Neural Networks and Deep Learning

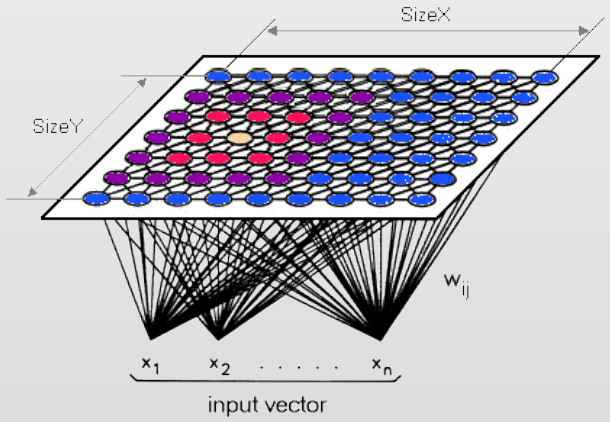
Charles Nicholson, Ph.D.

DSA/ISE 5103

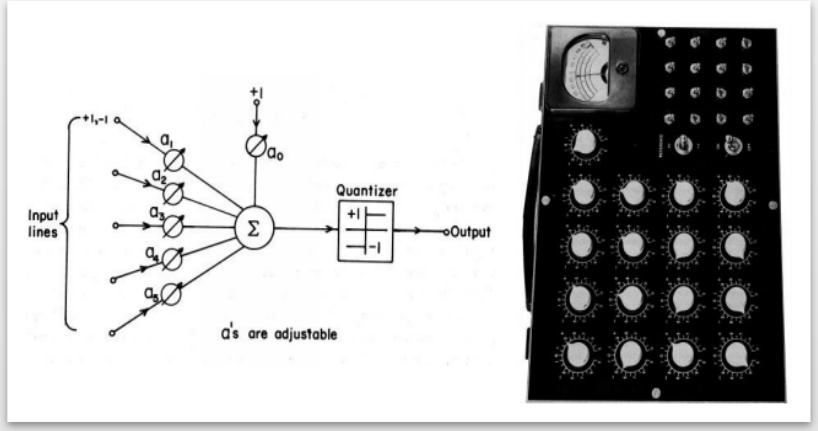
Some of this material is excerpted from or inspired by the resources at http://neuralnetworksanddeeplearning.com

Neural networks and deep learning quick history

- 40's, 50's, 60's prototypical concepts of artificial neural networks (ANNs) introduced

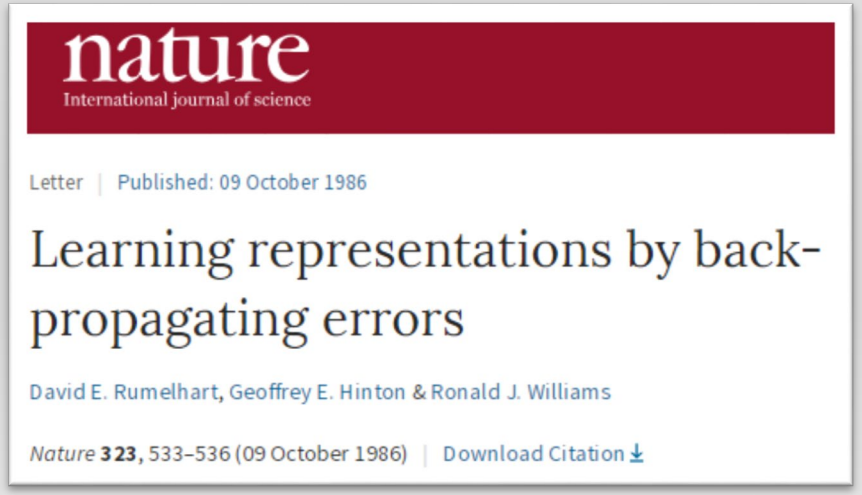


- Kohonen & Teuvo ('82) Self-Organizing Maps: ANN unsupervised approach for visualization of high-dimensional data



Widrow and Hoff, 1960's: Adaline/Madaline

- Recurrent Neural Networks – early 80's
- Rumelhart et al. 86 – great early success
- Interest subsides a bit in the late 90's as other models are introduced, e.g., SVMs


nature
International journal of science

Letter | Published: 09 October 1986

Learning representations by back-propagating errors

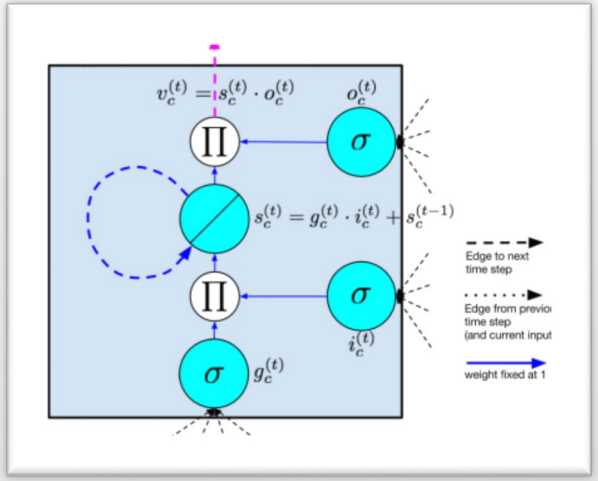
David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Nature **323**, 533–536 (09 October 1986) | Download Citation

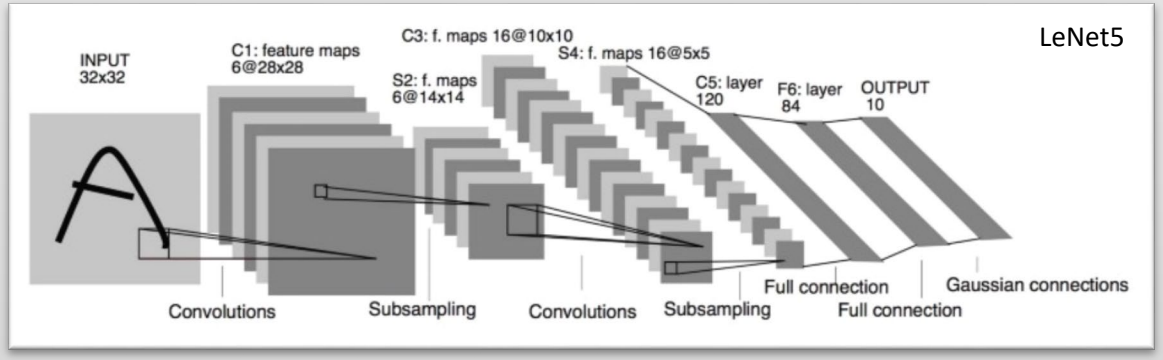
16,278 citations!

Neural networks and deep learning quick history

- Long-short term memory (LSTM) 1997, major impact in speech, text recognition



- Convolutional Neural Nets (CNN) –LeCun 1998



- 2012 -> Successes with supervised approaches overcoming vanishing gradient, etc. and more general applications: Use of GPU; Introduction of AlexNet (cited 32,000+ times!)

Characteristics of neural networks

- Ability to fit any function ← ***extremely flexible***
- Computationally, the training can be time-consuming
- Active area of research, especially in “Deep learning”
- Individual relations between input and the output variables are not developed by expert judgment ← ***classic black box***
- ***Need a lot of data***
- But, deep learning ***can benefit from lots of data***

Characteristics of neural networks

- Highly successful applications:
  - Pattern recognition, e.g. handwritten characters or face identification
  - Speech recognition
  - Human emotion detection
  - Automatic coloring of B&W images
  - Adding sounds to silent movies (http://news.mit.edu/2016/artificial-intelligence-produces-realistic-sounds-0613)
  - Object detection, identification, and location in photos
  - Handwriting generation (http://www.cs.toronto.edu/~graves/handwriting.cgi?text=hello+my+friend%21&style=&bias=0.15&samples=3 and https://arxiv.org/abs/1308.0850)

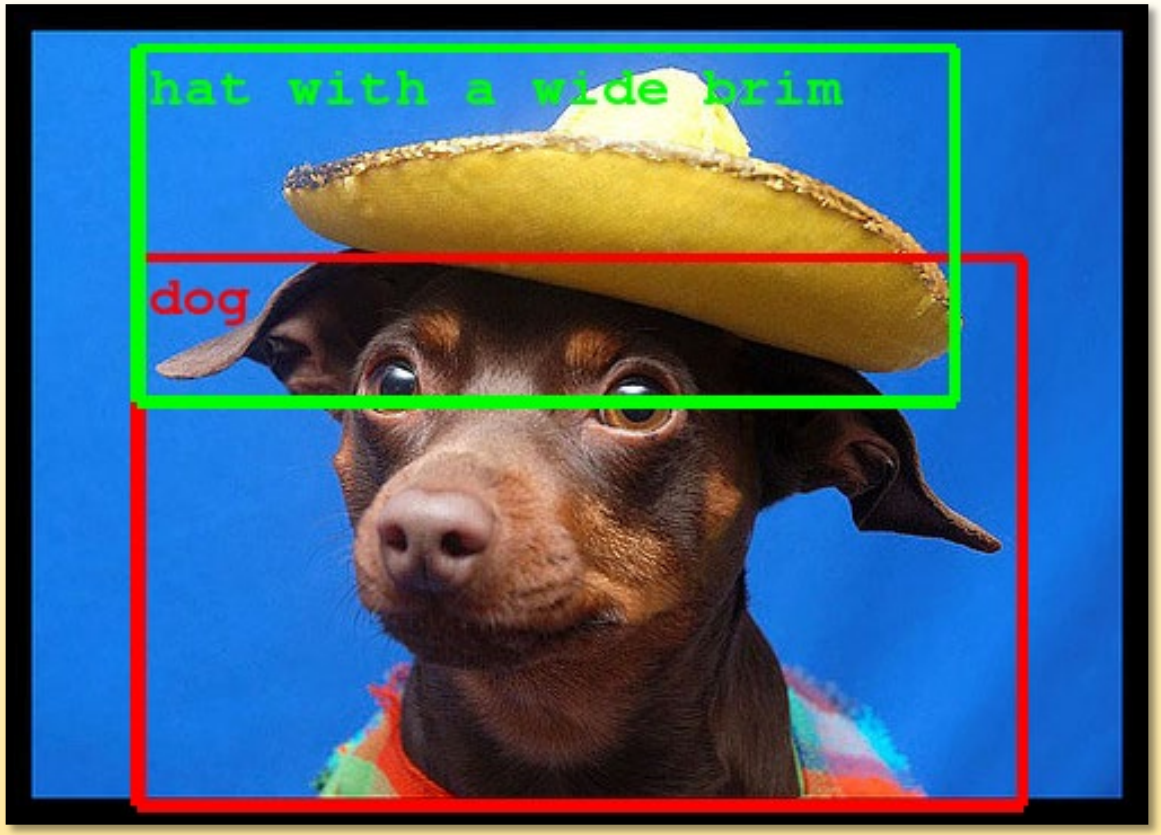
Example: Automatic Colorization of B&W Photos

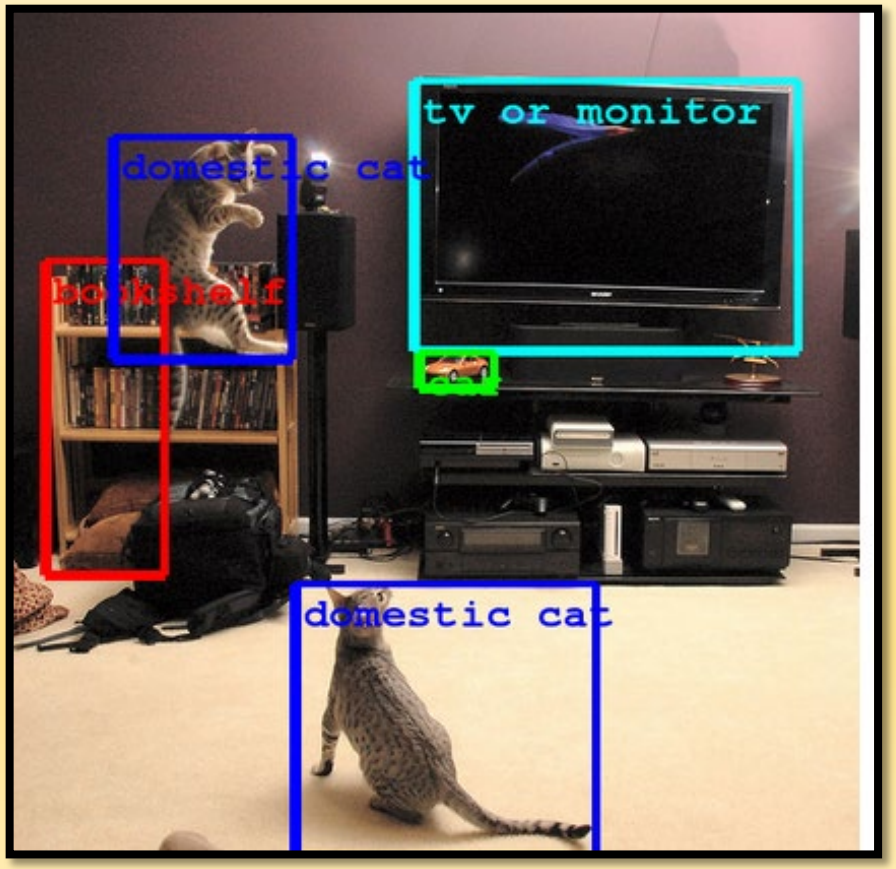
- neural networks used to [troll the subreddit /r/Colorization](#) - a community where people colorize historical black and white images manually using Photoshop
- Results were astonishing - what could take up to a month of manual labor could now be done in just a few seconds
  - See original work and publication list at http://richzhang.github.io/colorization/
  - Also check out: http://demos.algorithmia.com/colorize-photos/

Example: Object detection and location in photos

Blog: research.googleblog.com/2014/09/building-deeper-understanding-of-images.html

Paper: http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf



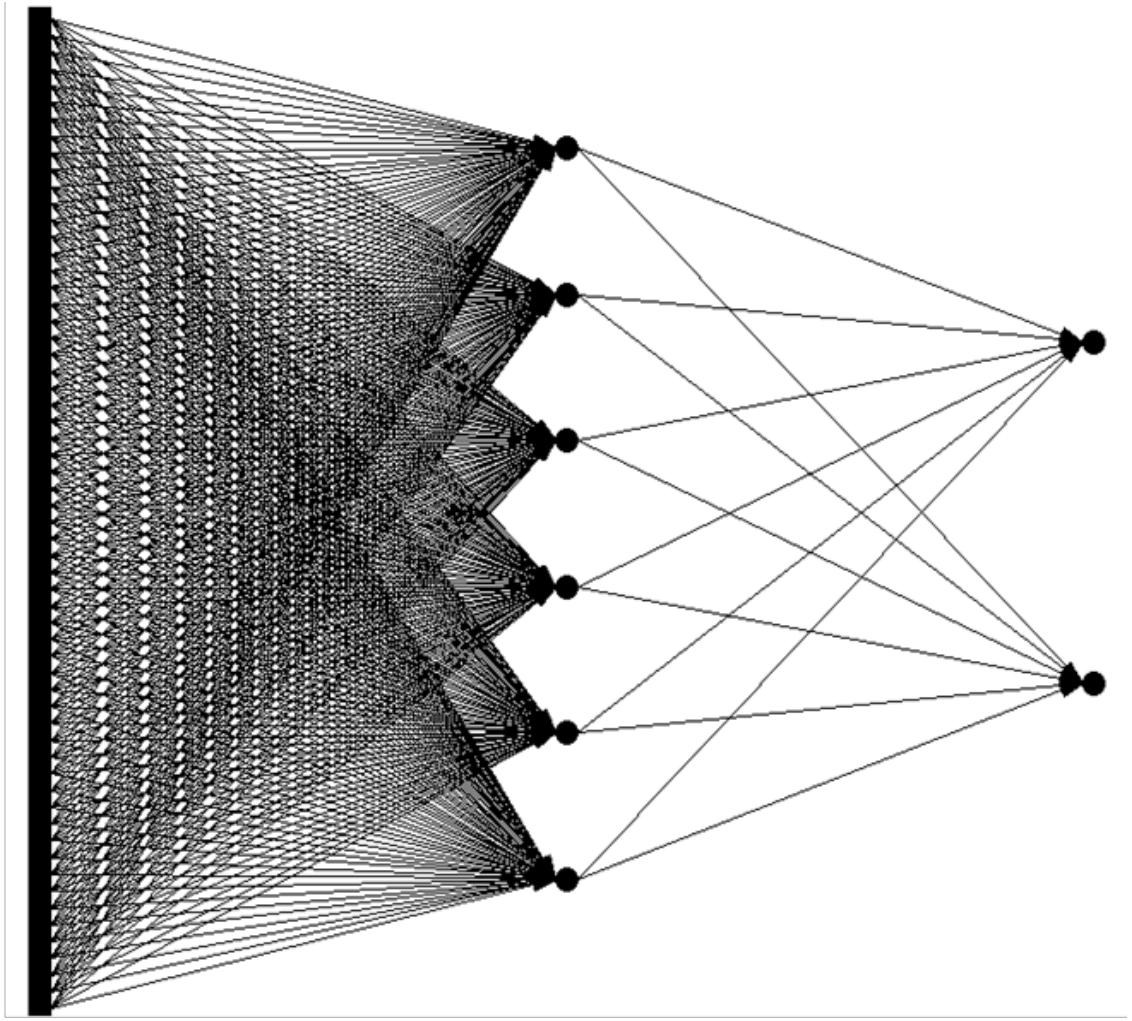


ImageNet large-scale visual recognition challenge

- annual event
- largest academic challenge in computer vision
- Goal: to test state-of-the-art technology in image understanding, both in the sense of recognizing objects in images and locating where they are

Agenda

- Motivation
- Explanation
- Training
- Deep Learning (DL)
- DL Architectures
- Implementation Tools

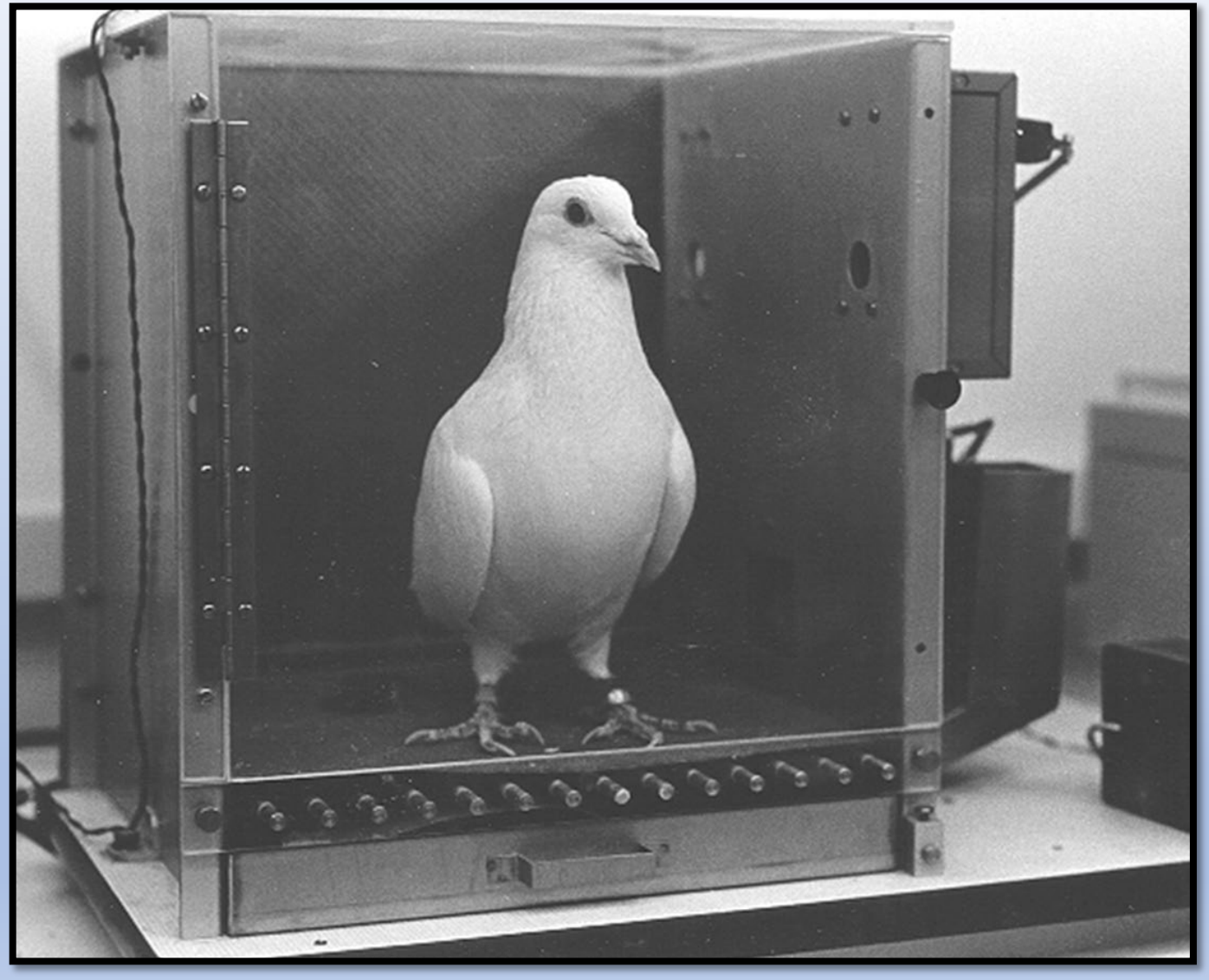


Pigeons in a box

(Watanabe *et al.* 1995)

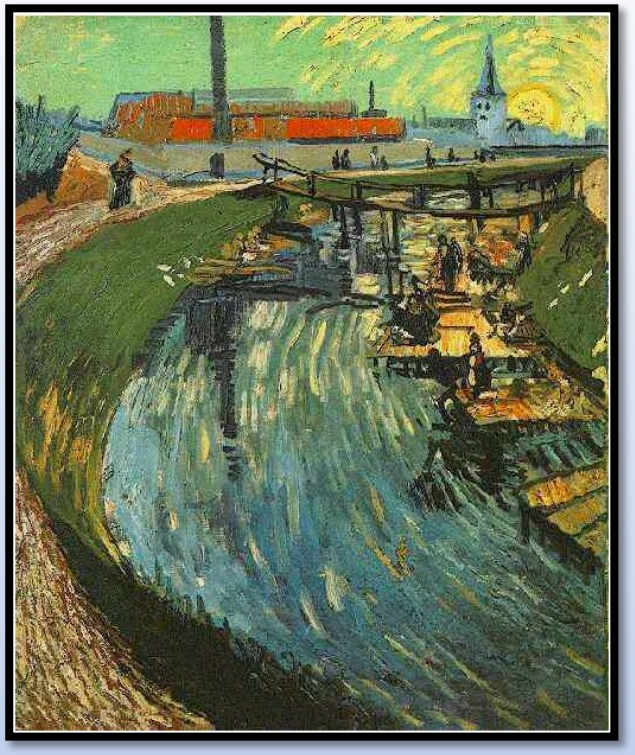
Experiment:

- Pigeon was placed in a closed box
- Present paintings of two different artists (e.g. Chagall / Van Gogh)
- Reward for pecking when presented a particular artist (e.g. Van Gogh)



Pigeons and Van Gogh

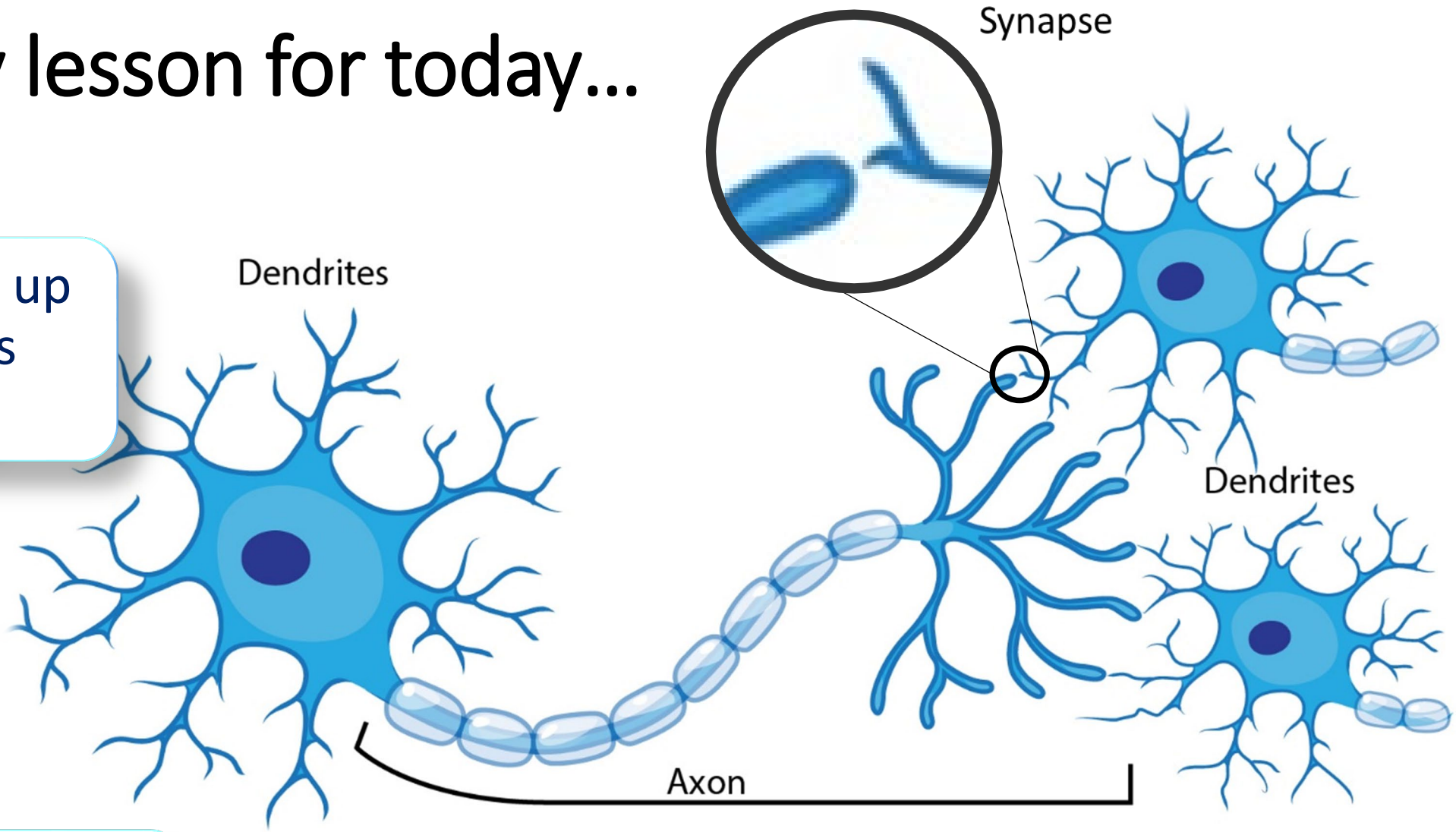
- Pigeons were able to discriminate between Van Gogh and Chagall with 95% accuracy (when presented with pictures they had been trained on)
- Discrimination still 85% successful for previously unseen paintings of the artists
- Did not simply memorize the pictures
  - but extract and recognize patterns (the ‘style’)
  - and generalize to make predictions
- This is what neural networks (biological and artificial) are good at (unlike conventional computer)



