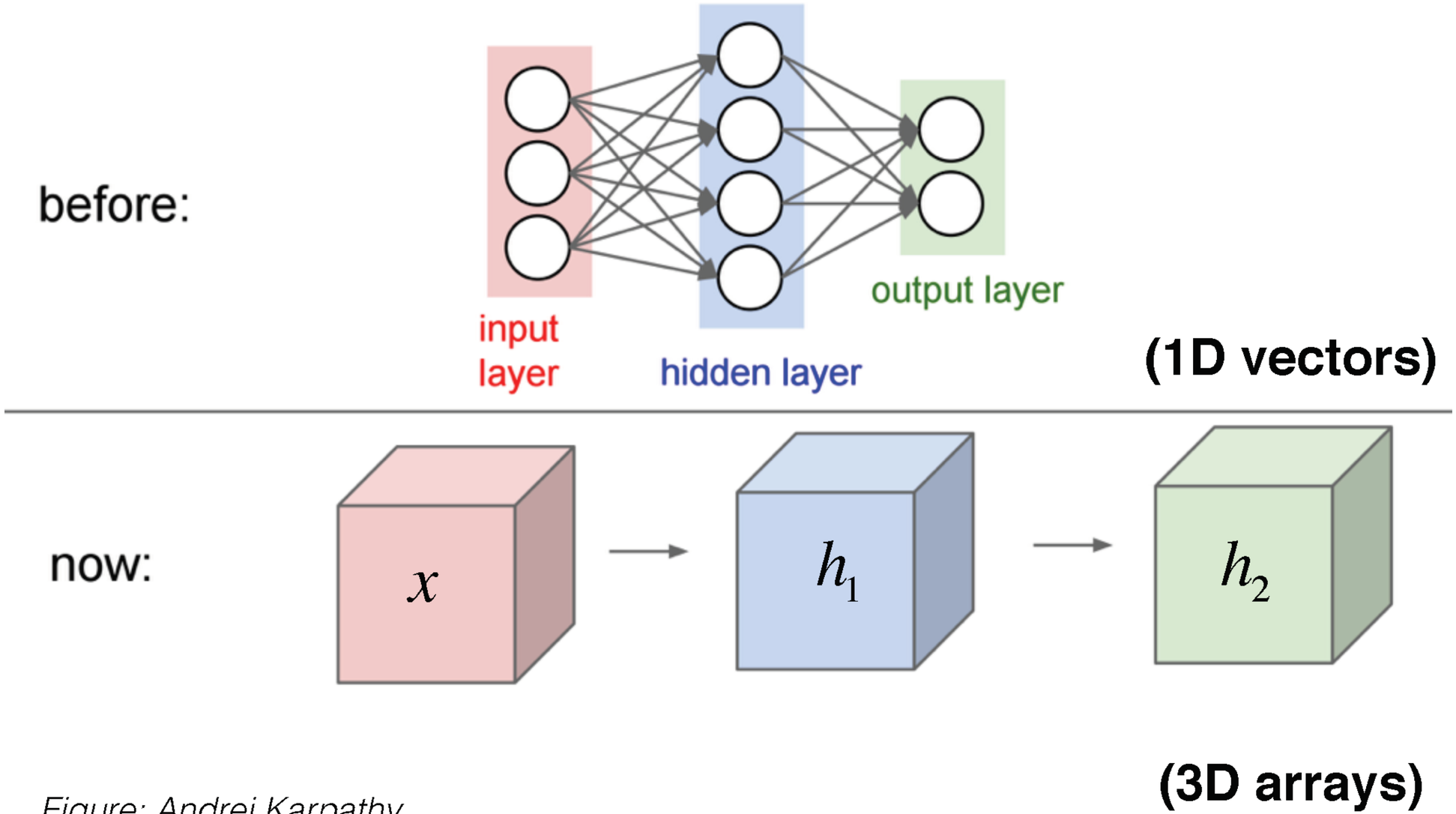


Figure: Andrej Karpathy

3D Activations

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

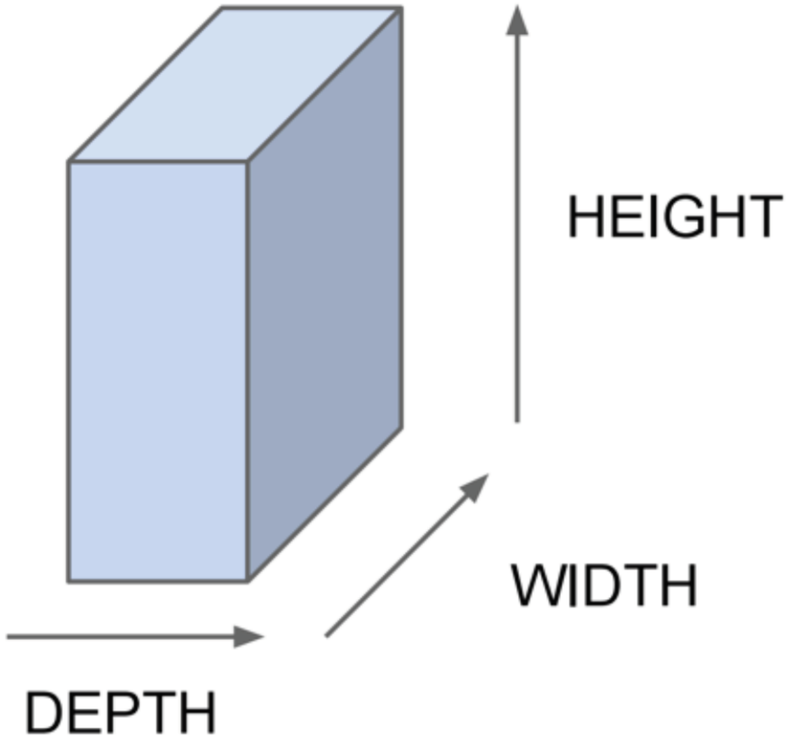
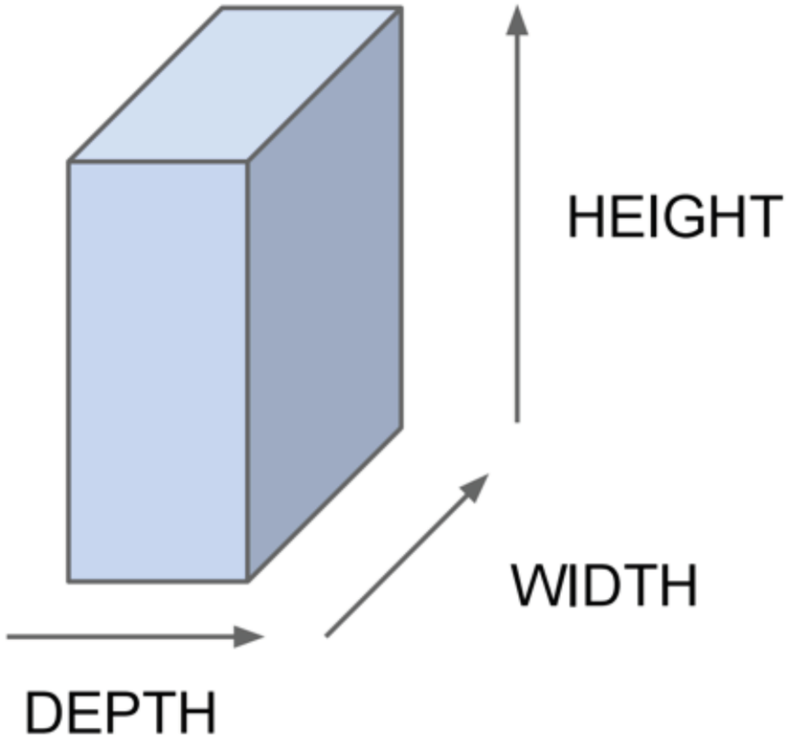


Figure: Andrej Karpathy

3D Activations

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



For example, a CIFAR-10 image is a 3x32x32 volume (3 depth — RGB channels, 32 height, 32 width)

Figure: Andrej Karpathy

3D Activations

1D Activations:

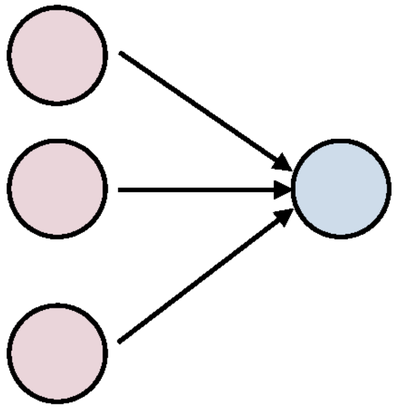
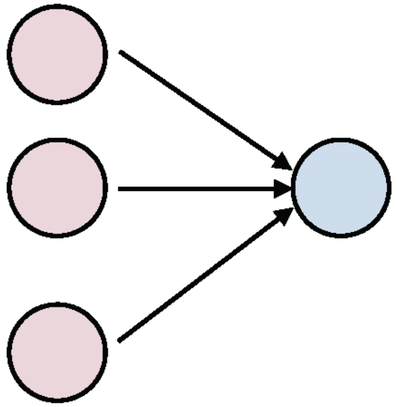


Figure: Andrej Karpathy

3D Activations

1D Activations:



3D Activations:

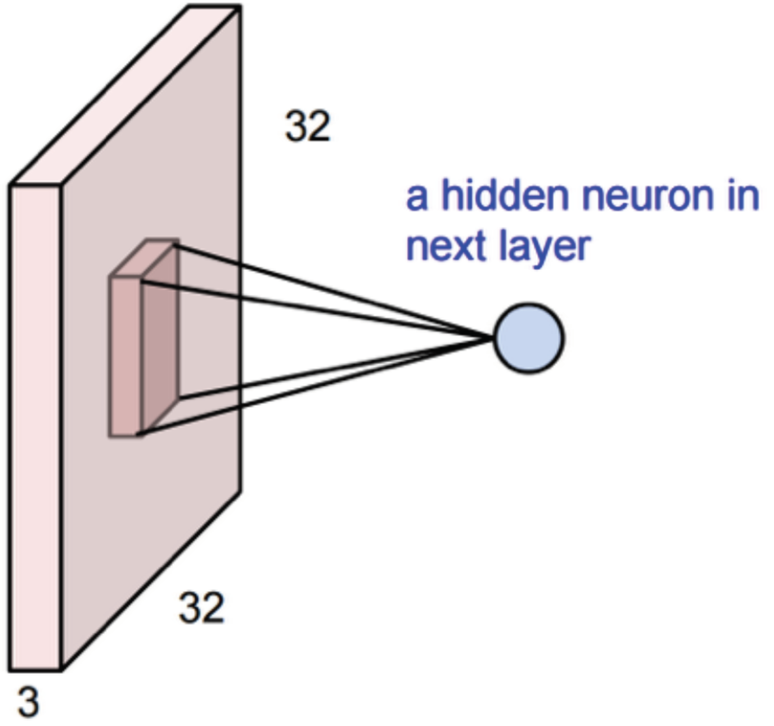
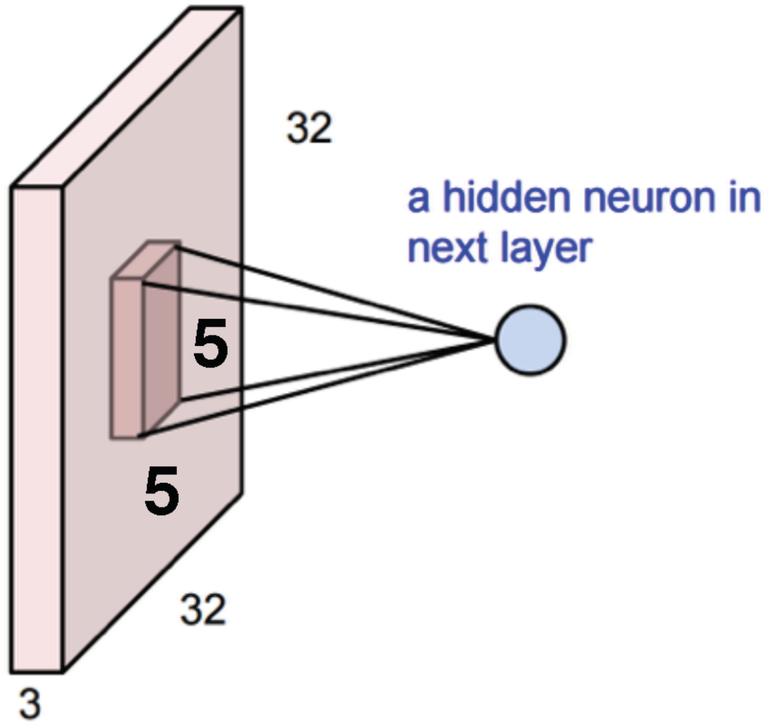


Figure: Andrej Karpathy

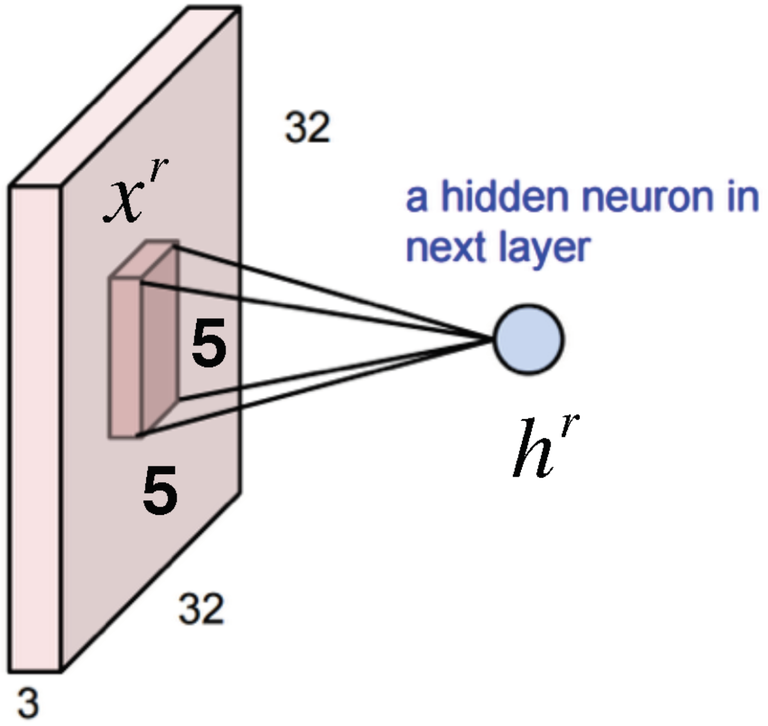
3D Activations



- The input is 3x32x32
- This neuron depends on a 3x5x5 chunk of the input
- The neuron also has a 3x5x5 set of weights and a bias (scalar)

Figure: Andrej Karpathy

3D Activations

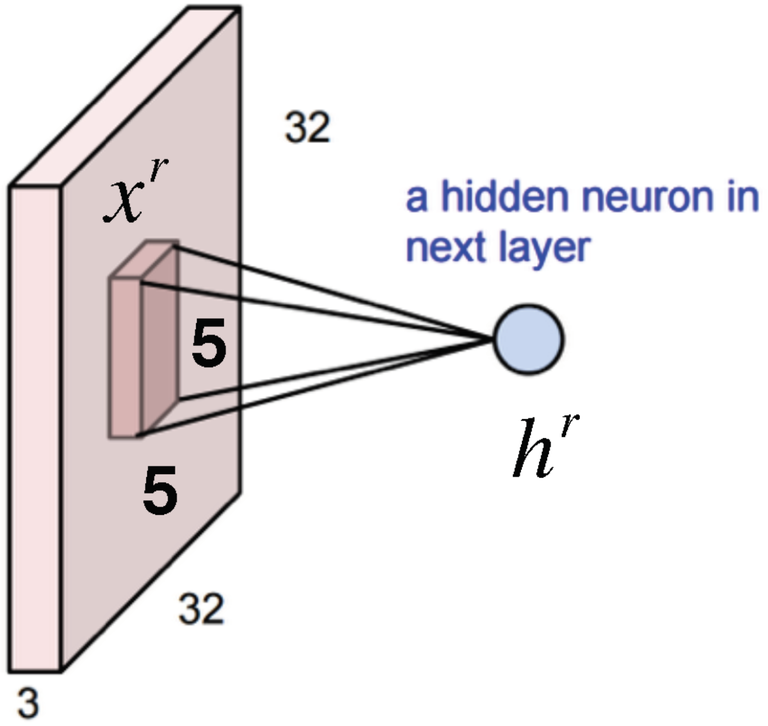


Example: consider the region of the input "x^r"

With output neuron h^r

Figure: Andrej Karpathy

3D Activations



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

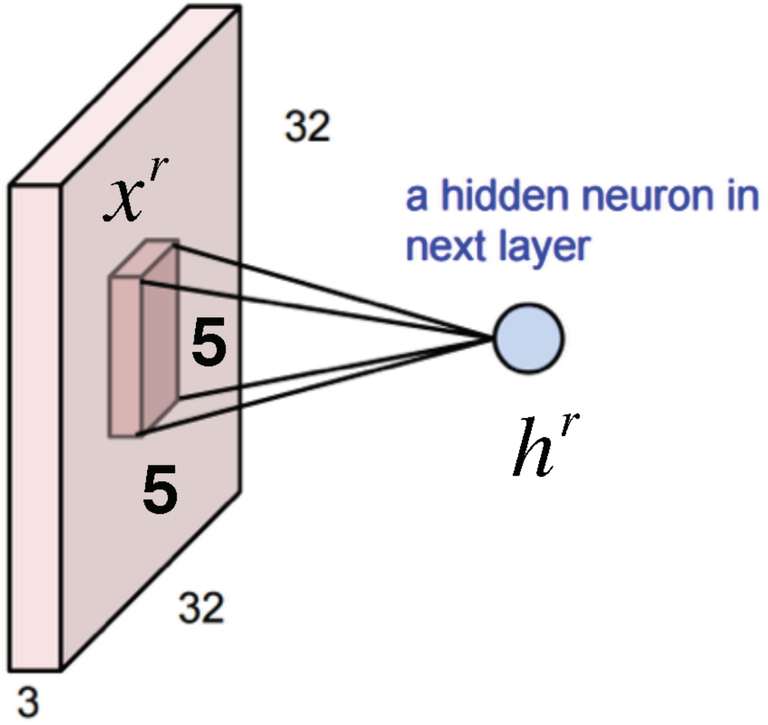
With output neuron h^r

Then the output is:

