# Outline

- Basic concepts
- Decision tree induction
- Evaluation of classifiers
- Naïve Bayesian classification
- Naïve Bayes for text classification
- Support vector machines
- **Linear regression and gradient descent**
- Neural networks
- K-nearest neighbor
- Ensemble methods
- Summary

# Linear regression

- Supervised learning has two main types
  - Classification: discrete predictive/output variable
  - Regression: continuous predictive/output variable

- We first study linear regression, i.e., the predictive function *h* is a linear function.

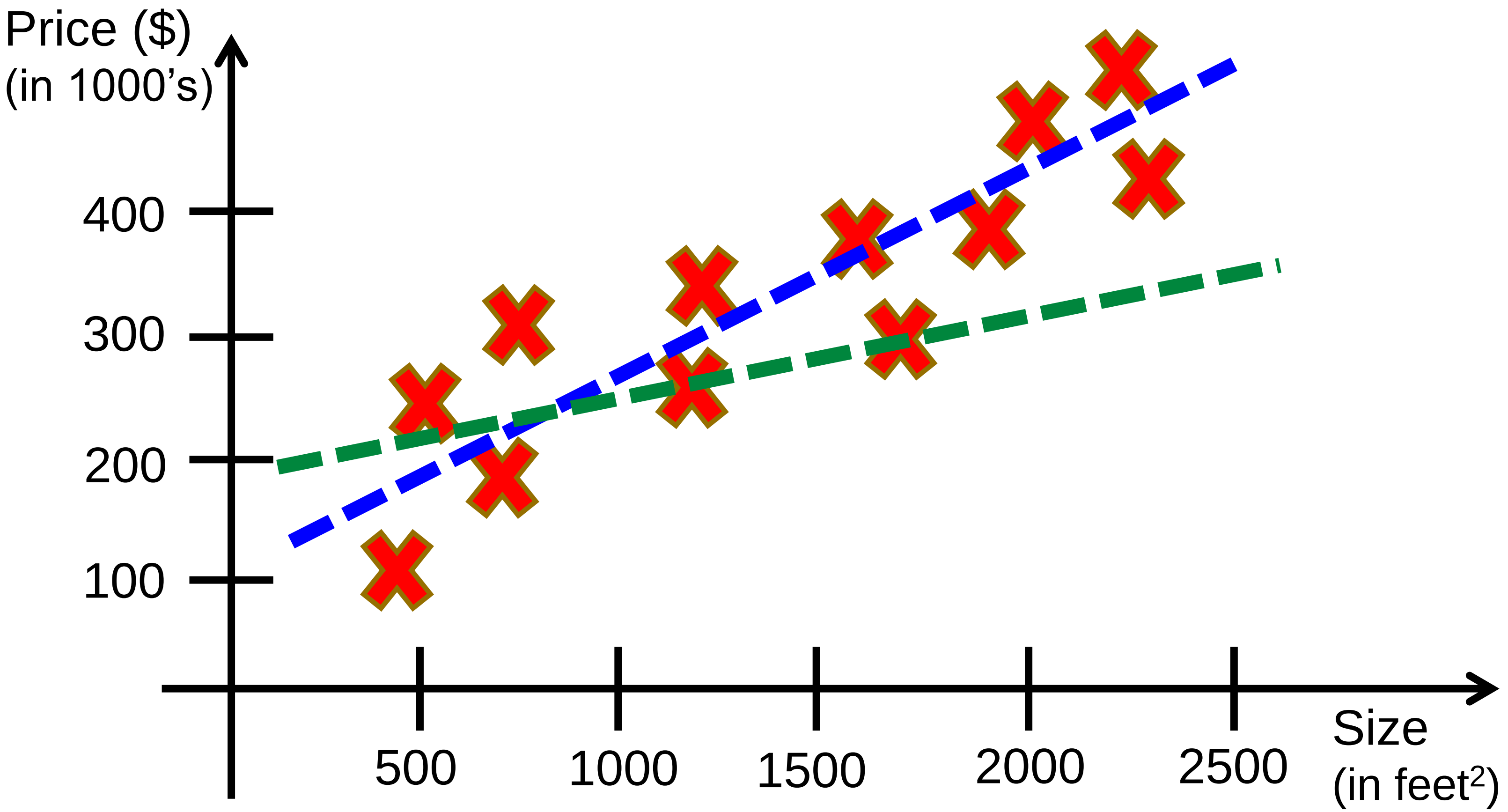
# An example: housing price prediction

- Given the size of a house, predict the price of the house.
- Notation:
  - $n$: Number of training examples
  - $x$: Input variable / feature (Size)
  - $y$: Output variable / target variable (Price)
  - $(x, y)$: One training example in general
  - $(x^i, y^i)$: $i^{th}$ training example

Training data

| Size in feet^2 (x) | Price ($) in 1000's (y) |
|---|---|
| 2104 | 460 |
| 1416 | 232 |
| 1534 | 315 |
| 852 | 178 |
| … | … |

# Training data and linear function



# Model representation

- This is a *univariate linear regression problem* as it has only one input variable $x$.
- The linear regression model in this case is as follows

$$y = h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x$$

  - There are two parameters $\theta_0$ and $\theta_1$.
  - $\boldsymbol{\theta}$ represents the parameter vector, i.e., $(\theta_0, \theta_1)$
- We use the training set to learn this model by optimizing a cost function, also called a *loss function (L)*.

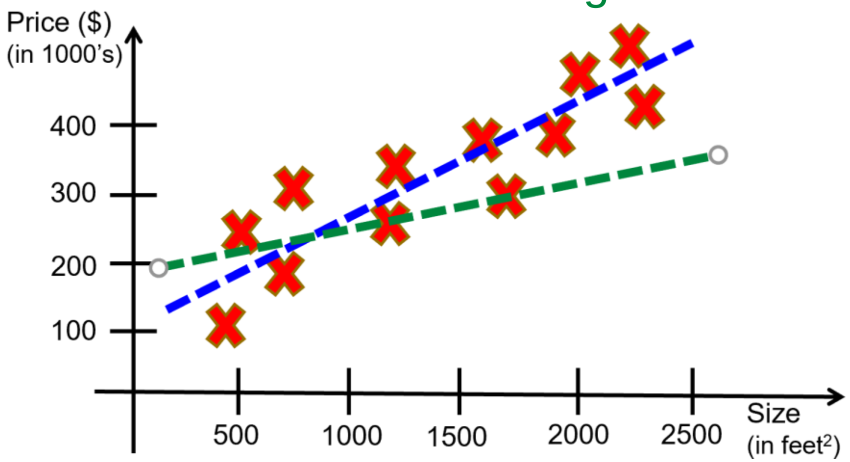
# Loss function

- Idea: select $\theta_0, \theta_1$ so that $h_{\boldsymbol{\theta}}(x)$ is close to $y$ for the training example $(x, y)$. This is expressed with a loss function.
- Loss function ($L$) used by linear regression:

$$L(\boldsymbol{\theta}) = L(\theta_0, \theta_1) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(x^i\right) - y^i\right)^2$$

where $h_{\boldsymbol{\theta}}\left(x^i\right) = \theta_0 + \theta_1 x^i$

Blue line is better than green line




- Learning goal: $\underset{\theta_0,\, \theta_1}{\text{argmin}}\; L(\theta_0, \theta_1)$