Biology

- Human brain is a dense interconnected network of approximately 86 billion neurons, each connected to, on average, 10,000 others
- Neuron activity is *excited* or *inhibited* through connections to other neurons

Biology lesson for today…



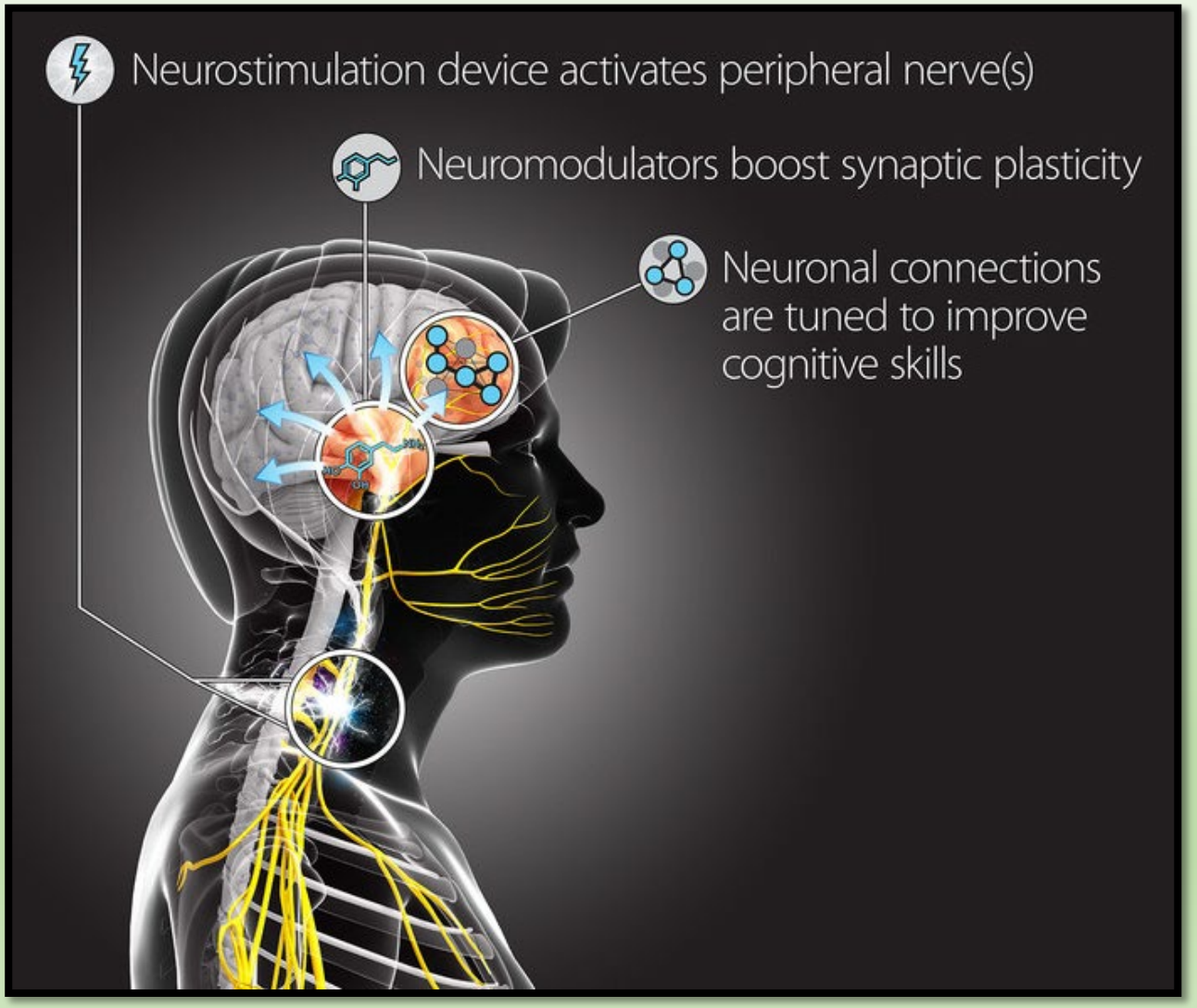
Input builds up on receptors (dendrites)

Cell has an input threshold

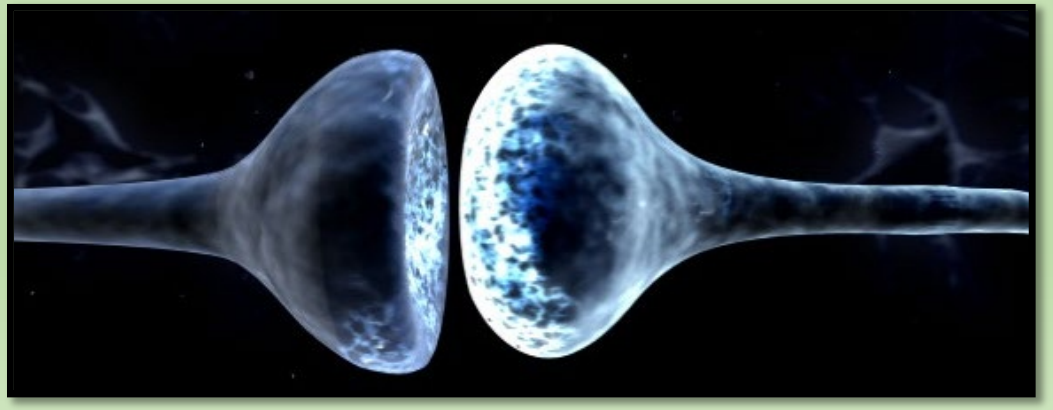
When threshold reached, activation is fired thru axon, to dendrites of other cells across the synapse

Biology

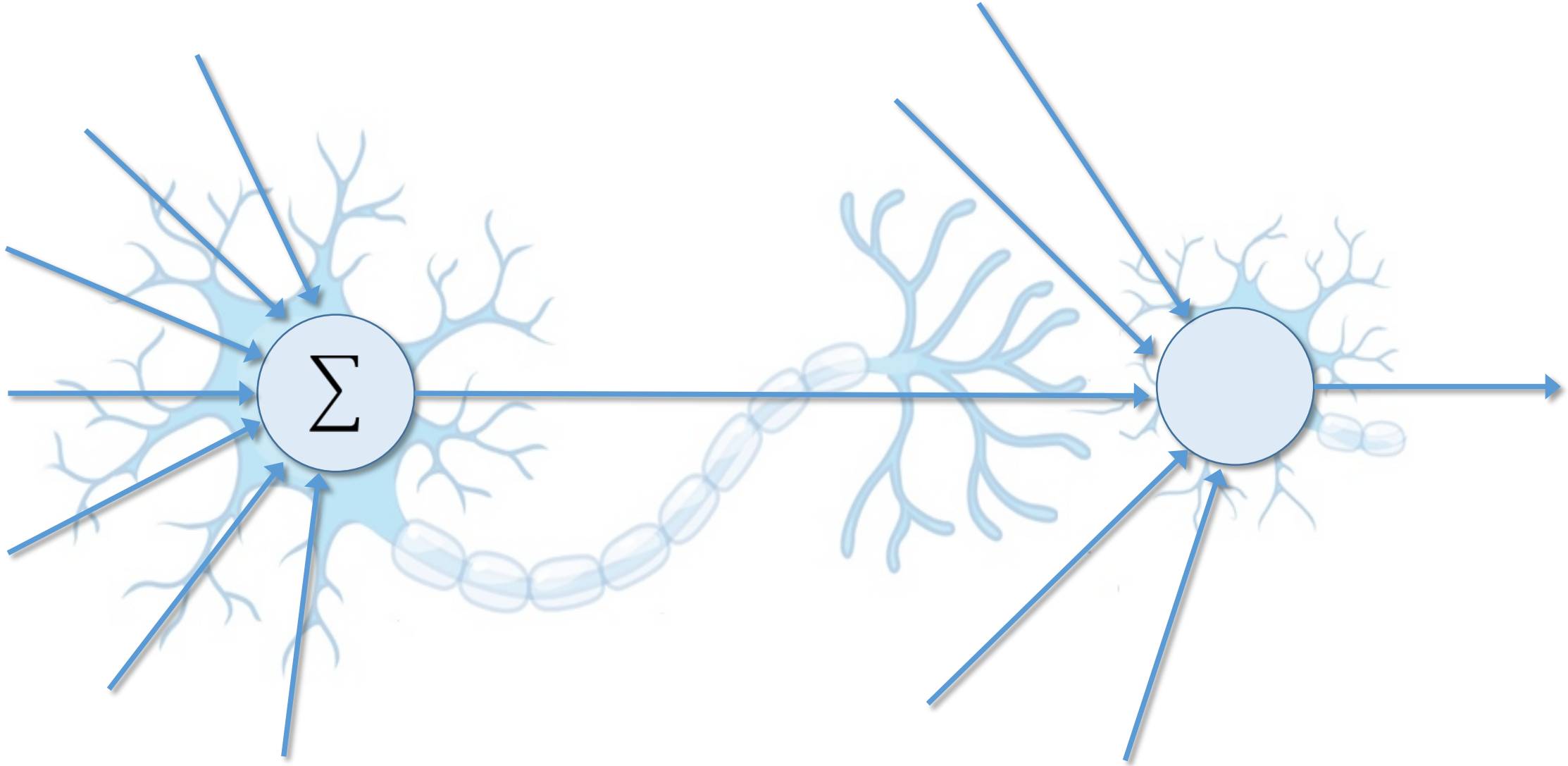
Hebbian learning (1949): *if one neuron frequently takes part in exciting another, some growth process or metabolic change takes place to make it more efficient*



- Synapses may increase/decrease strength of connection between neurons over time (synaptic plasticity) based on activity
- Synaptic plasticity is one of the important neurochemical foundations of learning and memory

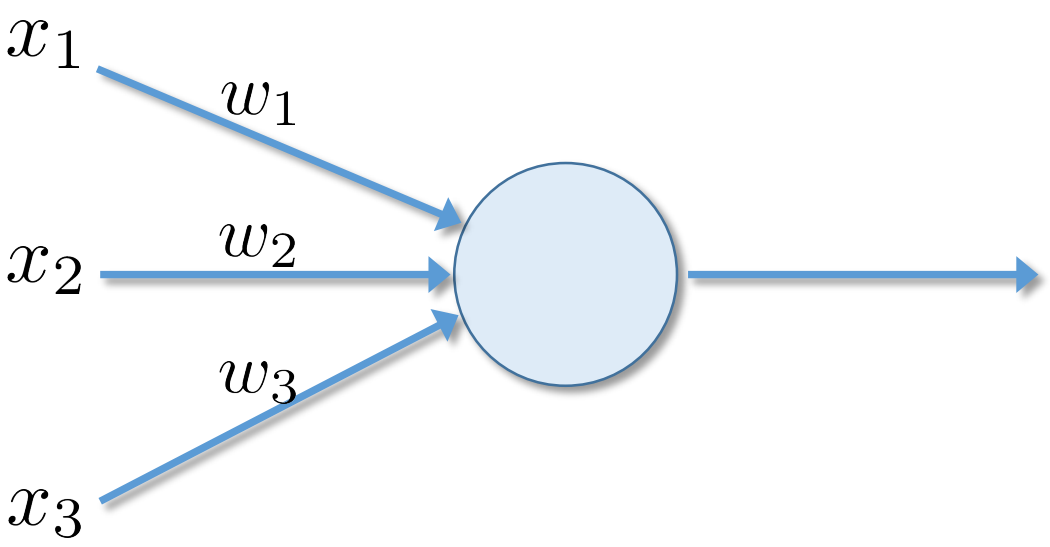


Biology inspired artificial neuron



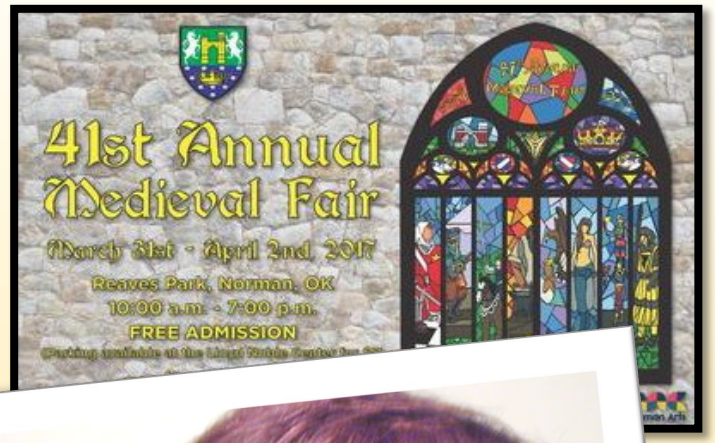
Perceptron: back to the basics

- artificial neuron called a *perceptron* (1950's)
- precursor to the *sigmoid neuron* (and other types of neurons)
- a perceptron takes several binary inputs, and produces a single binary output, e.g., $x_1, x_2, \ldots, x_n \in \{0, 1\}$



$$\text{output} = \begin{cases} 1, & \text{if } \sum_{j=1}^{3} w_j x_j > \text{threshold} \\ 0, & \text{otherwise} \end{cases}$$

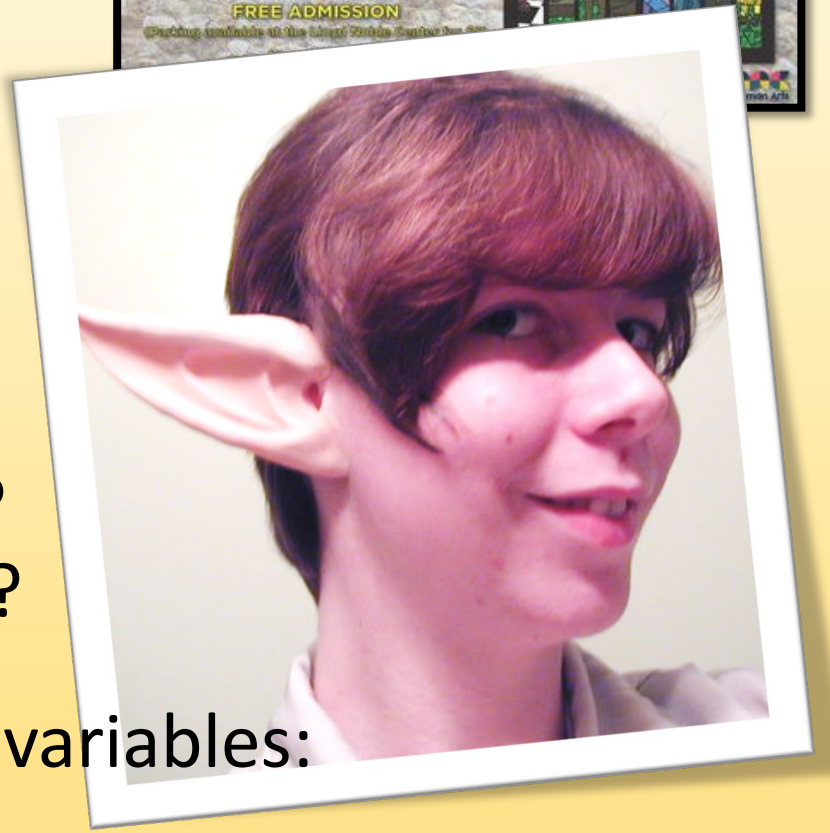
Medieval Fair and Decision-Making



The annual Medieval Fair is coming up and you are trying to decide whether or not to go.

You might make your decision by weighing up three factors:

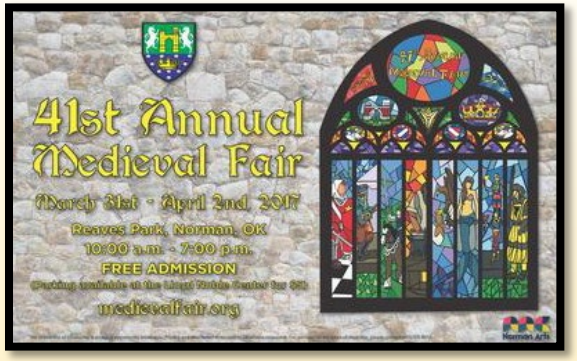
1. Is the weather good?
2. Does your boyfriend/girlfriend want to go?
3. Will your elf ears arrive from Amazon.com?



Let's represent these with binary variables:

e.g., let $x_1 = 1$ if the weather is good,
and $x_1 = 0$ if the weather is bad

Medieval Fair and Decision-Making



You absolutely adore swordplay.

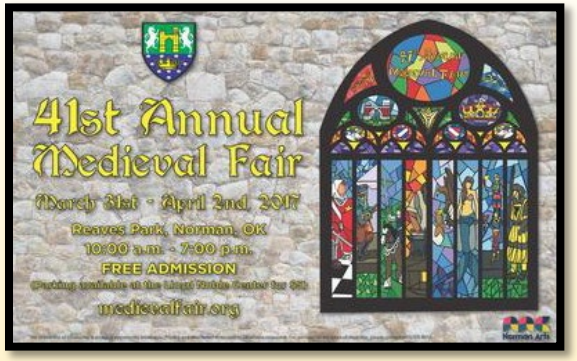
So much so that you're happy to go to the fair even if the weather is bad, and your boyfriend/girlfriend is uninterested. But perhaps you can't imagine attending without your beloved elf ears…

You can use perceptrons to model this kind of decision-making.

One way to do this is to choose particular weights for each variable and a good threshold.

Let weight $w_1 = 2$ for the weather, $w_2 = 2$ for the arm candy, and $w_3 = 6$ for the elf ears.

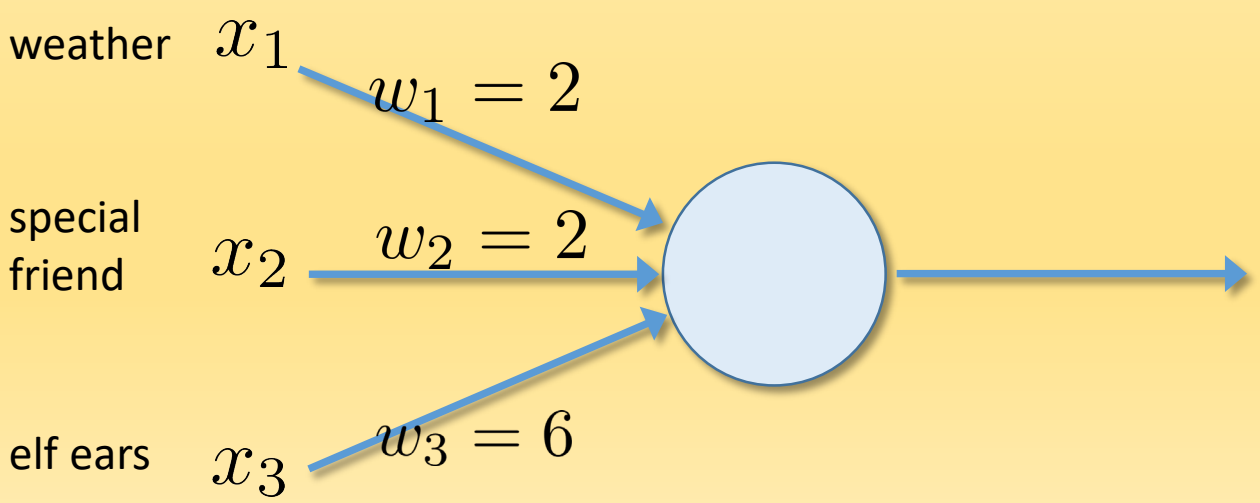
Medieval Fair and Decision-Making



Suppose you choose a threshold of 5 for the perceptron.

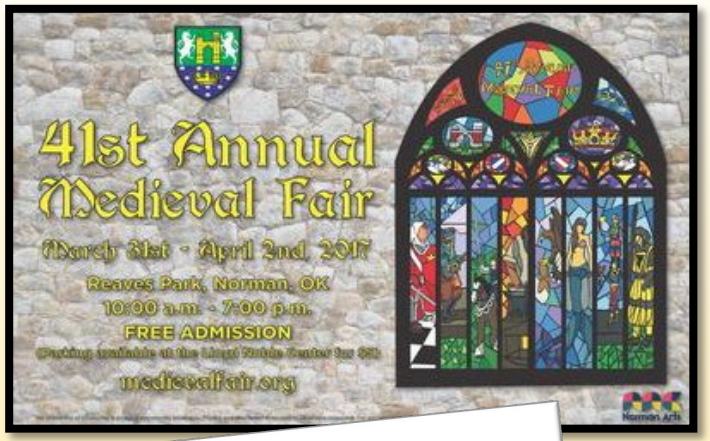
With these choices, the perceptron implements the desired decision-making model: *outputs 1 if the elf ears arrive in time, and 0 if they don't*.

It makes no difference to the output whether weather is good/bad or if your boyfriend/girlfriend wants to go.



$$\text{output} = \begin{cases} 1, & \text{if } \sum_{j=1}^{3} w_j x_j > 5 \\ 0, & \text{otherwise} \end{cases}$$

Medieval Fair and Decision-Making



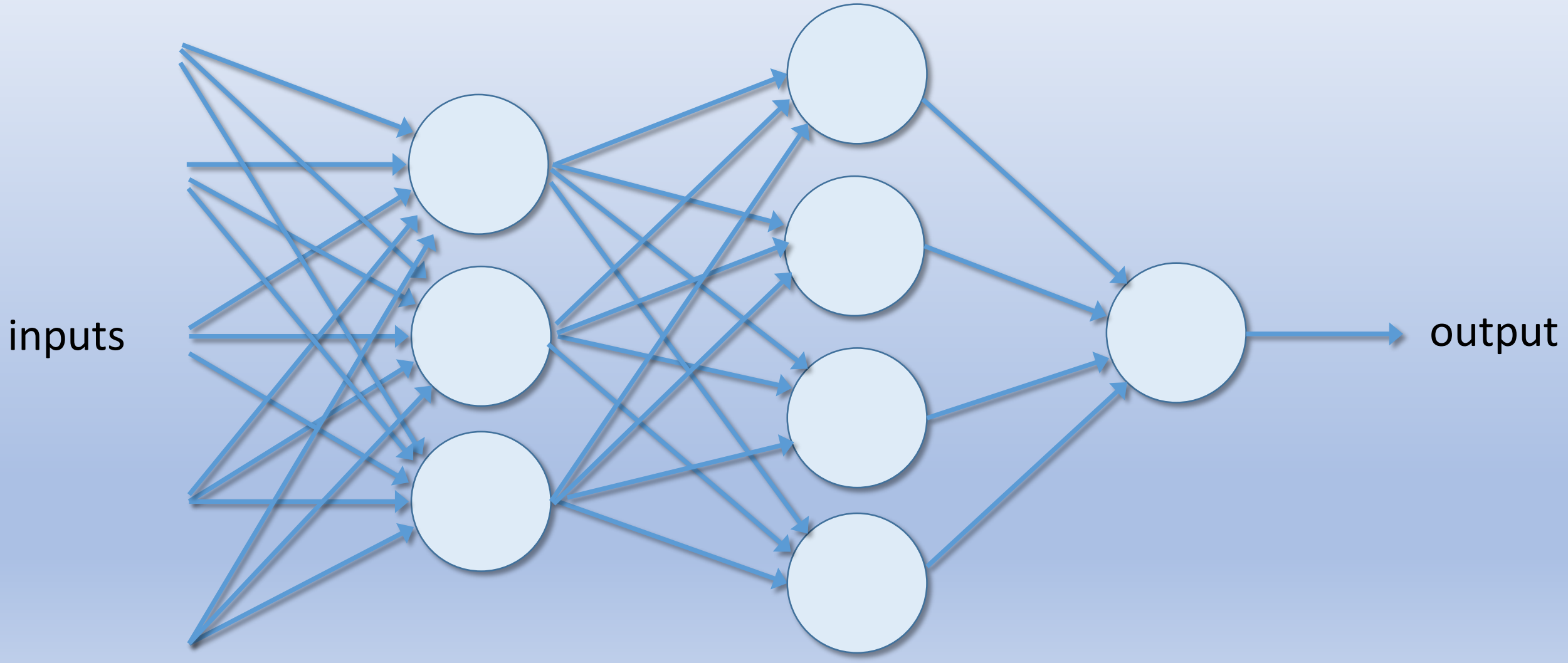
By varying the weights and the threshold, we get different models of decision-making.

E.g.: Suppose the threshold was set to 3.

Now, the perceptron would decide that you should go to the festival whenever the elf ears arrived *or* when *both* the weather was good and your boyfriend/girlfriend was willing to go.

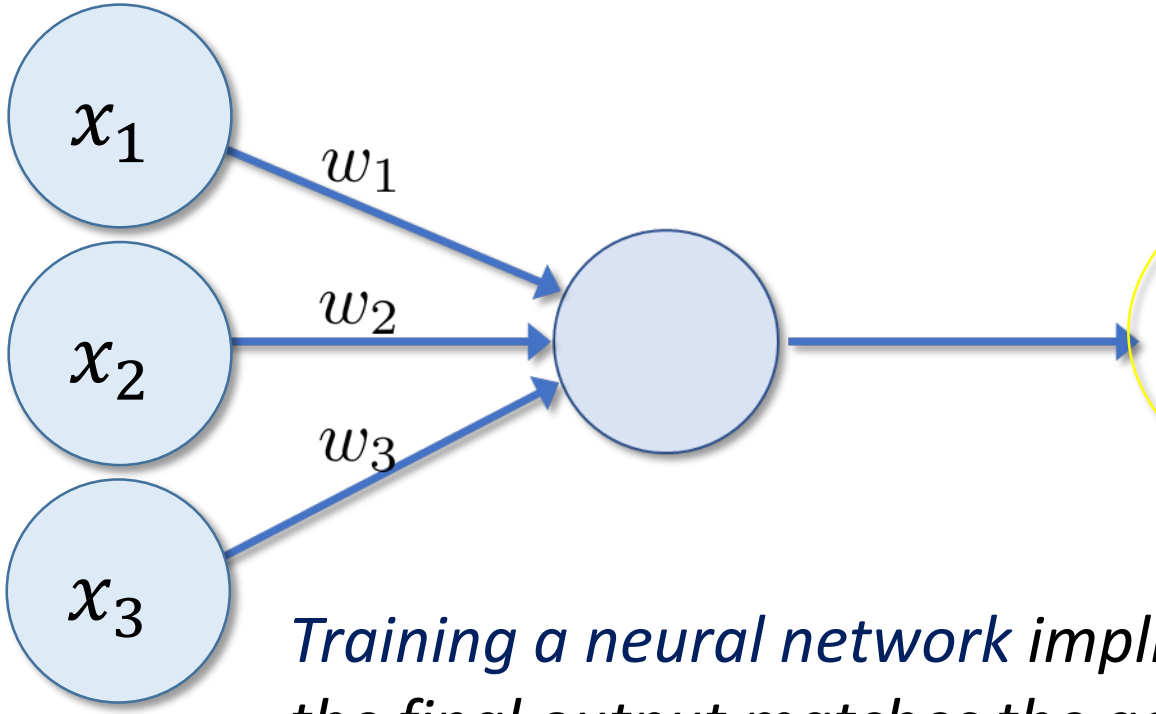
Such a happy couple!