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

Figure: Andrej Karpathy

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

Figure: Andrej Karpathy

3D Activations

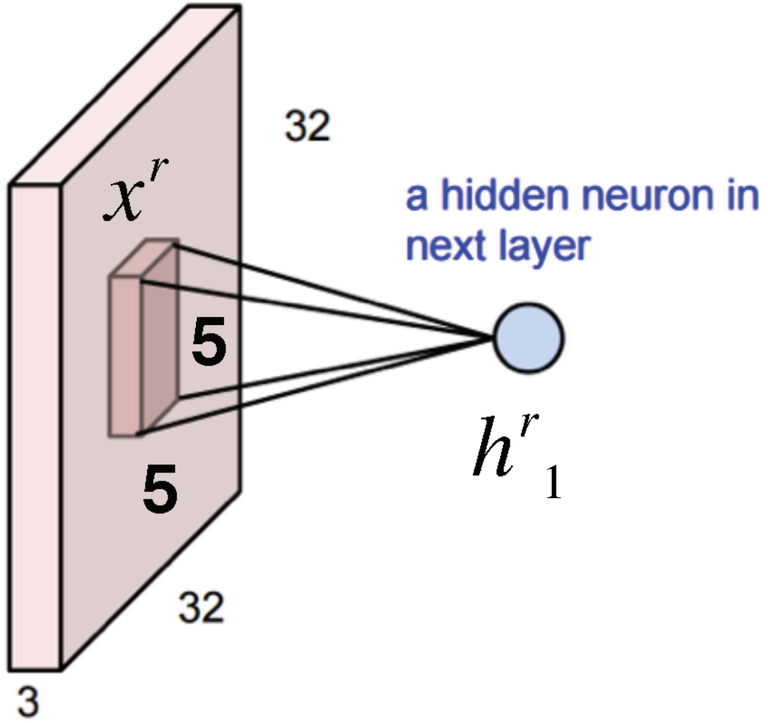


Figure: Andrej Karpathy

3D Activations

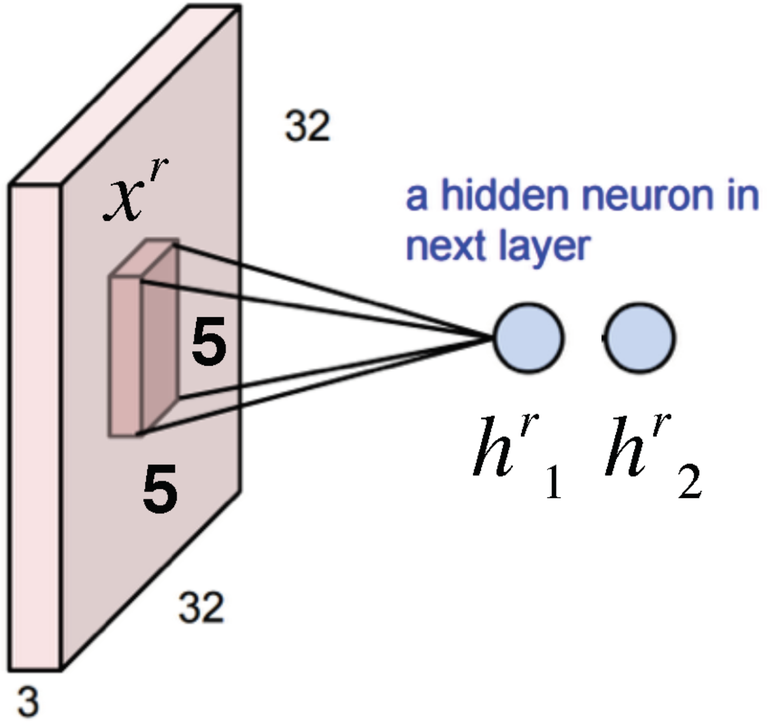
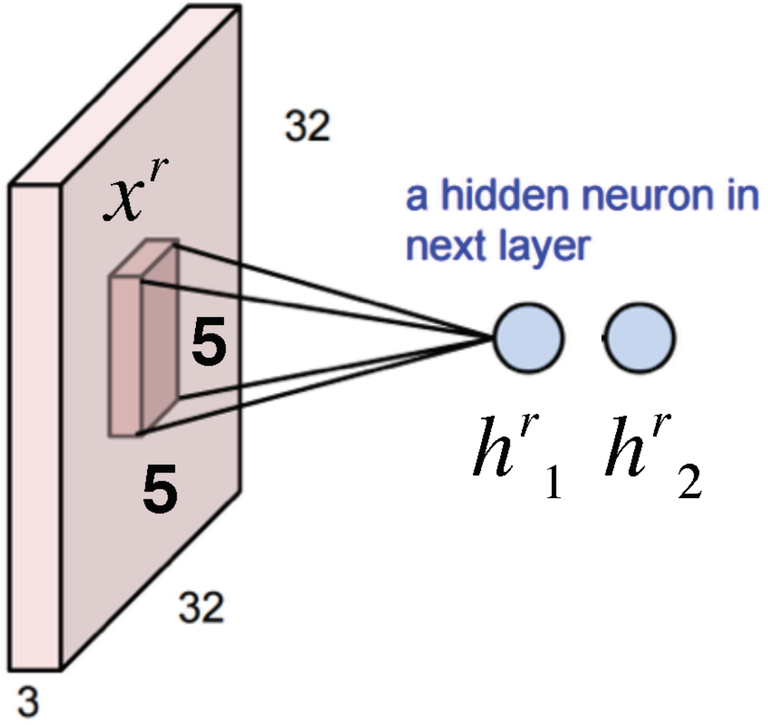


Figure: Andrej Karpathy

3D Activations



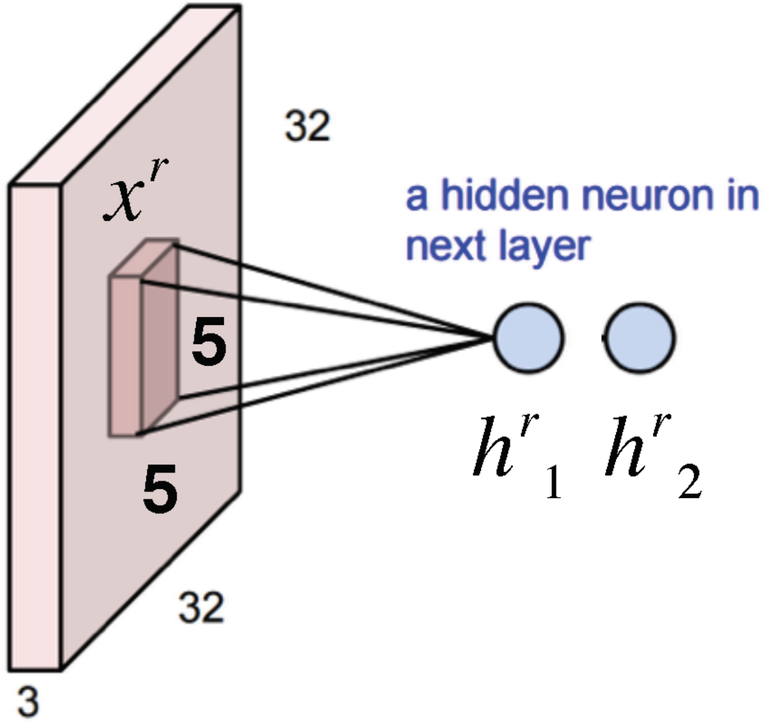
With **2** output neurons

$$h^r{}_1 = \sum_{ijk} x^r{}_{ijk} W_{1ijk} + b_1$$

$$h^r{}_2 = \sum_{ijk} x^r{}_{ijk} W_{2ijk} + b_2$$

Figure: Andrej Karpathy

3D Activations



With **2** output neurons

$$h^r{}_1 = \sum_{ijk} x^r{}_{ijk} W_{1ijk} + b_1$$

$$h^r{}_2 = \sum_{ijk} x^r{}_{ijk} W_{2ijk} + b_2$$

Figure: Andrej Karpathy

3D Activations

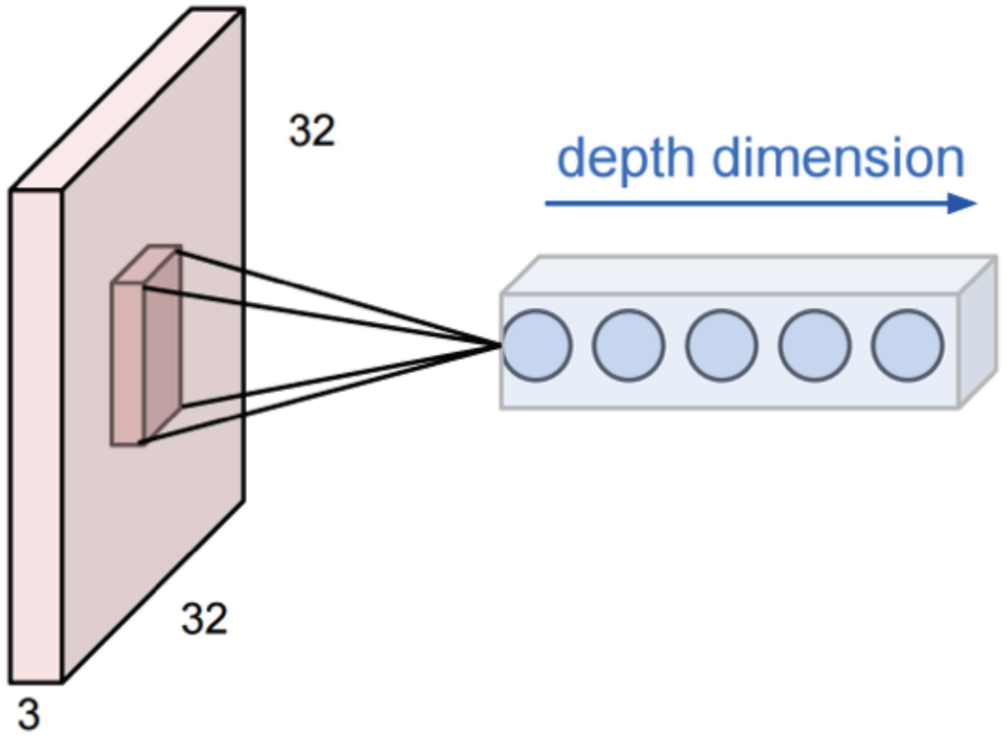
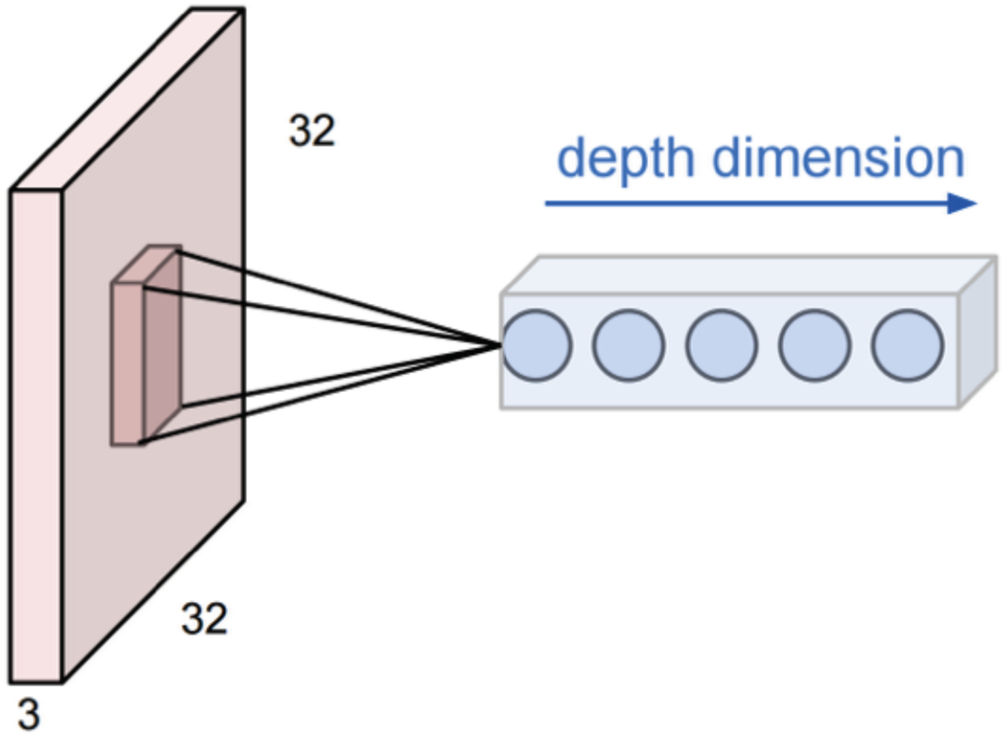


Figure: Andrej Karpathy

3D Activations

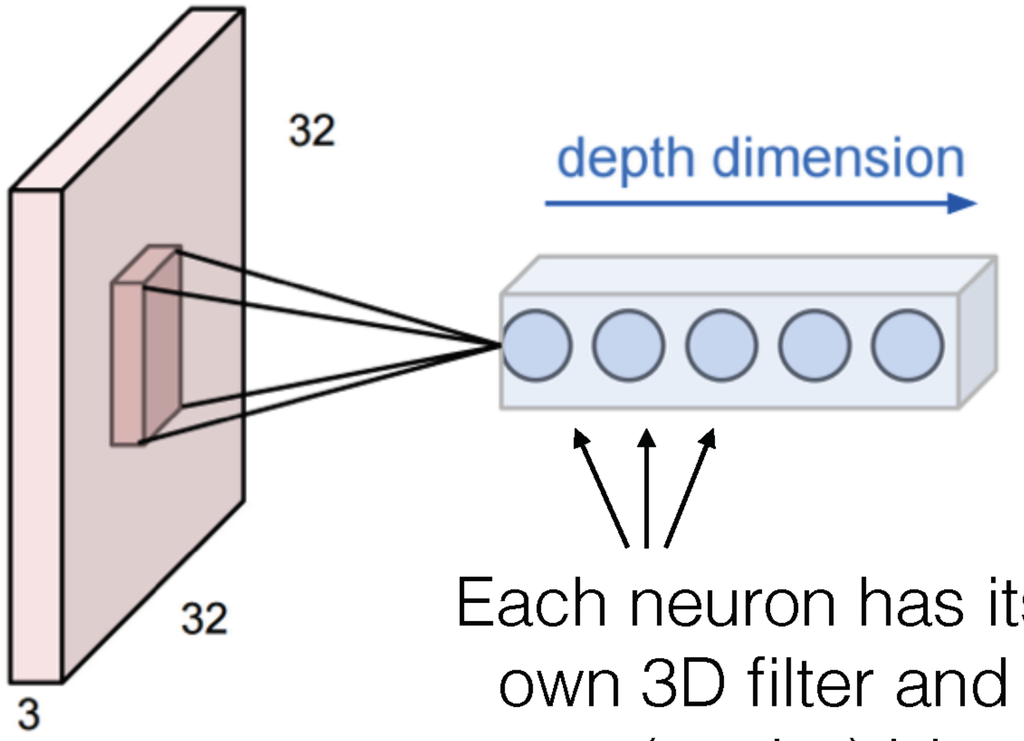


We can keep adding more outputs

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

Figure: Andrej Karpathy

3D Activations



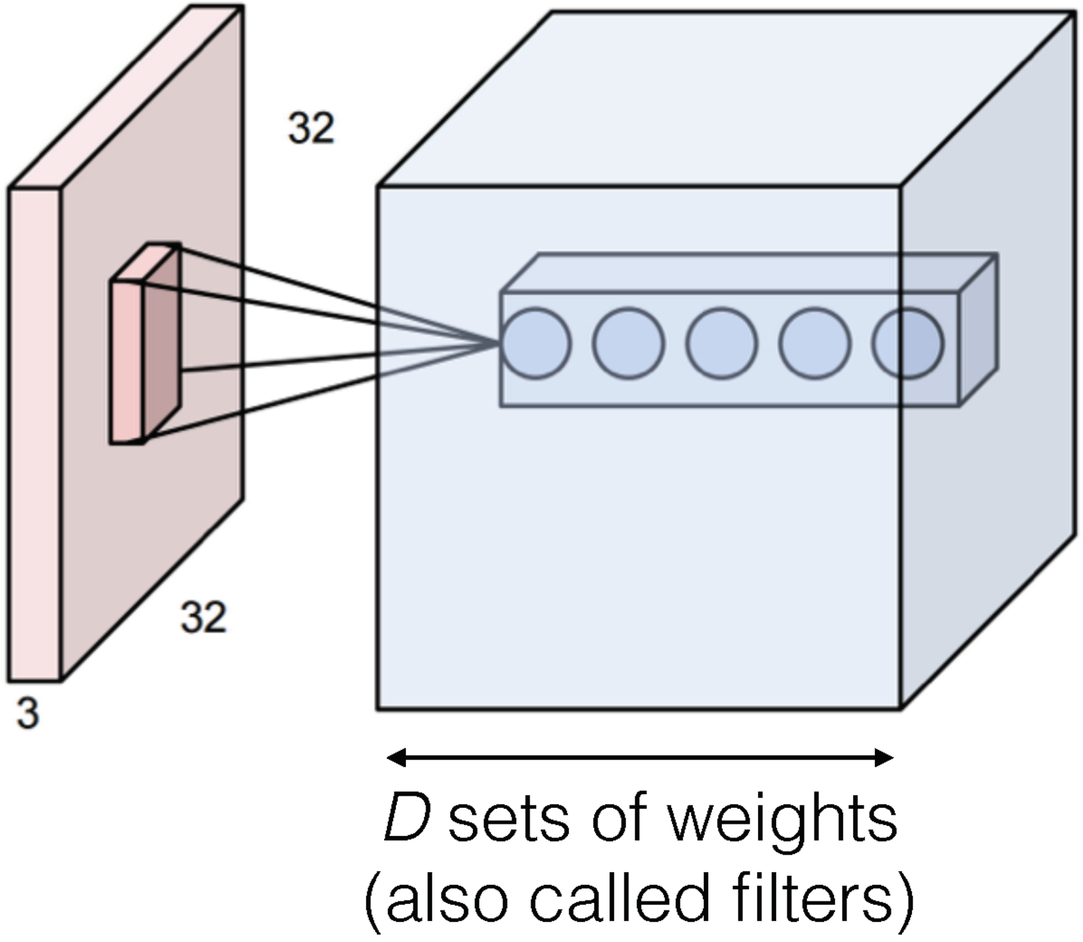
Each neuron has its own 3D filter and own (scalar) bias

We can keep adding more outputs

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

Figure: Andrej Karpathy

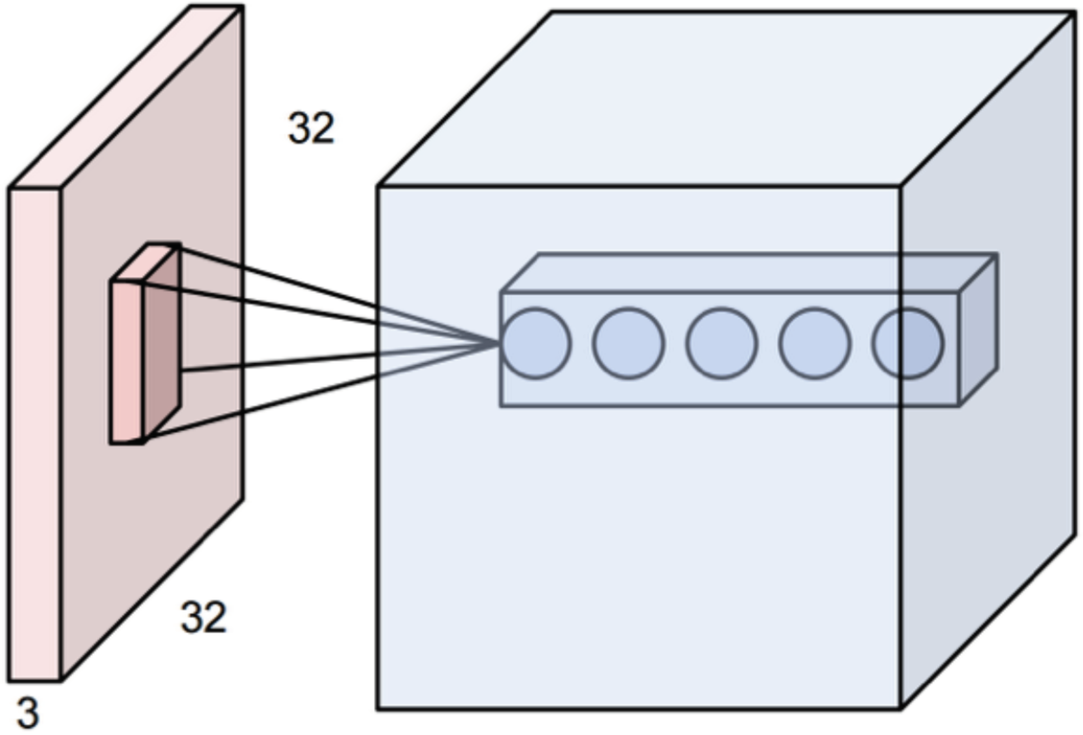
3D Activations



Now repeat this
across the input

Figure: Andrej Karpathy

3D Activations



D sets of weights
(also called filters)