# Solve the minimization problem

- The learning is done using a general technique called
  - gradient descent

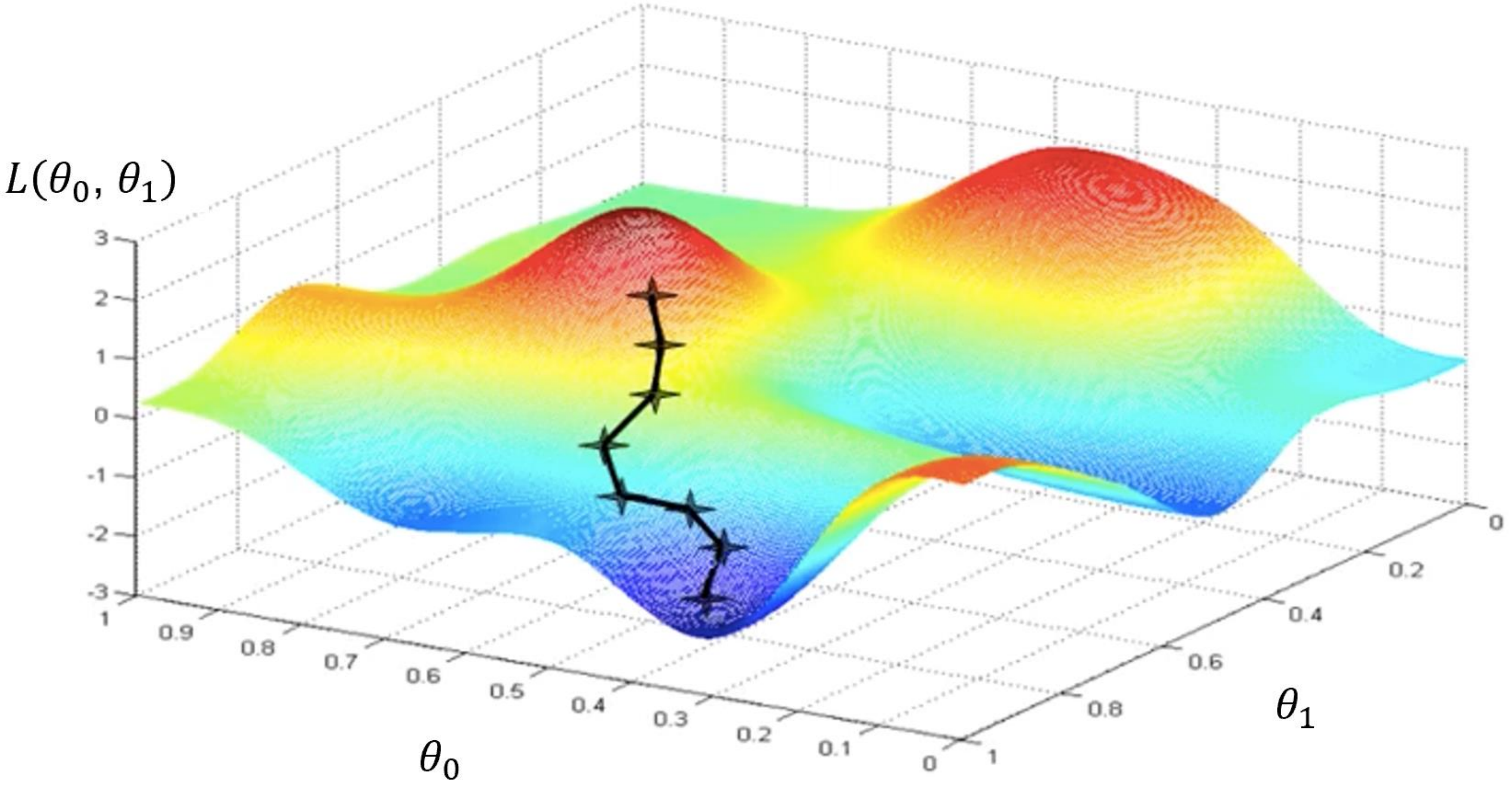
# Gradient descent

- Recall our univariate linear regression problem
  - Loss function: $L(\theta_0, \theta_1)$
  - Goal: $\underset{\theta_0,\, \theta_1}{\operatorname{argmin}} \; L(\theta_0, \theta_1)$

Steps:

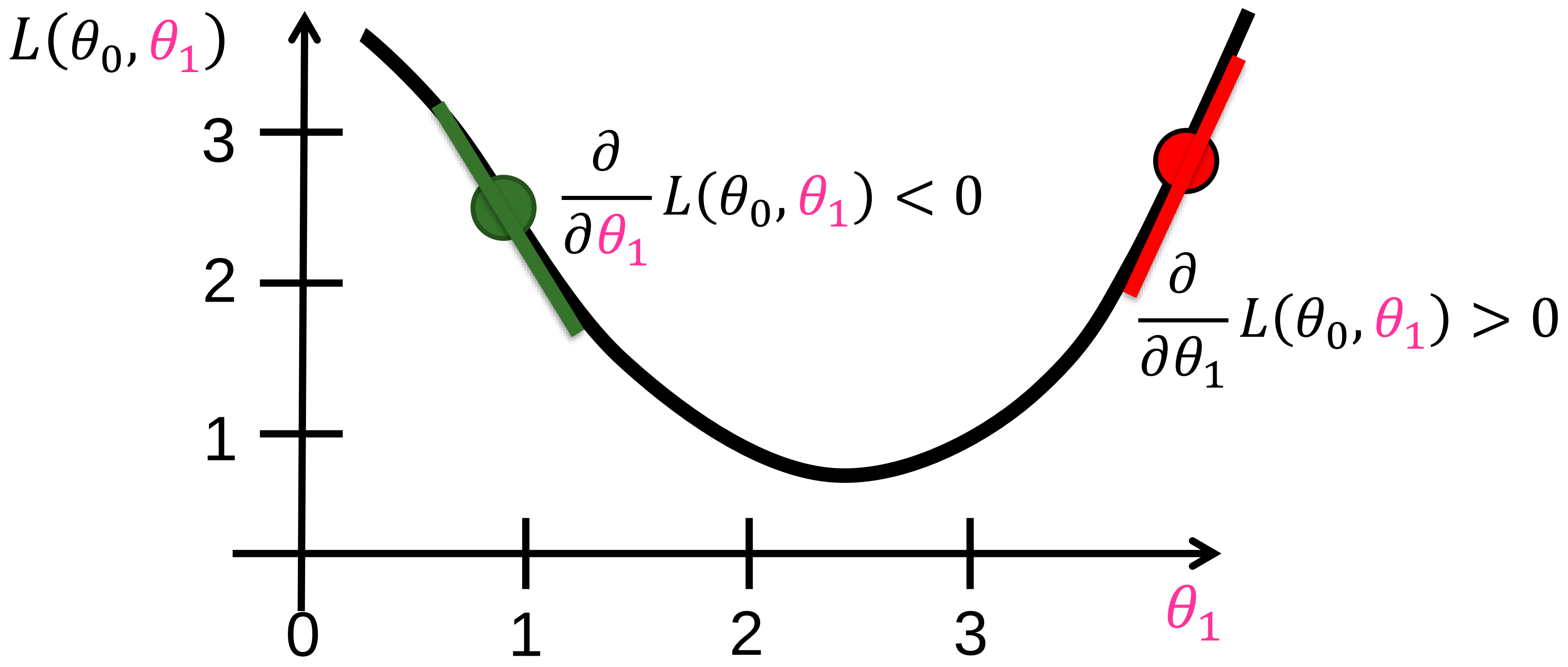
- Start with some initial $\theta_0, \theta_1$
- Keep changing $\theta_0, \theta_1$ to reduce $L(\theta_0, \theta_1)$ until we hopefully end up at minimum

# An illustration



# Keep going downhill

Learning rule: $\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} L(\theta_0, \theta_1)$



# Gradient descent algorithm

Repeat until convergence

{

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} L(\theta_0, \theta_1) \qquad \text{(for } j = 0 \text{ and } j = 1\text{)}$$

}

- $\alpha$: Learning rate (step size)
- $\frac{\partial}{\partial \theta_j} L(\theta_0, \theta_1)$: derivative (rate of change)

# How to update

**Correct: simultaneous update**

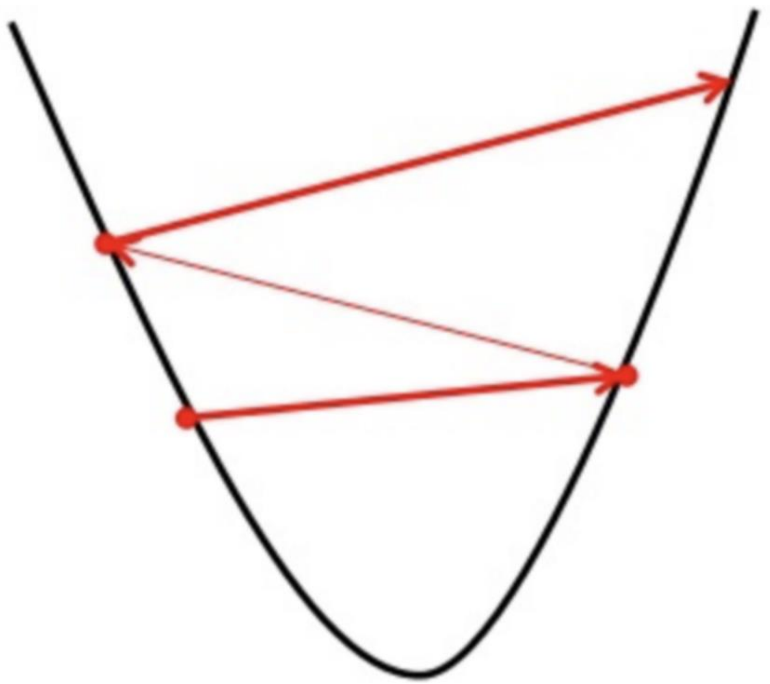
- $\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} L(\theta_0, \theta_1)$
- $\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} L(\theta_0, \theta_1)$
- $\theta_0 := \text{temp0}$
- $\theta_1 := \text{temp1}$

**Incorrect:**

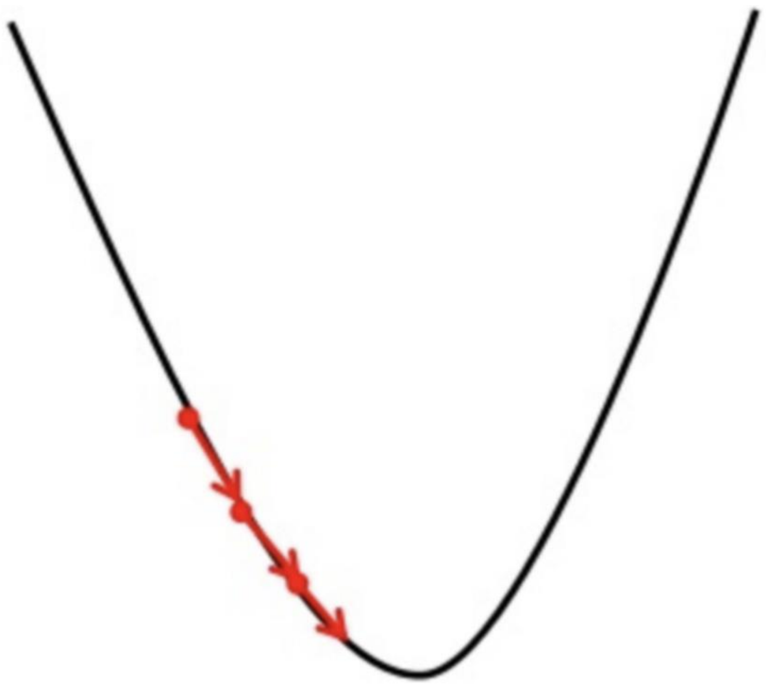
- $\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} L(\theta_0, \theta_1)$
- $\theta_0 := \text{temp0}$
- $\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} L(\theta_0, \theta_1)$
- $\theta_1 := \text{temp1}$