More complex decision frameworks



Modification of notation

- Typically, b (for bias) reflects the threshold, where b = -threshold; so notation can be revised appropriately
- Also, the inputs from the variables is usually depicted as a layer of neurons and called the *input layer*



$$\text{output} = \begin{cases} 1, \text{ if } w \cdot x + b \geq 0 \\ 0, \text{ otherwise} \end{cases}$$

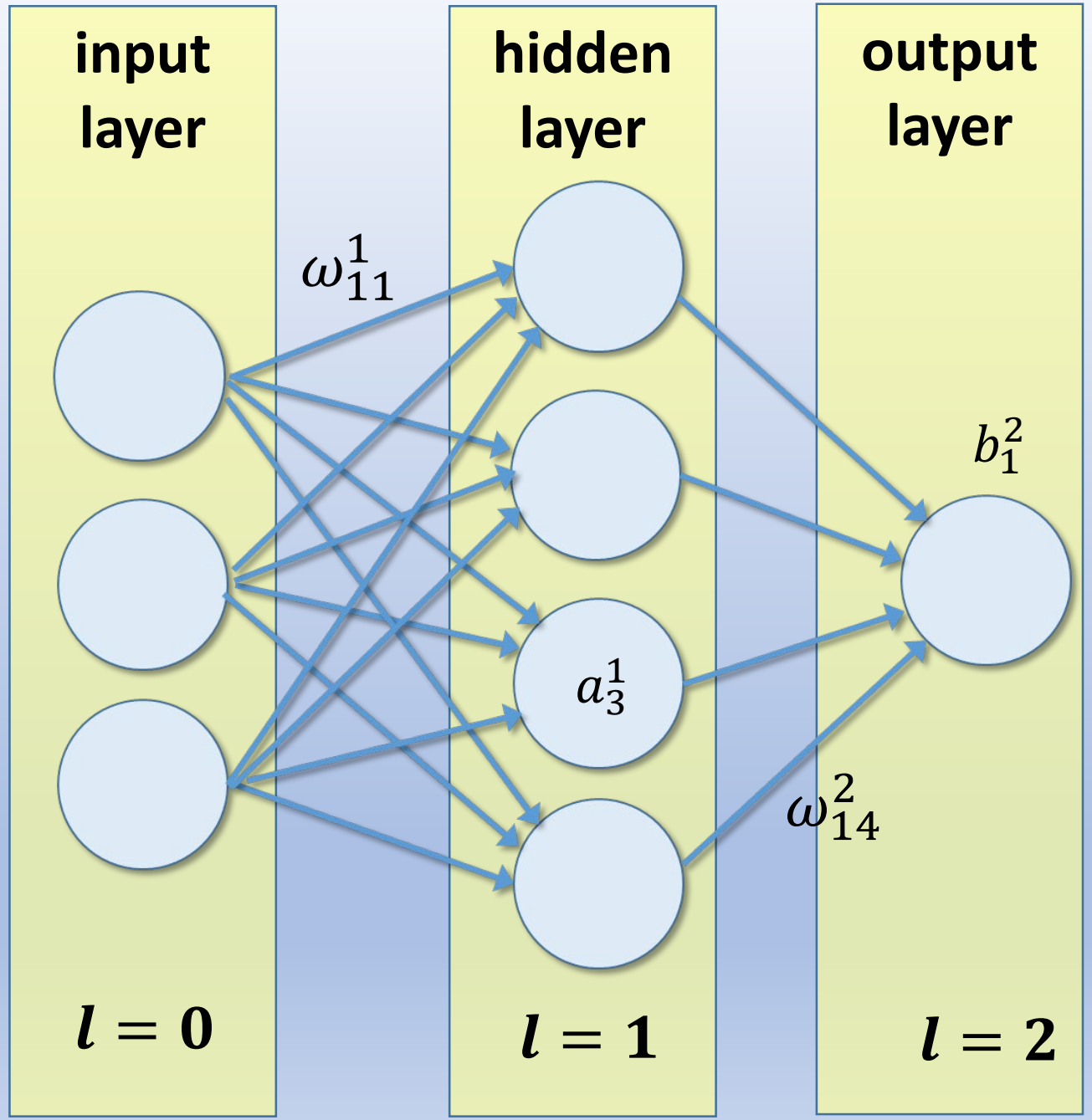
*Training a neural network implies adjusting the **weights** and **biases** so that the final output matches the actual values from training observations*

Clarification of notation

ω_{jk}^{l}: weight from the kth neuron in the $(l-1)^{\text{th}}$ layer to the jth neuron in the lth layer

b_{j}^{l}: bias of the jth neuron in the lth layer

a_{j}^{l}: activation of the jth neuron in the lth layer



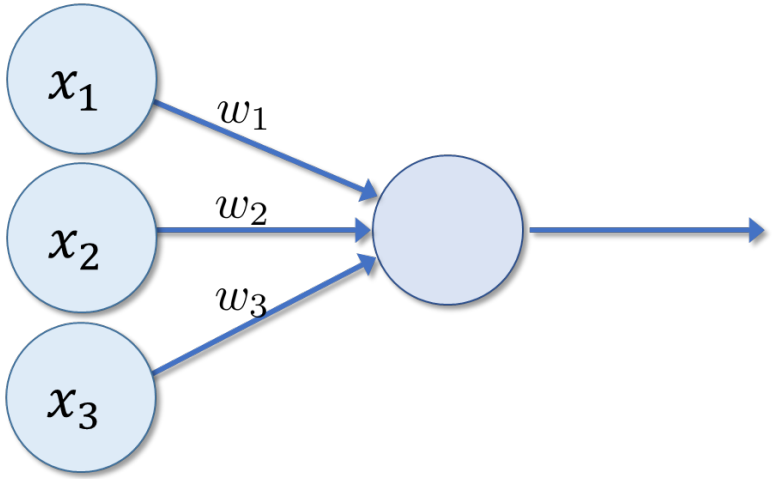
From perceptrons to neurons

- Output (activation value) from perceptron is always 1 or 0
  - Note: perceptrons are a.k.a. **Linear Threshold Units (LTU)**
- Need to be able to make small adjustments to weights (or biases) and get small changes in output

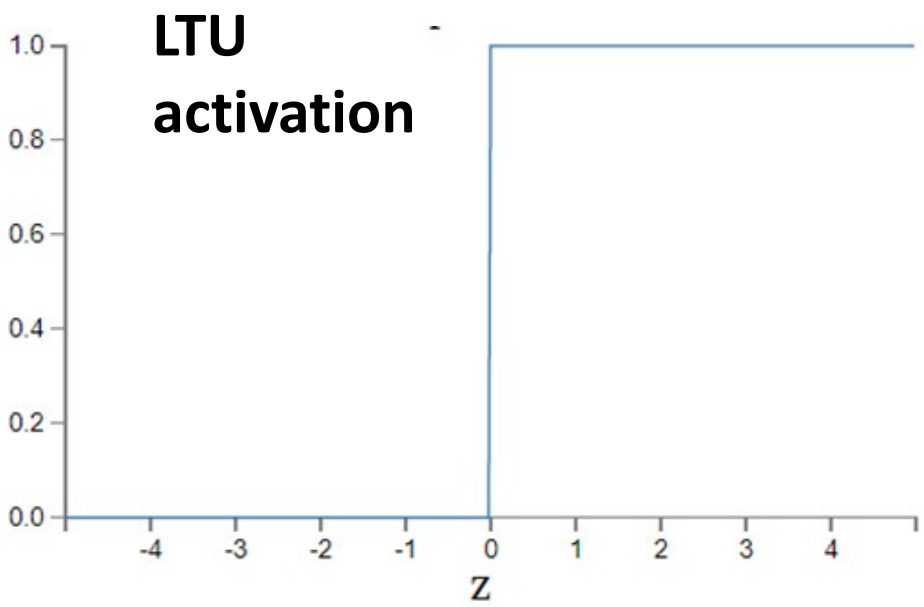
  → *but perceptrons do not allow this!*
- So: change activation function to make it smooth (i.e., differentiable)
- One common possibility is the *sigmoidal neuron*
- Remember this?

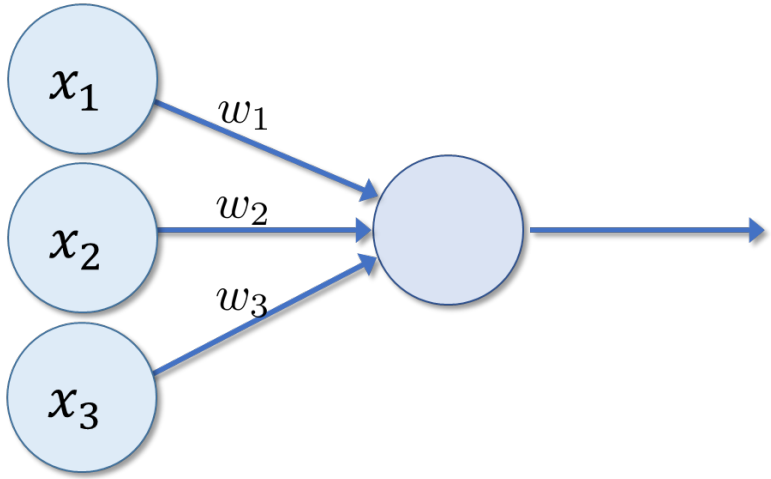
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

From perceptrons to neurons

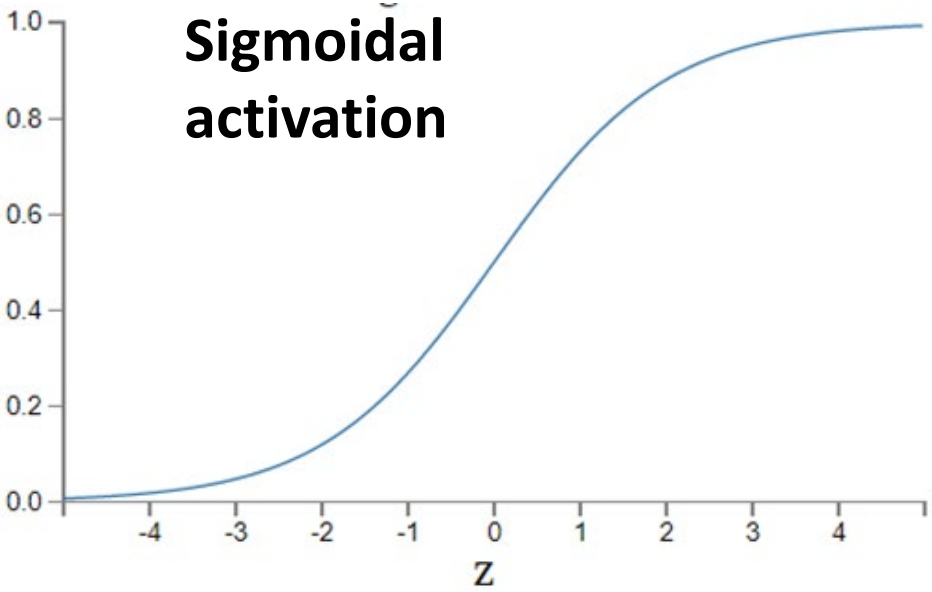


$$\begin{cases} 1, \text{ if } w \cdot x + b \geq 0 \\ 0, \text{ otherwise} \end{cases}$$



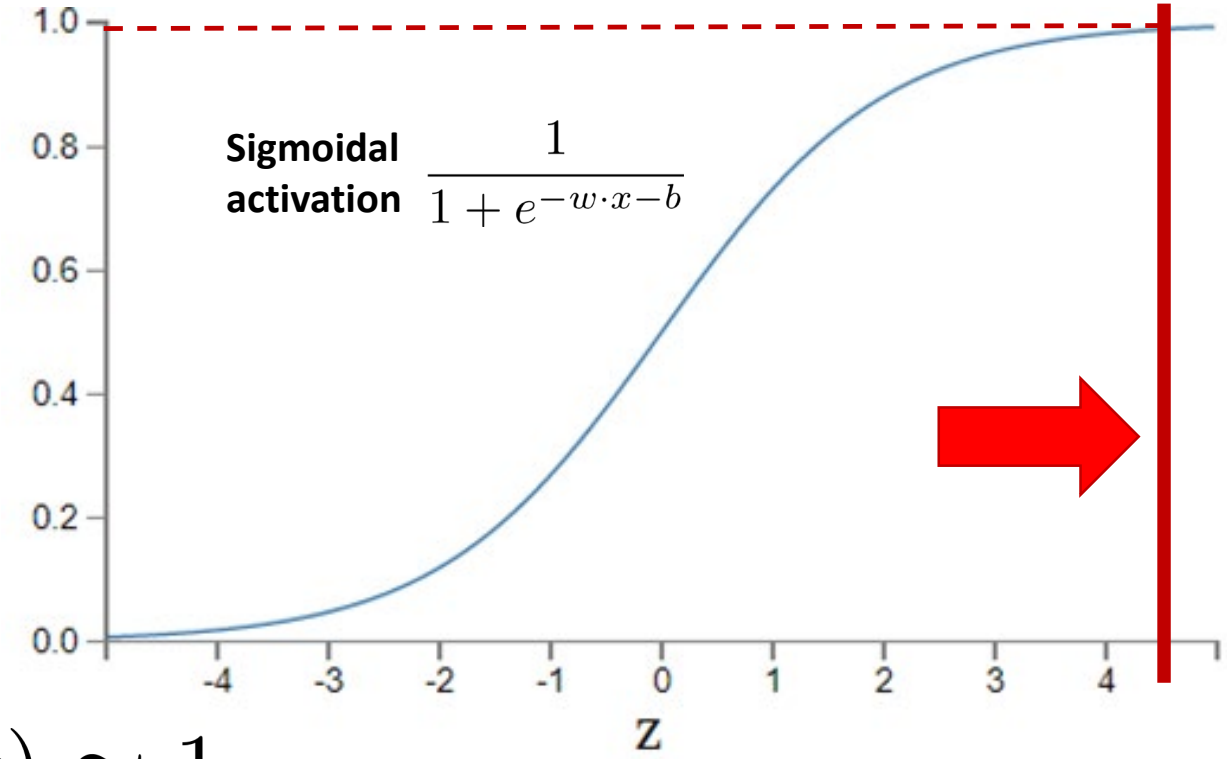


$$\frac{1}{1 + e^{-w \cdot x - b}}$$



Similarity of sigmoidal neurons and perceptrons

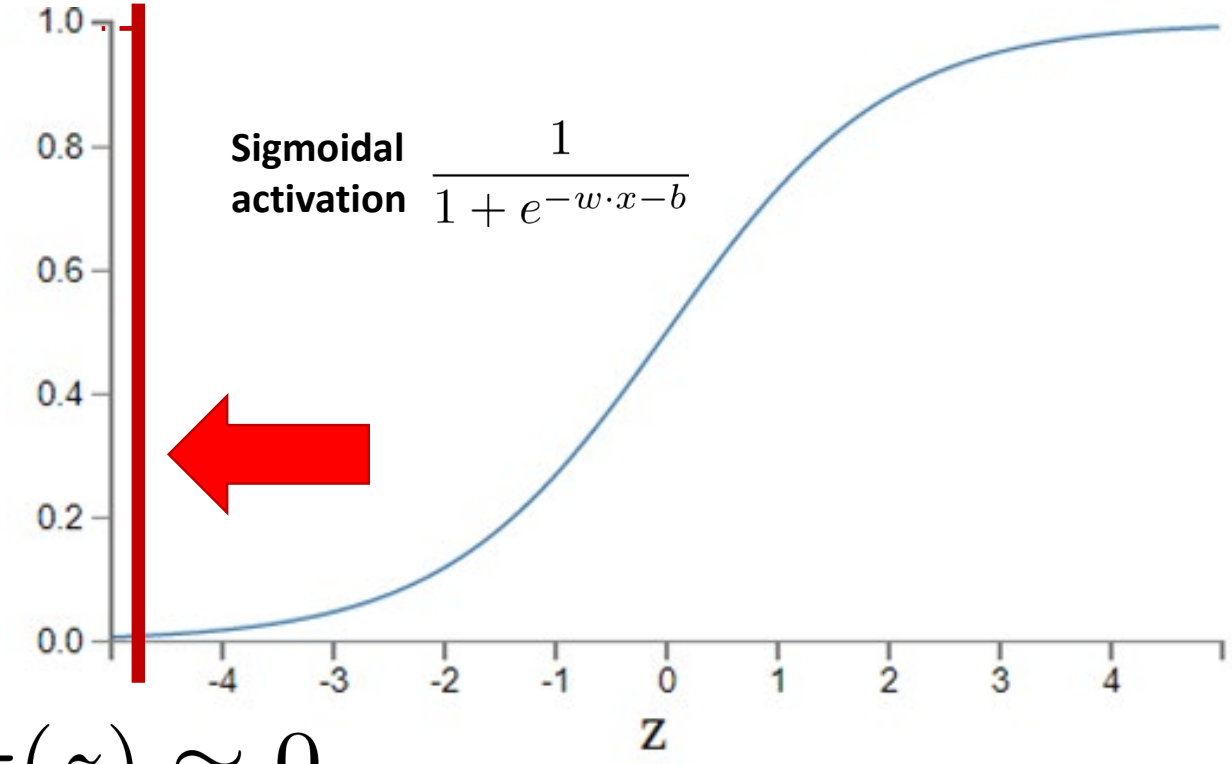
Suppose $z = w \cdot x + b$ is large, positive value



- Then, $e^{-z} \approx 0$ so $\sigma(z) \approx 1$
- That is, when z is large and positive, the output from the sigmoid neuron is approximately 1, *just as it would have been for a perceptron.*

Similarity of sigmoidal neurons and perceptrons

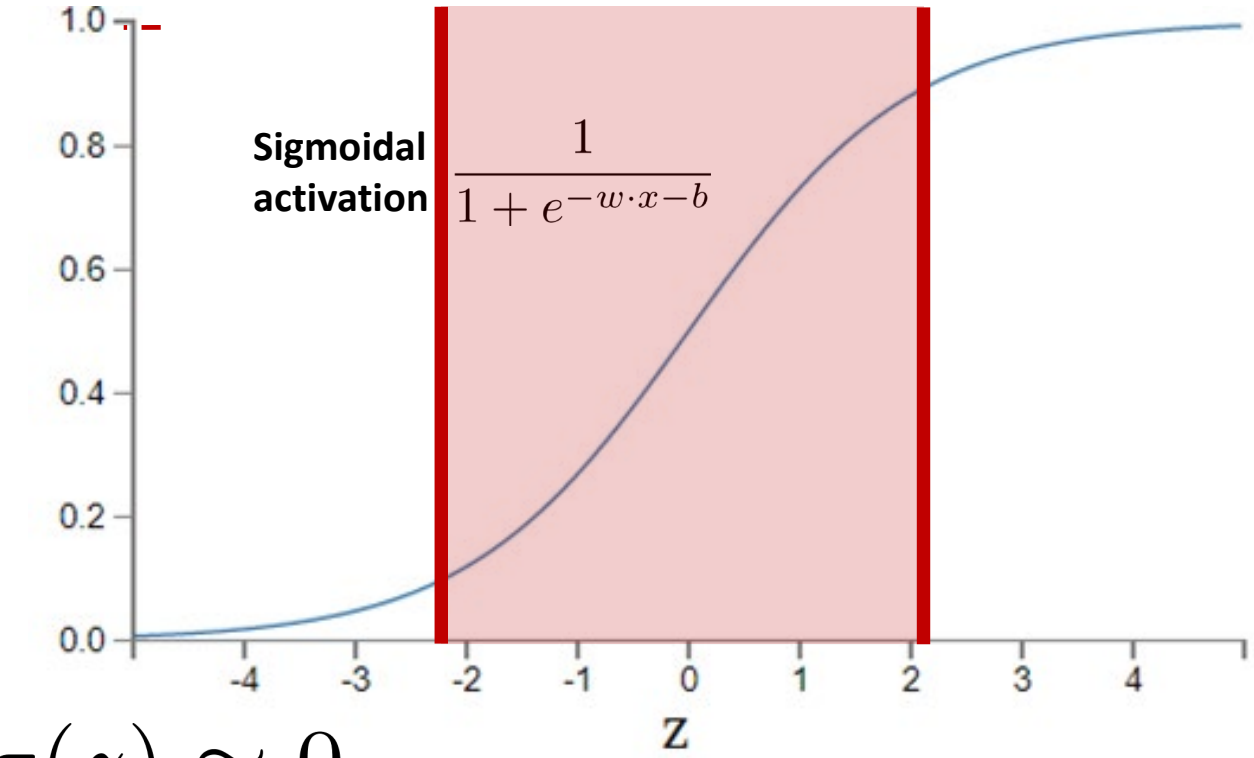
Suppose $z = w \cdot x + b$ is very negative



- Then, $e^{-z} \to \infty$ and $\sigma(z) \approx 0$
- That is, when z is very negative, a sigmoid neuron behaves like a perceptron

Similarity of sigmoidal neurons and perceptrons

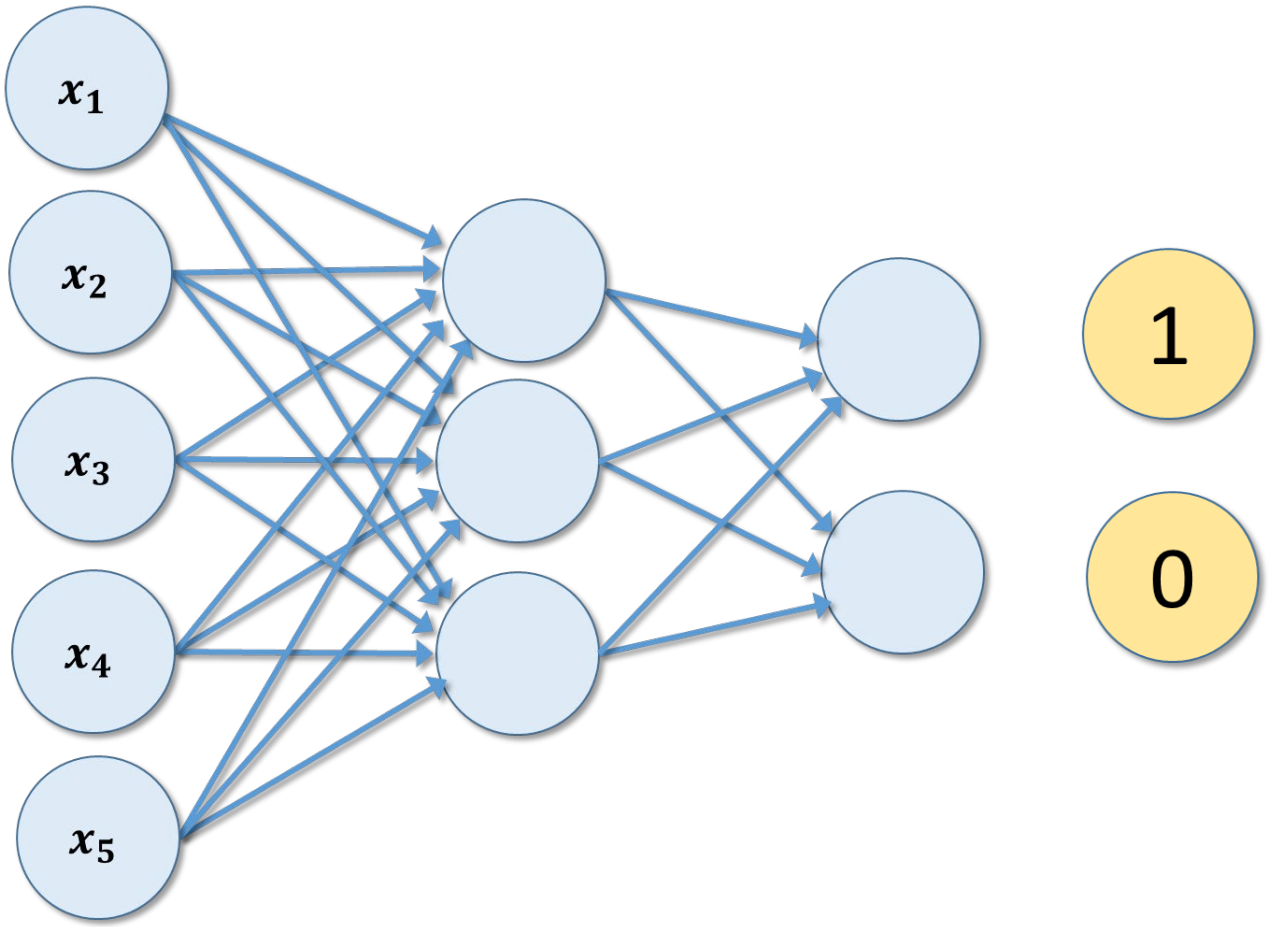
Suppose $z = w \cdot x + b$ is very negative



- Then, $e^{-z} \to \infty$ and $\sigma(z) \approx 0$
- That is, when z is very negative, a sigmoid neuron behaves like a perceptron
- It's only when z is of modest size the behavior deviates

Training a neural network

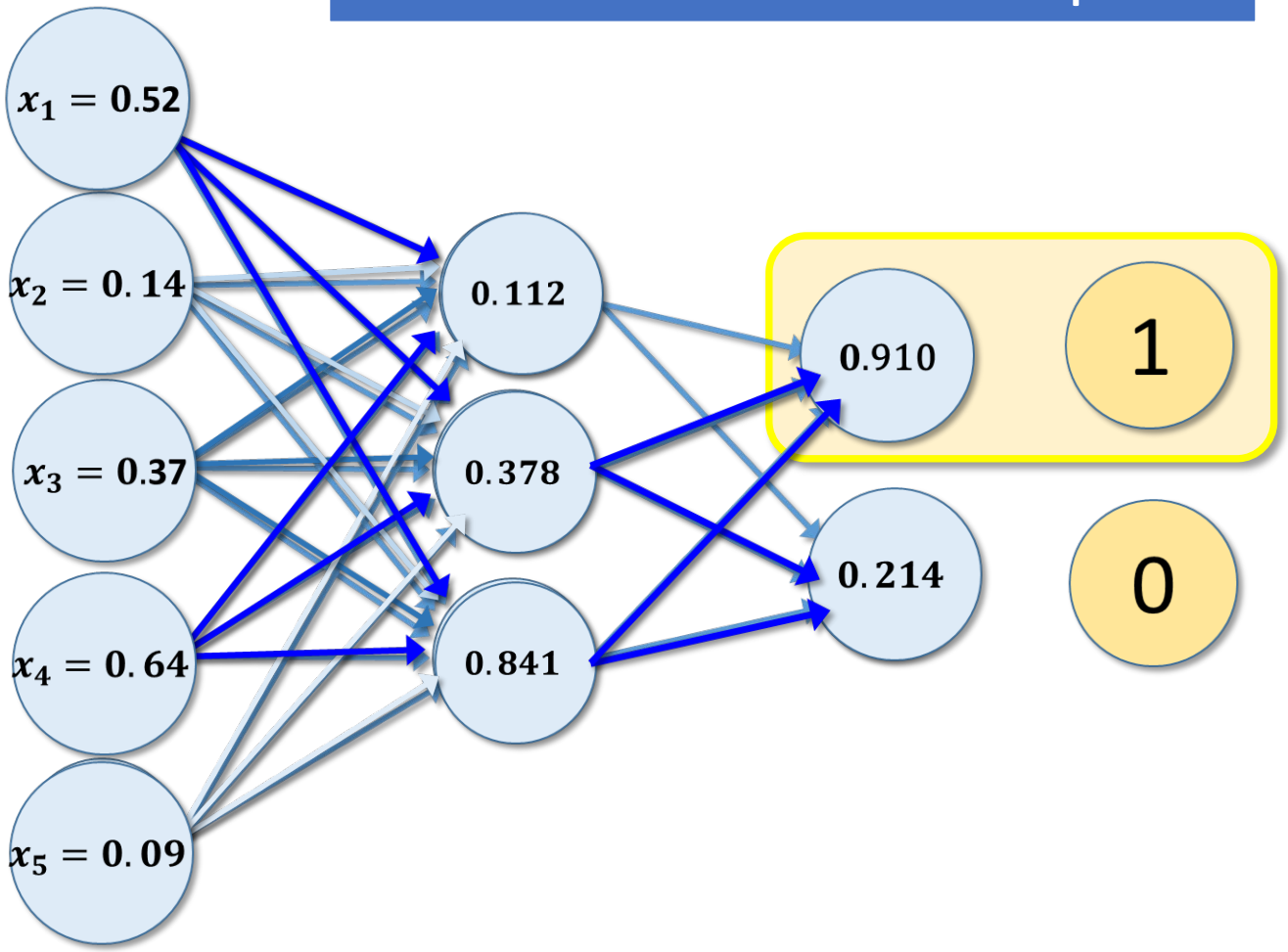
- For continuous models, usually one output node
- For binary classifier, one or two output nodes
  - E.g., at right: one node corresponds to a value of 1; the other to a value of 0
- For multi-class, usually one output node per class