Now repeat this across the input

Weight sharing:

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

Figure: Andrej Karpathy

3D Activations

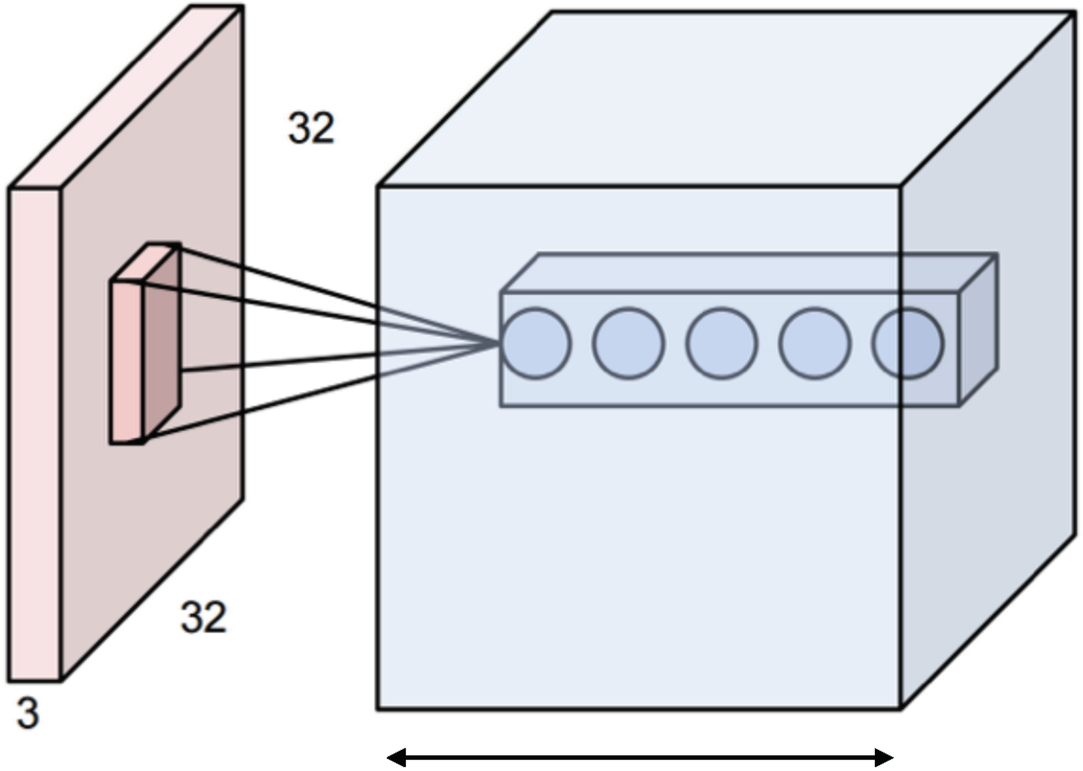
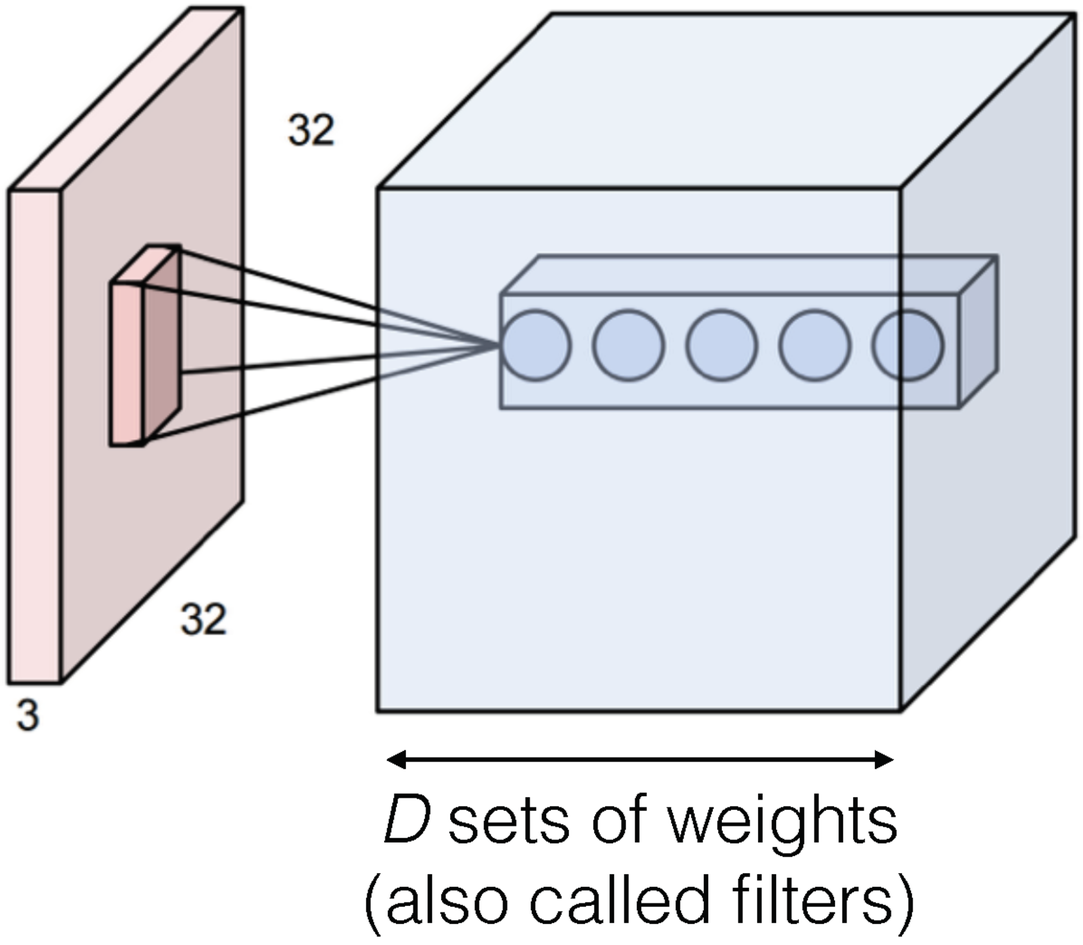


Figure: Andrej Karpathy

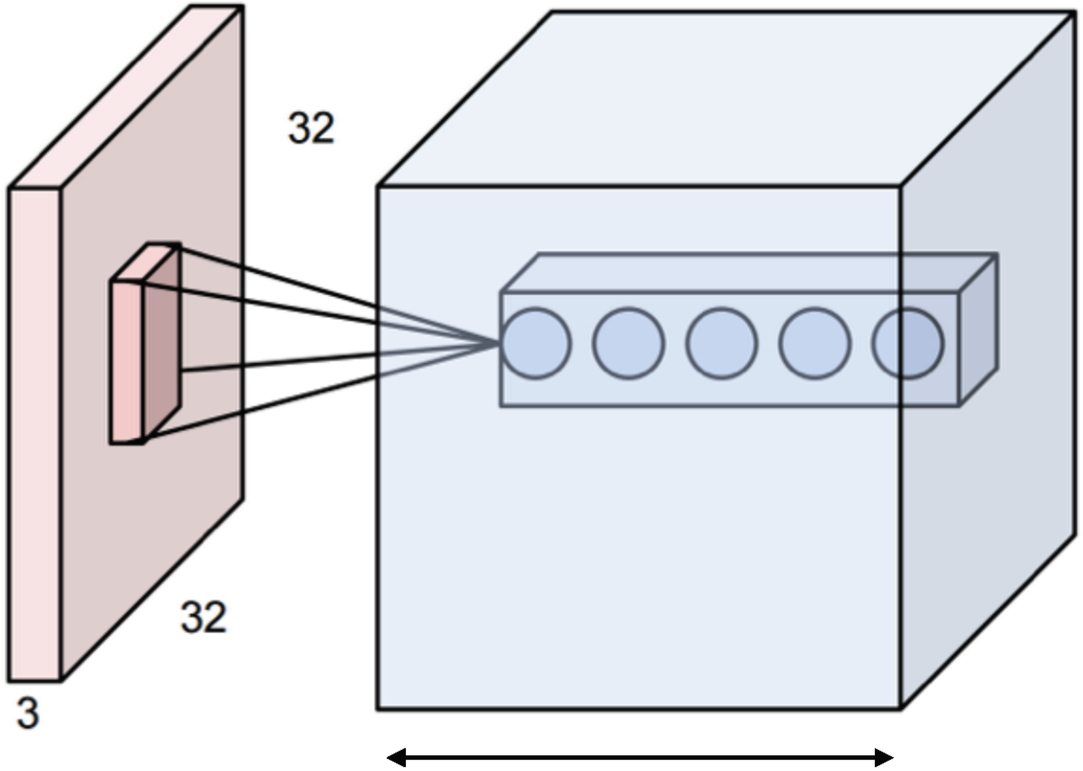
3D Activations



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

Figure: Andrej Karpathy

3D Activations



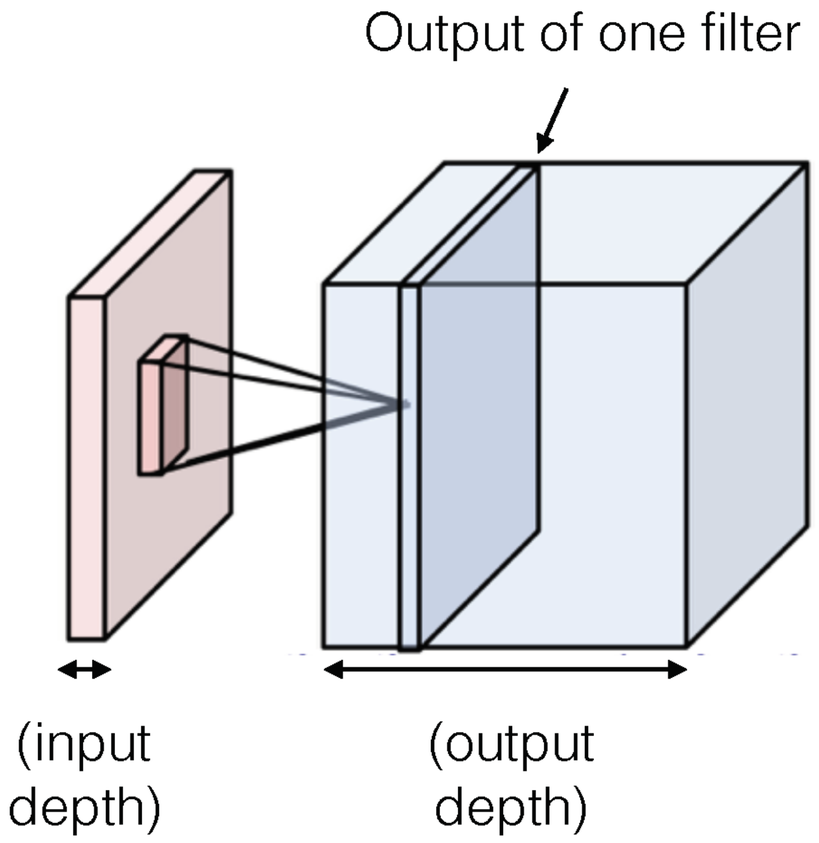
D sets of weights
(also called filters)

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

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

Figure: Andrej Karpathy

3D Activations

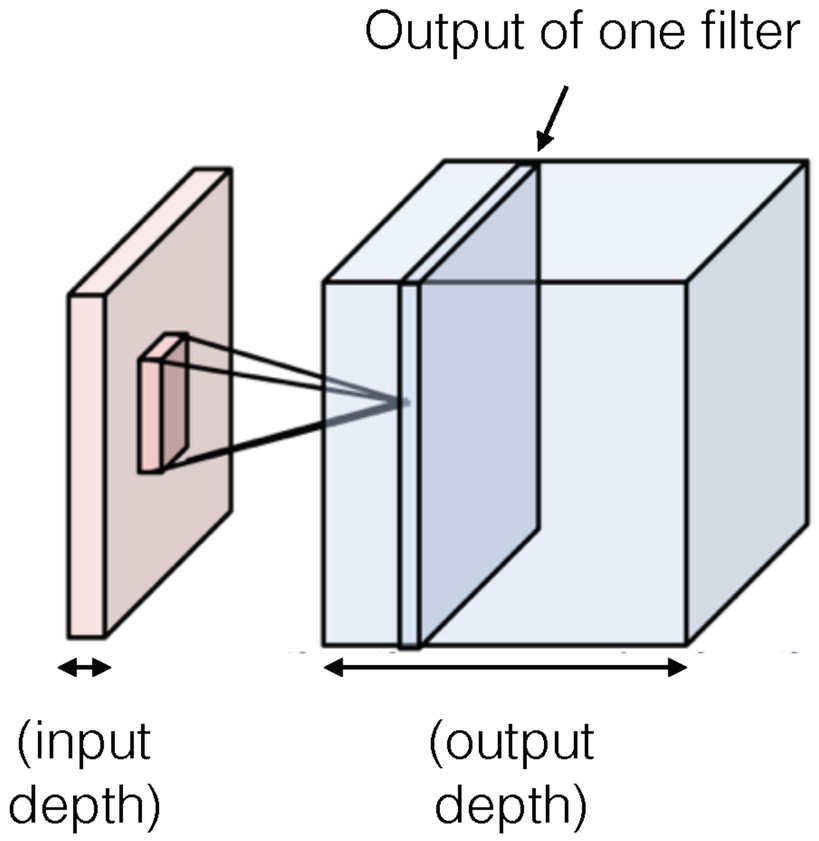


One set of weights gives one slice in the output

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)