# Learning rate

Too big learning rate



Small learning rate



# Recall: Loss function and learning goal

- Recall: Loss function (*L*) used by linear regression is:

$$L(\boldsymbol{\theta}) = L(\theta_0, \theta_1) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(x^i) - y^i\right)^2$$

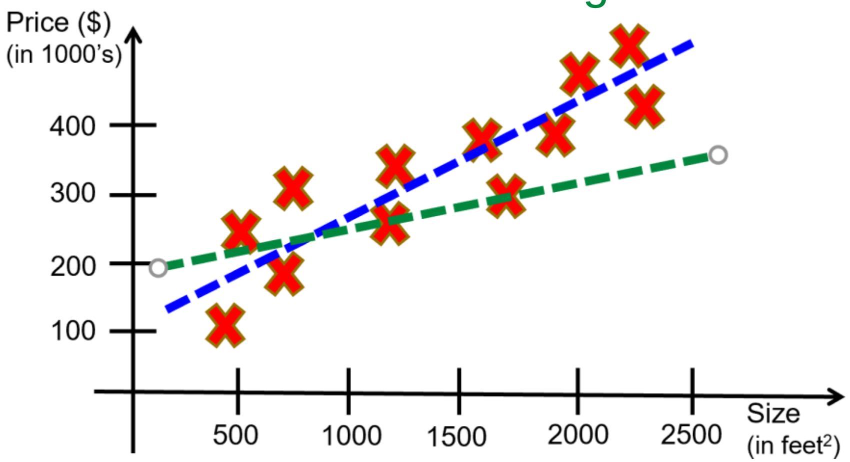
where $h_{\boldsymbol{\theta}}(x^i) = \theta_0 + \theta_1 x^i$

$h_{\boldsymbol{\theta}}(x^i)$ is an estimate of $y^i$

- Learning goal:

$$\underset{\theta_0, \theta_1}{\operatorname{argmin}}\ L(\theta_0, \theta_1)$$

Blue line is better than green line



# Computing partial derivative

- $\frac{\partial}{\partial \theta_j} L(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_j} \frac{1}{2n} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}}\left(x^i\right) - y^i\right)^2$

$$= \frac{\partial}{\partial \theta_j} \frac{1}{2n} \sum_{i=1}^{n} \left(\theta_0 + \theta_1 x^i - y^i\right)^2$$

- $j = 0$: $\frac{\partial}{\partial \theta_0} L(\theta_0, \theta_1) = \frac{1}{n} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}}\left(x^i\right) - y^i\right)$
- $j = 1$: $\frac{\partial}{\partial \theta_1} L(\theta_0, \theta_1) = \frac{1}{n} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}}\left(x^i\right) - y^i\right) x^i$

# Gradient descent for linear regression

Repeat until convergence

{

$$\theta_0 := \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} \left( h_{\boldsymbol{\theta}}\left(x^i\right) - y^i \right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{n} \sum_{i=1}^{n} \left( h_{\boldsymbol{\theta}}\left(x^i\right) - y^i \right) x_i$$

}

- Update $\theta_0$ and $\theta_1$ simultaneously

# Batch gradient descent

- Each step or update of gradient descent uses all (*n*) the training examples.
  - Sum over all *n* training examples for each step – slow
  - It is also memory demanding if the training data is huge.
- In a normal learning process, training needs many steps before convergence.
- The training process that covers all the training examples once is called an epoch.
  - In batch gradient descent, each step is an epoch.

# Stochastic gradient descent (SGD)

- SGD with one example per step: In SGD each step uses a single training example. Before each epoch, the data should be shuffled.
  - SGD converges faster when the dataset is large as it causes updates to the parameters more frequently.
    - The loss may fluctuate as only one example is used in each step.
- SGD with minibatch: each update/step uses a random ***minibatch*** of ***m*** out of ***n*** examples.
  - It is efficient, more stable, and more likely to jump out of a local minimum
- Batch Gradient Descent is more suitable for convex loss functions as it can converge directly to minima.

# Convex and non-convex function

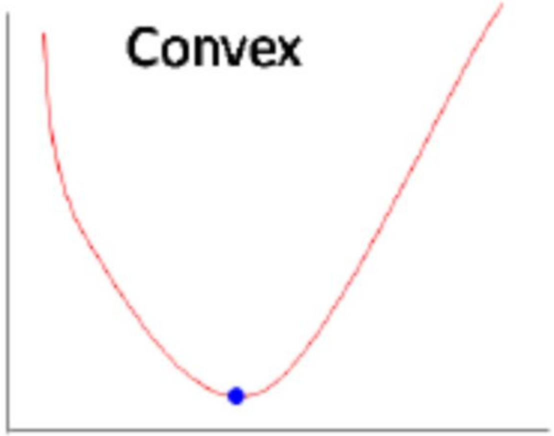
**Convex set *X***: for all *a* and *b* in *X*, the line segment connecting *a* and *b* is included in *X*.

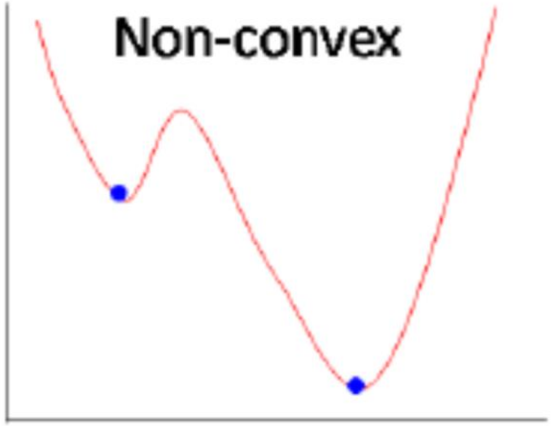
**Convex function:** a real-valued function is called **convex** if the line segment between any two points on the graph of the function does not lie below the graph between the two points.

- A convex function has one minimum.
  - For all $0 \leq \lambda \leq 1$ and all $x_1, x_2$ in a convex set *X* (e.g., an interval [a, b]), the following holds

$$f(\lambda x_1 + (1-\lambda)\, x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$$

- A non-convex function has local minima (valleys) that are not global minimum.



# Multivariate linear regression

- In our previous linear regression problem, we use only one input variable/feature (univariate). In general, the problem can have any number of input variables. Let the number of variables be *k*, $x_1, x_2, \ldots, x_k$.
- Training data: $D = \{\mathbf{x}^i, y^i\}_{i=1}^n$
- Multivariate linear regression model is

$$y = h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_k x_k$$

where $\boldsymbol{\theta}$ is the vector of all $\theta_i$ and $\mathbf{x}$ is the vector of all $x_i$.

# Multivariate linear regression (cont.)

- For convenience of notation, define $x_0 = 1$ ($x_0^j = 1$ for all examples *j*)
- $\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} \in R^{k+1}$ $\quad\quad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_k \end{bmatrix} \in R^{k+1}$
- $y = h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_k x_k = \boldsymbol{\theta}^{\top}\mathbf{x}$

# Univariate and multivariate gradient descent

- Univariate ($k = 1$)

Repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} \left( h_{\boldsymbol{\theta}}\left(x^i\right) - y^i \right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{n} \sum_{i=1}^{n} \left( h_{\boldsymbol{\theta}}\left(x^i\right) - y^i \right) x^i$$

}

- Multivariate ($k > 1$)

Repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{1}{n} \sum_{i=1}^{n} \left( h_{\boldsymbol{\theta}}\left(\mathbf{x}^i\right) - y^i \right) x_j^i$$