Training a neural network

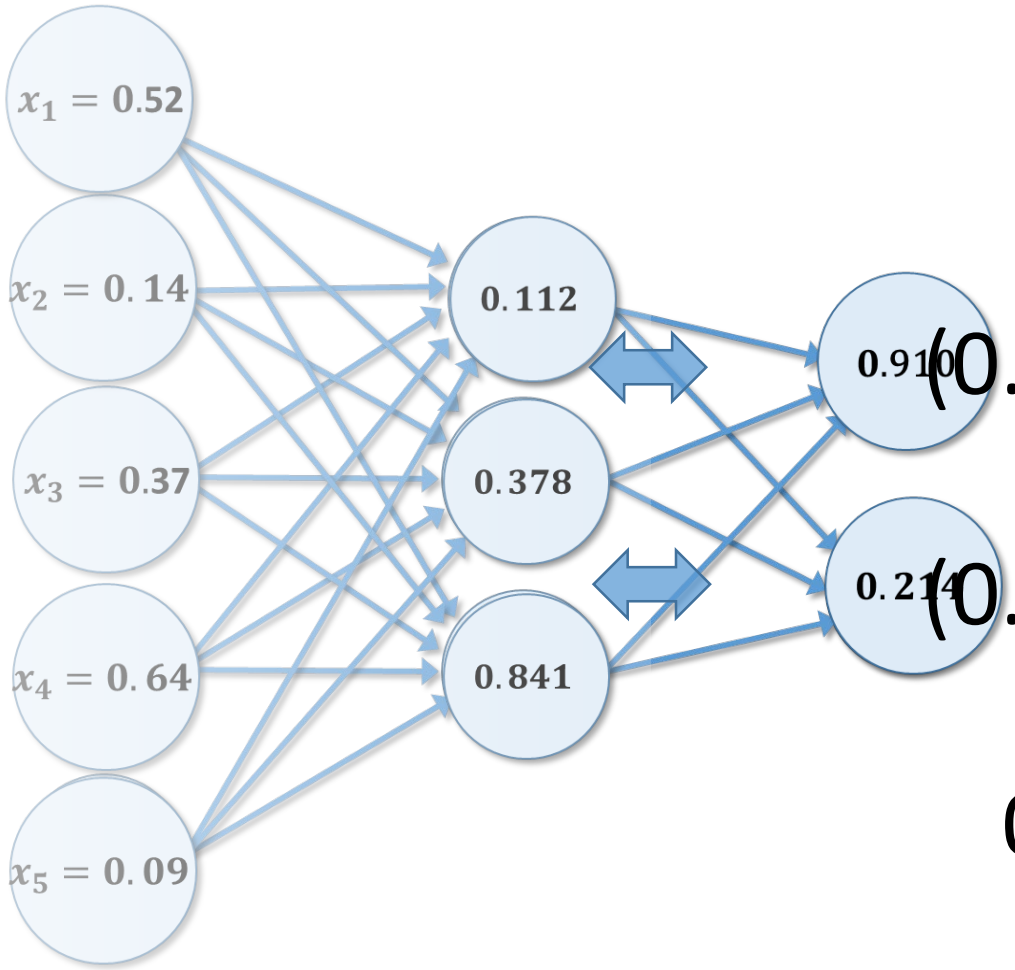
- For continuous models, usually one output node
- For binary classifier, one or two output nodes
  - E.g., at right: one node corresponds to a value of 1; the other to a value of 0
- For multi-class, usually one output node per class
- Learning means determining weights and biases based on the quality of the predictions

This is called a "forward pass"



Training a neural network

- The weights and biases are all randomly generated* at first; and the cost is computed.
- The cost function provides a penalty based on the output node and the true value.
- One common cost function is simply MSE (there are others too!, e.g., https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications)
- Here the cost of this one training example is 0.0657

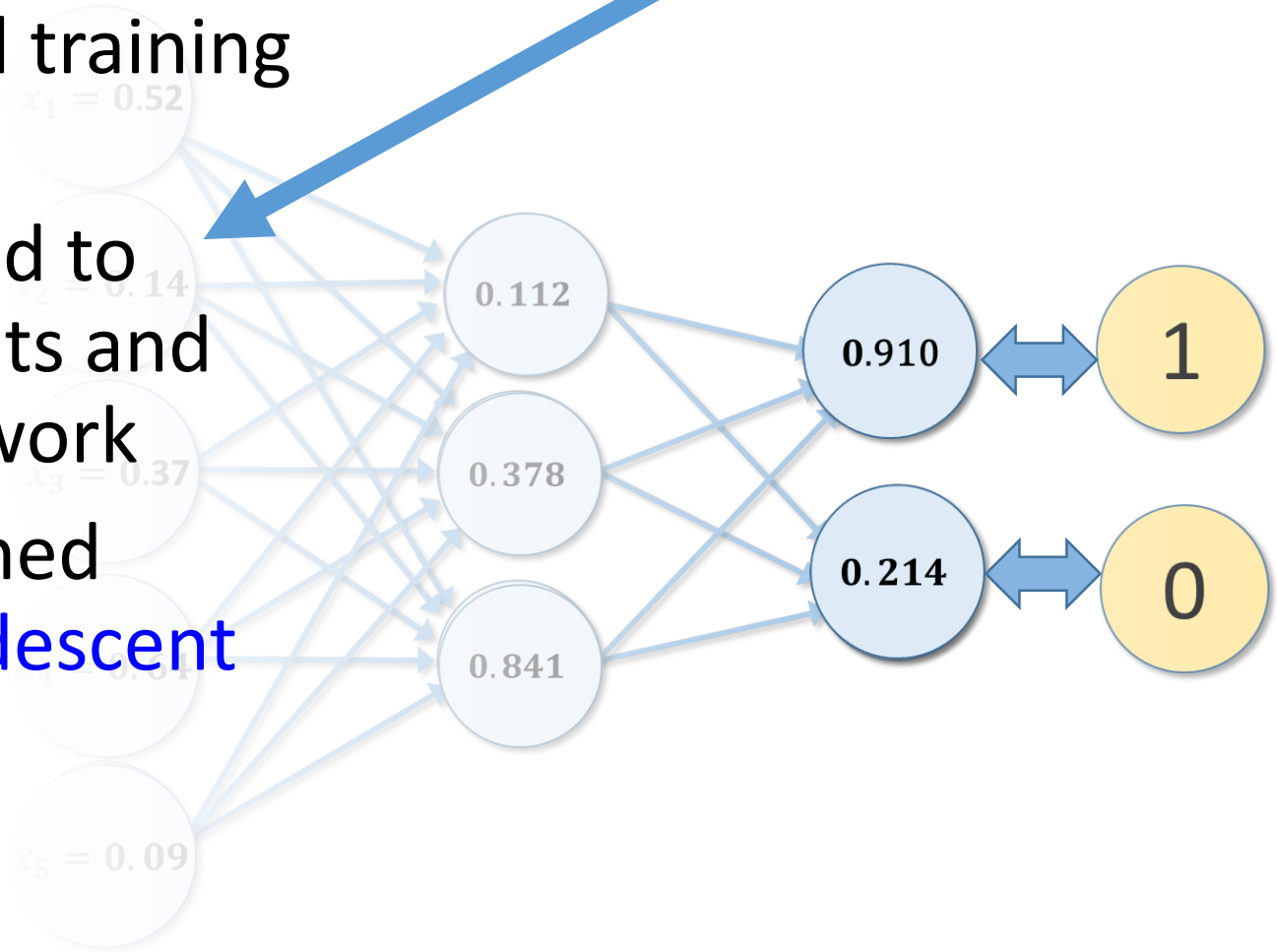


$$(0.91 - 1)^2 + (0.24 - 0)^2 = 0.0657$$

Training a neural network

- The goal is to minimize the penalty for all training samples
- To do this we need to update the weights and biases in the network
- This is accomplished using a gradient descent method

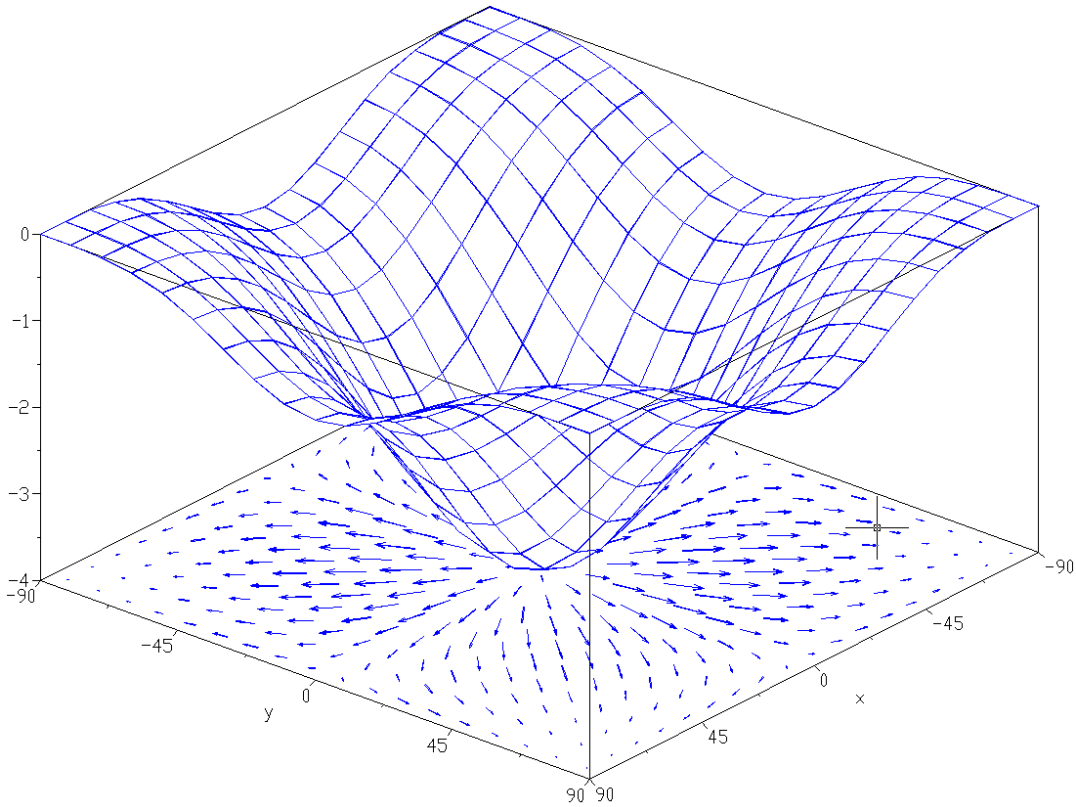
Typically, via “back propagation”



$$(0.91 - 1)^2 + (0.24 - 0)^2 = 0.0657$$

Remind me what a gradient is…

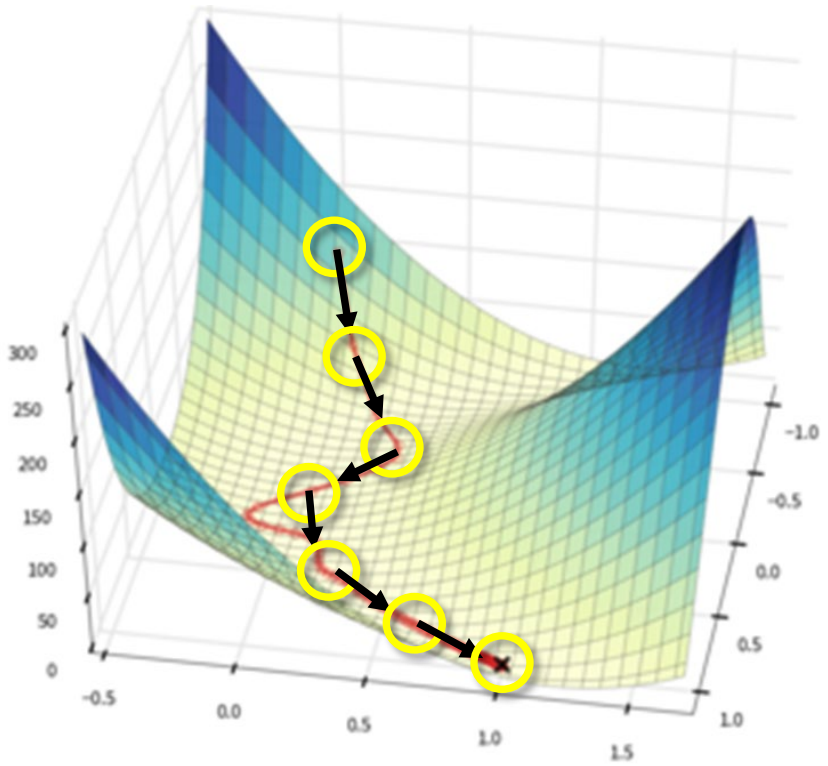
- How should we adjust the hundreds (or thousands) of weights and biases to reduce the overall cost?
- The **gradient** of a m-dimensional function f is a m-dimensional vector of partial derivatives denoted with the symbol ∇
- $\nabla f(x_1, x_2, \dots, x_m) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_m}\right)$
- The gradient has a value everywhere (on smooth surfaces)
- The direction of the gradient is the direction of greatest uphill slope



Gradient Descent

Gradient descent is method to minimize a function – which in our case is the cost function

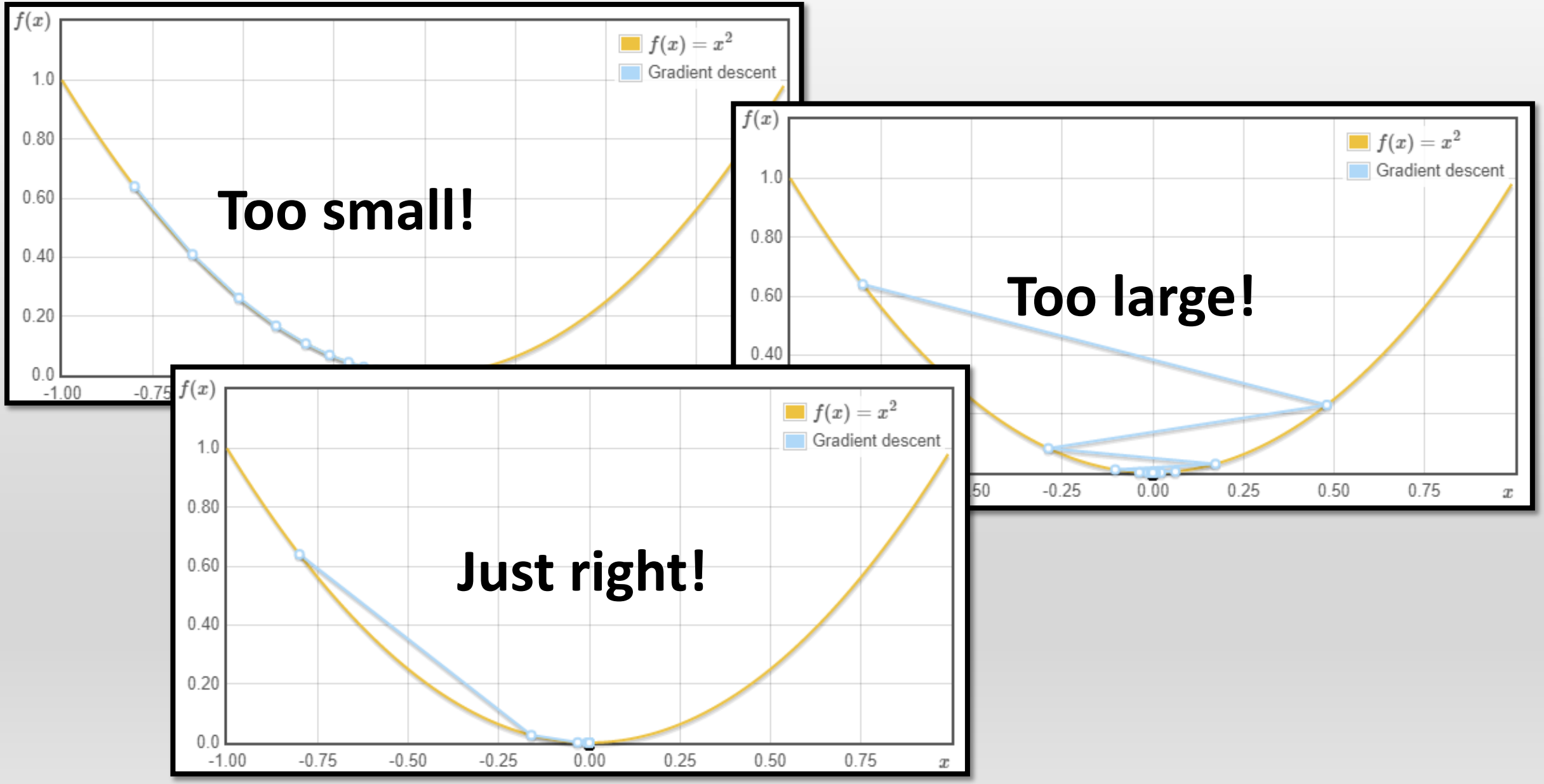
1. Generate a starting point (e.g., random weights and biases)
2. Calculate the gradient at the current point
3. Take a “step” in the opposite direction of the gradient (descent instead of ascent)
4. Is it time to stop? If so, you are done. Otherwise, go to step 2.



How big of a “step” should you take?

- The magnitude of the gradient vector is a indication of how steep the slope is at that point.
- However, we may want to take a larger (or smaller) step
- Please note:
  - **Too small and it may take a long time to get the minimum value**
  - **Too big and you might miss it altogether**

Step size


Too small!
Too large!
Just right!
$f(x)$
$f(x) = x^2$
Gradient descent
1.0
0.80
0.60
0.40
0.20
0.0
-1.00
-0.75
-0.50
-0.25
0.00
0.25
0.50
0.75
x

Gradient Descent for Neural Networks

- Let **x** denote the m-dimensional vector of weights and biases
- Let **f** denote the MSE cost function in the neural net
- Then an example negative gradient might look like this

$$-\nabla f(x) = \begin{pmatrix} 0.27 \\ -1.48 \\ 0.02 \\ 1.05 \\ \vdots \\ -0.41 \end{pmatrix}$$

Increase the first weight some

Decrease the second one a lot

Increase the third one a little

Increase the fourth one a lot

Decrease the mth one some

Which allows you to understand whether or not the associated weight or bias should be increased or decreased; additionally, it indicates how much it should be adjusted

Algorithmic Overview

(with some simplifications)

- Randomly choose initial weights and biases*
- While error is too large
  - For each training observation (presented in random order)
    - Calculate output for every neuron from input layer, through the hidden layer(s), to the output layer
    - Calculate error at the outputs

      forward pass
    - Use the error to iteratively, compute weight and bias adjustments from output back towards input layer
    - Apply the adjustments

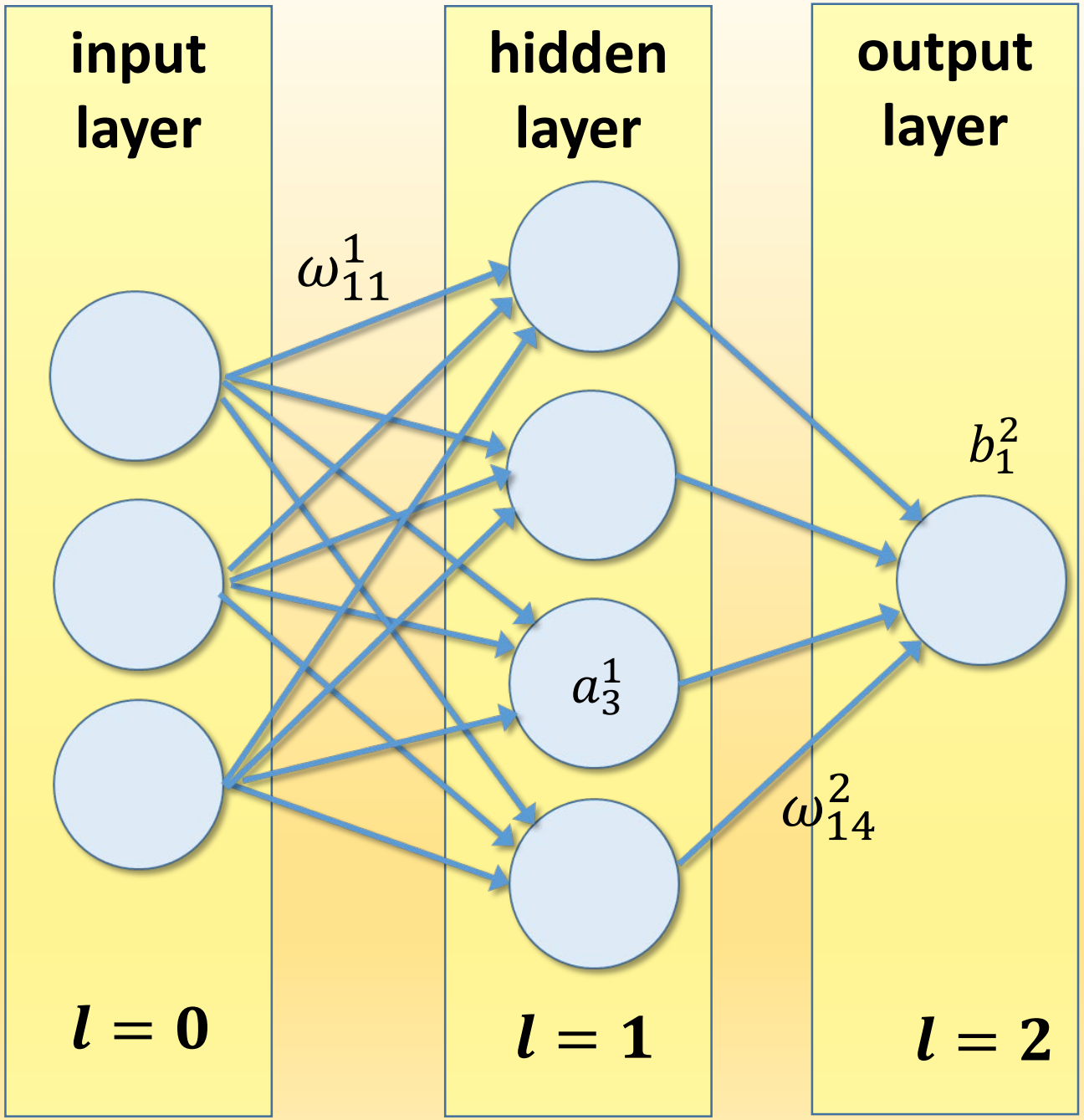
      backpropagation
  - Periodically evaluate the network performance

Reminder of notation

ω_{jk}^{l}: weight from the kth neuron in the $(l-1)^{\text{th}}$ layer to the jth neuron in the lth layer

b_{j}^{l}: bias of the jth neuron in the lth layer

a_{j}^{l}: activation of the jth neuron in the lth layer



Backpropagation: some more notation

ω_{jk}^{l}: weight from the kth neuron in the $(l-1)^{\text{th}}$ layer to the jth neuron in the lth layer

b_j^l: bias of the jth neuron in the lth layer

a_j^l: activation of the jth neuron in the lth layer

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

Vectorize this to improve the notation:

$$a^l = (a_1^l, \ldots, a_m^l)$$
$$w^l = (w_1^l, \ldots, w_m^l)$$
$$b^l = (b_1^l, \ldots, b_m^l)$$

Backpropagation: some more notation

ω_{jk}^{l}: weight from the kth neuron in the $(l-1)^{\text{th}}$ layer to the jth neuron in the lth layer

b_j^l: bias of the jth neuron in the lth layer

a_j^l: activation of the jth neuron in the lth layer

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right)$$

Let's also create intermediate quantity to further improve notation:

$$z^l = w^l a^{l-1} + b^l \quad \Rightarrow \quad \boxed{a^l = \sigma\left(z^l\right)}$$

Backpropagation: gradient and cost function

Let C denote the cost function.

For now we will use MSE (well, actually half of MSE – we'll see why later):

$$C = \frac{1}{2n} \sum_{x \in T} \left(y(x) - a^L(x) \right)^2$$

where

- T is the set of all n training examples;
- $y(x)$ is the observed outcome of training example x;
- and L is the output layer

Backpropagation: gradient and cost function

Let C denote the cost function.

$$C = \frac{1}{2n} \sum_{x \in T} \left(y(x) - a^L(x) \right)^2$$

$$z^l = w^l a^{l-1} + b^l$$

Let C denote the cost function.

Since $a^L(x)$ is a function of the weights and biases, in backpropagation we need to compute the gradient, i.e.:

- **partial derivatives $\partial C / \partial w$ with respect to each weight w**
- **partial derivatives $\partial C / \partial b$ with respect to each bias b**

Backpropagation: Equations!

Define a new quantity to help with notation: $$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

Let's compute the impact on the cost function with respect to change in the inputs to the output layer.

This requires the use of the chain rule to resolve: $$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}$$

Since $\frac{\partial a_k^L}{\partial z_j^L}$ is 0 if $k \neq j$, this simplifies to:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial \sigma\left(z_j^L\right)}{\partial z_j^L} = \boxed{\frac{\partial C}{\partial a_j^L} \sigma'\left(z_j^L\right)}$$

Backpropagation: Equations!