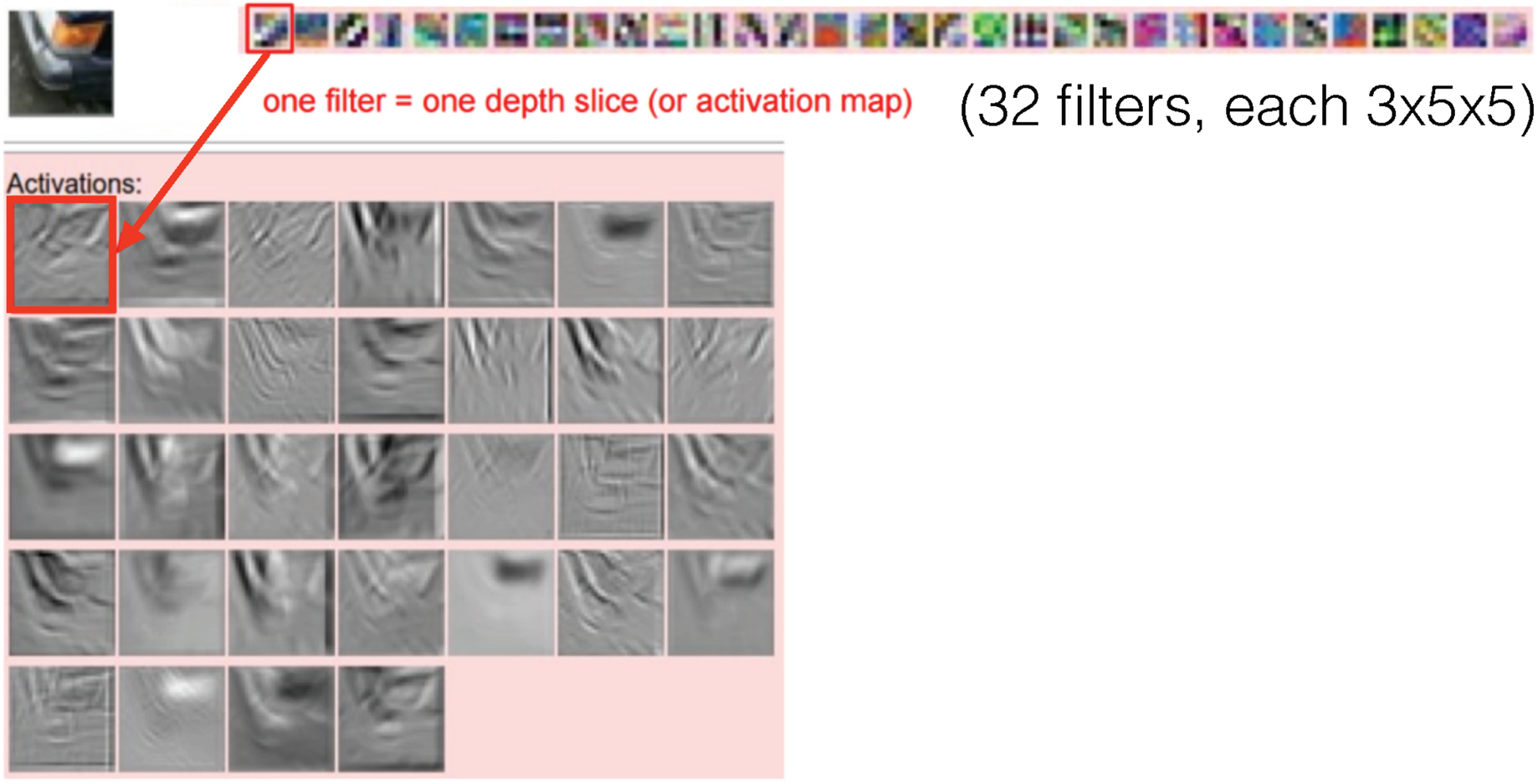


Figure: Andrej Karpathy

3D Activations

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

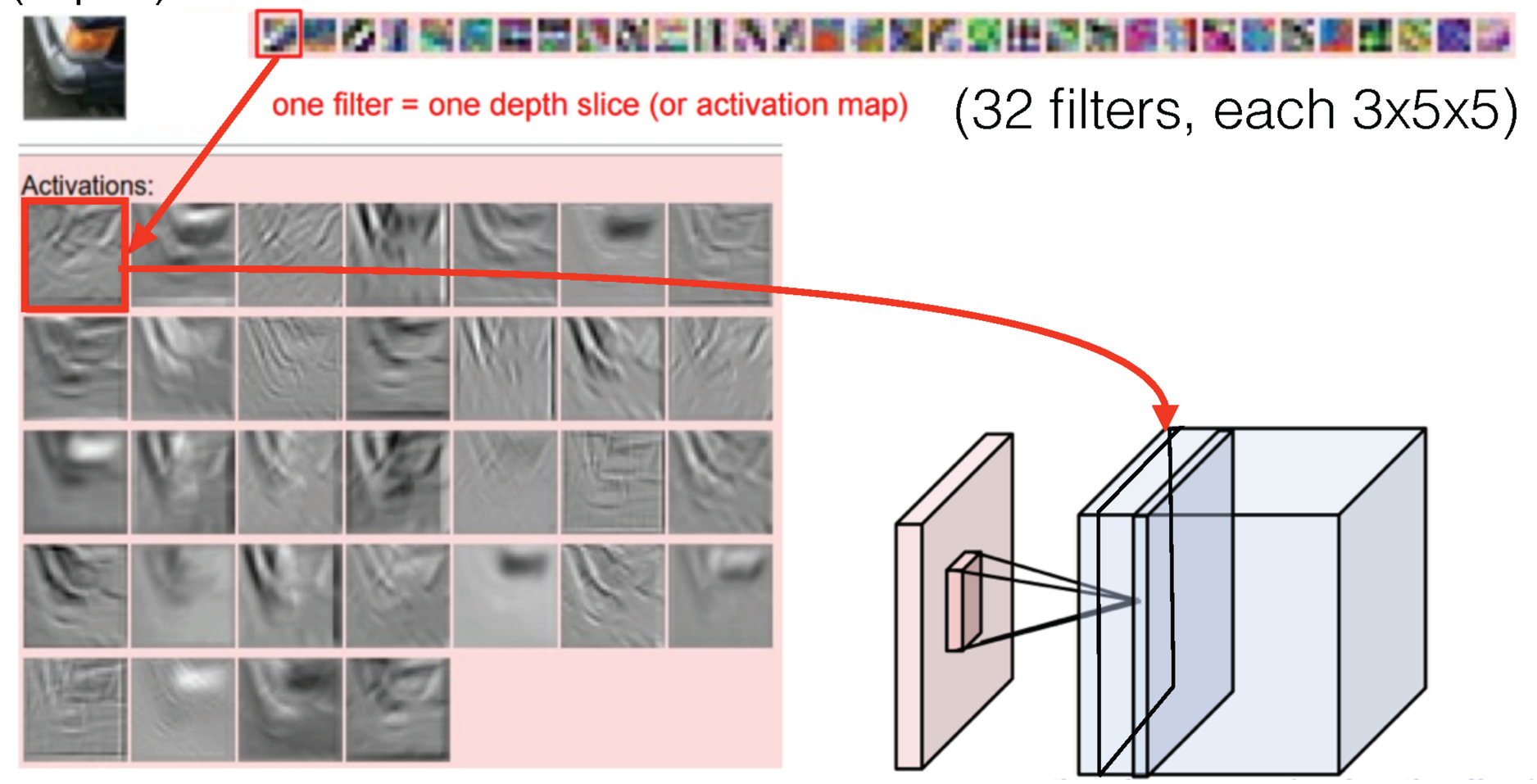


Figure: Andrej Karpathy

3D Activations

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

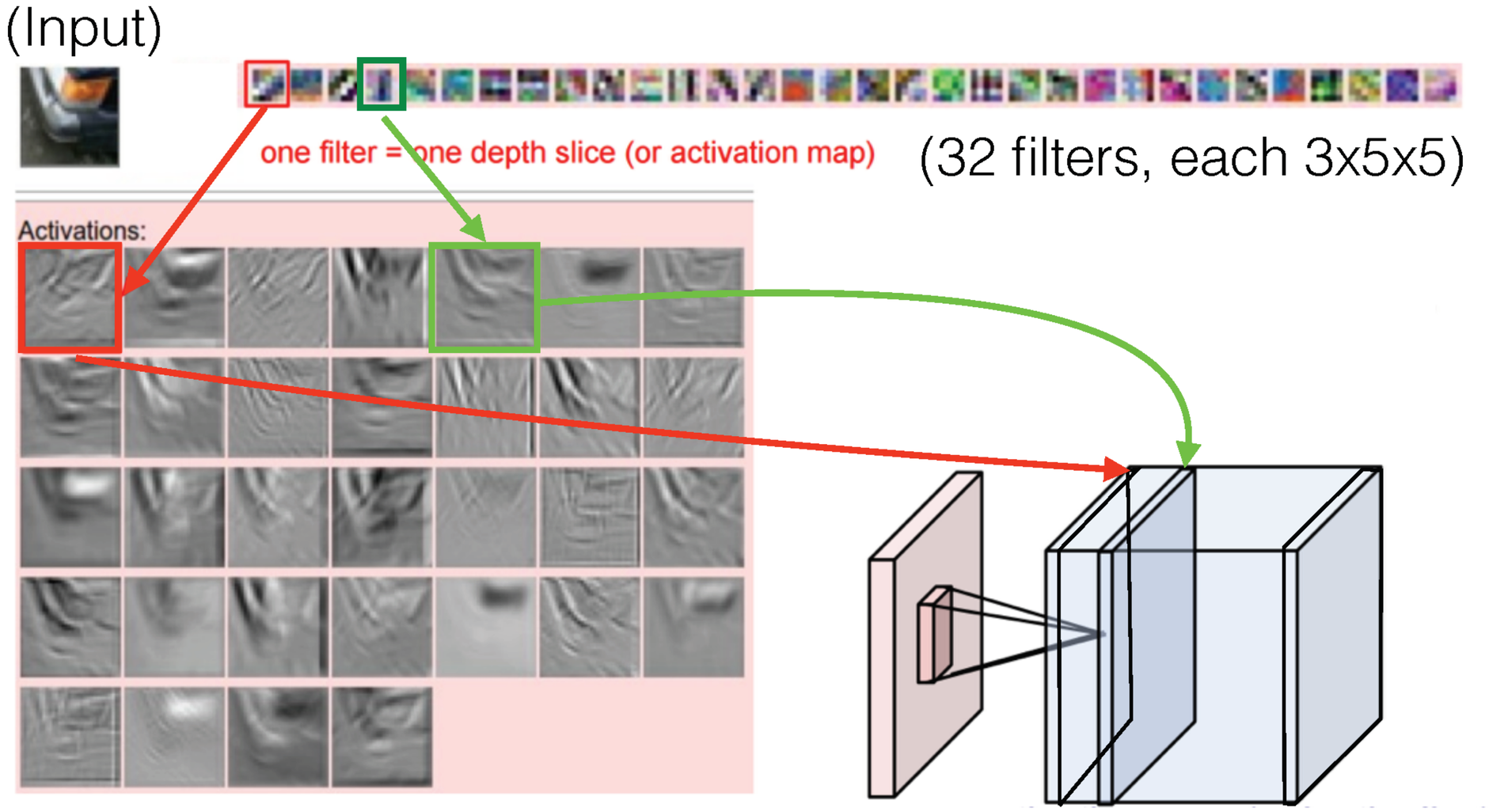


Figure: Andrej Karpathy

3D Activations

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

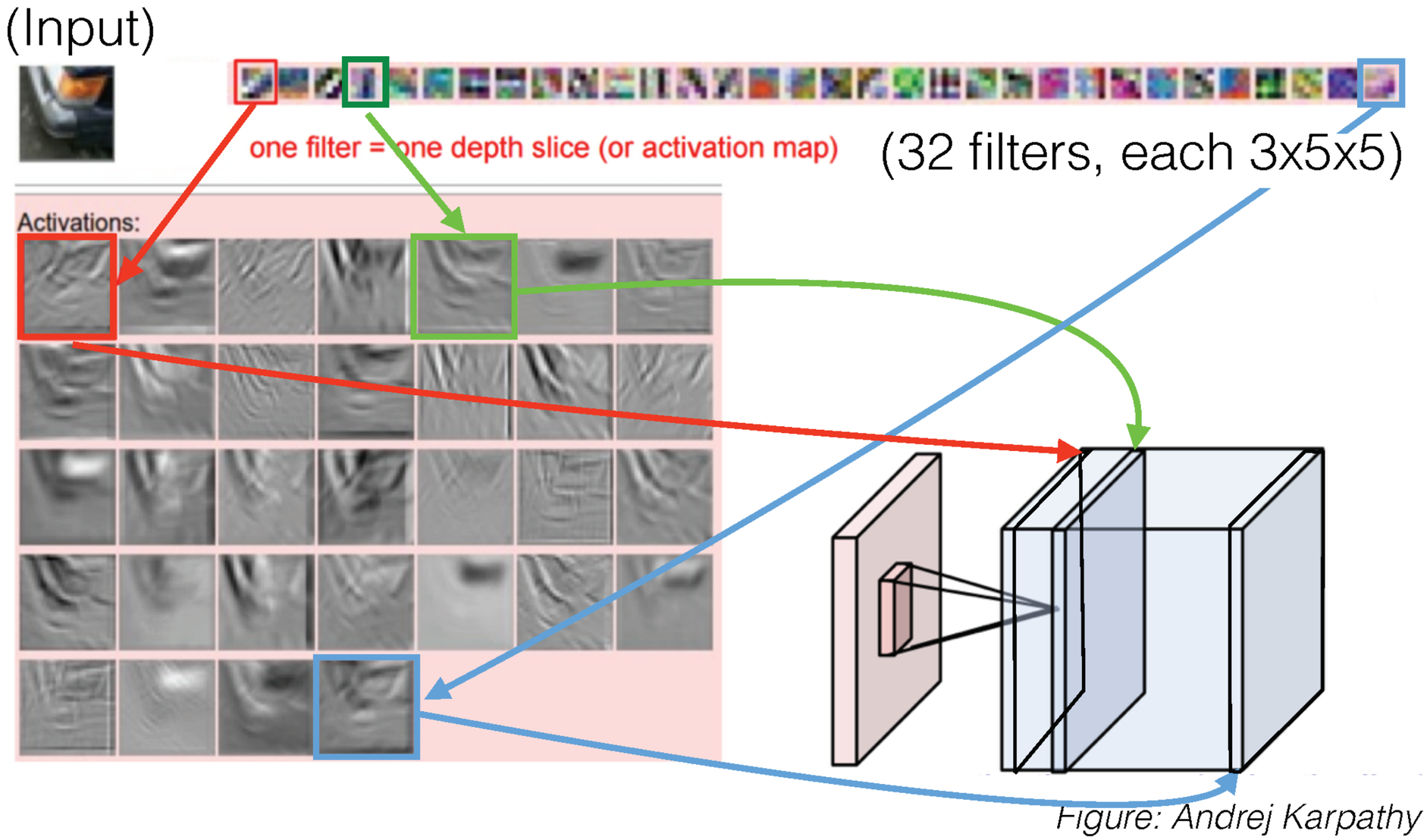
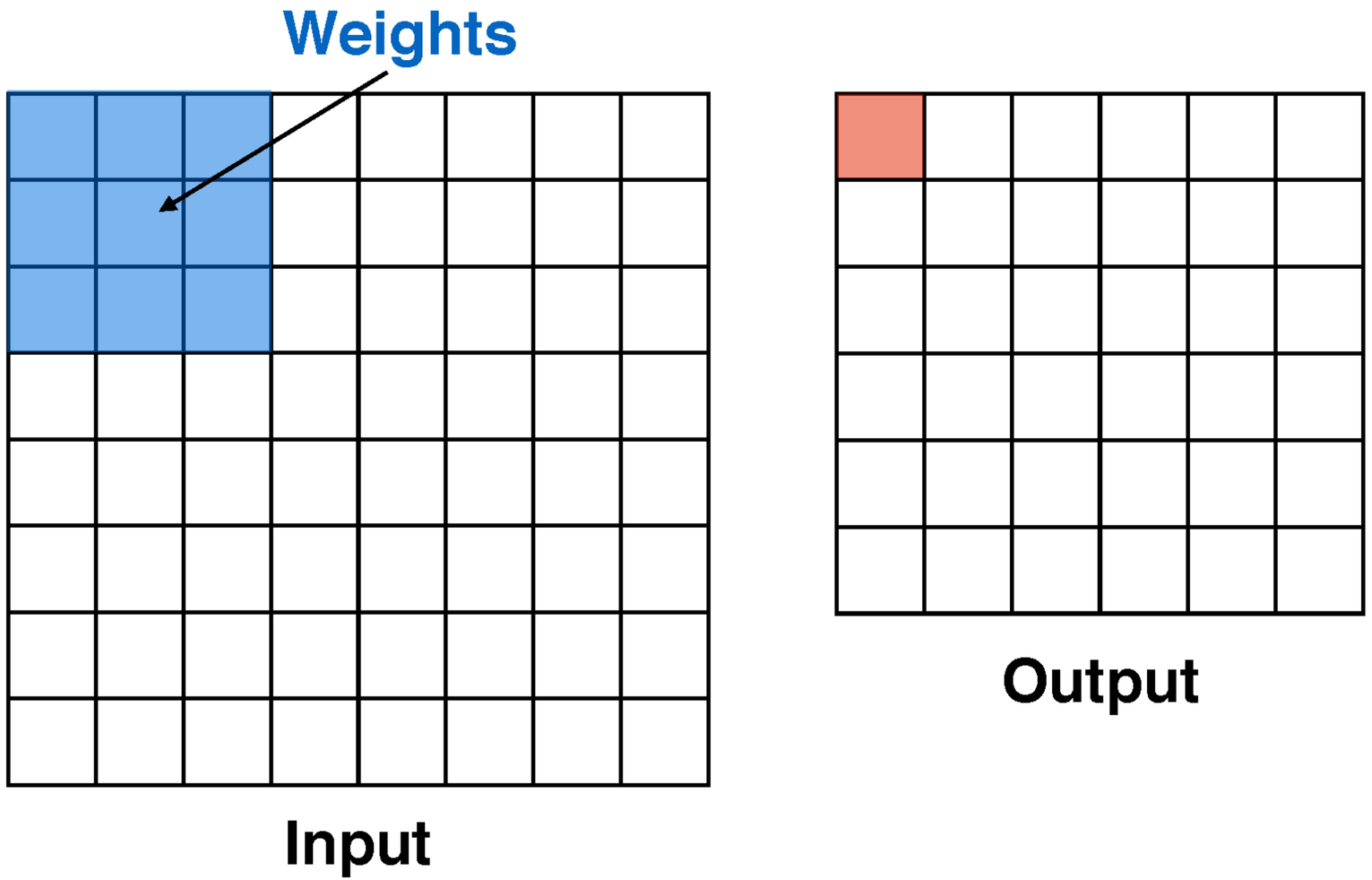