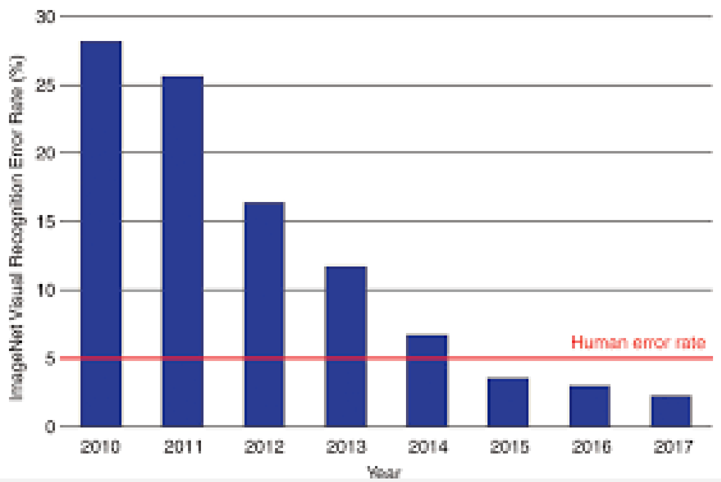
}

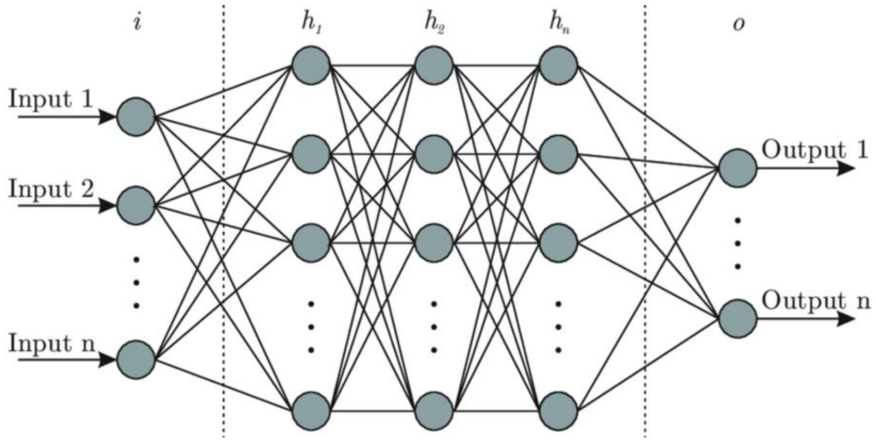
Simultaneously update $\theta_j$, for $j = 0, 1, \cdots, k$

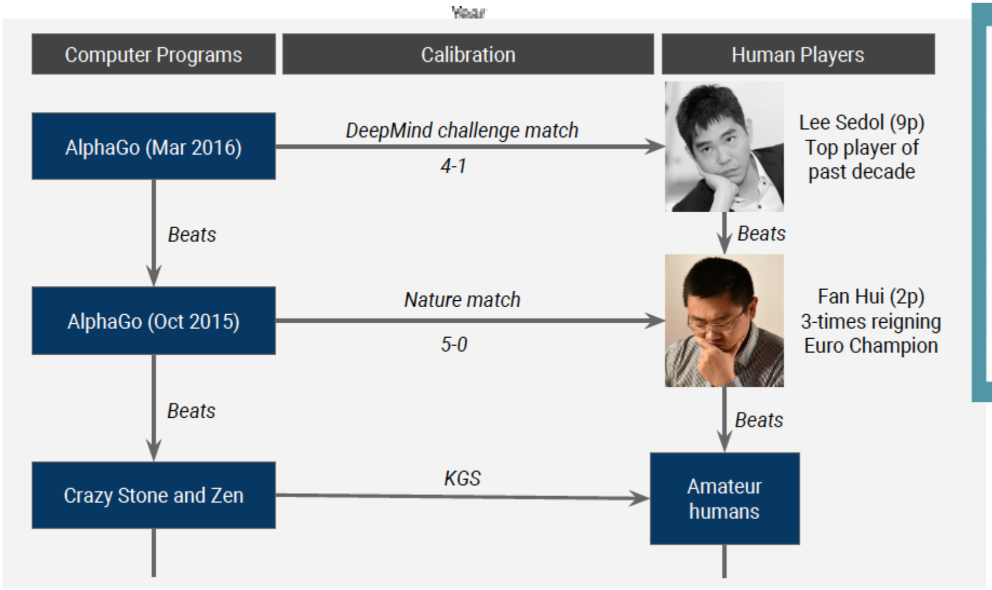
# Outline

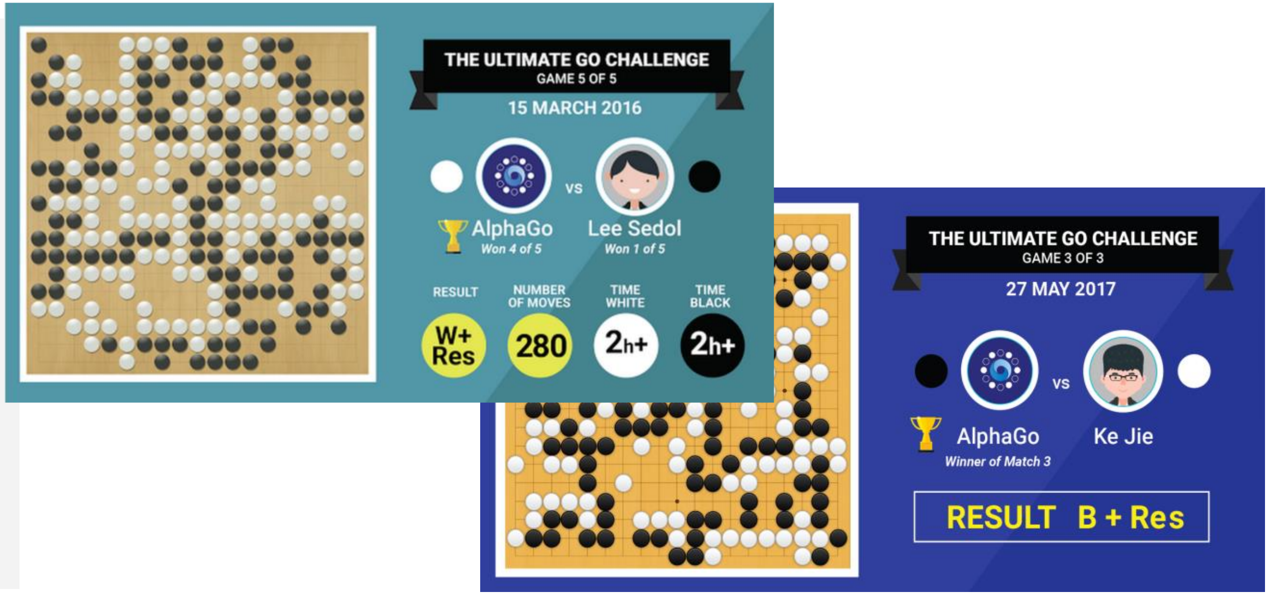
- Basic concepts
- Decision tree induction
- Evaluation of classifiers
- Naïve Bayesian classification
- Naïve Bayes for text classification
- Support vector machines
- Linear regression and gradient descent
- **Neural networks**
- K-nearest neighbor
- Ensemble methods
- Summary