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial \sigma\left(z_j^L\right)}{\partial z_j^L}$$

$$= \boxed{\frac{\partial C}{\partial a_j^L}}\boxed{\sigma'\left(z_j^L\right)}$$

measures how fast the cost is changing as a function of the jth output activation

measures how fast the activation function σ is changing at z_j^L

Backpropagation: Equations!

$$= \frac{\partial C}{\partial a_j^L} \sigma'\left(z_j^L\right)$$

measures how fast the cost is changing as a function of the jth output activation

measures how fast the activation function σ is changing at

E.g., if for a particular training example $C = \frac{1}{2} \sum_j \left(y_j - a_j^L\right)^2$

then, $$\frac{\partial C}{\partial a_j^L} = a_j^L - y_j$$

Backpropagation: Equations!

$$= \frac{\partial C}{\partial a_j^L} \sigma'\left(z_j^L\right)$$

measures how fast the cost is changing as a function of the jth output activation

measures how fast the activation function σ is changing at

And if we are considering a sigmoidal neuron, then…

$$\sigma'(z) = \frac{d}{dz} \frac{1}{1+e^{-z}} = \dots$$

Trip down memory lane…

$$\sigma'(z) = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \frac{d}{dz}(1+e^{-z})^{-1}$$

$$= -(1+e^{-z})^{-2}(-e^{-z})$$

$$= \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}}{1+e^{-z}}\right)$$

$$= \left(\frac{1}{1+e^{-z}}\right)\left(1-\frac{1}{1+e^{-z}}\right)$$

$$= \sigma(z)\,(1-\sigma(z))$$

Backpropagation: Equations!

$$\frac{\partial C}{\partial a^L}\sigma'\left(z^L\right) = \left(a^L - y\right) a^L \left(1 - a^L\right)$$

Backpropagation: Equations!

So the output layer is easy… $\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'\left(z_j^L\right)$

Now we want to "back propagate" this error to the previous layer.

To do so, we want to express δ_j^l as a function of δ_j^{l+1}

Backpropagation: Equations!

Let's start with our defined quantity again: $\delta_j^l = \frac{\partial C}{\partial z_j^l}$

And employ the chain rule.

$$\delta_j^l = \sum_k \boxed{\frac{\partial C}{\partial z_k^{l+1}}} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

$$= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$

Backpropagation: and more derivations

Now how do we compute this: $\delta_j^l = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$

Well, $z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma\left(z_j^l\right) + b_k^{l+1}$

And thus: $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'\left(z_j^l\right)$

So: $\boxed{\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'\left(z_j^l\right)}$

Backpropagation: Equations!

Compute the impact on cost with respect to change in the weights.

This requires the use of the chain rule to resolve: $\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l}$

$$\frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a_k^{l-1} = \boxed{\delta_j^l a_k^{l-1}}$$

Backpropagation: Equations!

Compute the impact on cost with respect to change in the biases

This requires the use of the chain rule to resolve: $\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l}$

$$\frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \boxed{\delta_j^l}$$

Backpropagation Algorithm

1. **Input x:** Set the corresponding activations a^1 of the input layer
2. **Feedforward**: For each $l = 2,3, \dots, L$ compute z^l and $a^l = \sigma(z^l)$
3. **Backpropagation**:
   - Compute δ^L for final output layer
   - Use δ^L to then compute δ^l each $l = L - 1, \dots, 2, 1$
4. **Gradient output**: $\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$ $\quad \frac{\partial C}{\partial b_j^l} = \delta_j^l$
5. **Update weights and biases:** $w_{jk}^l \leftarrow w_{jk}^l - \eta \left(\delta_j^l a_k^{l-1}\right)$

   $b_j^l \leftarrow b_j^l - \eta \left(\delta_j^l\right)$

   (η is the gradient "step size";
   also called the "learning rate")

Backpropagation Algorithm

1. **Input x:** Set the corresponding activations a^1 of the input layer
2. **Feedforward:** For each $l = 2,3, \ldots, L$ compute z^l and $a^l = \sigma(z^l)$

   Do i = 1 to epochs
3. **Backpropagation**:
   - Compute δ^L for final output layer
   - Use δ^L to then compute δ^l each $l = L-1, \ldots, 2, 1$
4. **Gradient output**: $\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$ $\quad \frac{\partial C}{\partial b_j^l} = \delta_j^l$

   Loop
5. **Update weights and biases:** $w_{jk}^l \leftarrow w_{jk}^l - \eta \left(\delta_j^l a_k^{l-1}\right)$

   $b_j^l \leftarrow b_j^l - \eta \left(\delta_j^l\right)$

   (η is the gradient "step size"; also called the "learning rate")

Epochs: number of times the algorithm "sees" all of the training data

Backpropagation Algorithm

Uh oh…. Computing the gradients based on the entire data set each time is time-consuming….

Can we get “good” gradients even if we don’t look at all of the data every time we update them?

Yes. Yes we can → SGD (next slide)

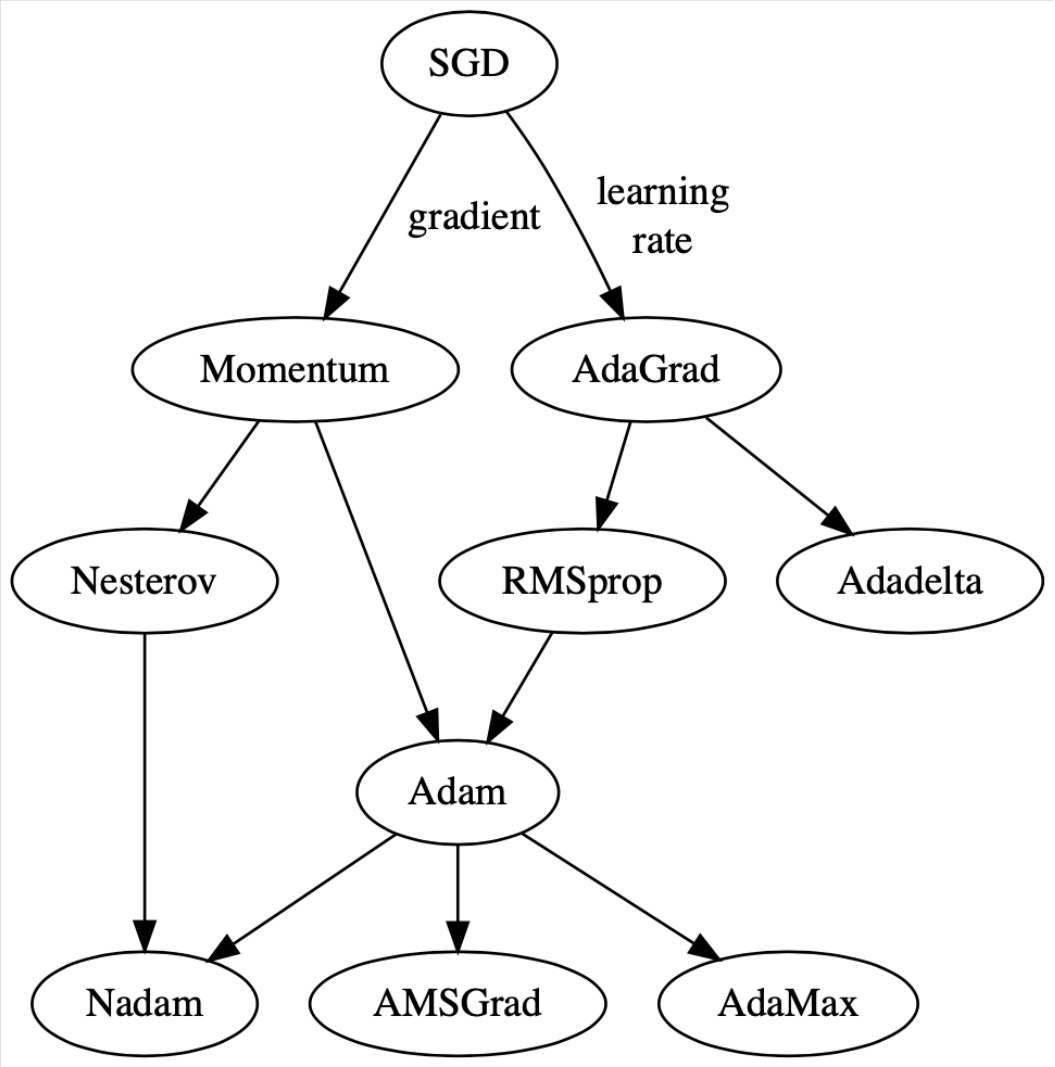
Stochastic Gradient Descent and Mini-batch SGD

By choosing examples at random from our data set, we estimate (albeit, noisily) a big average from a much smaller one.

- **Stochastic gradient descent**(**SGD**) takes this idea to the extreme--it uses only a single example (a batch size of 1) chosen at random per iteration. Given enough iterations, SGD works but is very noisy.
- **Mini-batch stochastic gradient descent** (**mini-batch SGD**) is a compromise between full-batch iteration and SGD. A mini-batch is typically between 10 and 1,000 examples, chosen at random. Mini-batch SGD reduces the amount of noise in SGD but is still more efficient than full-batch.

Disclaimer

- SGD is only one way to go about updating the weights and biases.
- There are many others
  - some look at the moving average of the gradients (as opposed to simply the most recent gradient)
  - Others adapt the learning rate based on the gradient values
  - Others combine the two approaches

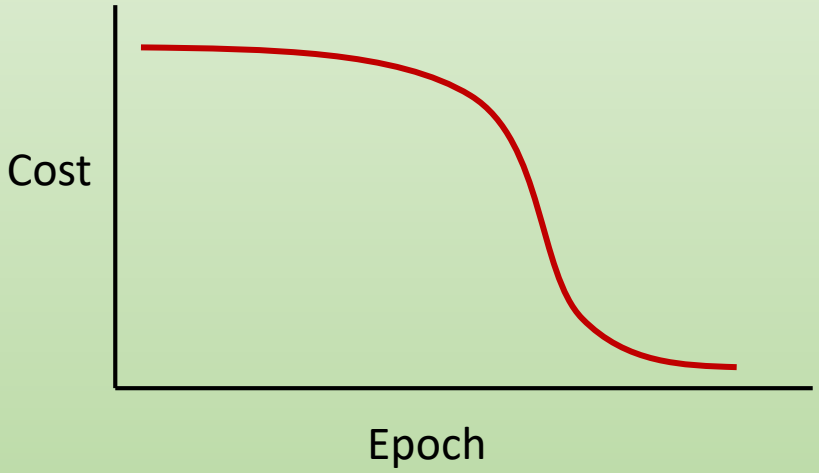


https://towardsdatascience.com/

Slow learning problem

- Typically (and hopefully), humans learn quickly when they are very wrong
  e.g., “it is fun to put your hand in the pretty fire”

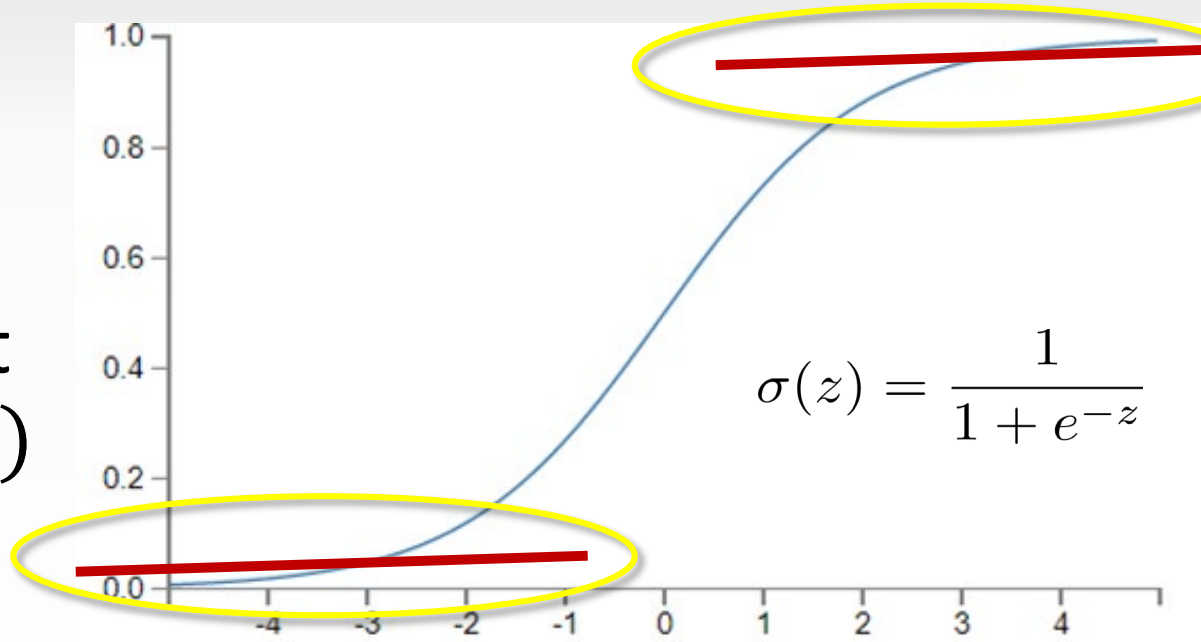
- However, our sigmoidal based neural network does not generally exhibit this behavior: *even being “very” wrong, ANNs may take many epochs to learn better weights and biases*



- To understand why, recall neurons learn by changing weight and bias at a rate determined by the gradient

Slow learning problem

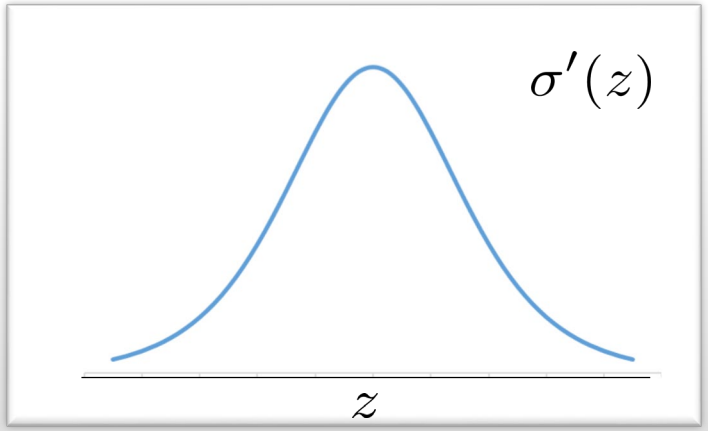
For sigmoidal neurons with a quadratic cost function, the gradient is a function of the derivative of $\sigma(z)$



Consider simple case:
single sigmoidal neuron
input: x=1, target value y=0

For brevity, only consider partial derivative w.r.t. to weight

$$\frac{\partial C}{\partial w} = (\sigma(z) - y)\sigma'(z)x = \sigma(z)\sigma'(z)$$



One approach to the slow learner problem

- Change the cost function!
- You are much more likely to see and use (and *get better, faster results*) from **cross-entropy loss (a.k.a. log-loss)**

$$-\sum_i \left[y_i \log p_i + (1 - y_i) \log(1 - p_i)\right]$$

where:
$y \in \{0, 1\}$ for dichotomous targets
or, $0 \leq y \leq 1$ for continuous targets (scaled)

or

$$-\sum_{i,c \in C} y_{i,c} \log p_{i,c}$$

where:
$y_{i,c} \in \{0, 1\} \, \forall c \in C$ (set of classes)

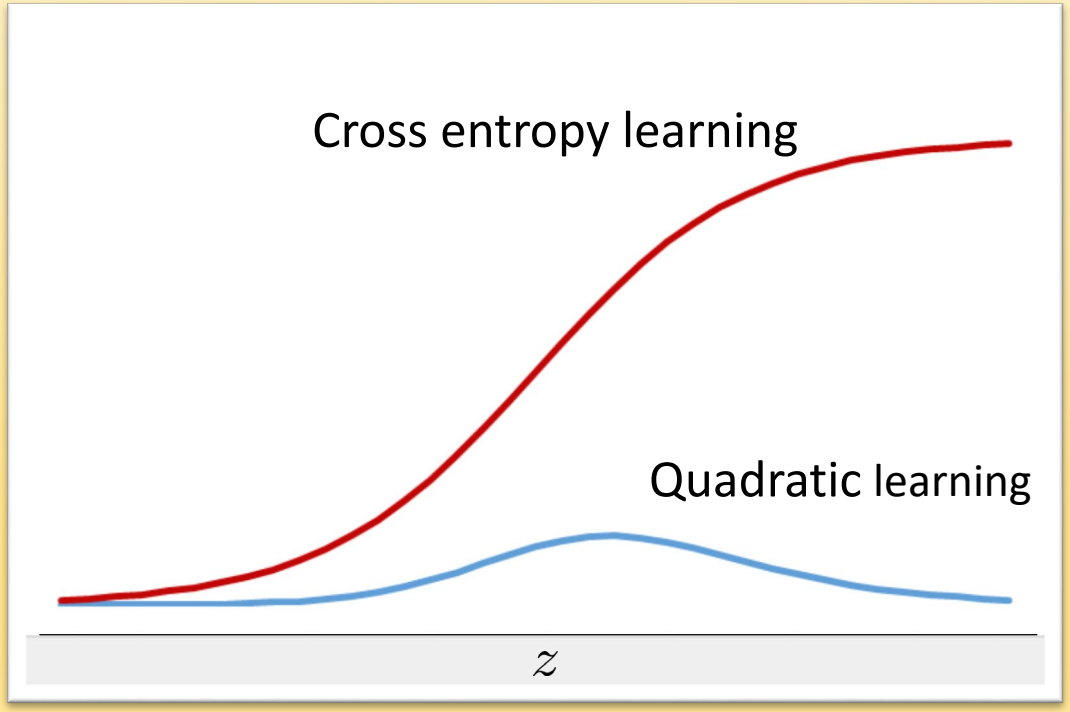
Slow learning problem

For sigmoidal neurons with a **cross entropy loss**, the gradient is a function of ~~the derivative of~~ $\sigma(z)$

For our simple case,

$$\frac{\partial C}{\partial w} = x(\sigma(z) - y)$$

Now, compare the gradients →



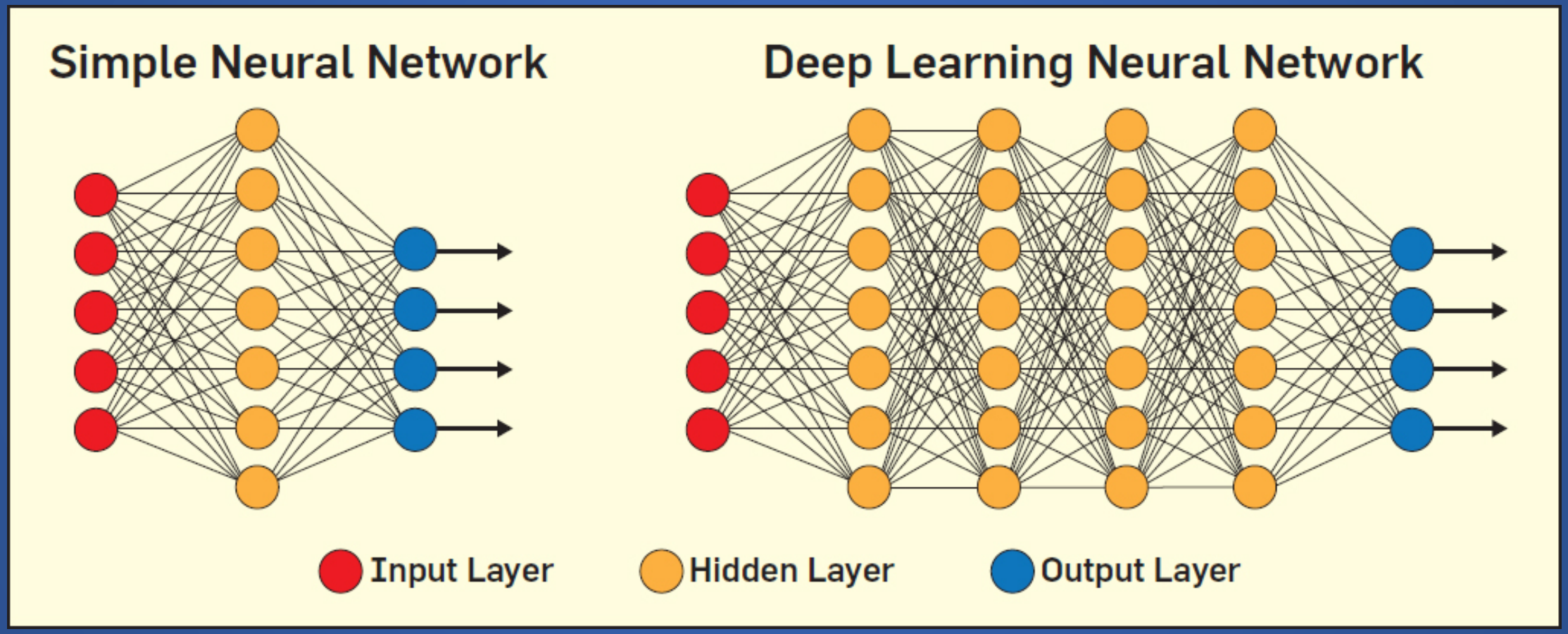
And while I'm being honest…

There are many options for cost functions, e.g.,

- Huber loss (from robust regression lectures)
- Mean absolute error (a.k.a. L1 error – also from robust regression lectures)
- Kullback–Leibler divergence (a.k.a. Information Gain from the lectures on decision trees)
- Itakura-Saito distance

…and regularization (e.g., like we saw with Ridge Regression and LASSO) can be used in conjunction with loss functions

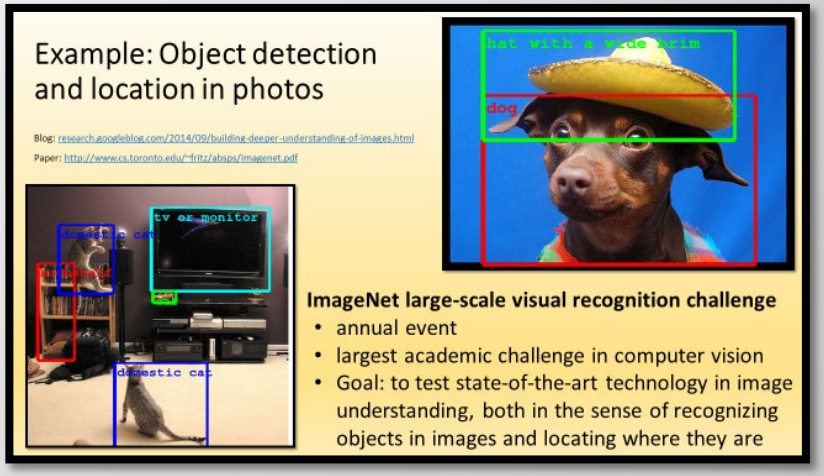
Deep Learning



Deep Learning Overview

1. Yes, deep learning is all the rage right now.
2. Yes, it is an awesome approach for many (but not all) problems.

Deep learning refers to neural networks with more than one hidden layer (often, there are several hidden layers) → current trend of *increasing* depth



ImageNet Large Scale Visual Recognition Challenge

Winners
- **2012, 2013: 8 layers***
- **2014: 19-22 layers***
- **2015: 152 layers***
- **2016: 269+ layers***

Implications of this:
- Highly flexible, highly non-linear
- In some sense, automatic feature engineering
- Not interpretable
- Must be careful to not over fit
- Time consuming to train

Vanishing Gradient Problem

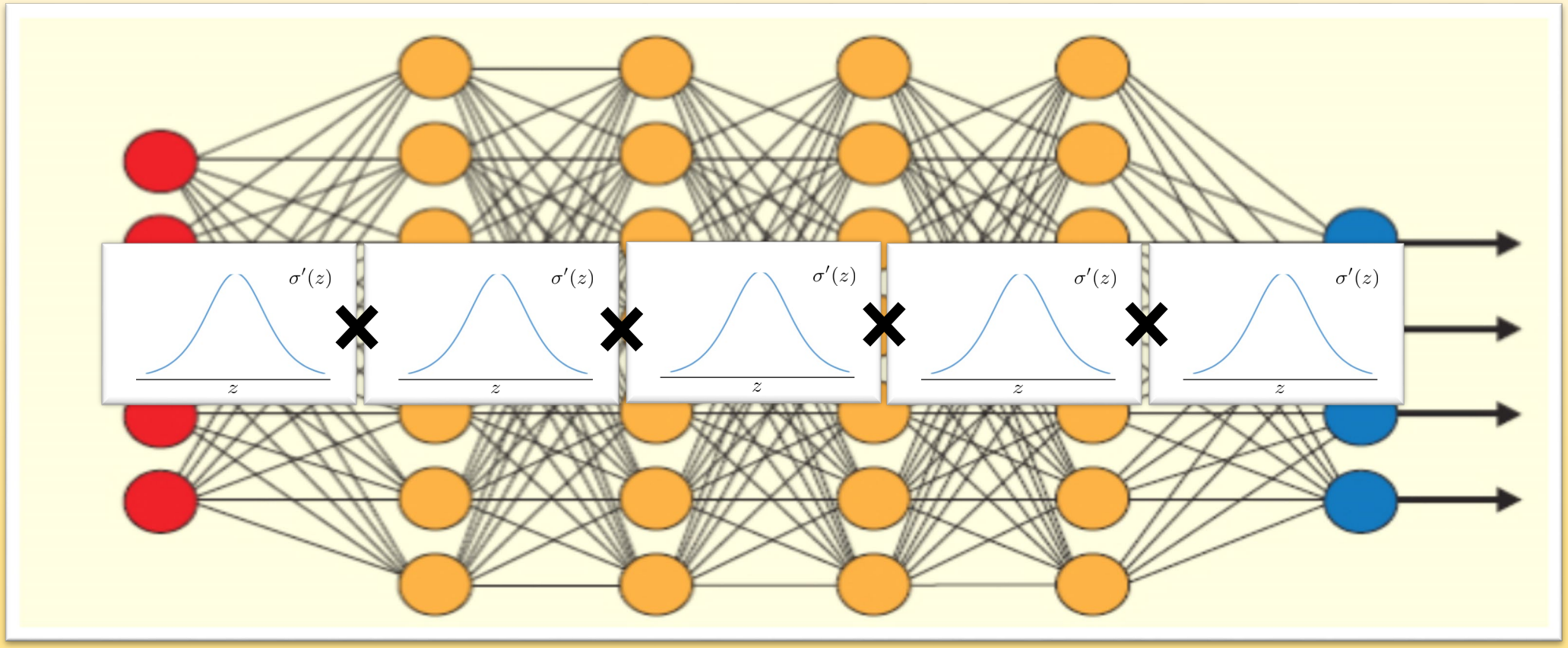
Another unfortunate implication of many layers when using backpropagation is the ***vanishing gradient problem***

Vanishing gradient problem: In backpropagation, the gradient in the earlier layers is a product of the gradients from the later layers; these latter gradients may be small (< 1), and their product may become vanishingly small. As such, early layers may not be trained.