Figure: Andrej Karpathy

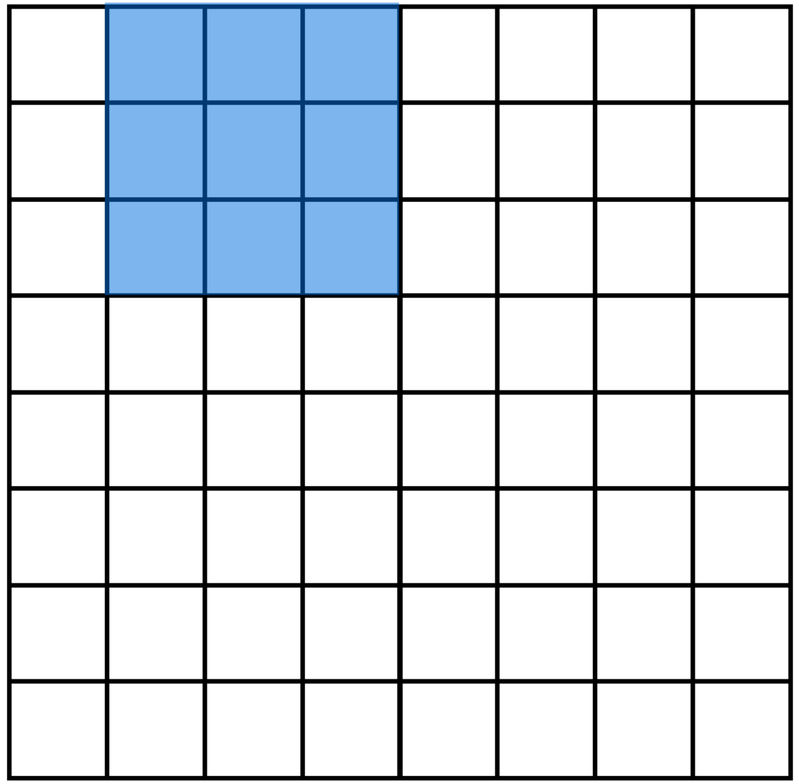
Convolution: Stride

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

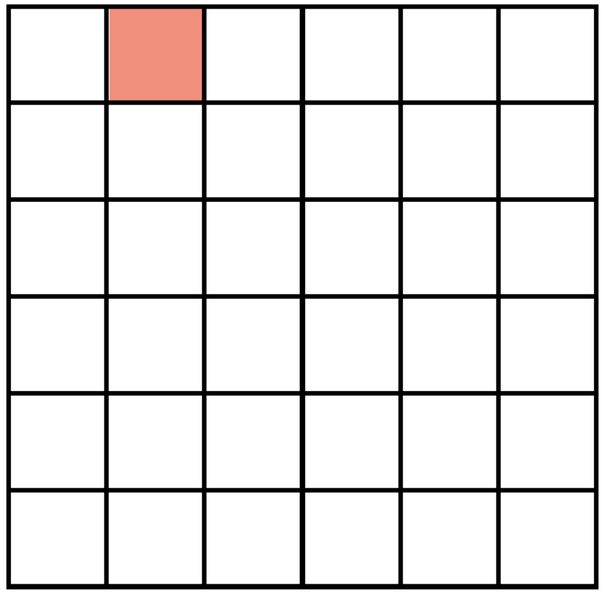


Convolution: Stride

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



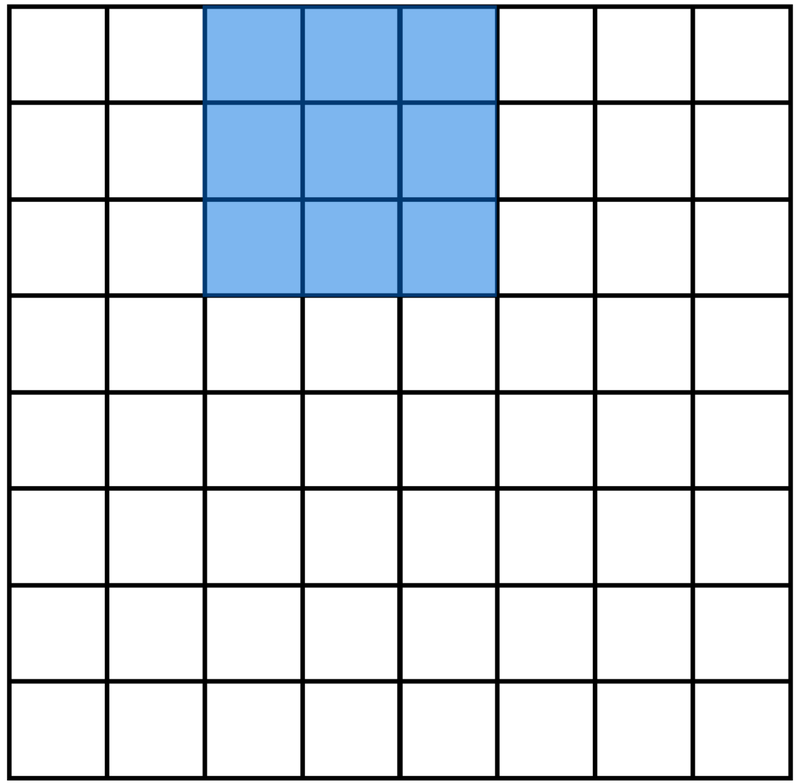
Input



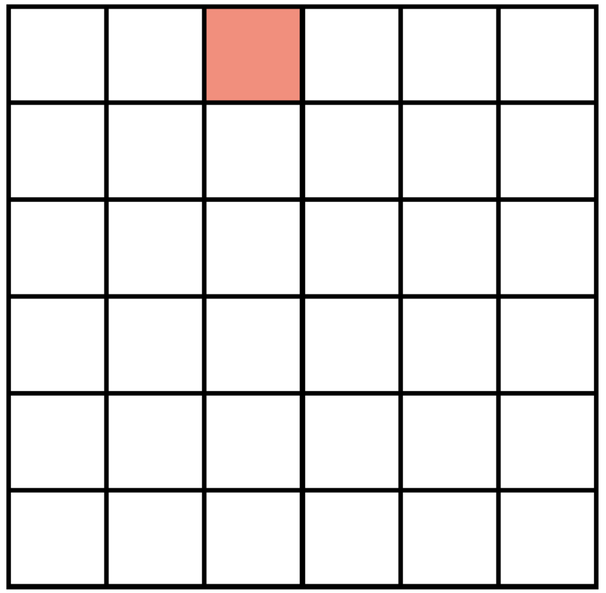
Output

Convolution: Stride

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



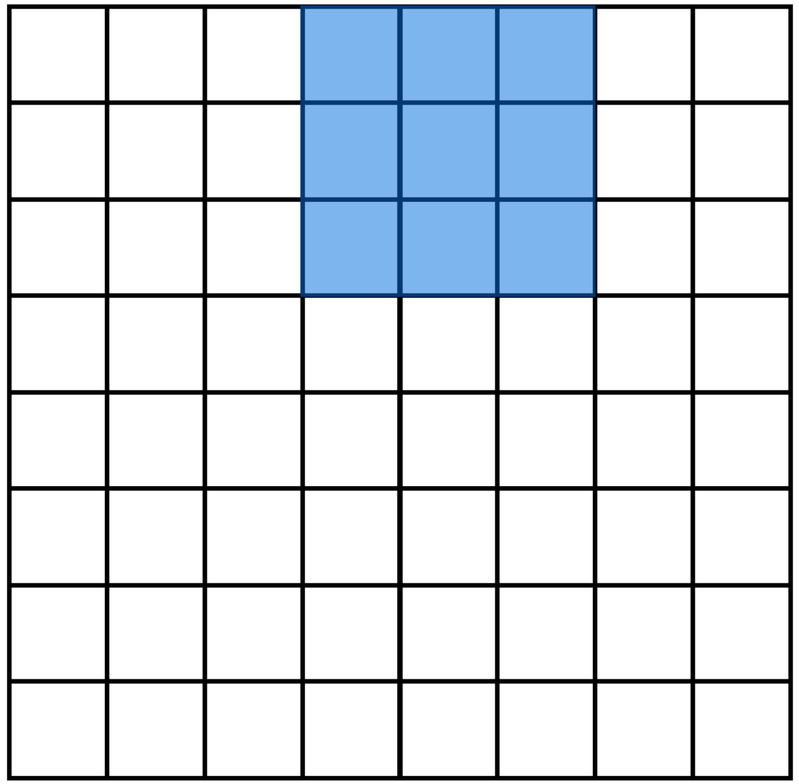
Input



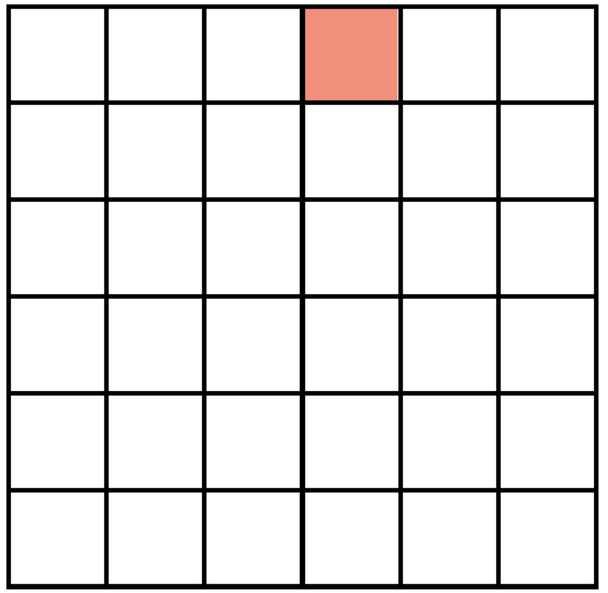
Output

Convolution: Stride

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



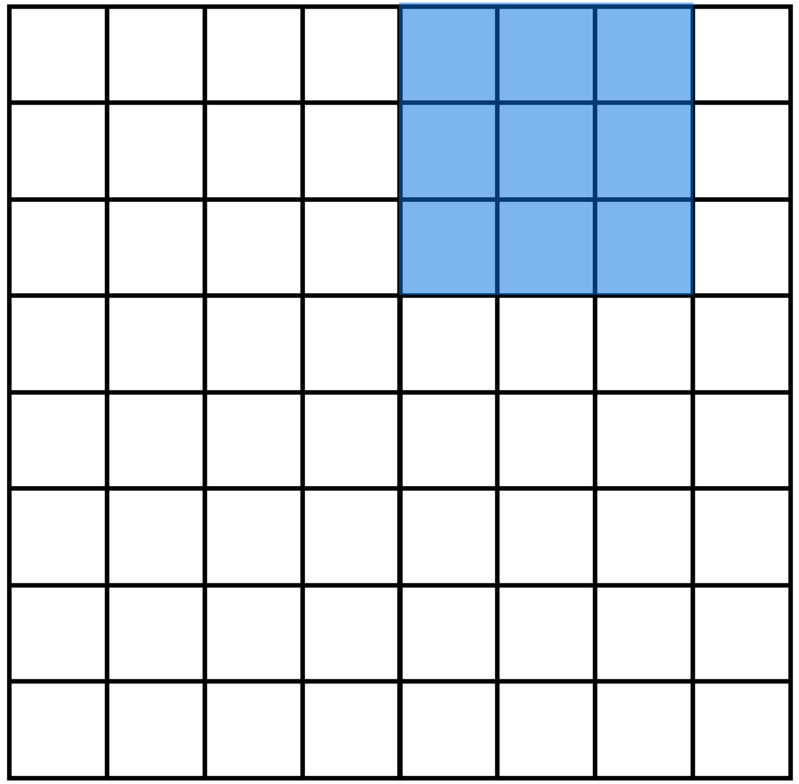
Input



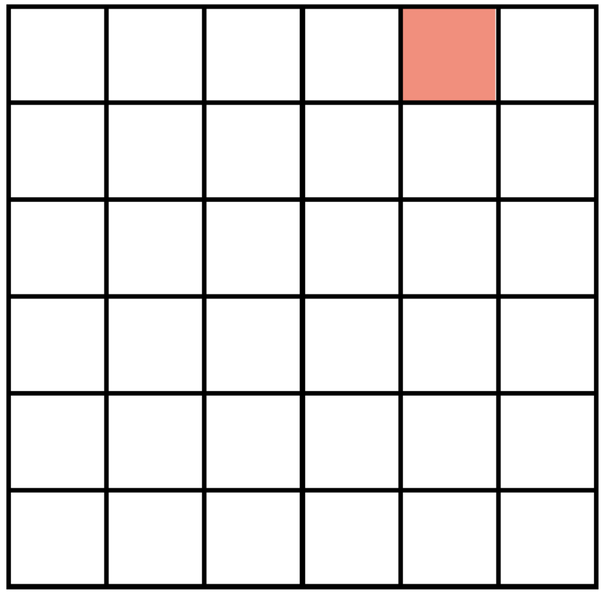
Output

Convolution: Stride

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



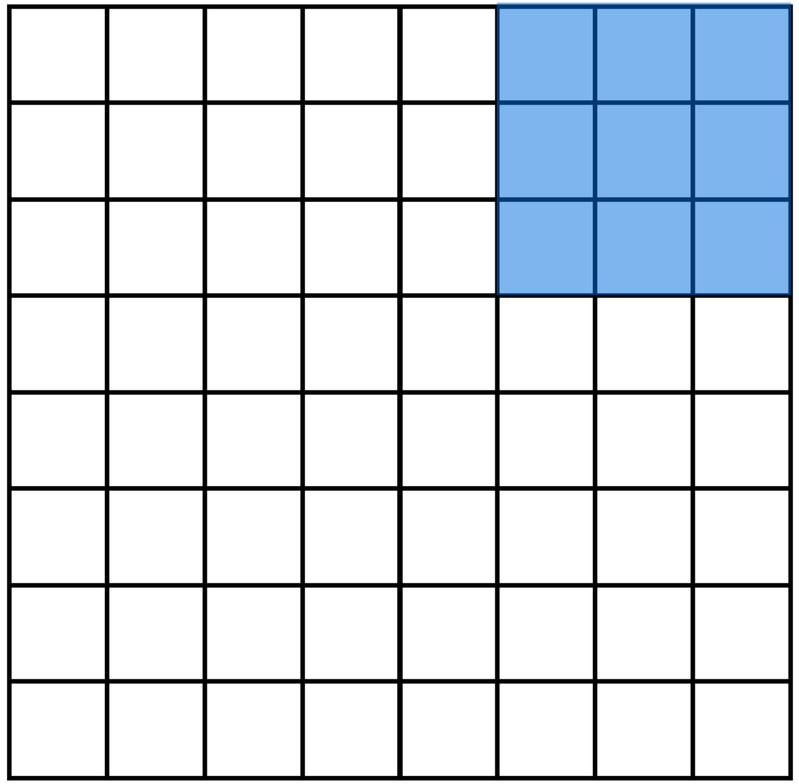
Input



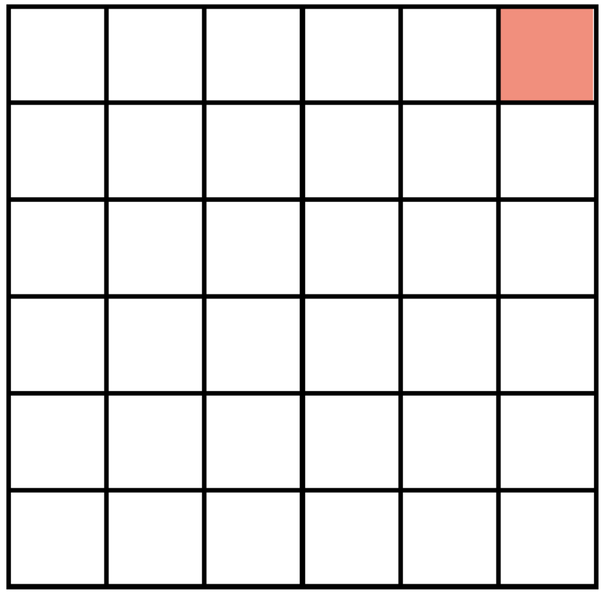
Output

Convolution: Stride

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



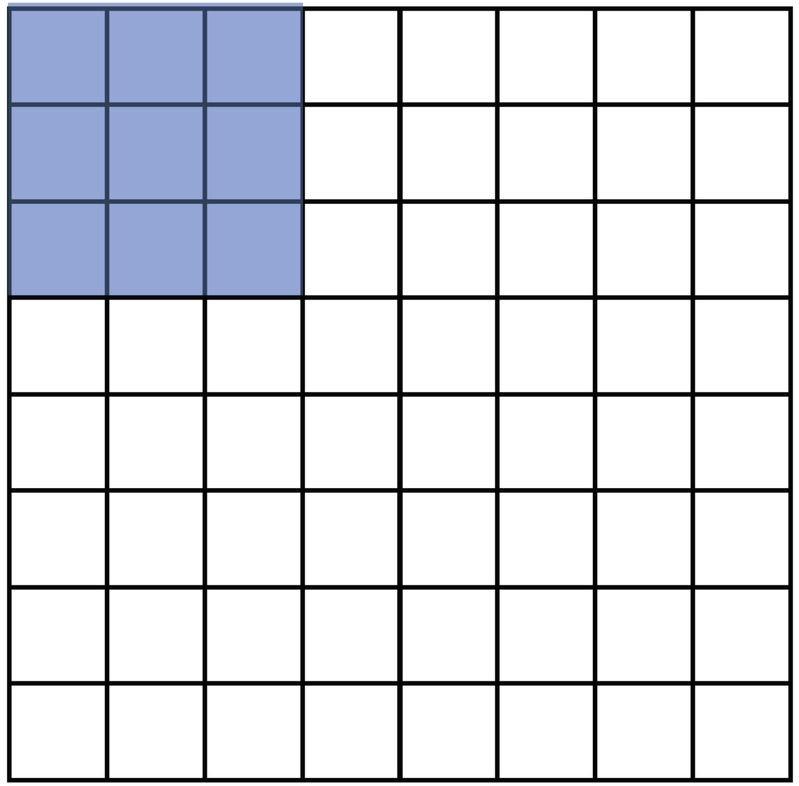
Input



Output

Convolution: Stride

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



Input

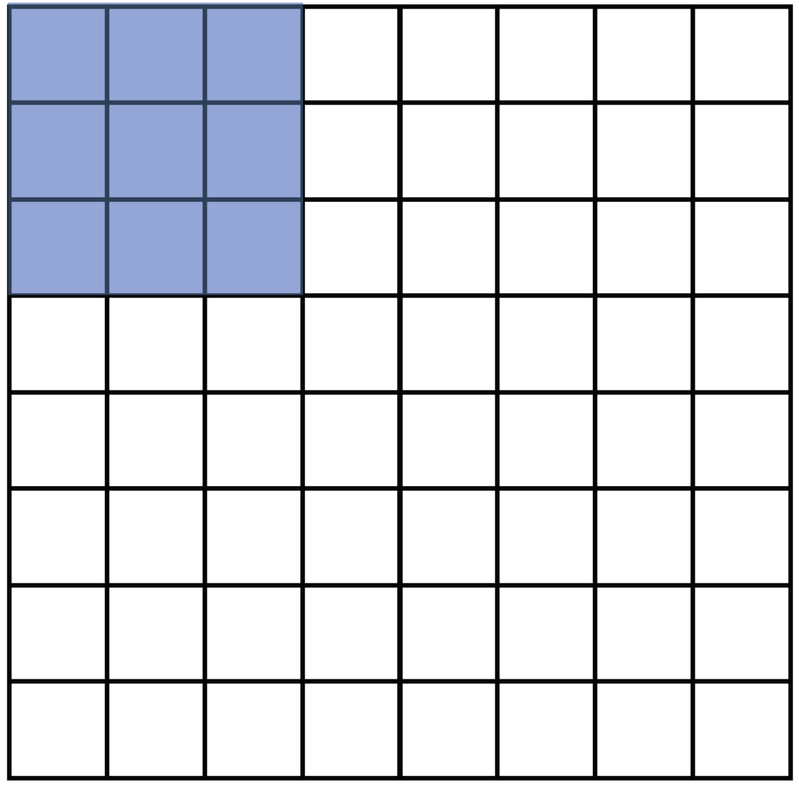
Recall that at each position, we are doing a **3D** sum:

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

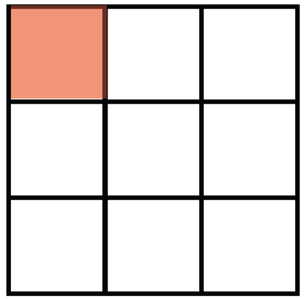
(channel, row, column)