# Some example successes of neural networks



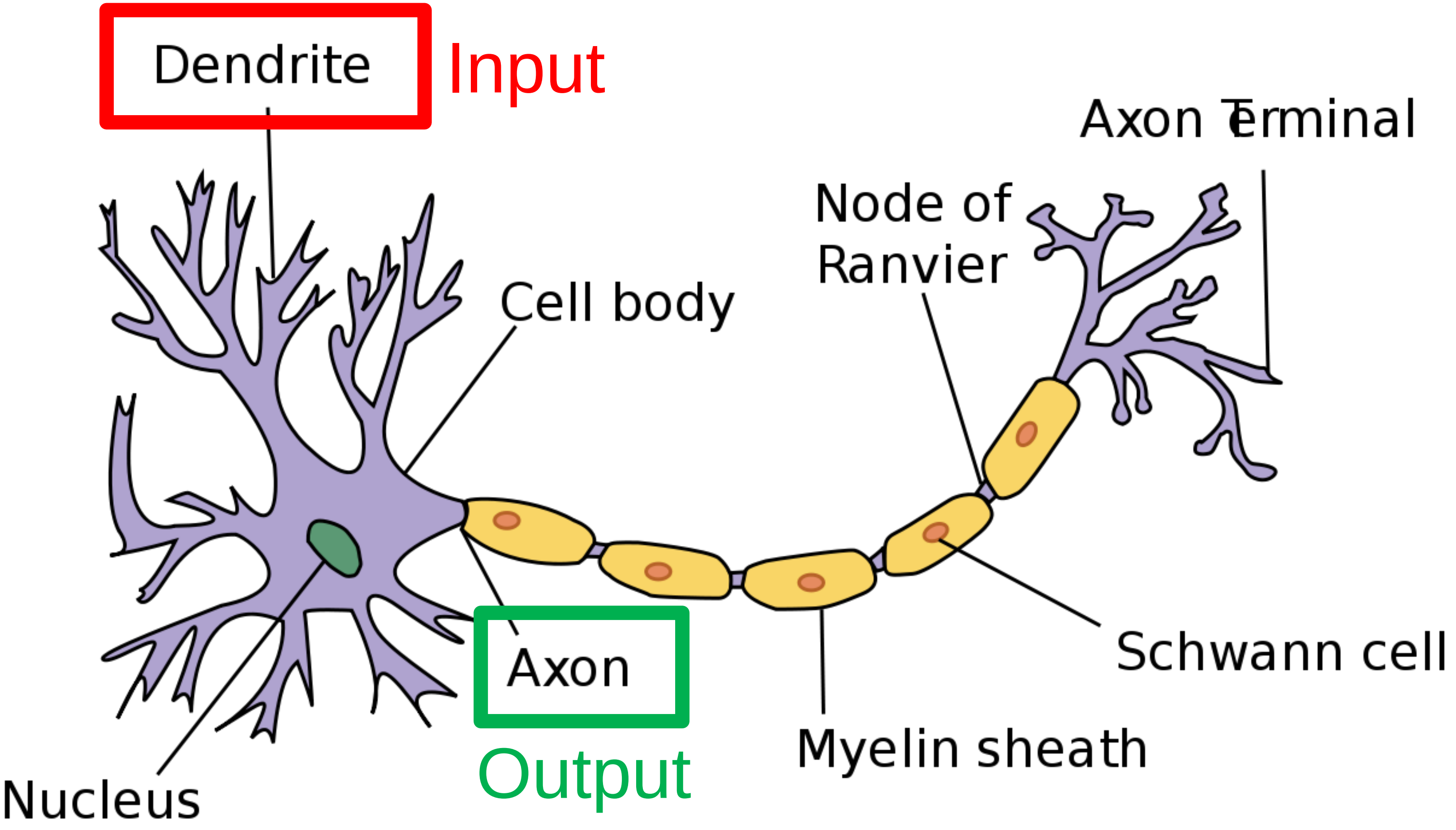



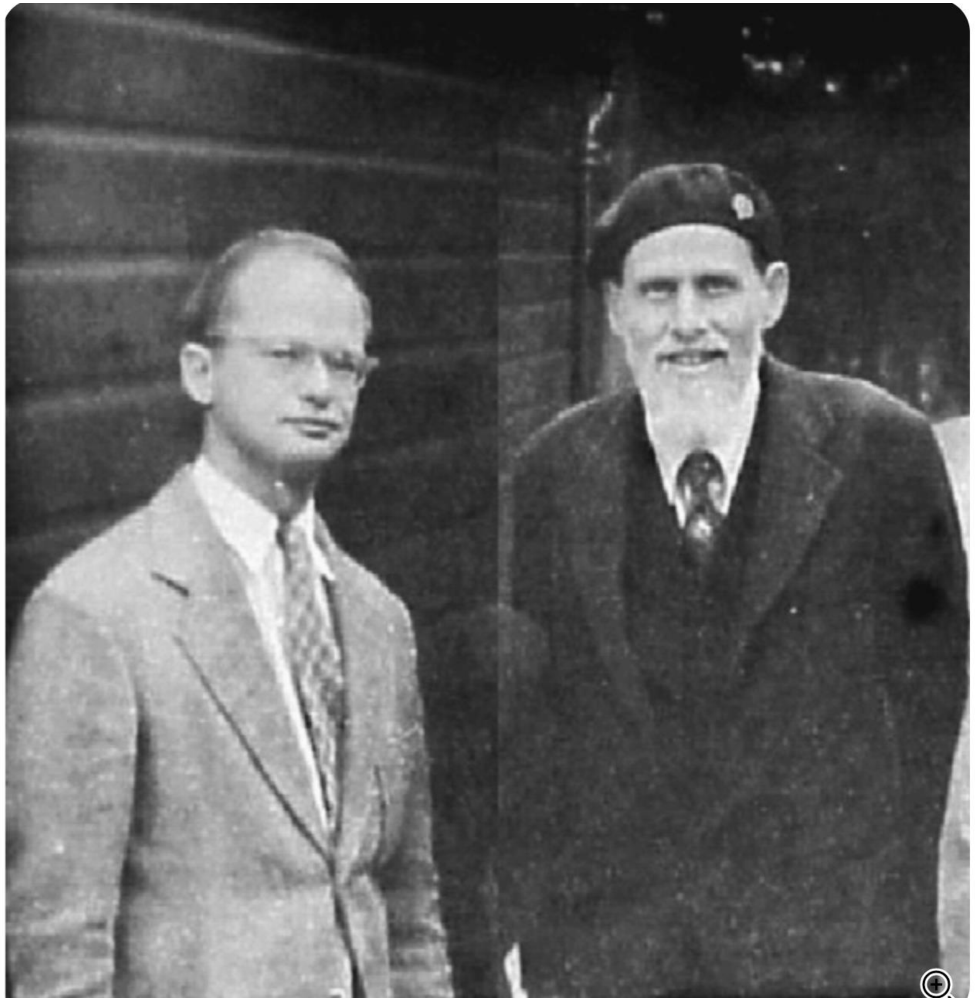
# Resurgence of neural networks

- Origin: Algorithms that try to mimic the brain (1943).
- Was very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art results in many applications.
- It works especially well for computer vision and natural language processing (including speech recognition).
  - It has revolutionized the two fields in recent years.
  - It has spread to almost every machine learning area and application in practice.

# A single neuron in the brain



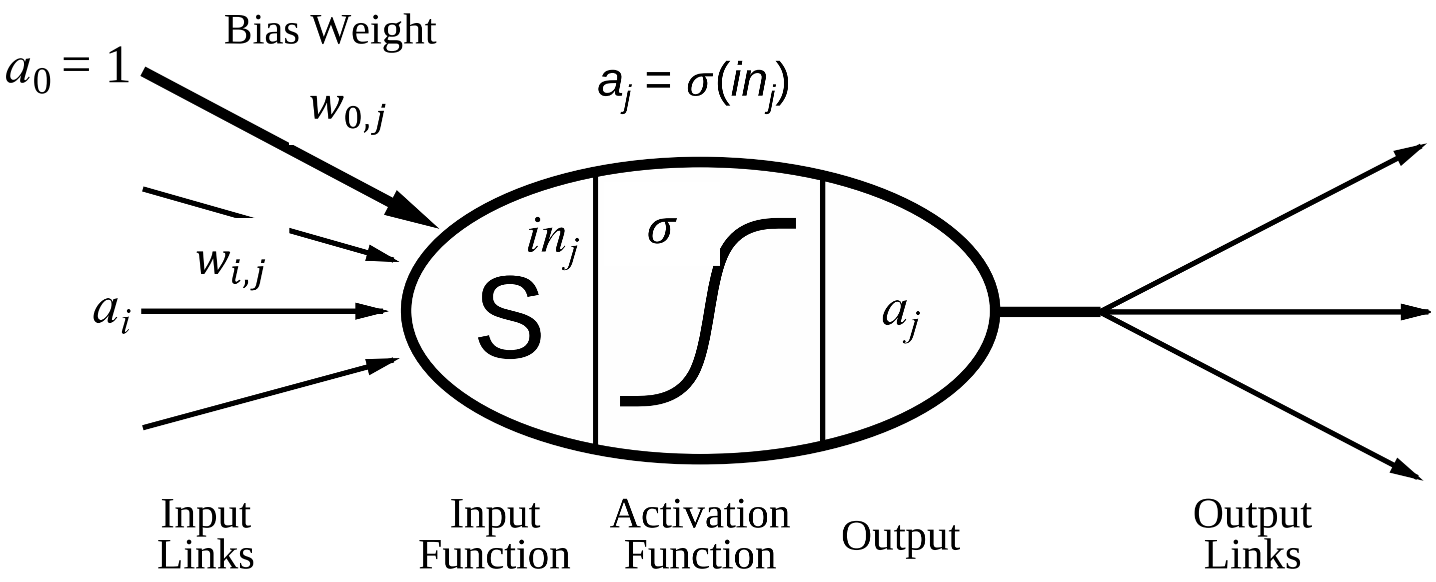
# The first neural network (McCulloch & Pitts, 1943)



In 1943 American neurophysiologist and cybernetician of the University of Illinois at Chicago Warren McCulloch and self-taught logician and cognitive psychologist Walter Pitts published "A Logical Calculus of the ideas Imminent in Nervous Activity," describing the "McCulloch - Pitts neuron", "the first mathematical model of a neural network.

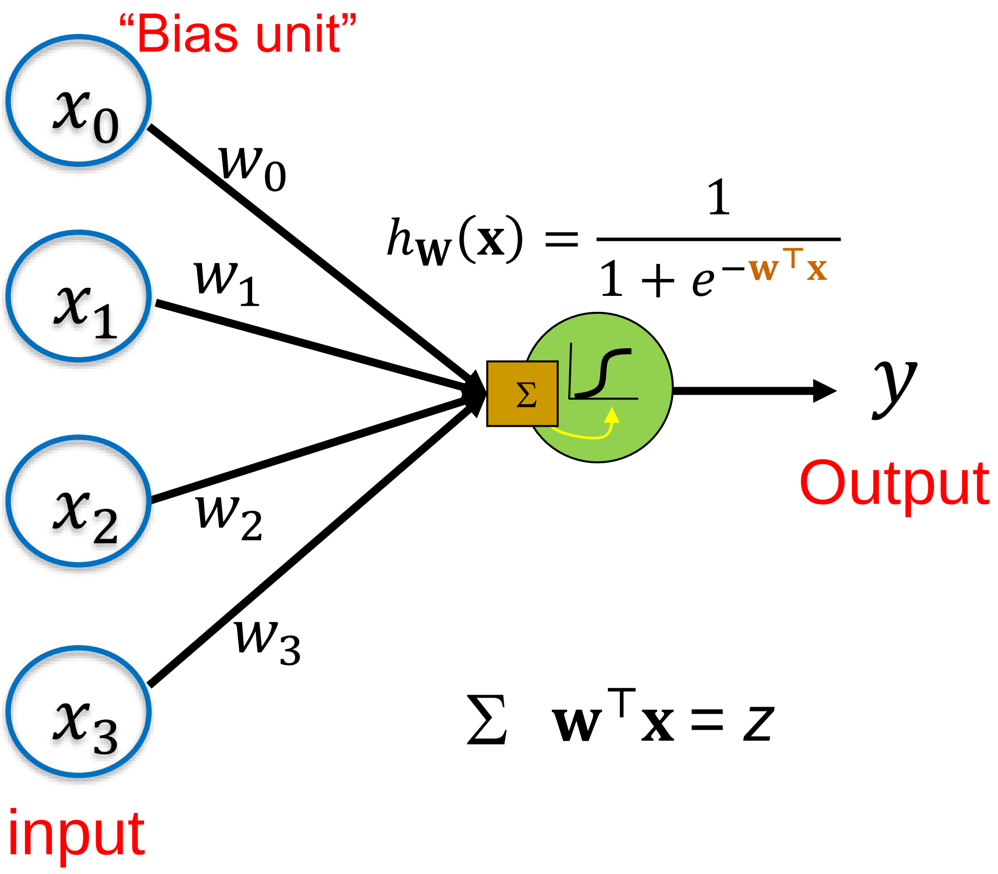
Building on ideas in Alan Turing's "On Computable Numbers", McCulloch and Pitts's paper provided a way to describe brain functions in abstract terms, and showed that simple elements connected in a neural network can have immense computational power. The paper