$$\frac{\partial C}{\partial W_l} \propto f'(z^{(l)}) f'(z^{(l+1)}) f'(z^{(l+2)}) \ldots$$

$$\text{where } f \text{ is an activation function}$$

Vanishing Gradient Problem



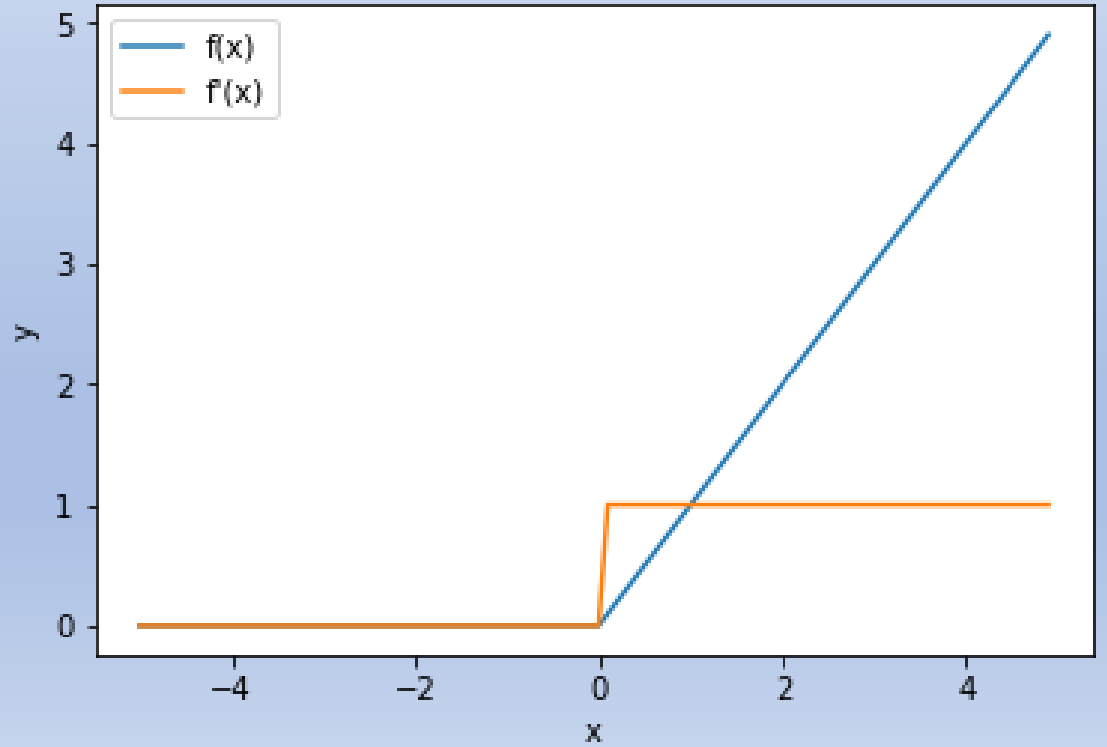
$$0.25 \times 0.25 \times 0.25 \times 0.25 \times 0.25 \approx 0.000976$$

Rectified linear unit activation function

- Use better activation functions
  - s.t., the product of the derivative will not get too small
- Basic idea behind **rectified linear unit activation function** (a.k.a. **ReLU**)

$$\text{ReLU: } f(z) = \max(0, z)$$

$$f'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$

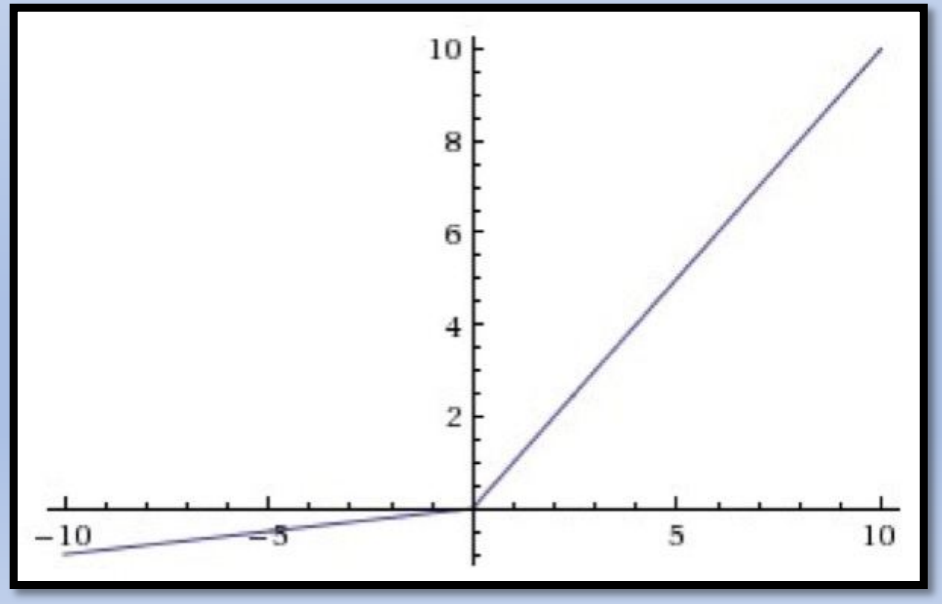


Leaky rectified linear unit activation function

- One problem with the ReLU activation:
  - back-propagated error and learning is impacted whenever there is a negative input into a given neuron (i.e., $z \leq 0$), since $\frac{\partial C}{\partial W_l} = 0$
  - Thus, no way for the associated weights to update in the right direction

$$\text{Leaky ReLU: } f(z) = \max(0.01z, z)$$

$$f'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0.01, & \text{otherwise} \end{cases}$$



Parameterized ReLU

$$\text{Parameterized ReLU: } f(z) = \max(\alpha z, z)$$

$$f'(z) = \begin{cases} 1, & \text{if } z > 0 \\ \alpha, & \text{otherwise} \end{cases}$$

where α is a trainable parameter

One more activation function

Commonly for classification problems, there is an output layer node for every class. It is convenient to interpret the output activation as an estimate of the probability of the classification.

E.g., assume set of C classes, then

$$0 \leq f(z_c^L) \leq 1 \quad \forall c \in C \text{ and } \sum_{c \in C} f(z_c^L) = 1$$

If the last layer is Leakly ReLU (or ReLU or Sigmoidal or tanh), this is not a natural outcome.

Softmax activation

- Activation of neuron c in output layer is function of all output activations

$$\text{Softmax: } f_c(z) = \frac{e^{z_c^L}}{\sum_{\gamma \in C} e^{z_\gamma^L}}$$

$$0 \leq f(z_c^L) \leq 1 \;\; \forall c \in C \text{ and } \sum_{c \in C} f(z_c^L) = 1$$

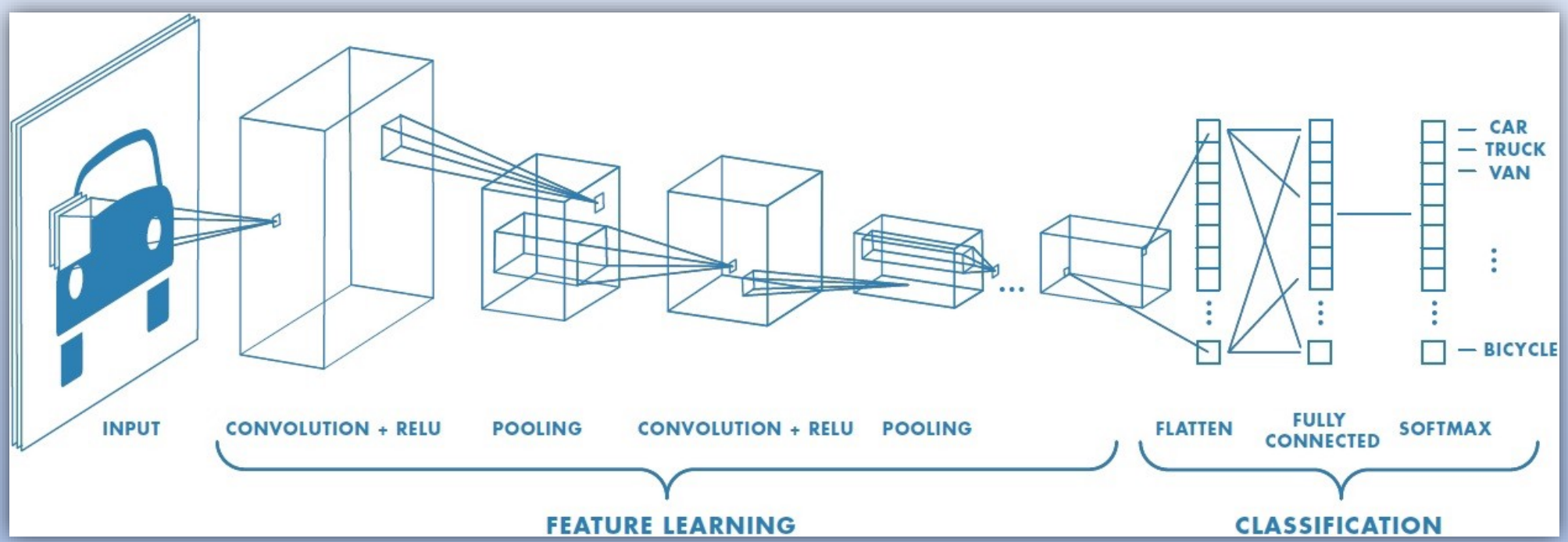
Finally, deep learning

- One neuron per input
- Many hidden layers*
- Cross-entropy cost function (or others)
- Hidden layer activations often ReLU (or parameterized ReLU)
- Typically, for classification
  - One output neuron per class
  - Output layer activation is softmax
- Typically, for regression
  - One output neuron
  - Output activation is linear or sigmoid

Deep Learning Network Architecture

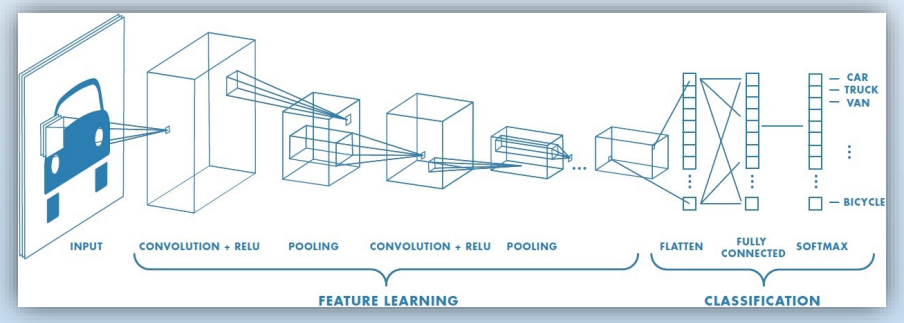
- Deep Feed Forward Neural Network
- Convolutional Neural Network (CNN)
  - Commonly used in Computer Vision
  - Major core of deep learning
- Recurrent Neural Network (RNN)
  - Useful when time or sequence is important, e.g., speech to text, question answering
  - Siri, Cortana, Alexa, etc.
- Generative Adversarial Networks (GAN)

Convolutional Neural Networks (CNNs)



Convolutional Neural Networks

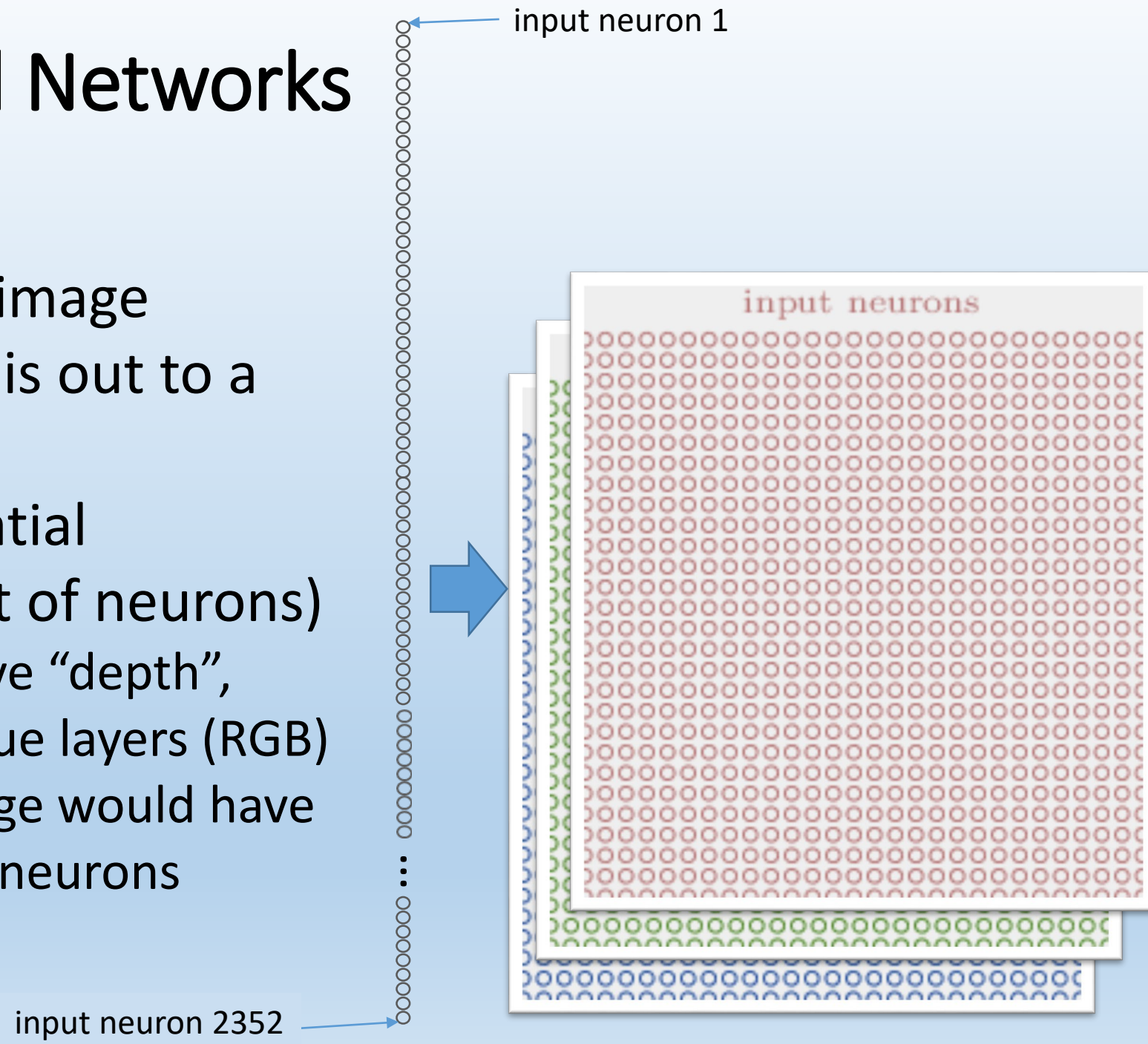
- CNNs first developed for image analysis
- Similarities:
  - Neurons, links, weights, biases, activations
  - Many hidden layers
  - Backpropagation*



- Major differences:
  - Not all neurons are connected to all neurons in next layer (locally connected, receptive field)
    - Very specific pattern for connecting inputs to hidden layer neurons
  - Weights and biases are often shared (convolving, stride length, zero padding, feature maps)
  - Pooling a.k.a. sampling

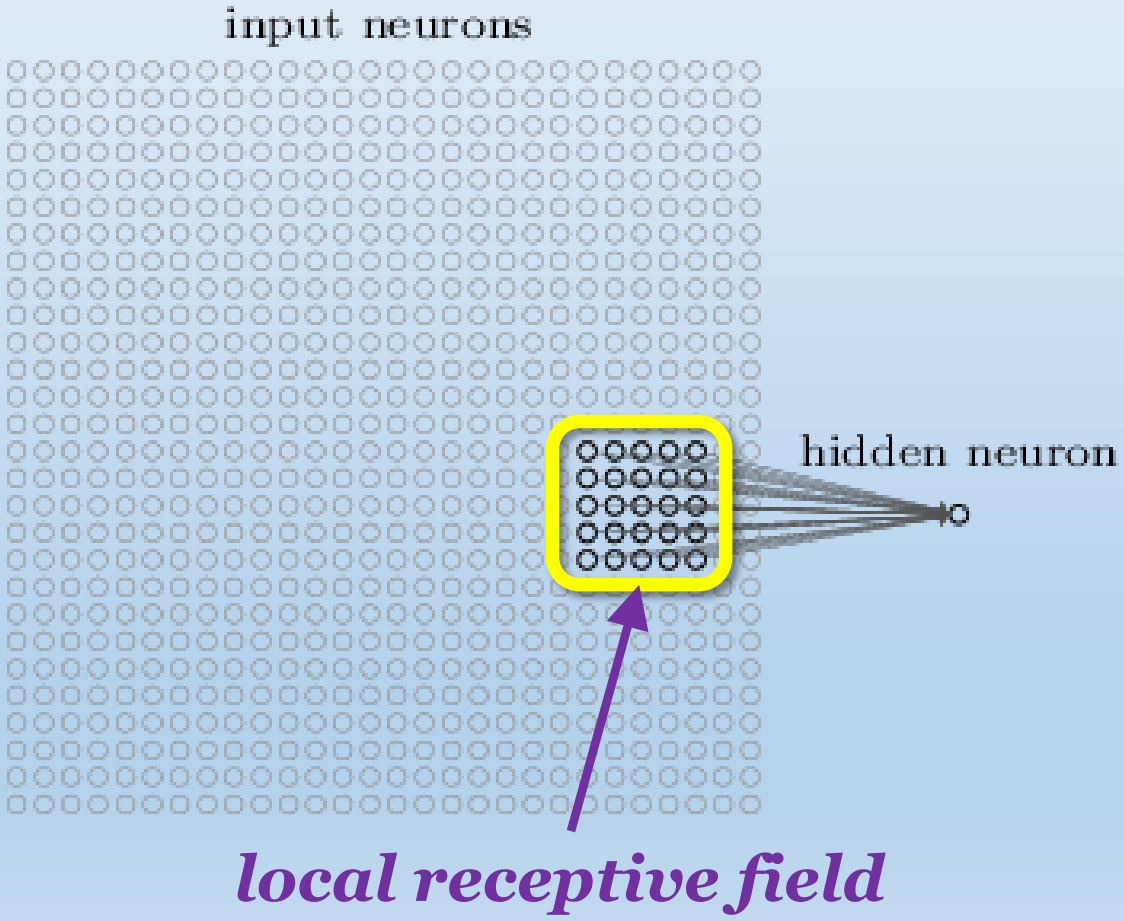
Convolutional Neural Networks

- Assume a 28x28 B&W image
- Traditionally, stretch this out to a 784 input vector
- CNN's preserve the spatial structure (28x28 layout of neurons)
  - Note: Images often have "depth", e.g., red, green, and blue layers (RGB)
  - e.g., a 28x28 color image would have 28x28x3 = 2,352 input neurons

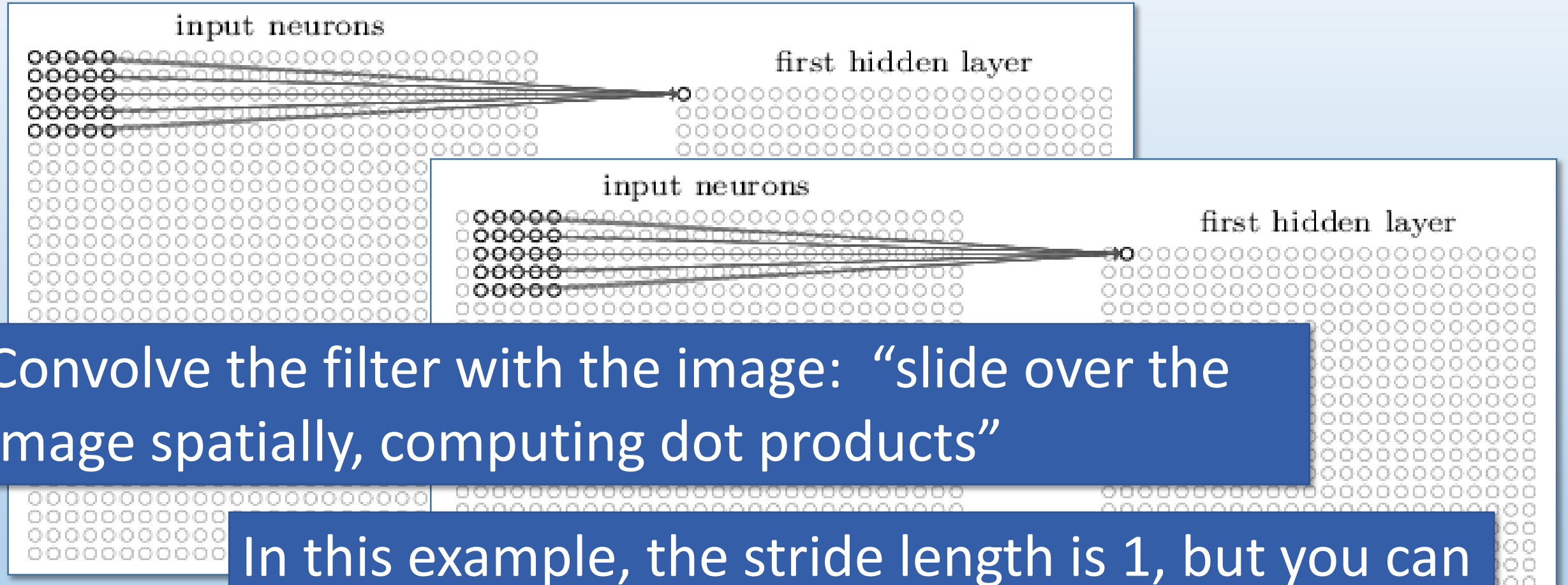


Convolutional Neural Networks: Receptive Field

- Sets of related input neurons are grouped together and connected to hidden neuron
  - *local receptive field*
  - closely related to *filter*, *kernel*
- e.g., with an image a neuron in first hidden layer will be connected to a small local region of the input neurons of perhaps a 5×5 region, corresponding to 25 input pixels



CNNs: Convolving and stride length



Convolve the filter with the image: "slide over the image spatially, computing dot products"

In this example, the stride length is 1, but you can use a longer stride length, which would reduce the number of hidden neurons in the first layer

Convolving

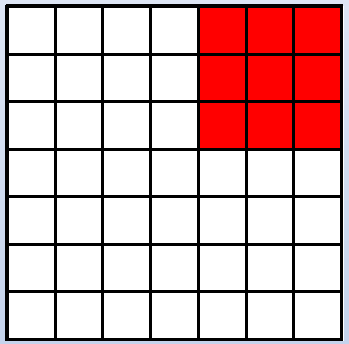
Output size for *N*x*N* image, *F*x*F* filter and stride length *S*

$$\frac{(N-F)}{S}+1$$

Must be integer!

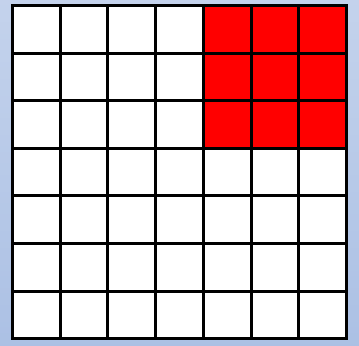
7x7 input

stride length: 1



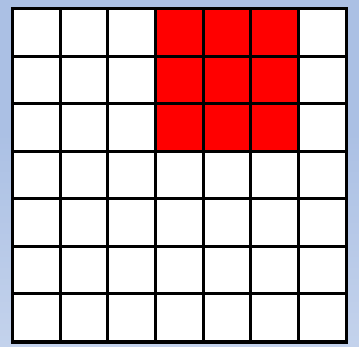
5x5 output

stride length: 2



3x3 output

stride length: 3



No!

Does not fit! Cannot apply 3x3 filter on 7x7 input with stride 3

Convolving: Zero-Padding

- Common to zero pad the image, e.g. a single pixel border of 0's
- Output size for *N*x*N* image, *F*x*F* filter, stride *S*, and padding *P*

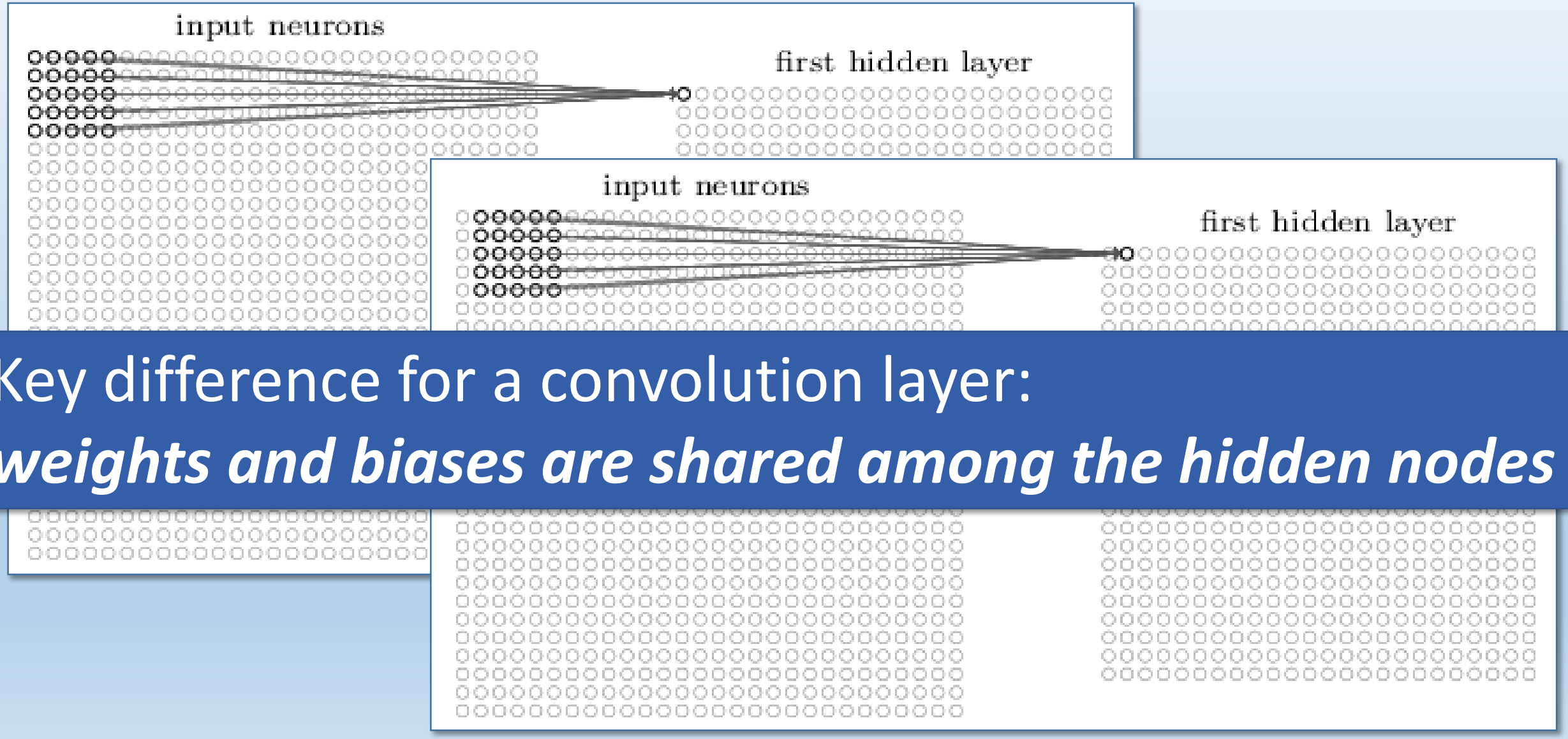
$$\frac{(N + 2P - F)}{S} + 1$$

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

What is output size of 7x7 image, with 3x3 filter, stride length 3, and single pixel border padding of 0's? → **Output size: 3x3**

What is output size of 7x7 image, with 3x3 filter, stride length 1, and single pixel border padding of 0's? → **Output size: 7x7**