Convolution: Stride

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



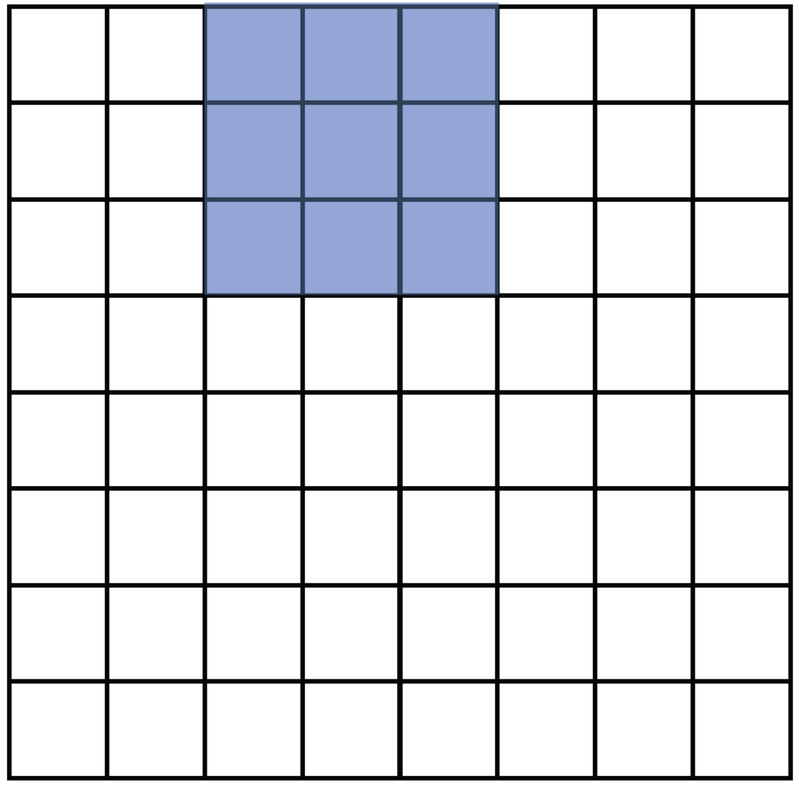
Input



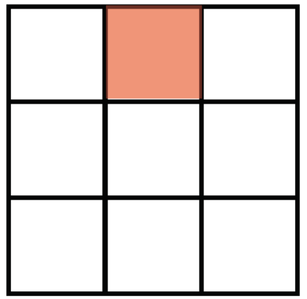
Output

Convolution: Stride

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



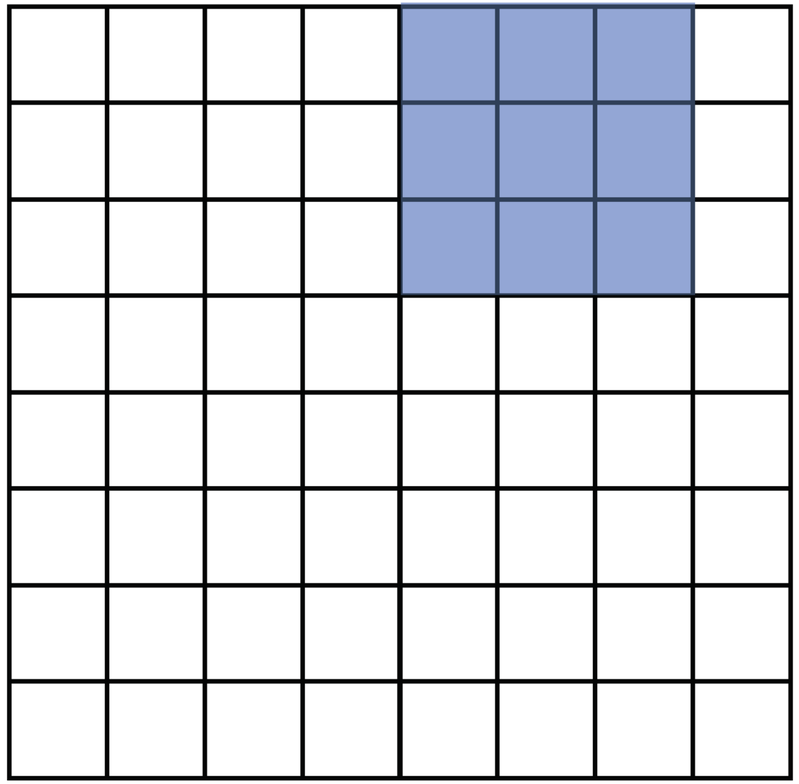
Input



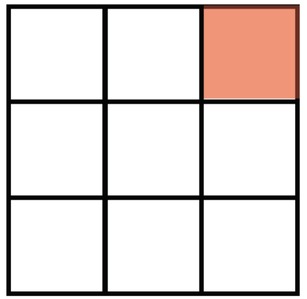
Output

Convolution: Stride

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



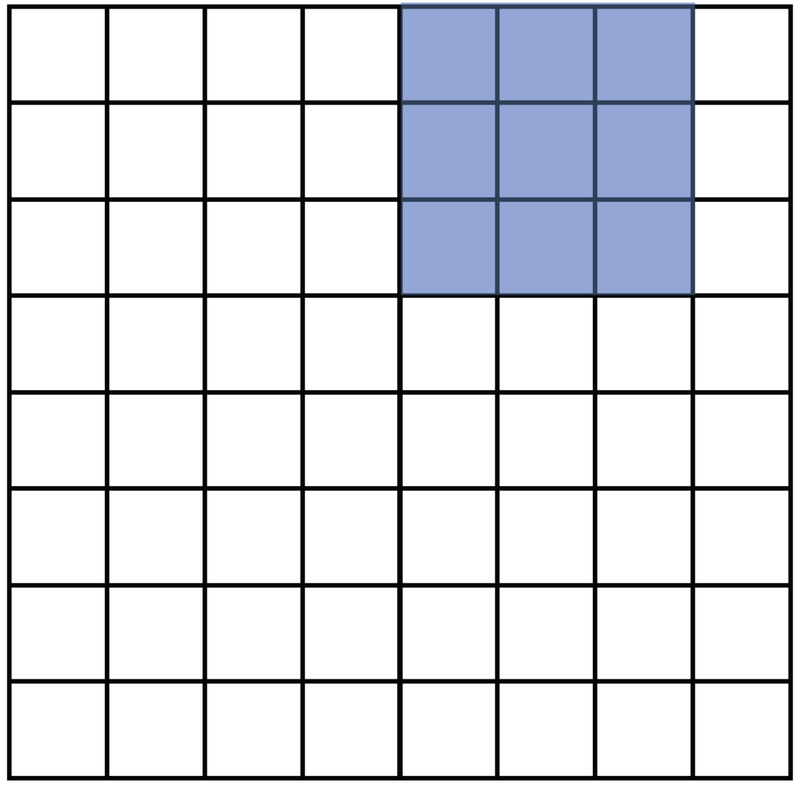
Input



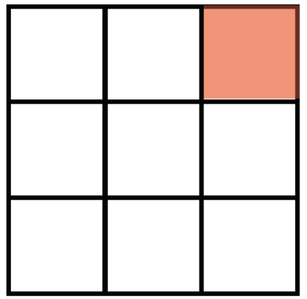
Output

Convolution: Stride

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



Input



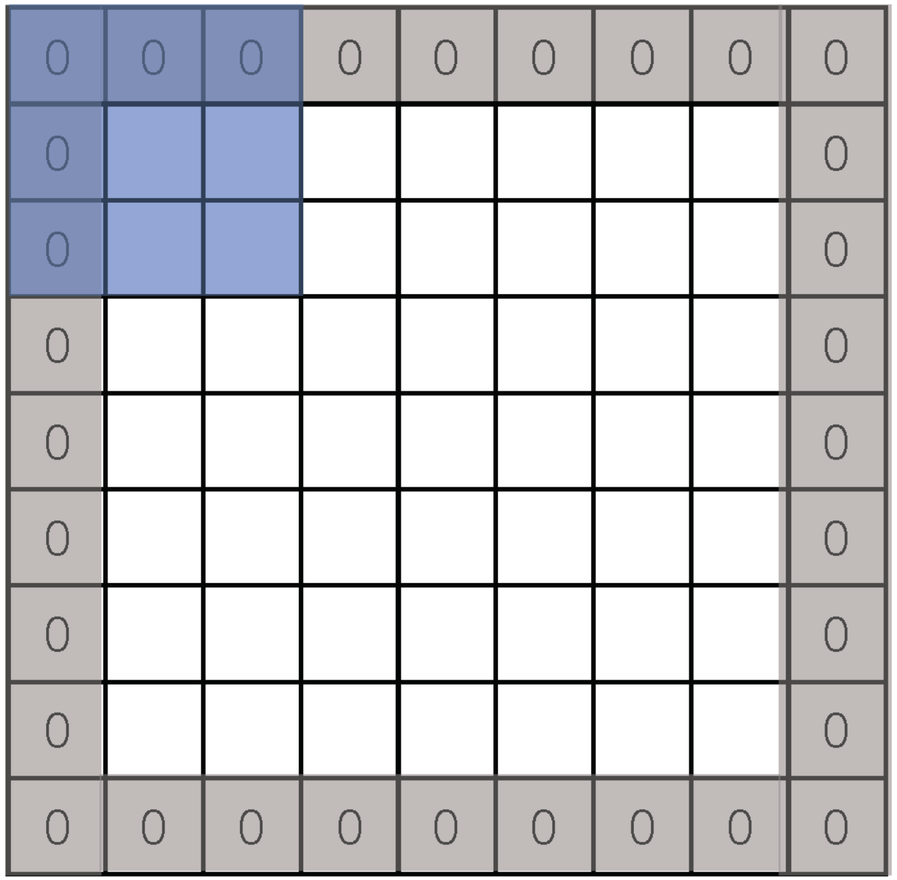
Output

- Notice that with certain strides, we may not be able to cover all of the input

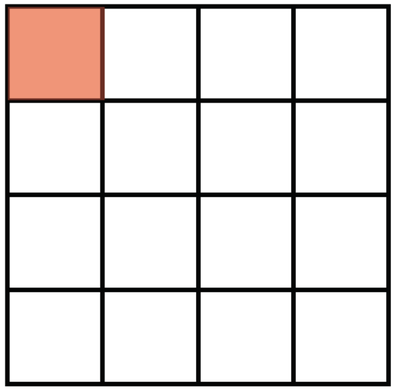
- The output is also half the size of the input

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



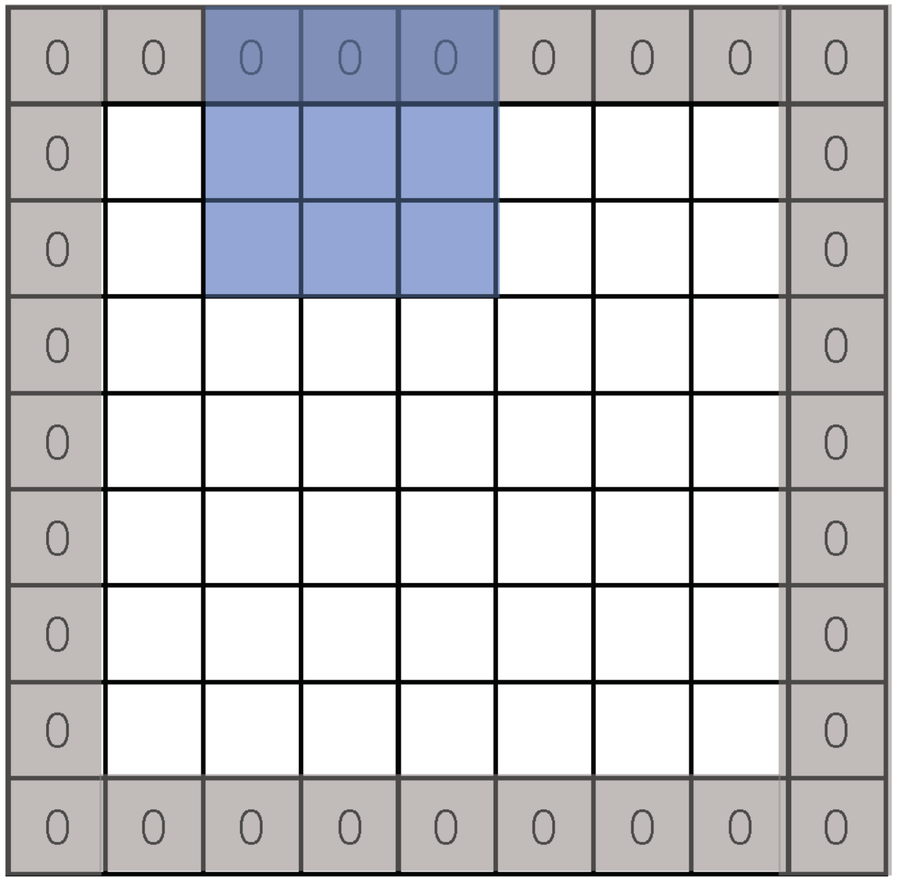
Input



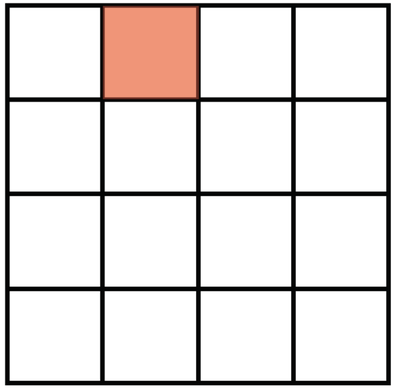
Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



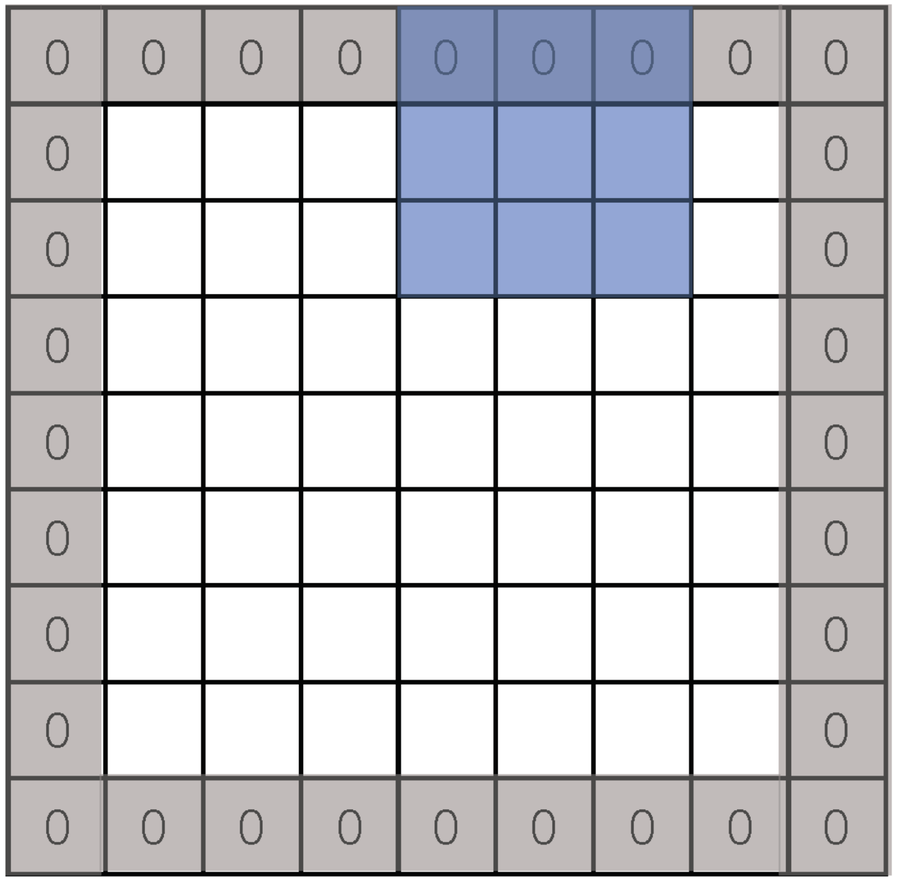
Input



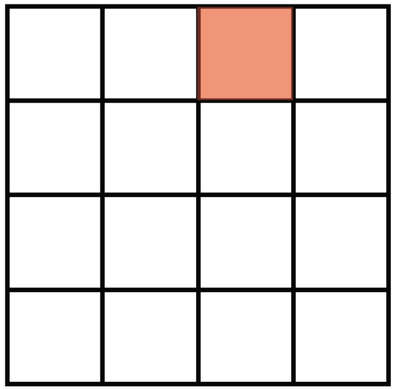
Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



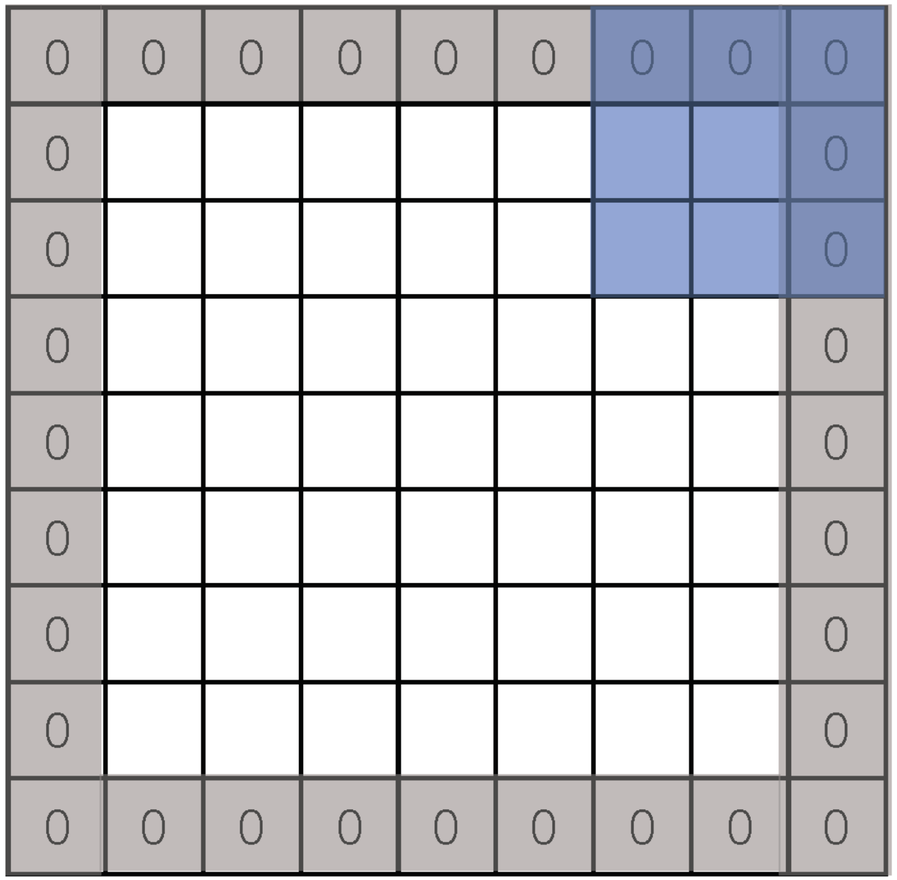
Input



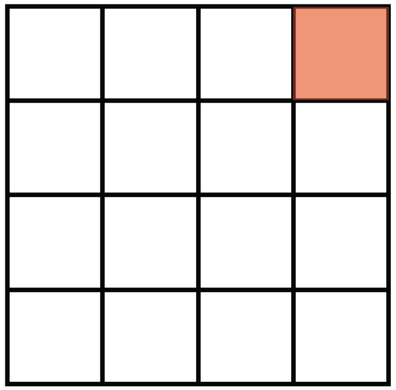
Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



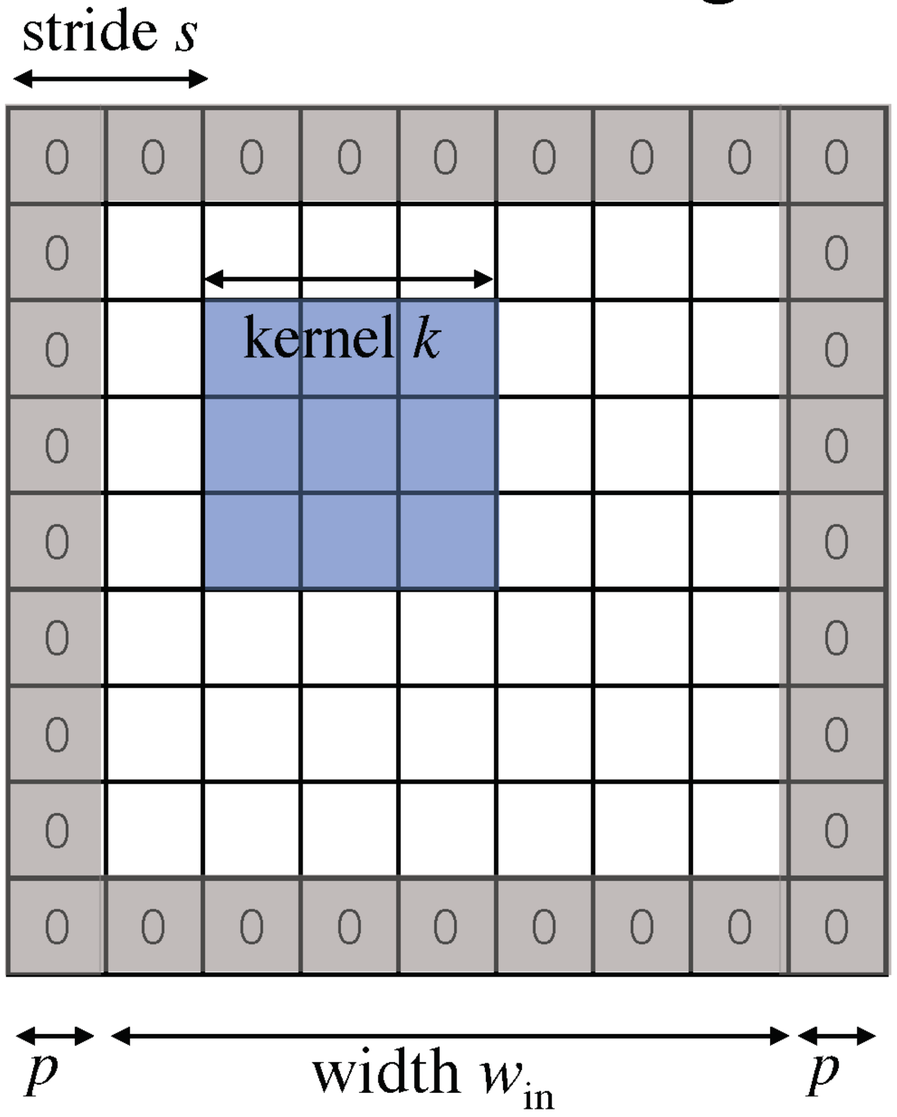
Input



Output

Convolution:

How big is the output?