# Simple model of a neuron (McCulloch & Pitts, 1943)




- Inputs $a_i$ come from the output of node $i$ to this node $j$ (or from "outside")
- Each input link has a ***weight*** $w_{i,j}$
- There is an additional fixed input $a_0$ (bias) with weight $w_{0,j}$
- The total input is $in_j = \Sigma_i \, w_{i,j} \, a_i$
- The output is $a_j = \sigma(in_j) = \sigma(\Sigma_i \, w_{i,j} \, a_i) = \sigma(\mathbf{w.a})$

# Logistic regression in a figure




$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}$$

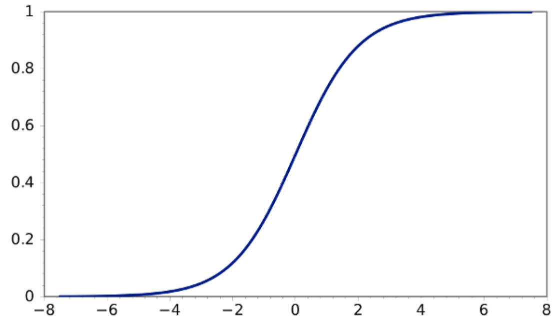
$$\Sigma \quad \mathbf{w}^\top \mathbf{x} = z$$

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

"Weights"
"Parameters"

$$y = h_{\mathbf{w}}(\mathbf{x}) = \sigma\left(\mathbf{w}^\top \mathbf{x}\right),$$

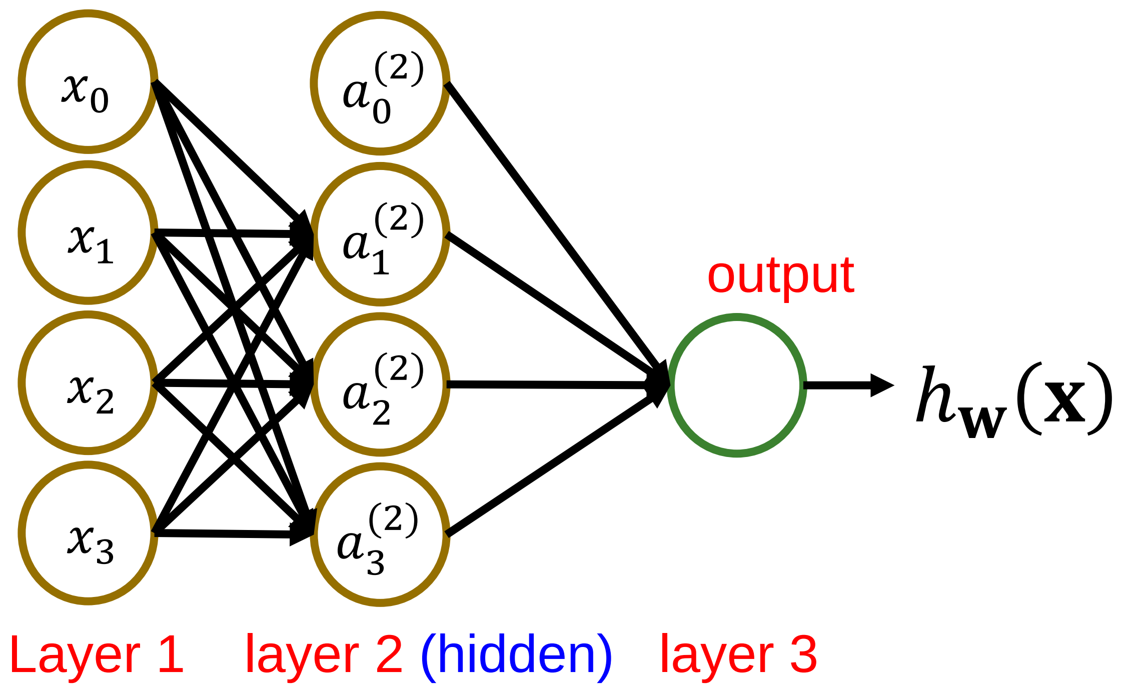
where $\sigma(z) = \frac{1}{1+e^{-z}}$



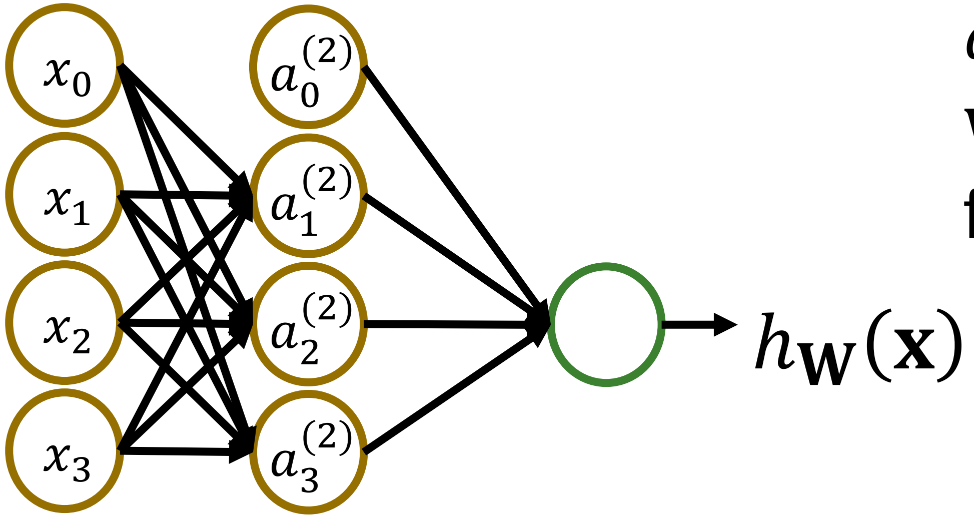

Sigmoid (logistic) function

# An artificial neuron: a logistic unit

- A neuron is a logistic unit
  - $\sigma(\mathbf{w}^\top \mathbf{x})$ is called activation function.
  - Activation function does not have to be sigmoid.
- A neural network is a composition of many logistic units organized in layers.
  - It can also be seen as a logistic regression model with one or more hidden layers.



# Neural network: an example



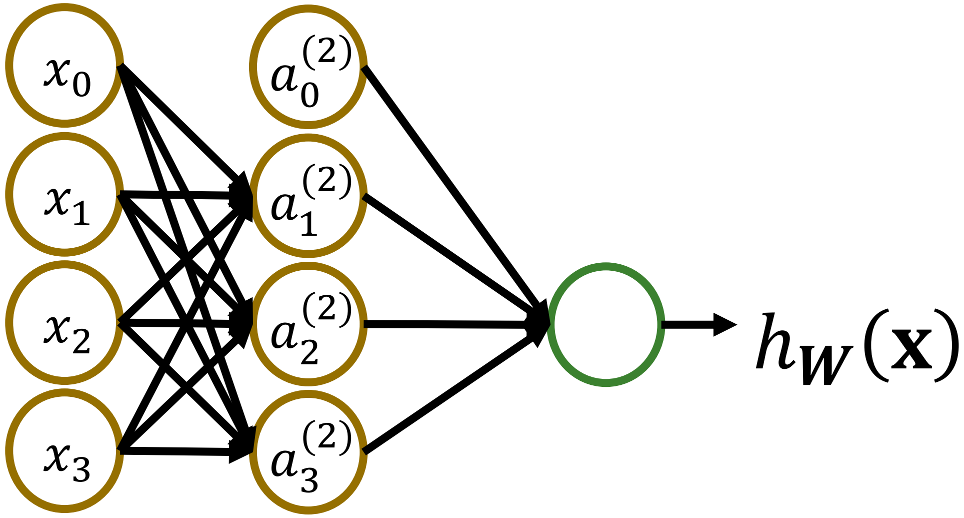

$a_i^{(j)}$ = "activation" of unit $i$ in layer $j$
$\mathbf{W}^{(j)}$ = matrix of weights controlling function mapping from layer $j$ to layer $j+1$