CNNs: Shared weights, biases

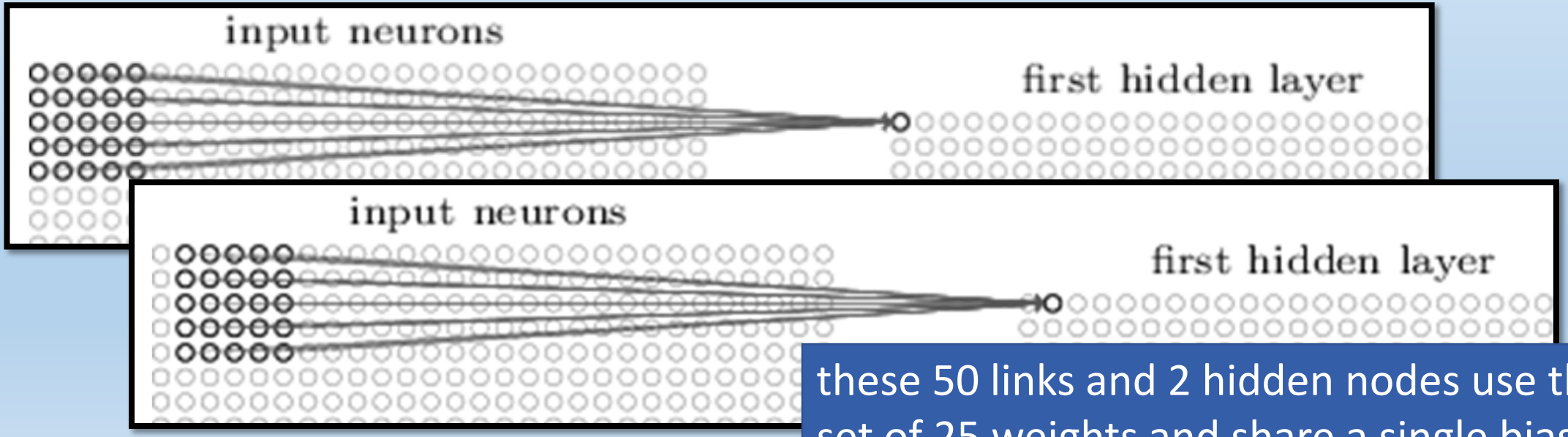


Key difference for a convolution layer:

weights and biases are shared among the hidden nodes

CNNs: Shared weights, biases

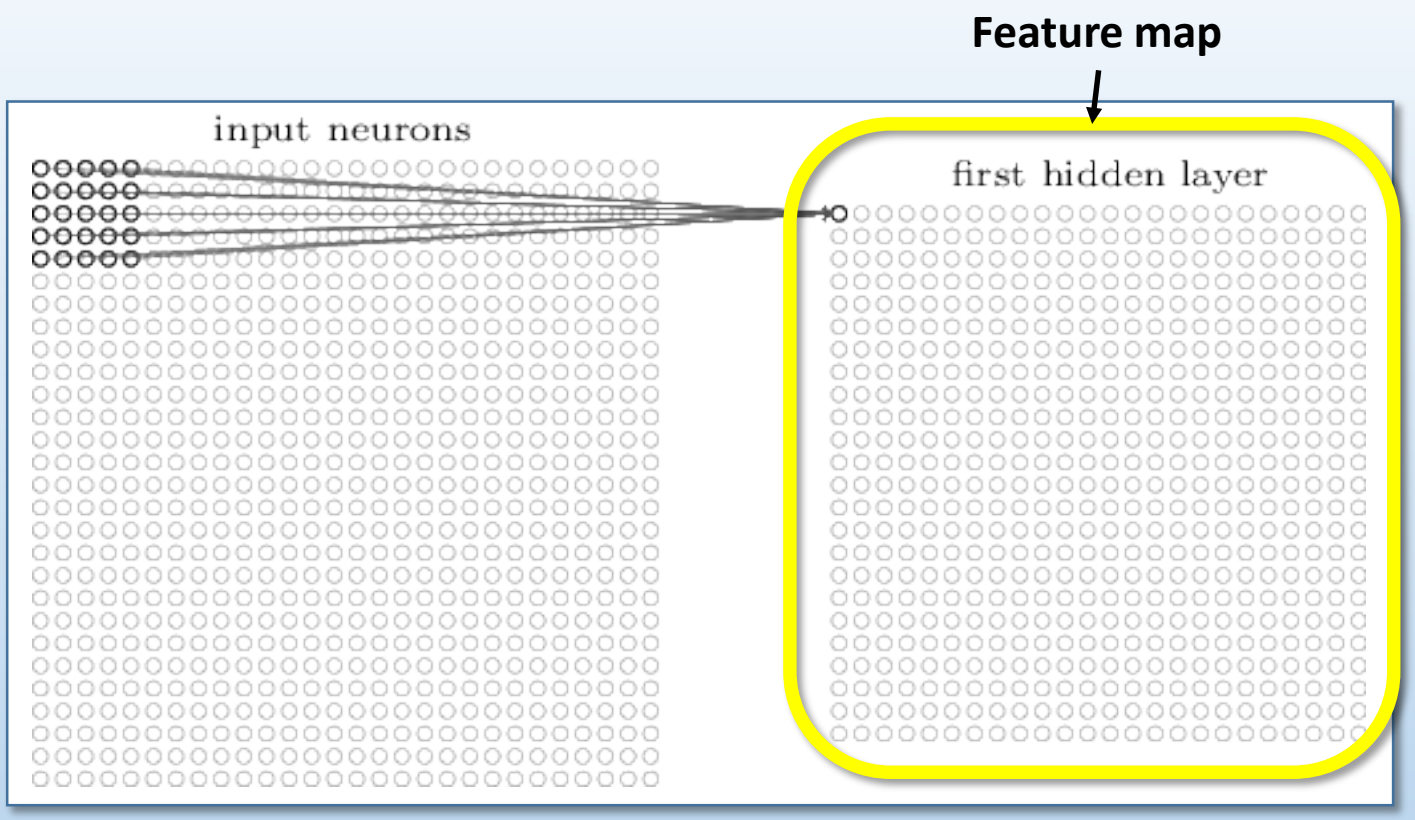
- In the case of a 5x5 filter, with stride 1, for a 28x28 B&W image, how many weights? biases?
  - instead of 5x5x24x24=14,400 weights, there would be only 25 weights
  - instead of 24x24=576 biases, there is only 1 bias



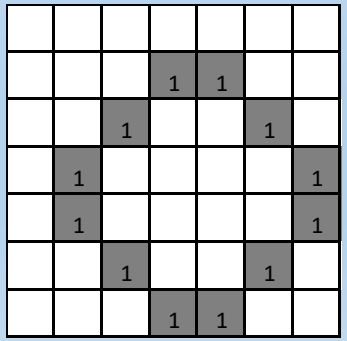
these 50 links and 2 hidden nodes use the same set of 25 weights and share a single bias value

CNNs: Feature maps

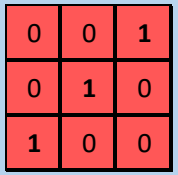
- After convolving the filter over the image, the activations compose a “feature map”
- The idea is that the network will learn a feature that exists in the network, e.g.,

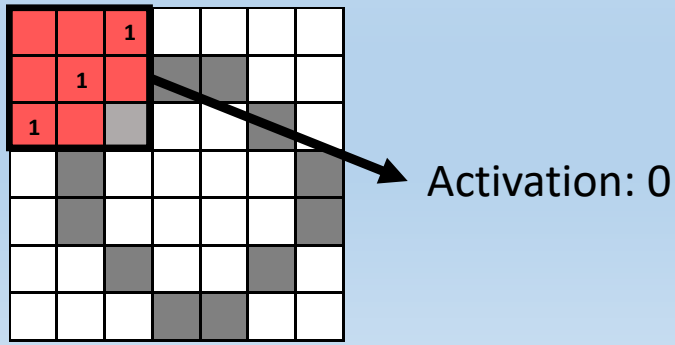


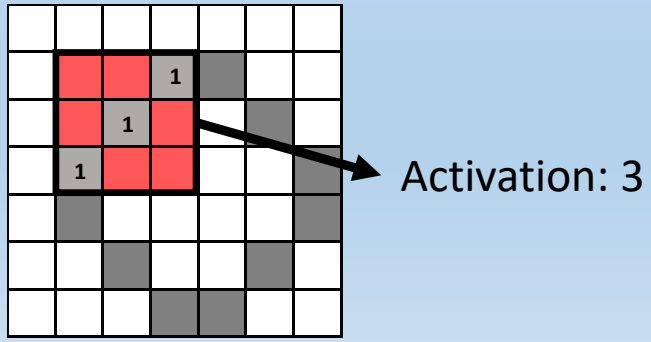
Image



Filter

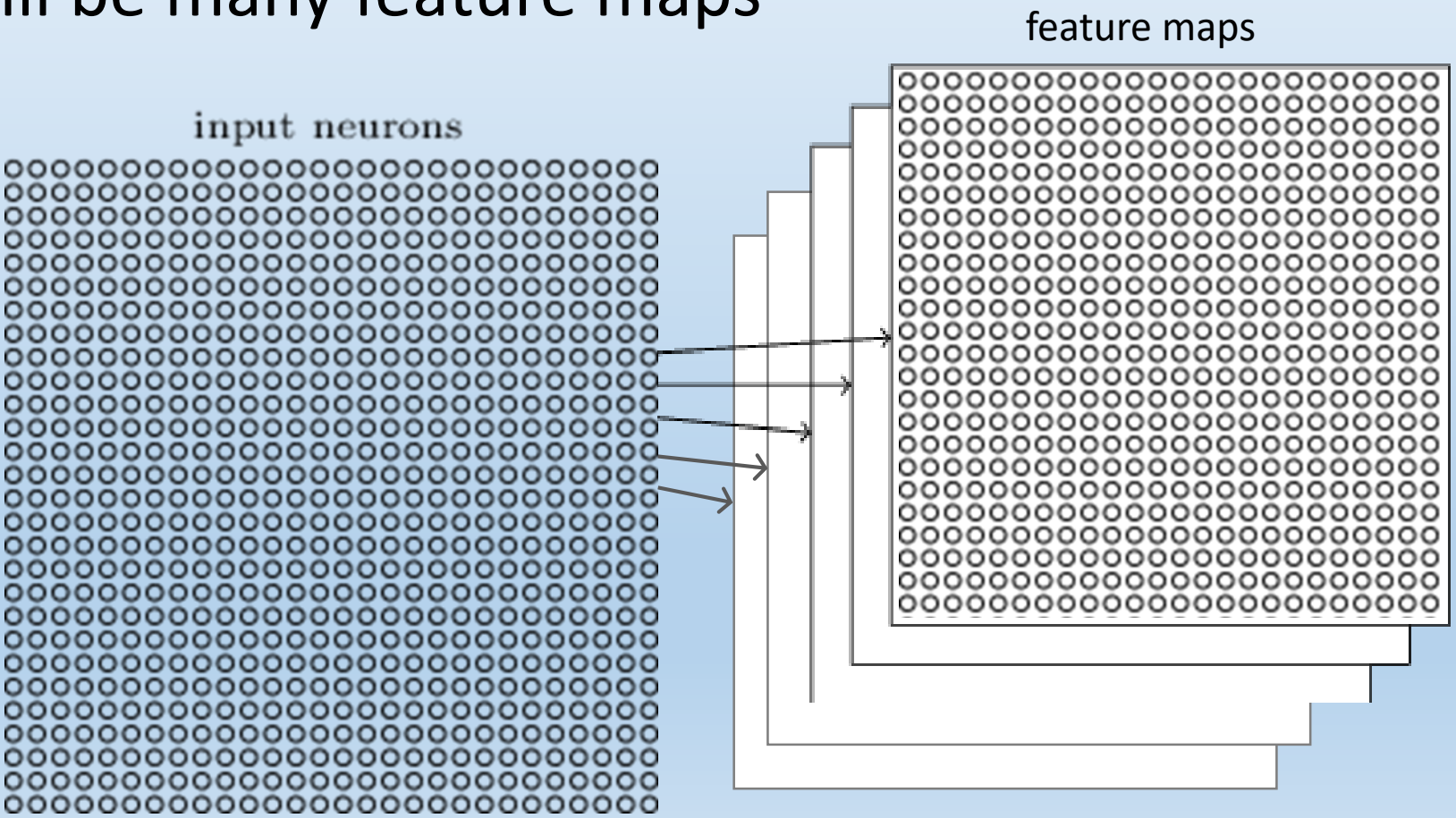






CNNs: Feature maps

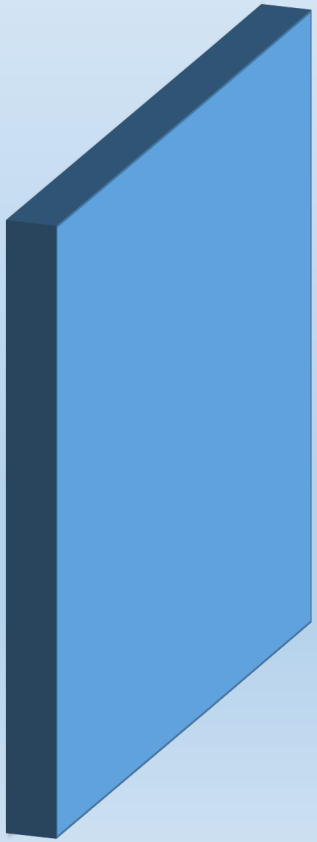
- CNN's usually wish to identify many features
- As such, there will be many feature maps



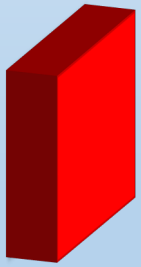
CNNs: Feature maps

- Images usually have “depth”, e.g., RGB levels

28 X 28 X 3 image

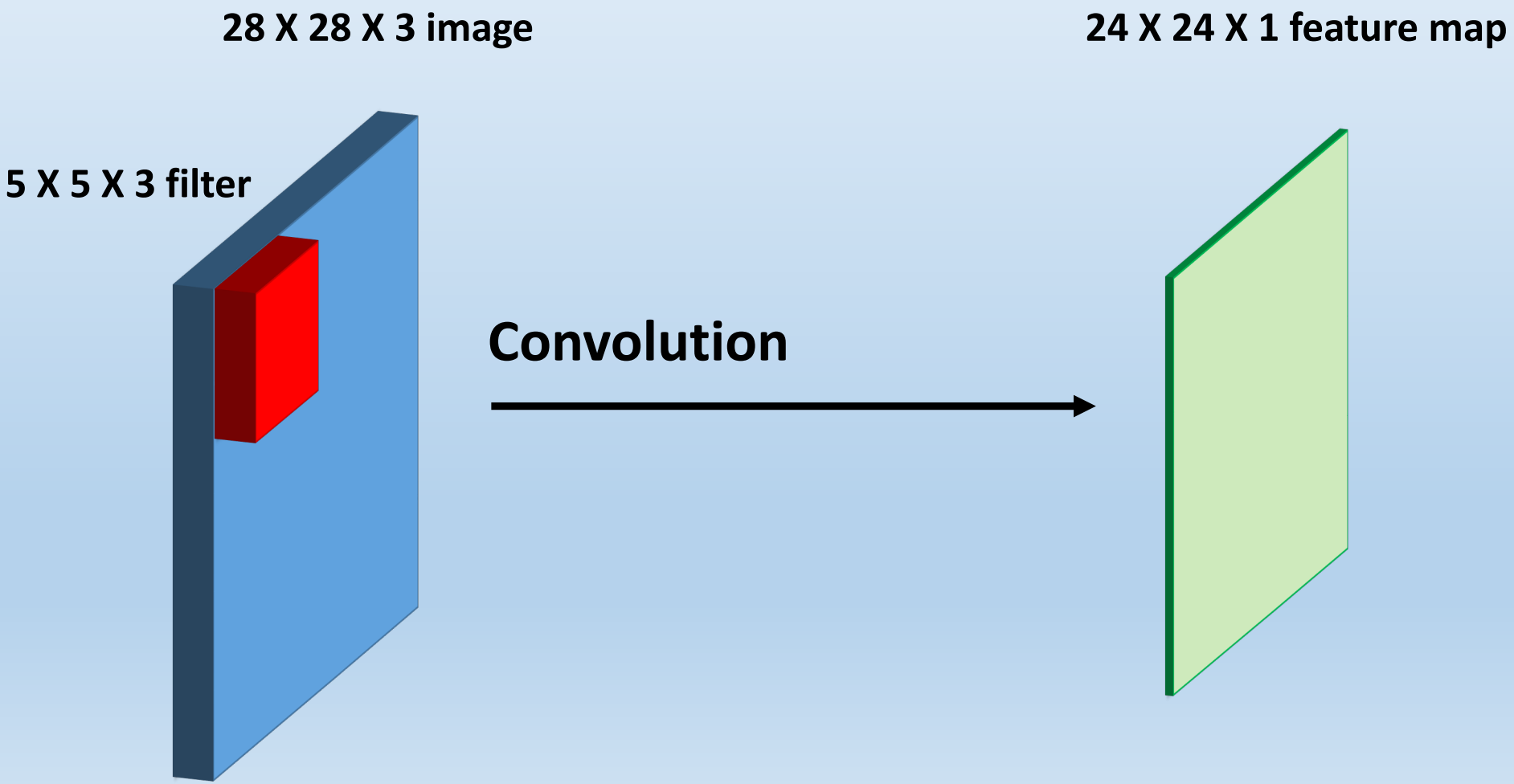


5 X 5 X 3 filter



CNNs: Feature maps

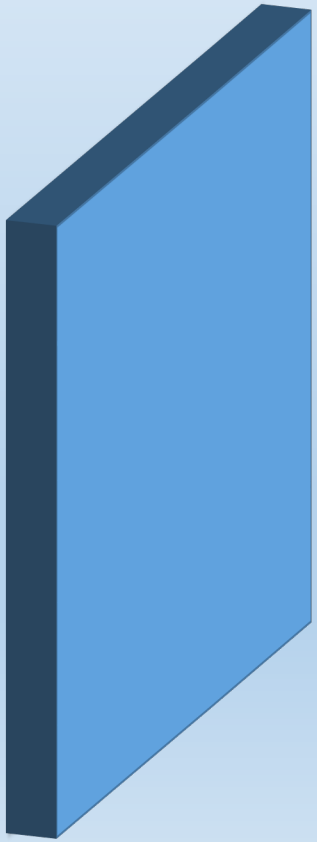
- Images usually have “depth”, e.g., RGB levels



CNNs: Feature maps

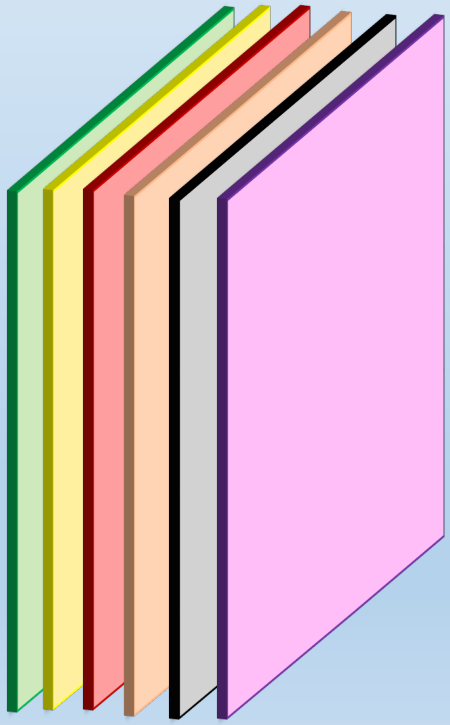
- Images usually have “depth”, e.g., RGB levels

28 X 28 X 3 image

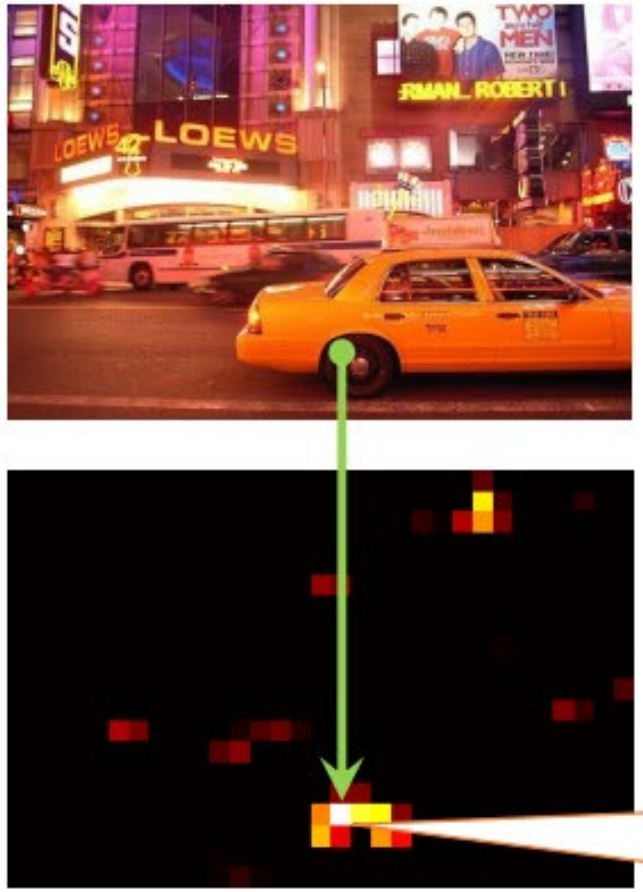


If we had 6 different 5x5 filters, we’ll get 6 separate feature maps

24 X 24 X 6 output volume



Individual feature maps in an output volume are sometimes called “slices” or “depth slices”

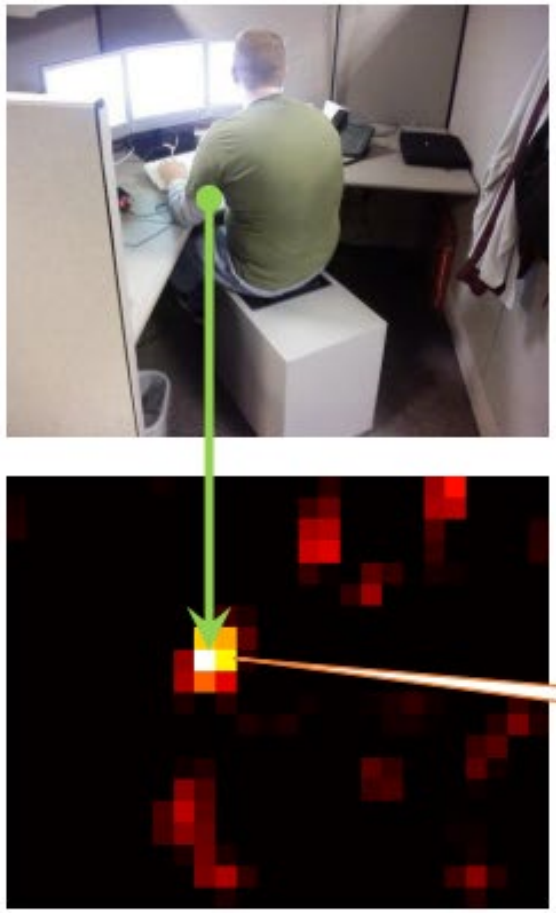


one feature map of $conv_5$
(#55 in 256 channels of a model trained on ImageNet)

ImageNet images with strongest responses of this channel

Intuition of *this* response:
There is a "circle-shaped" object (likely a tire) at this position.

What **Where**



one feature map of $conv_5$
(#66 in 256 channels of a model trained on ImageNet)

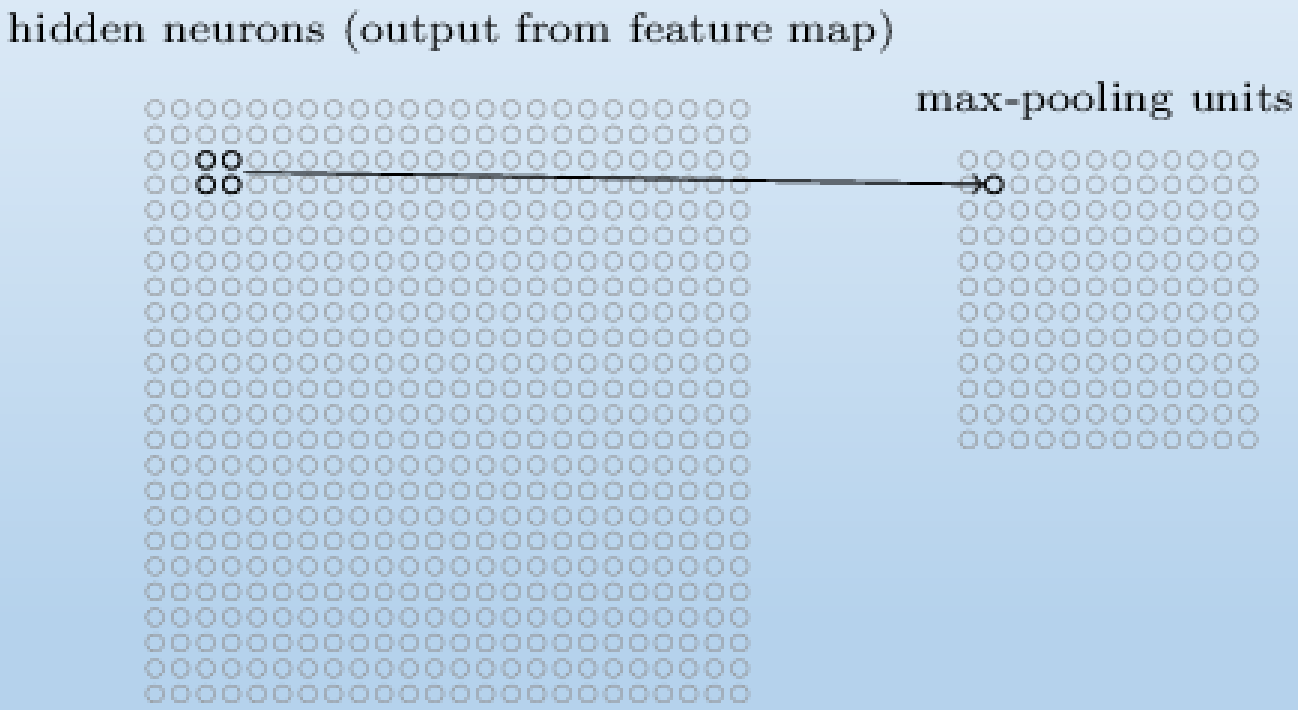
ImageNet images with strongest responses of this channel

Intuition of *this* response:
There is a "λ-shaped" object (likely an underarm) at this position.