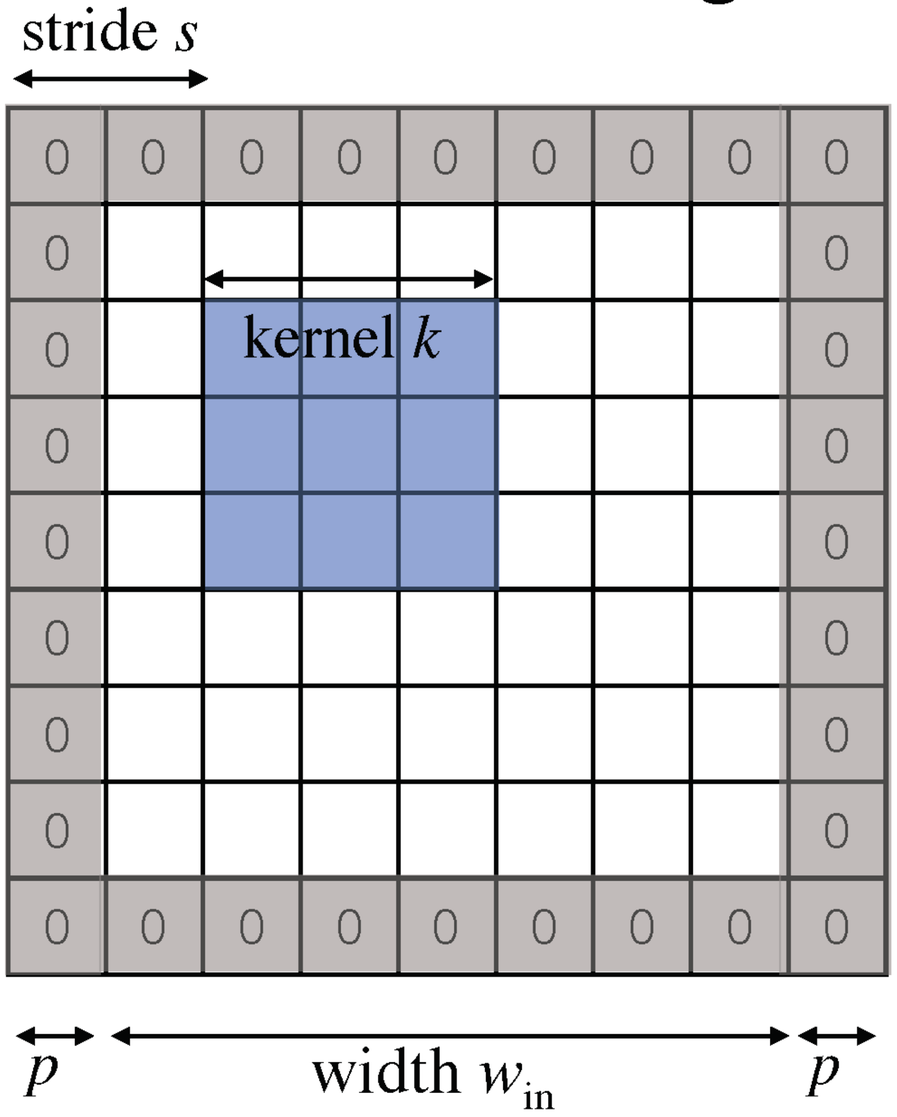


In general, the output has size:

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

Convolution:

How big is the output?



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

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

$$= \left\lfloor \frac{w_{\text{in}} + 2 - 3}{1} \right\rfloor + 1$$

$$= w_{\text{in}}$$

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?

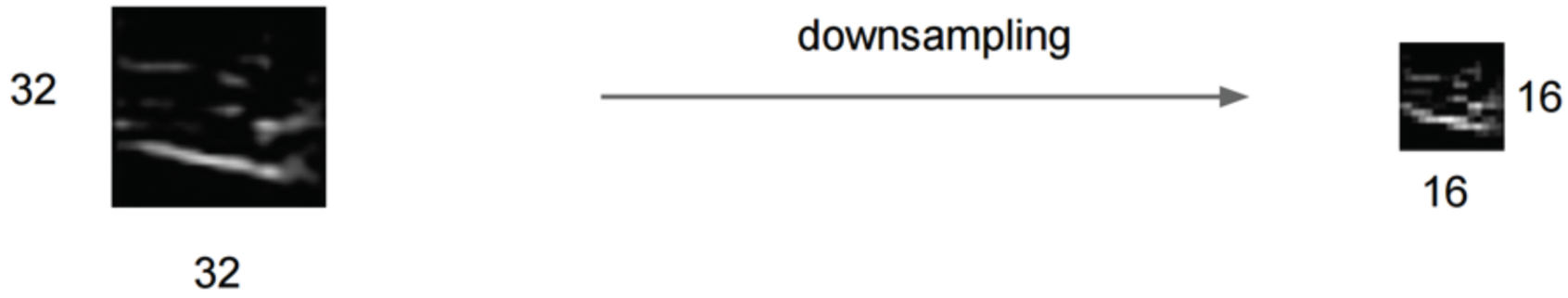
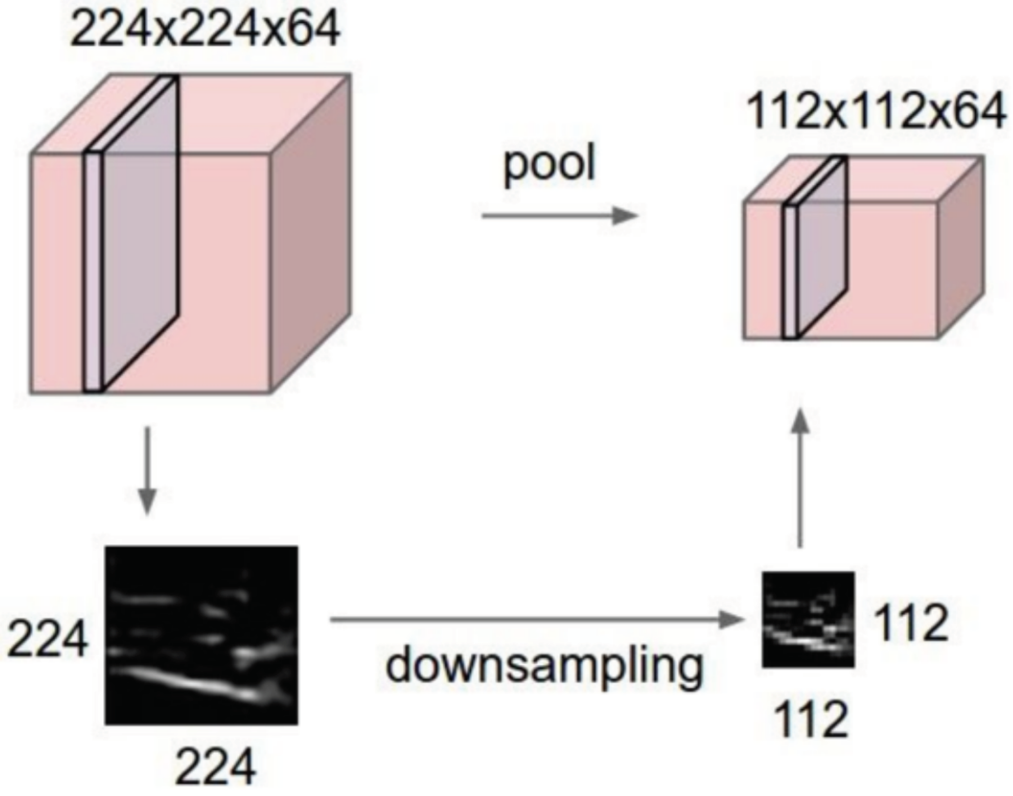


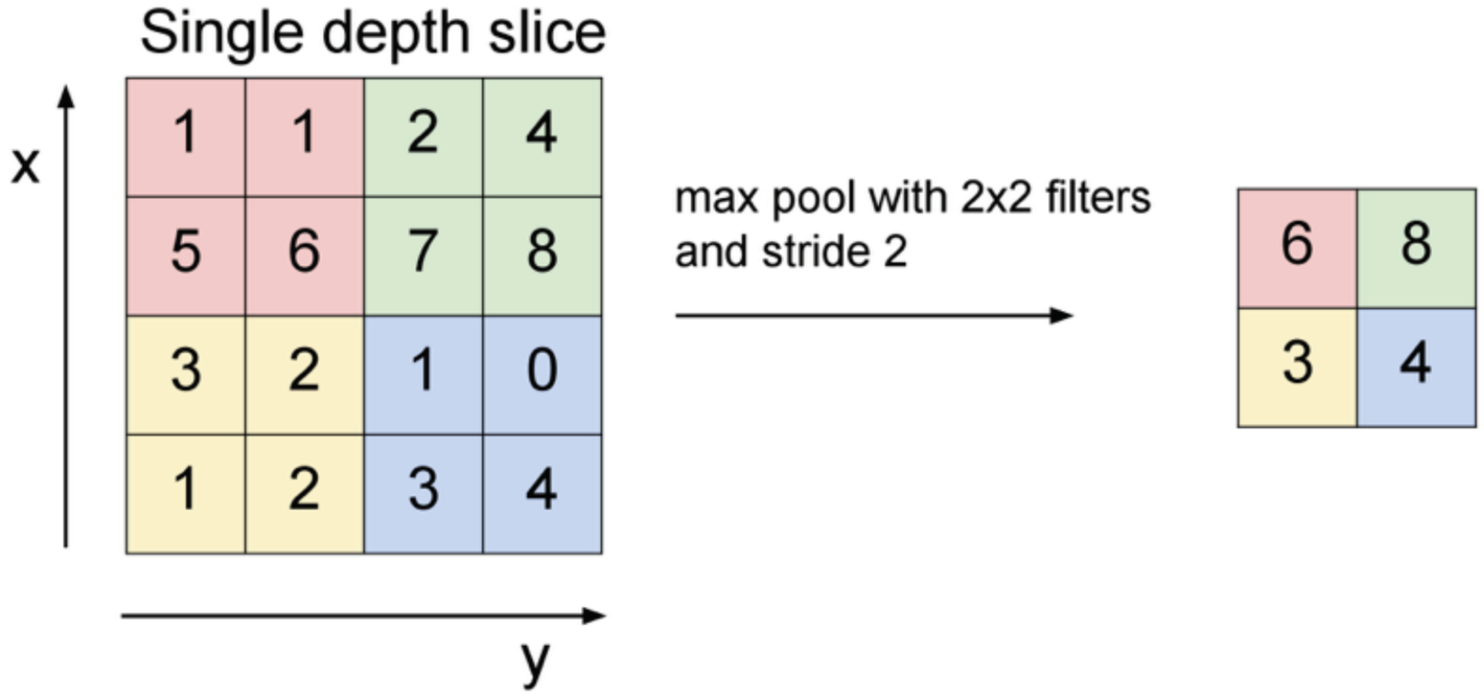
Figure: Andrej Karpathy

Pooling

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



Max Pooling

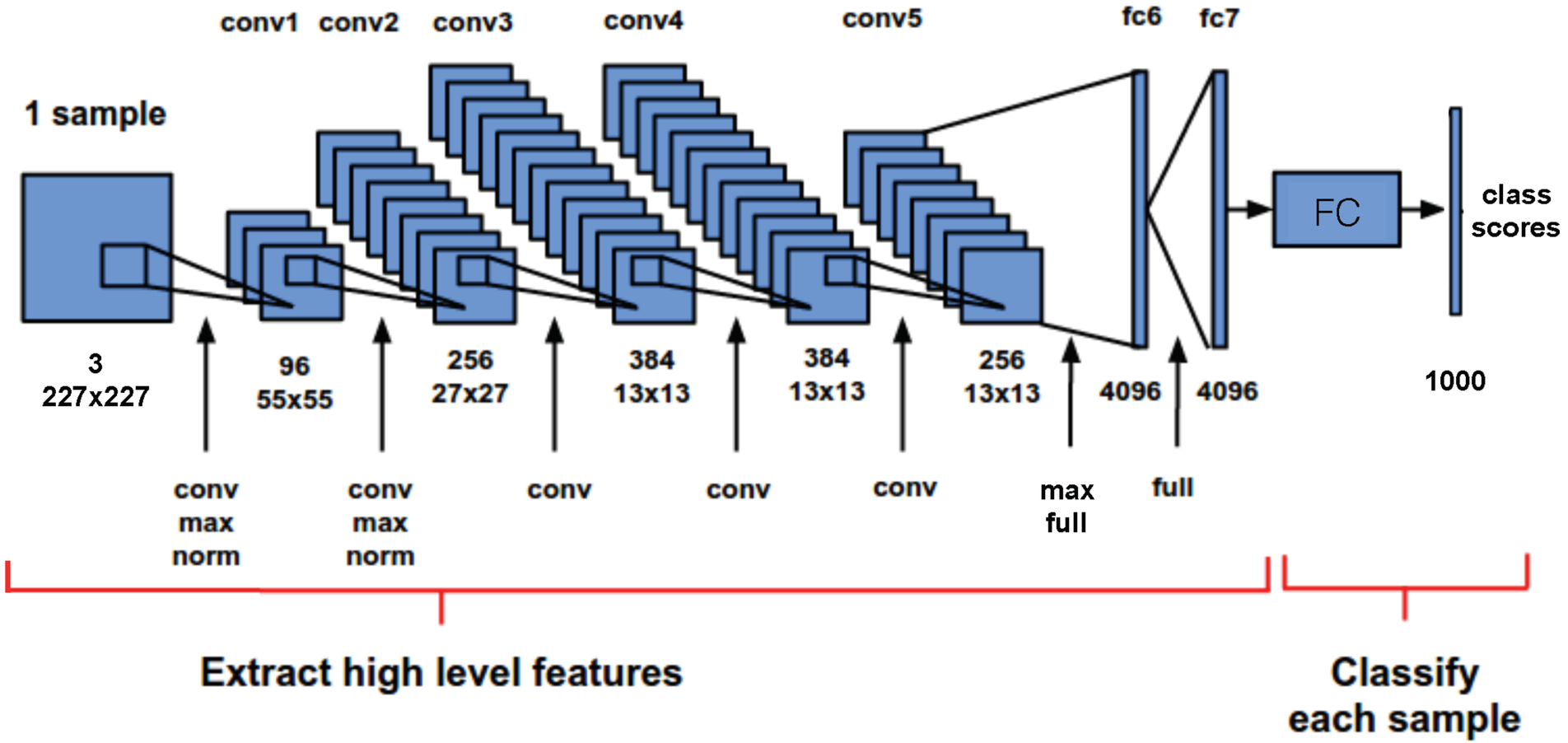


What's the backprop rule for max pooling?

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

Figure: Andrej Karpathy

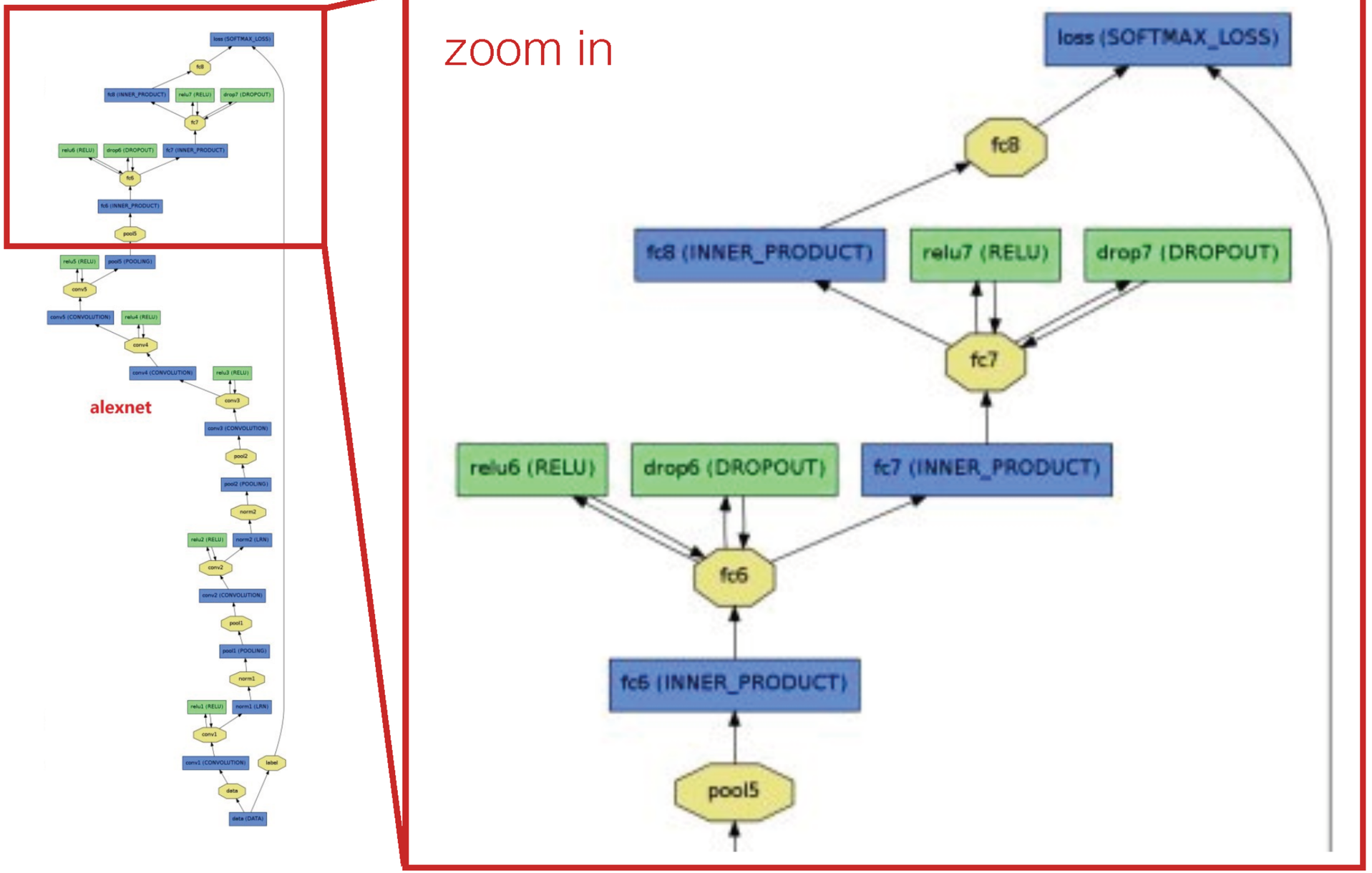
Example: AlexNet [Krizhevsky 2012]