$s_j$ units in layer $j$

$s_{j+1}$ units in layer $j+1$

$$a_1^{(2)} = \sigma\left(\mathbf{W}_{10}^{(1)}x_0 + \mathbf{W}_{11}^{(1)}x_1 + \mathbf{W}_{12}^{(1)}x_2 + \mathbf{W}_{13}^{(1)}x_3\right)$$
$$a_2^{(2)} = \sigma\left(\mathbf{W}_{20}^{(1)}x_0 + \mathbf{W}_{21}^{(1)}x_1 + \mathbf{W}_{22}^{(1)}x_2 + \mathbf{W}_{23}^{(1)}x_3\right)$$
$$a_3^{(2)} = \sigma\left(\mathbf{W}_{30}^{(1)}x_0 + \mathbf{W}_{31}^{(1)}x_1 + \mathbf{W}_{32}^{(1)}x_2 + \mathbf{W}_{33}^{(1)}x_3\right)$$
$$h_{\mathbf{W}}(x) = \sigma\left(\mathbf{W}_{10}^{(2)}a_0^{(2)} + \mathbf{W}_{11}^{(2)}a_1^{(2)} + \mathbf{W}_{12}^{(2)}a_2^{(2)} + \mathbf{W}_{13}^{(2)}a_3^{(2)}\right)$$

# Neural network: an example



"Pre-activation"

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \mathrm{z}^{(2)} = \begin{bmatrix} \mathrm{z}_1^{(2)} \\ \mathrm{z}_2^{(2)} \\ \mathrm{z}_3^{(2)} \end{bmatrix}$$

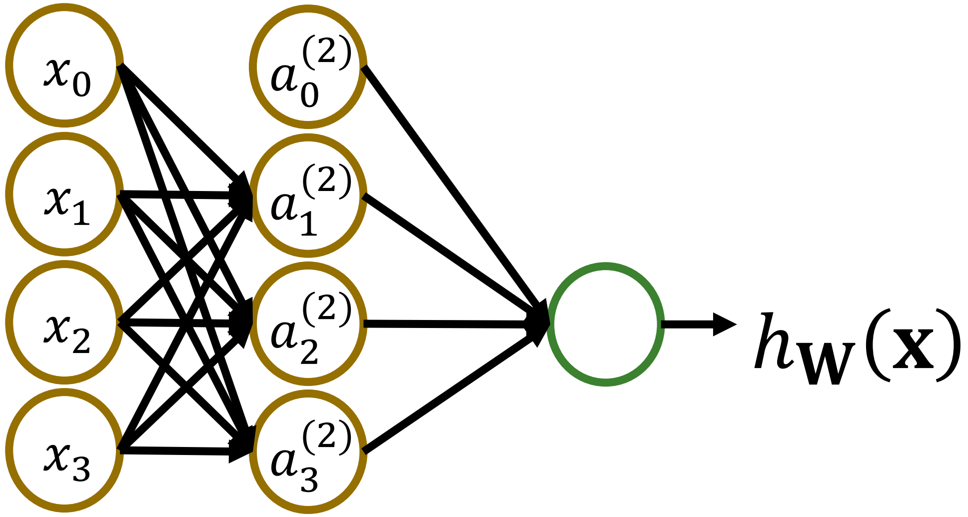
$$a_1^{(2)} = \sigma\left(\mathbf{W}_{10}^{(1)} x_0 + \mathbf{W}_{11}^{(1)} x_1 + \mathbf{W}_{12}^{(1)} x_2 + \mathbf{W}_{13}^{(1)} x_3\right) = \sigma(\mathrm{z}_1^{(2)})$$

$$a_2^{(2)} = \sigma\left(\mathbf{W}_{20}^{(1)} x_0 + \mathbf{W}_{21}^{(1)} x_1 + \mathbf{W}_{22}^{(1)} x_2 + \mathbf{W}_{23}^{(1)} x_3\right) = \sigma(\mathrm{z}_2^{(2)})$$

$$a_3^{(2)} = \sigma\left(\mathbf{W}_{30}^{(1)} x_0 + \mathbf{W}_{31}^{(1)} x_1 + \mathbf{W}_{32}^{(1)} x_2 + \mathbf{W}_{33}^{(1)} x_3\right) = \sigma(\mathrm{z}_3^{(2)})$$

$$h_{\mathbf{W}}(x) = \sigma\left(\mathbf{W}_{10}^{(2)} a_0^{(2)} + \mathbf{W}_{11}^{(2)} a_1^{(2)} + \mathbf{W}_{12}^{(2)} a_2^{(2)} + \mathbf{W}_{13}^{(2)} a_3^{(2)}\right) = \sigma(z^{(3)})$$

# Neural network: an example




$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

"Pre-activation"

$$\mathbf{z}^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$$a_1^{(2)} = \sigma(z_1^{(2)})$$
$$a_2^{(2)} = \sigma(z_2^{(2)})$$
$$a_3^{(2)} = \sigma(z_3^{(2)})$$
$$h_{\mathbf{W}}(\mathbf{x}) = \sigma(z^{(3)})$$

$\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{x} = \mathbf{W}^{(1)}\mathbf{a}^{(1)}$ // layer 1: $\mathbf{a}^{(1)} = \mathbf{x}$