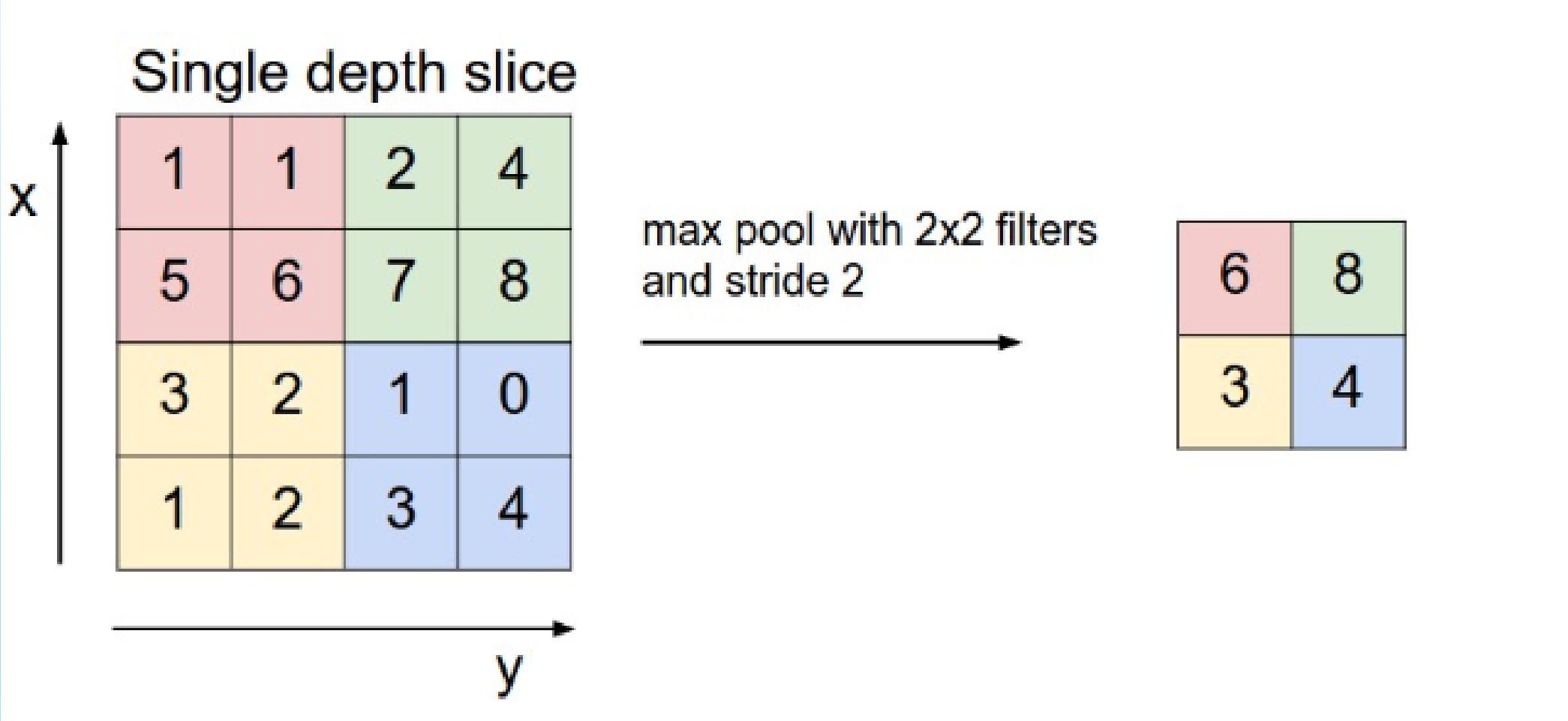
What **Where**

CNNs: Pooling

- Makes the representations smaller and more manageable
- Operates over each *feature map* independently
- Also called “sampling” layer
- Max pooling is typical
  - e.g., only keeps max value within a region



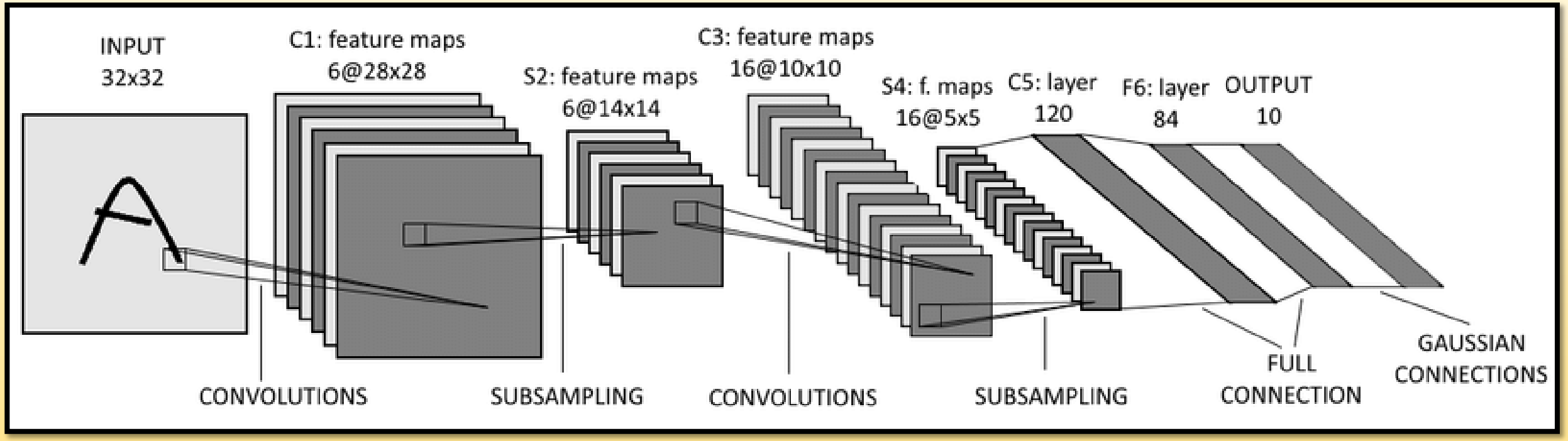
CNNs: Max pooling example



CNNs: Overall Architecture

- Spatially structured input neurons
- Many filters convolved with input to produce multiple feature maps (each with shared weights and biases)
  - Activation function is commonly ReLU or parameterized ReLU
- Pooling a.k.a. sampling is used after a convolution layer
  - No activation function needed
- One or more subsequent convolutional layers and pooling are used
- Final layers are often fully connected
  - Commonly use softmax for connections to final output layer

LeNet5 (1998)



Filters were 5x5, applied at stride 1

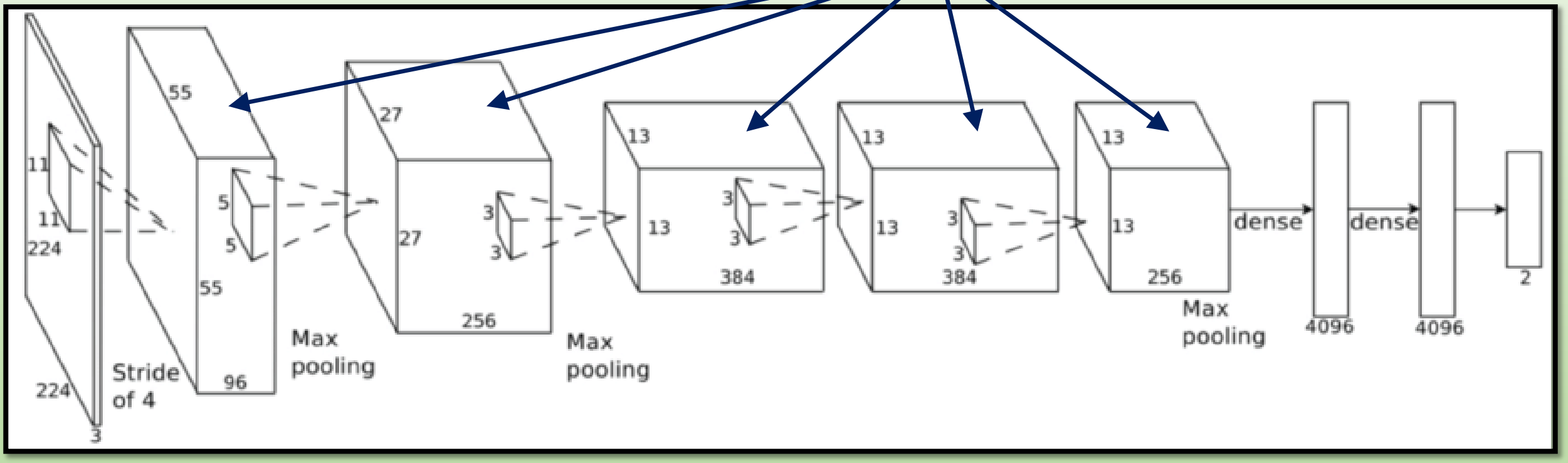
Subsampling (Pooling) layers were 2x2 applied at stride 2

Architecture is [CONV-POOL-CONV-POOL-FC-FC]

~60,000 trainable parameters

AlexNet (2012)

5 Convolution Layers (96, 256, 384, 384, 256 filters)



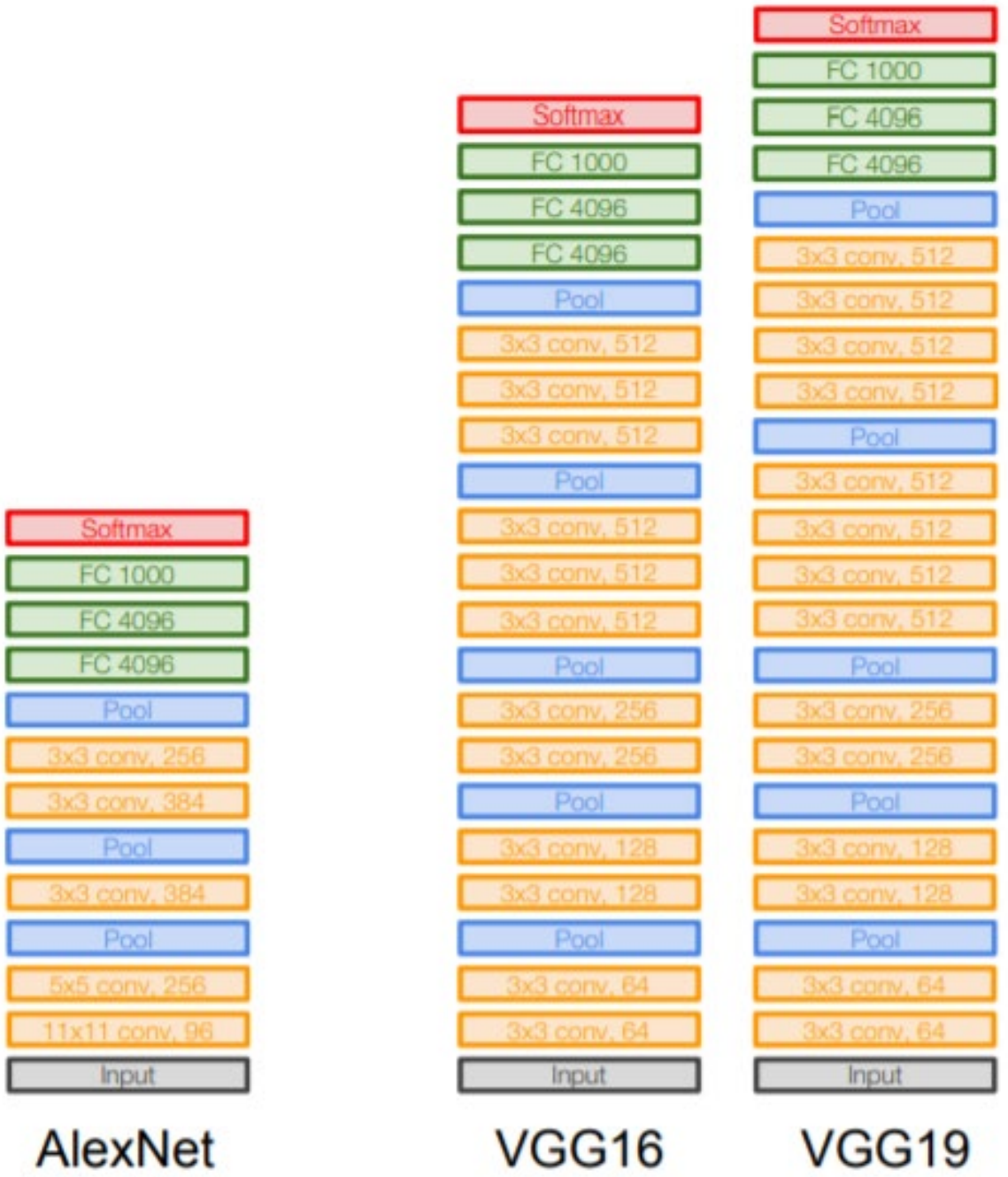
Conv1: Filters were 11x11, applied at stride 1; Conv2: 5x5 filters at stride 1, pad 2; Conv3-5: 3x3 filters at stride 1, pad 1

All pooling layers: 3x3 applied at stride 2

~60,000,000 trainable parameters

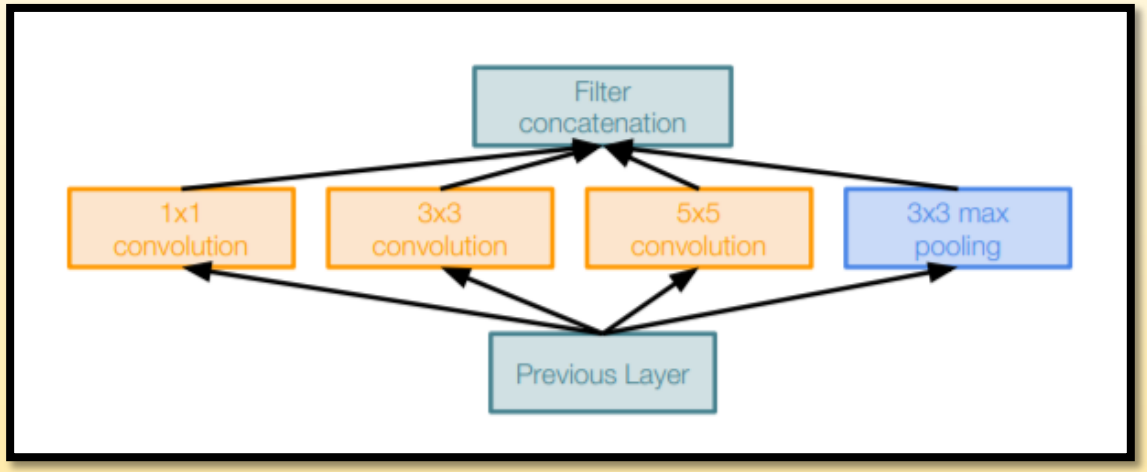
VGG16, VGG19

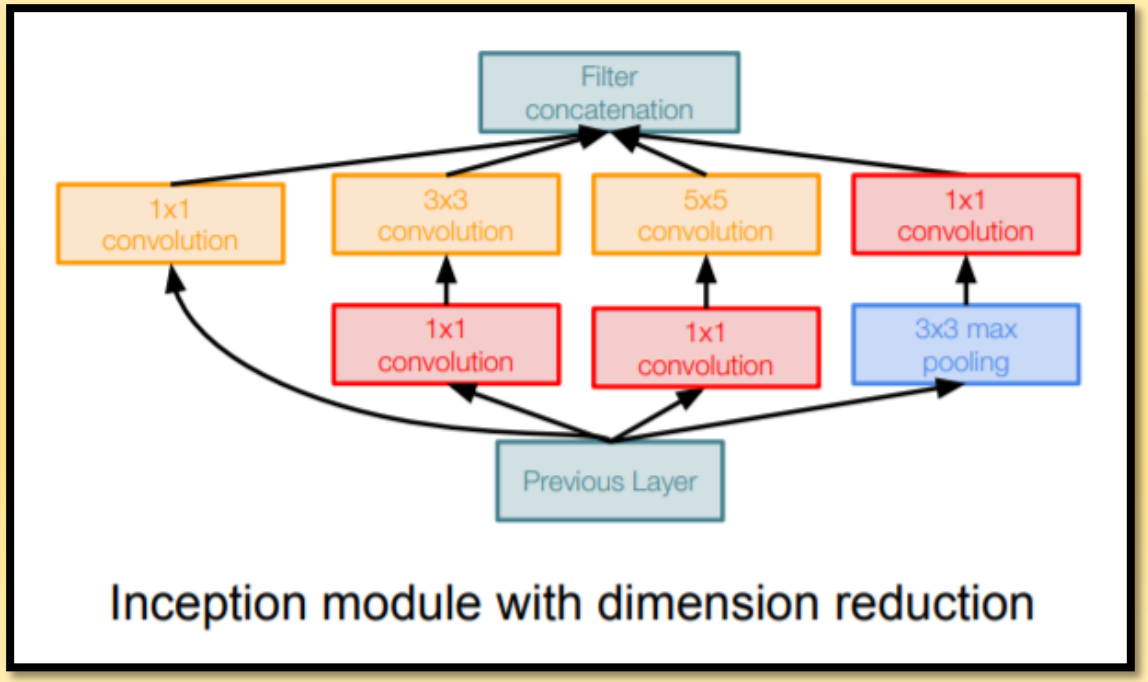
- VGG16 and VGG19 had 16 and 19 layers, respectively
- Used:
  - 3x3 convolutions, stride 1, pad 1
  - 2x2 max pooling, stride 2
  - FC layers



GoogLeNet and Inception Modules (2014)

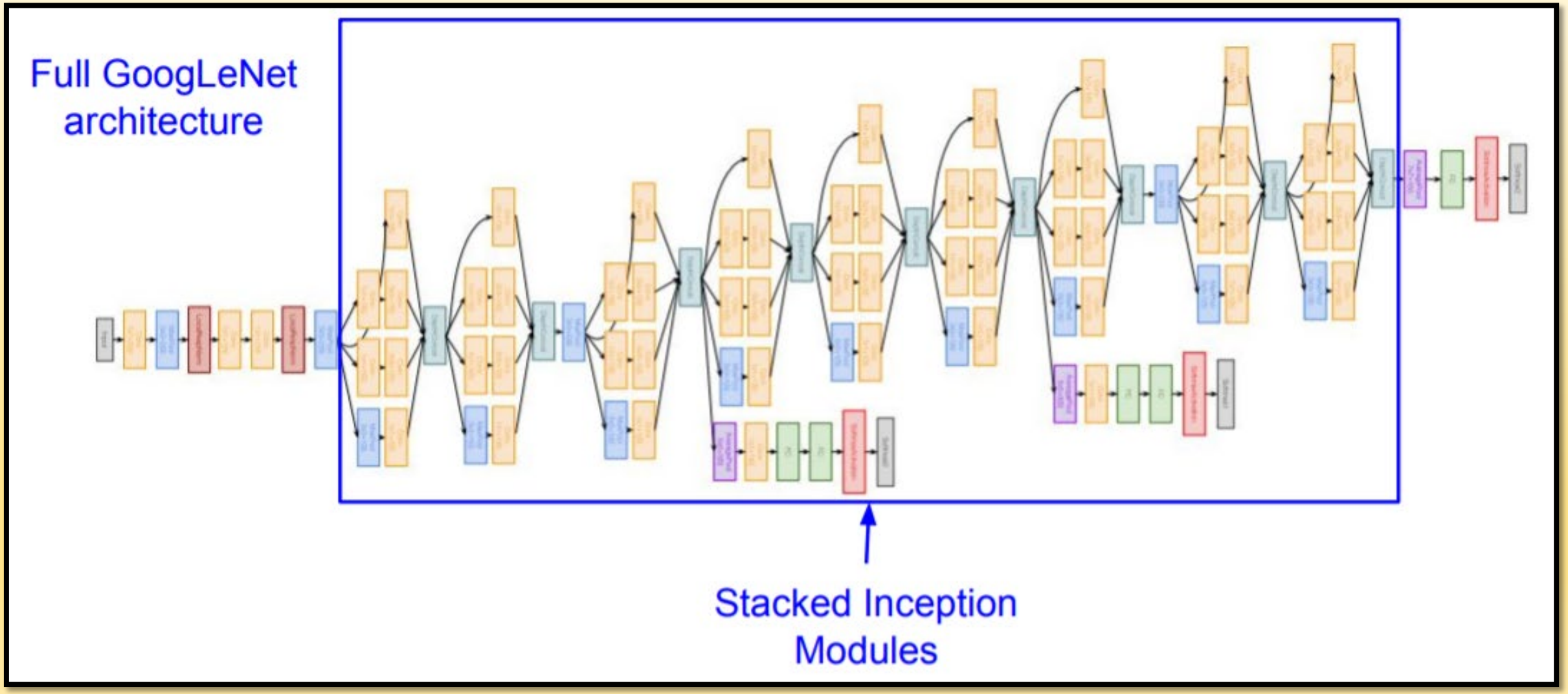
- Inception: Apply parallel filters (of different sizes and depths) to previous layer
- Combine the filters
- Computationally expensive, so dimension reduction is also used via 1x1 convolutions



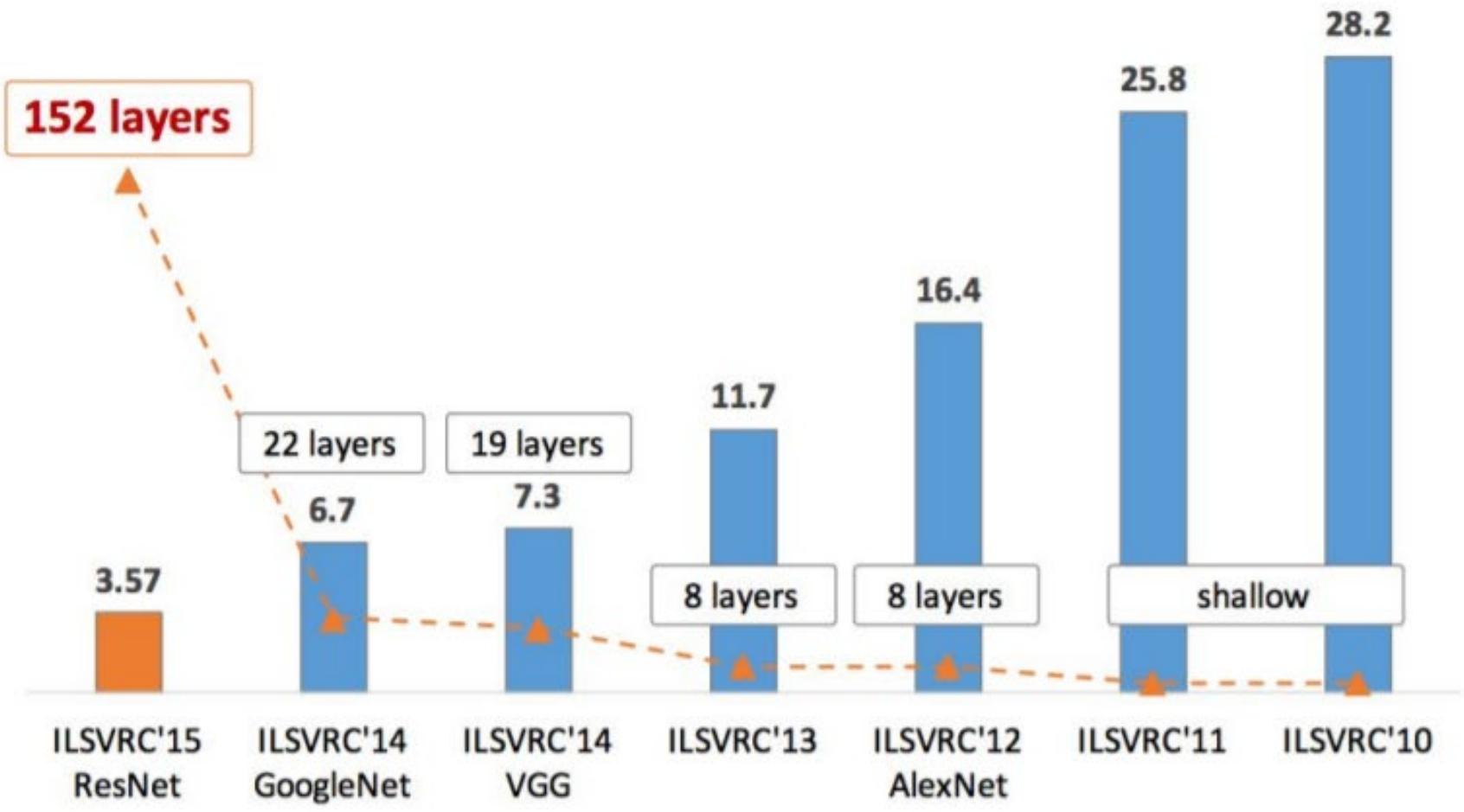


Inception module with dimension reduction

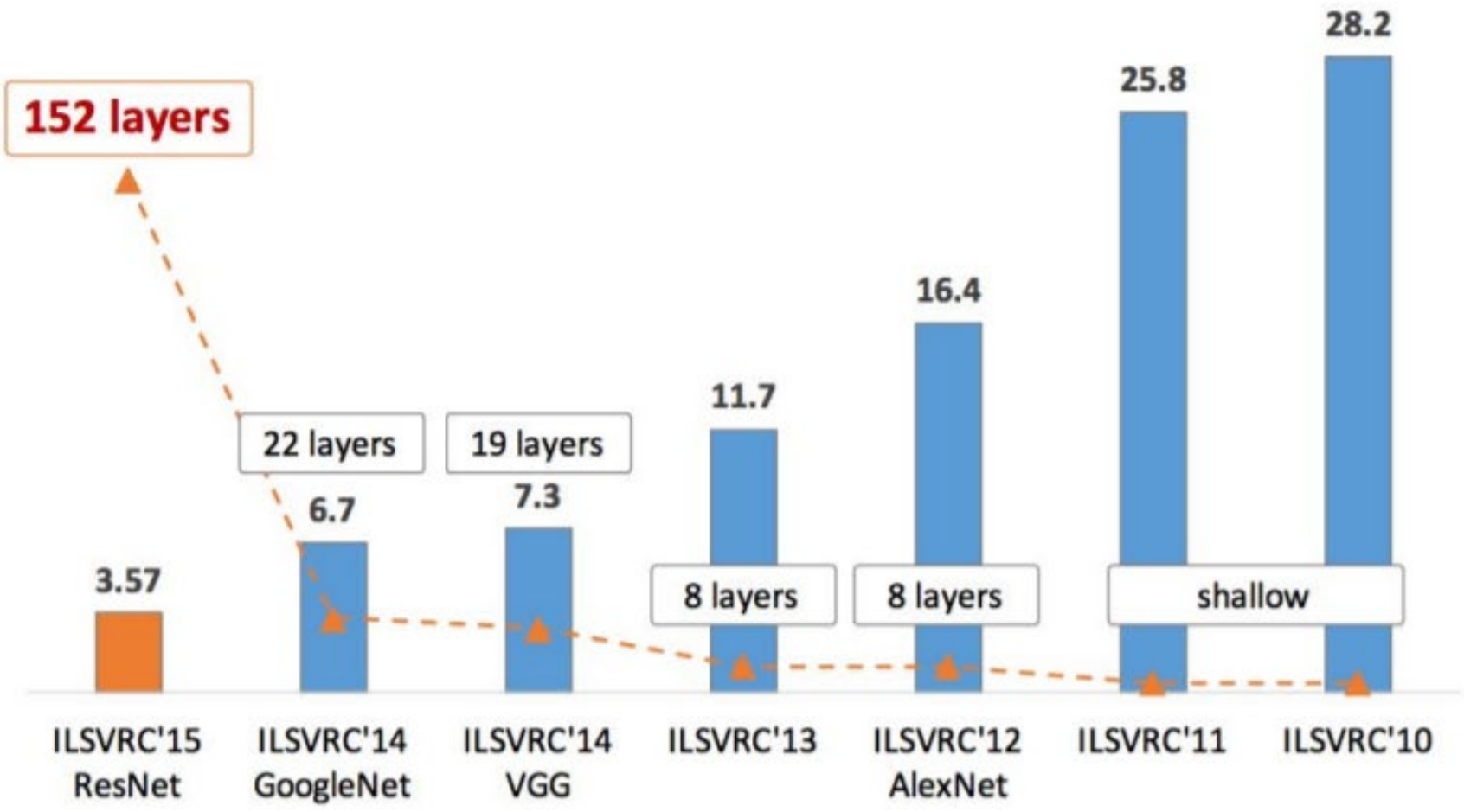
GoogLeNet (2014)



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Other Deep Learning Classes

- Recurrent Neural Networks (RNN)
  - Long-short term memory (LSTM)
- Generative Adversarial Networks (GAN)