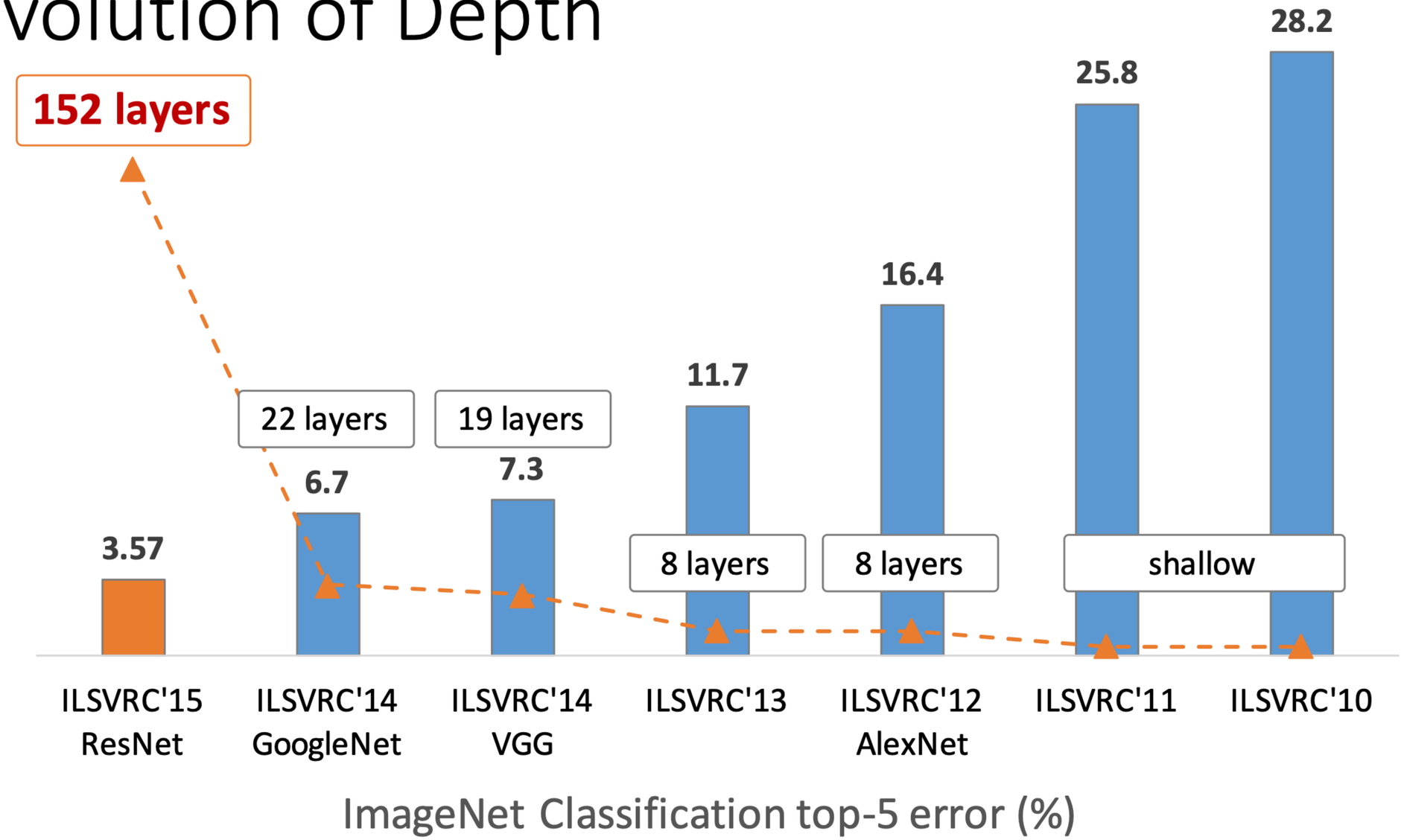
"max": max pooling
"norm": local response normalization
"full": fully connected

Figure: [Karnowski 2015] *(with corrections)*

Example: AlexNet [Krizhevsky 2012]

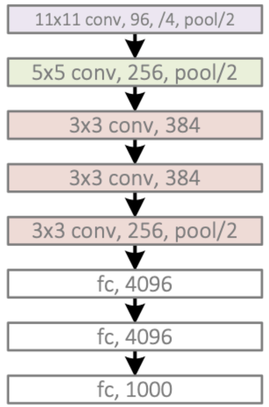


Revolution of Depth

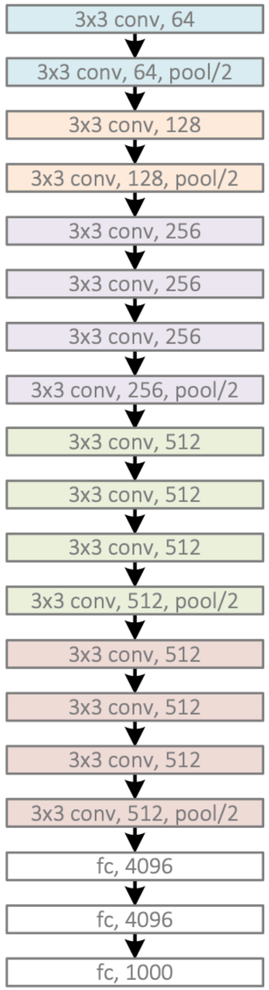


Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)

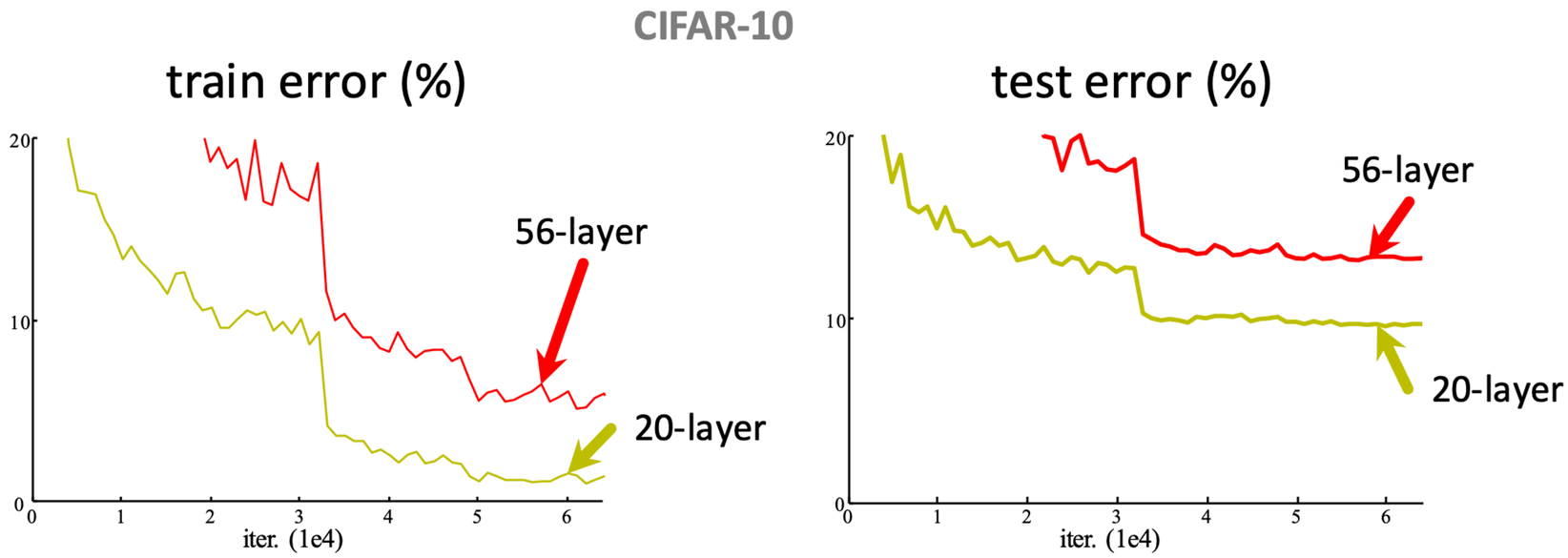


GoogleNet, 22 layers
(ILSVRC 2014)

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

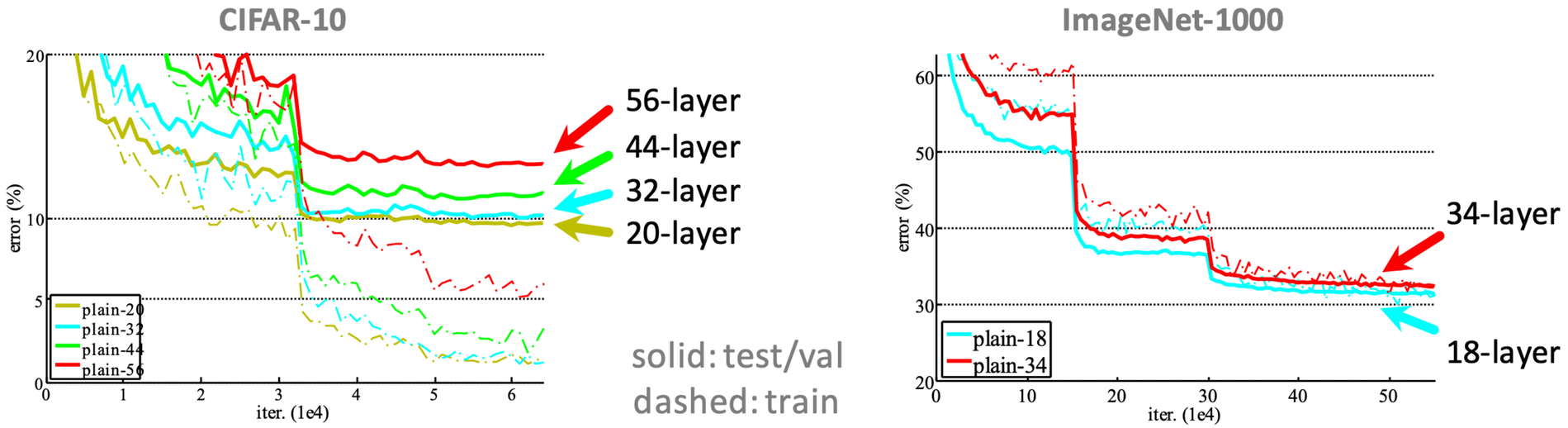
Going deeper?

- Is learning better networks as simple as stacking more layers?



Going deeper?

- Is learning better networks as simple as stacking more layers?

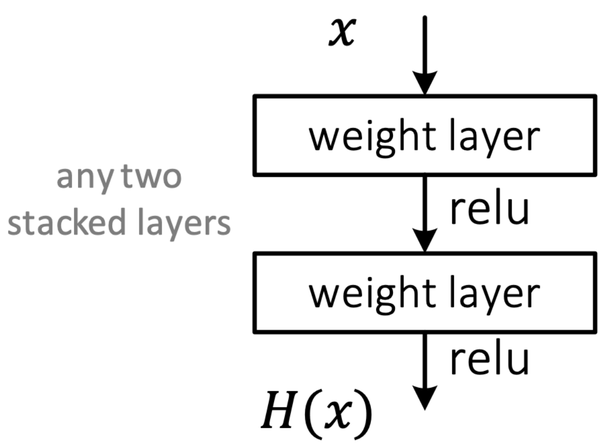


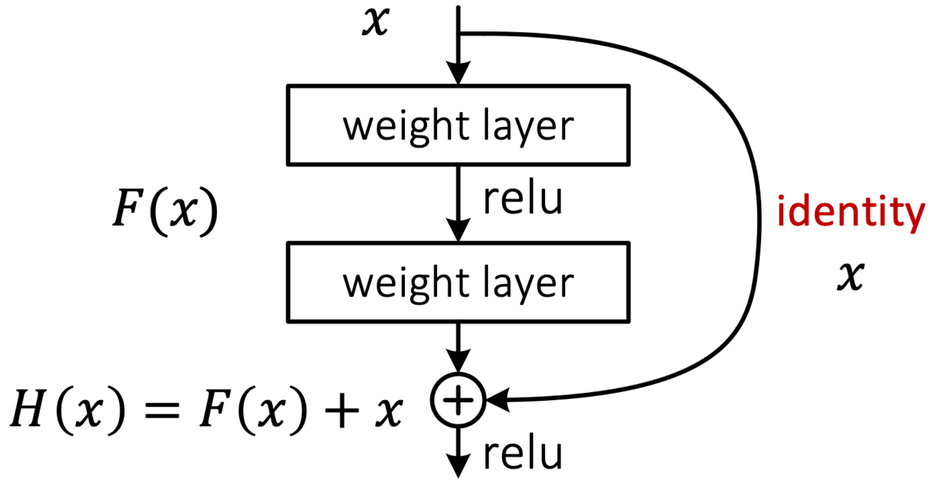
Going deeper?

- Is learning better networks as simple as stacking more layers?
- Richer solution space
- Gradient vanishing / explosion
- Solvers cannot find the solution when going deeper

Going deeper?

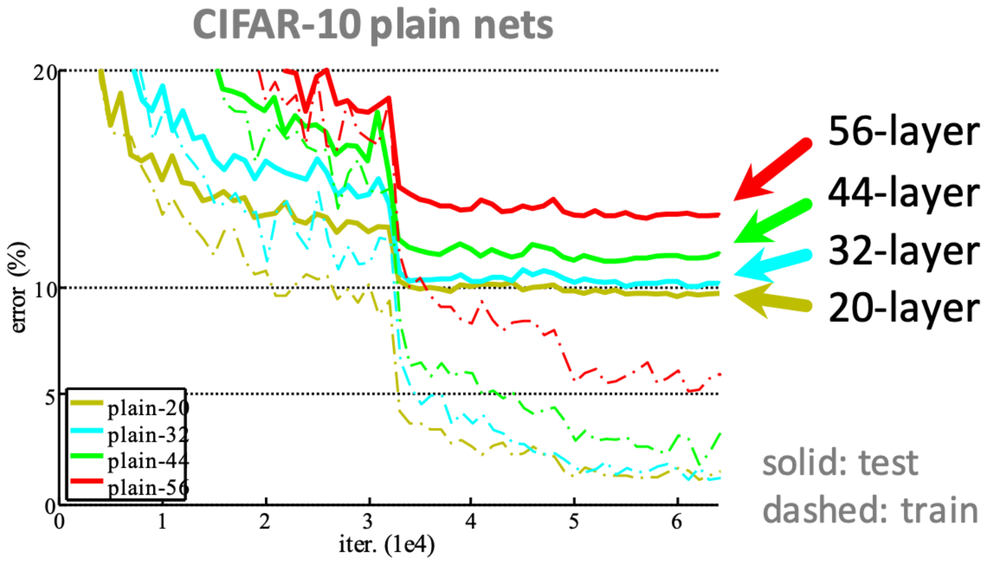
- A solution by construction: add shortcuts to the model.

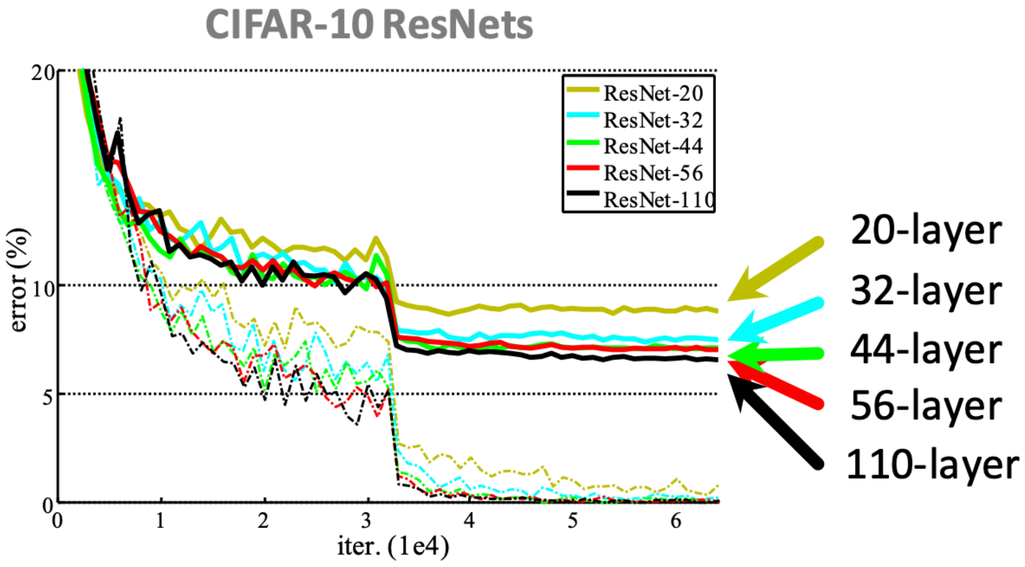




ResNet

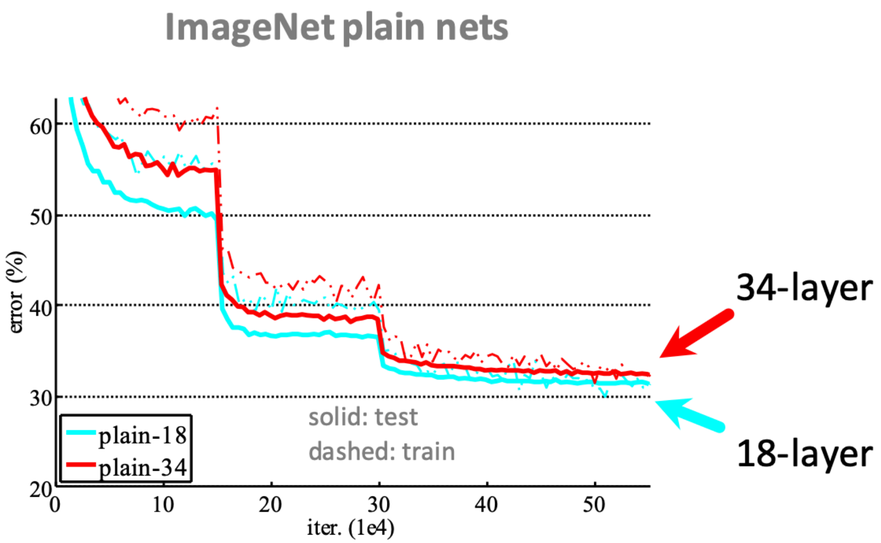
- Deeper ResNets have lower training error and also lower test error.

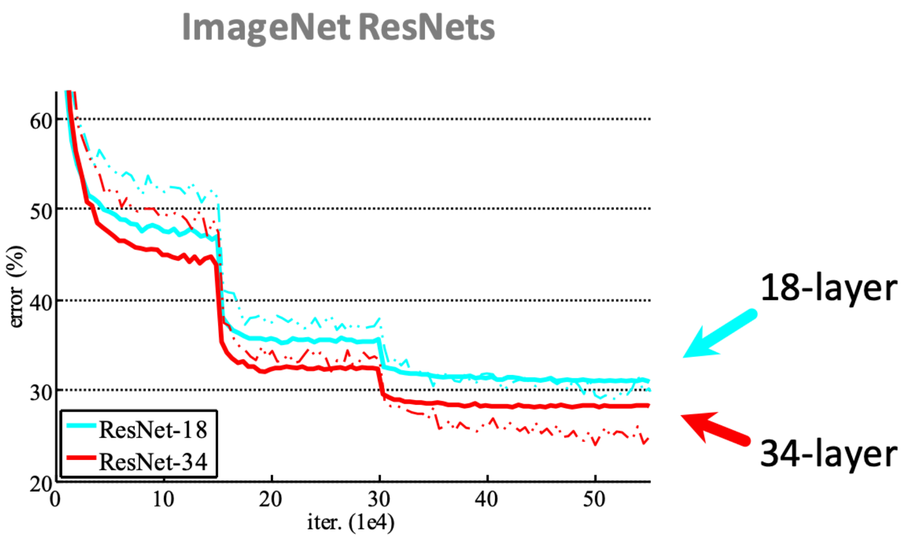




ResNet

- Deeper ResNets have lower training error and also lower test error.





Acknowledgement

- Many contents are taken from Stanford CS231N, CMU 16-385, TTIC 31040, and tutorial from Kaiming He