$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$

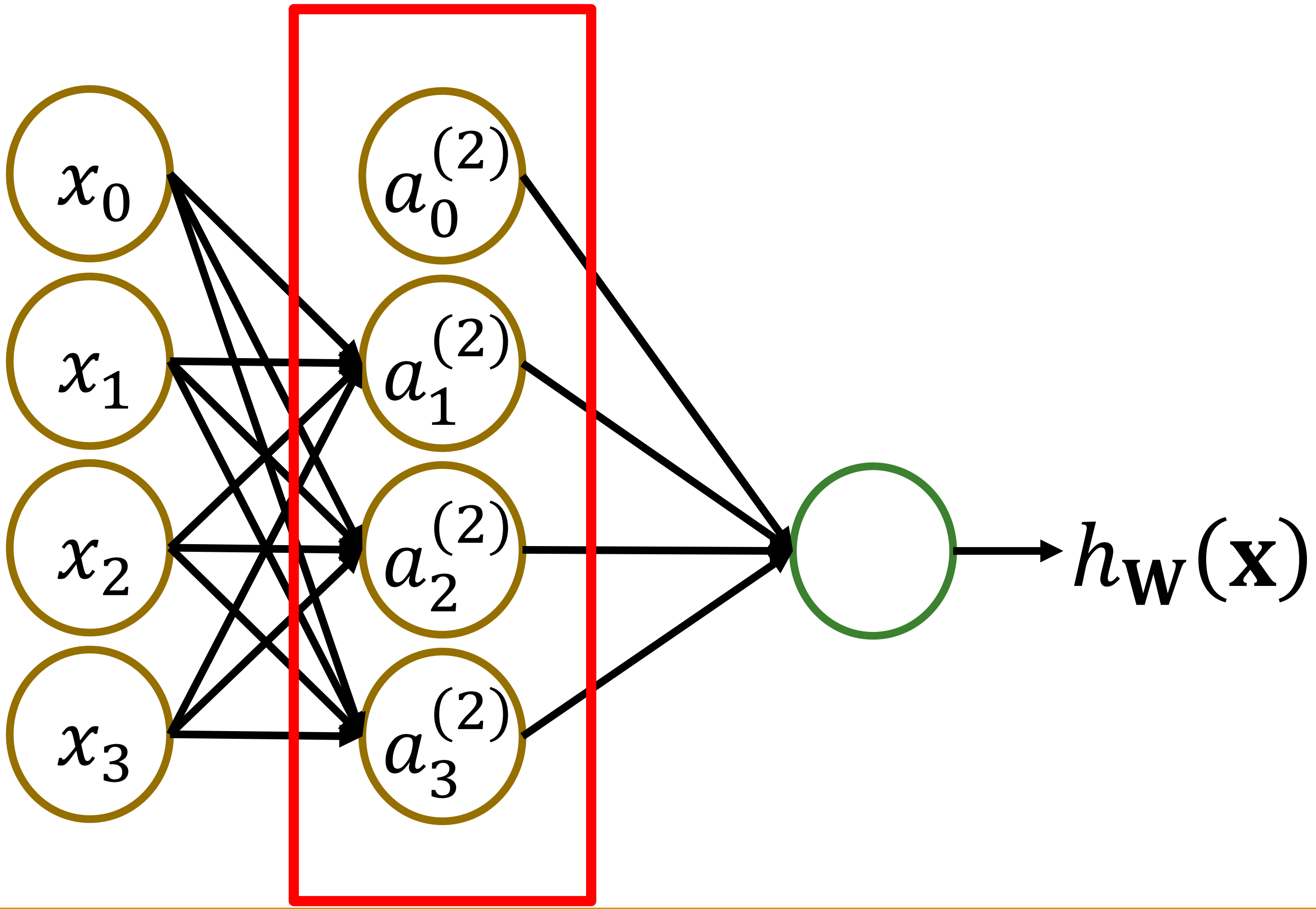
Add $a_0^{(2)} = 1$

$\mathbf{z}^{(3)} = \mathbf{W}^{(2)}\mathbf{a}^{(2)}$

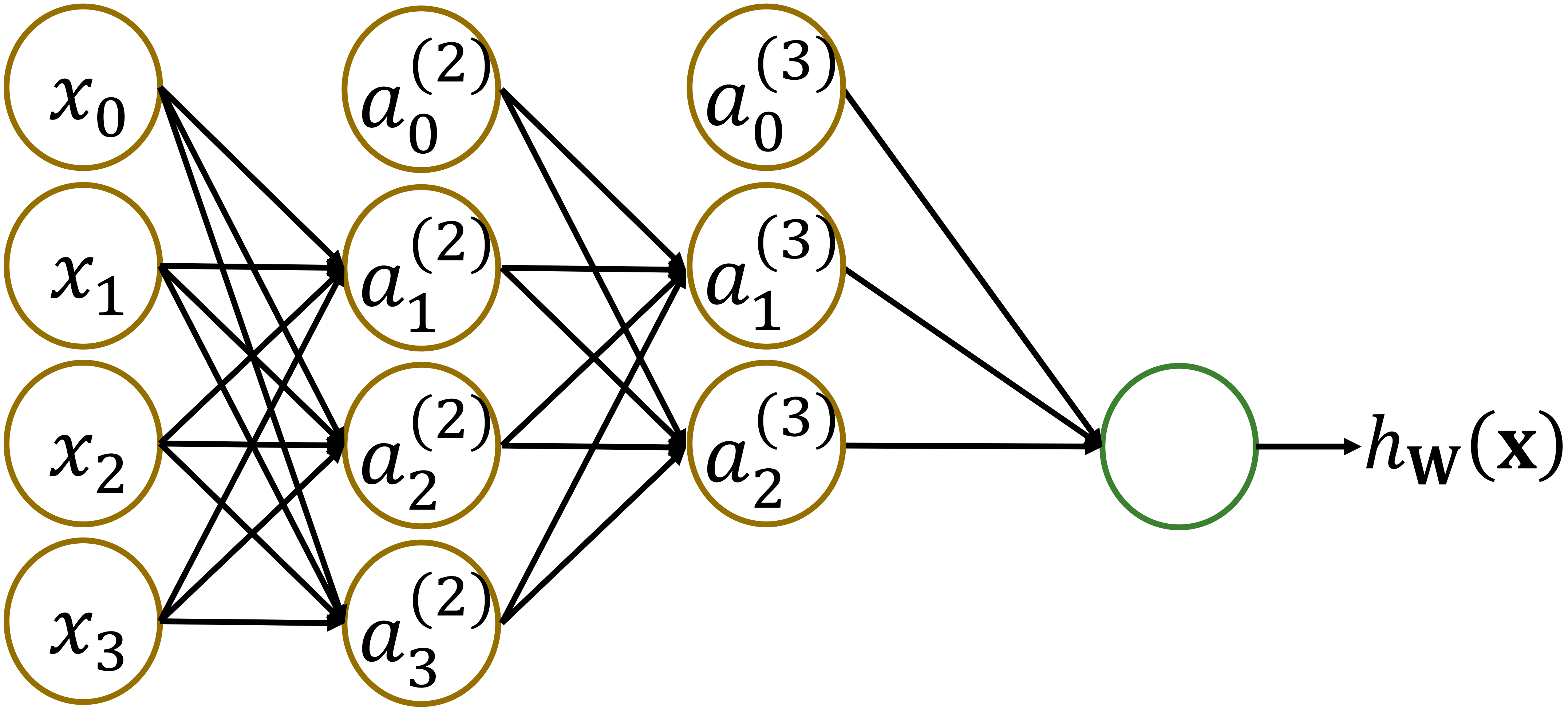
$h_{\mathbf{W}}(\mathbf{x}) = \mathbf{a}^{(3)} = \sigma(\mathbf{z}^{(3)})$

# Neural network learning its own features

- Other machine learning models directly use the input features to build models.
- But a neural network can learn higher level features that consider the interactions of the input features.

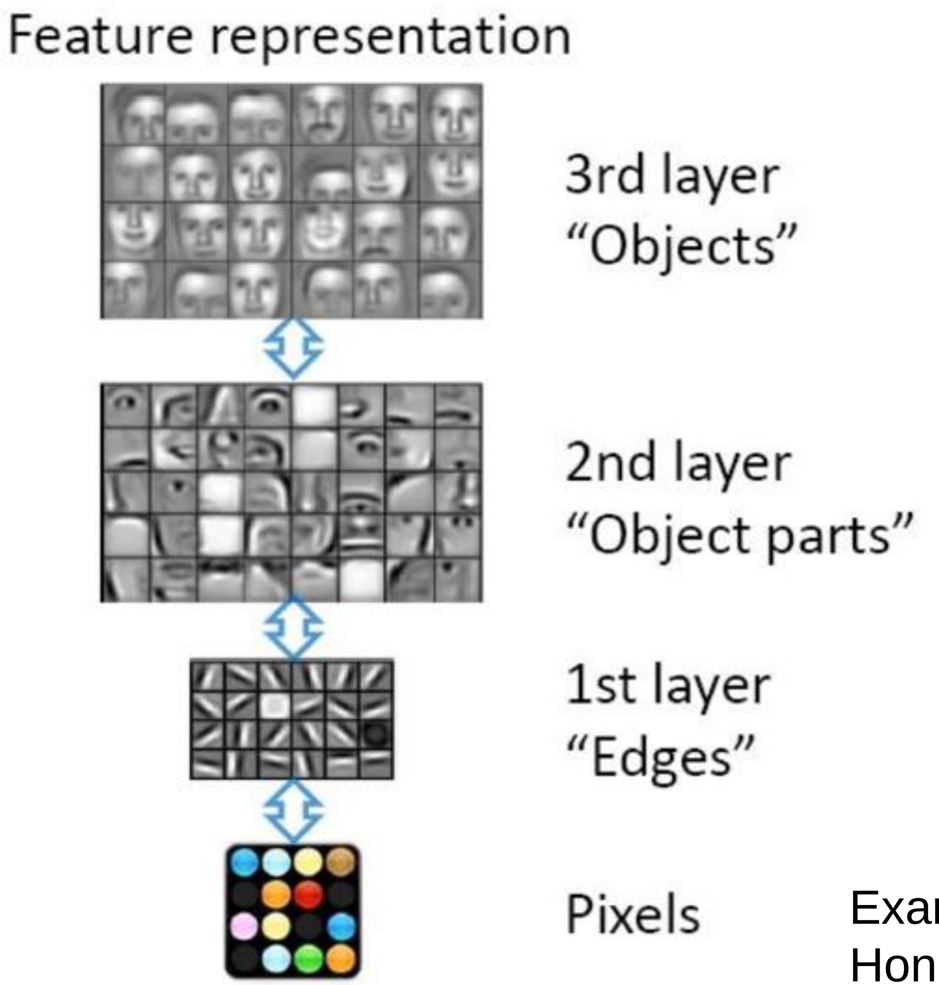


# More layers



# More layers give different levels of abstraction

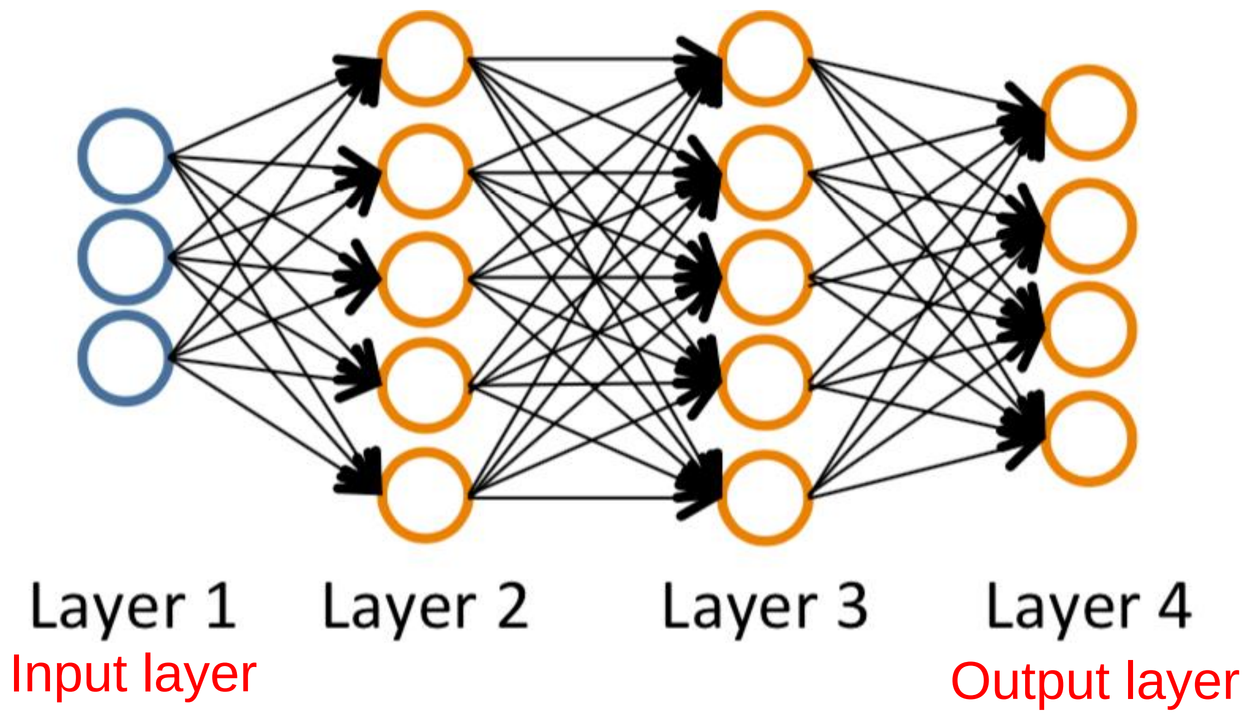
- We don't know the "right" levels of abstraction
- So let the model figure it out!
- **Face Recognition:**
  - Deep network can build up increasingly higher levels of abstraction
  - Lines, parts, regions




Example from Honglak Lee (NIPS 2010)

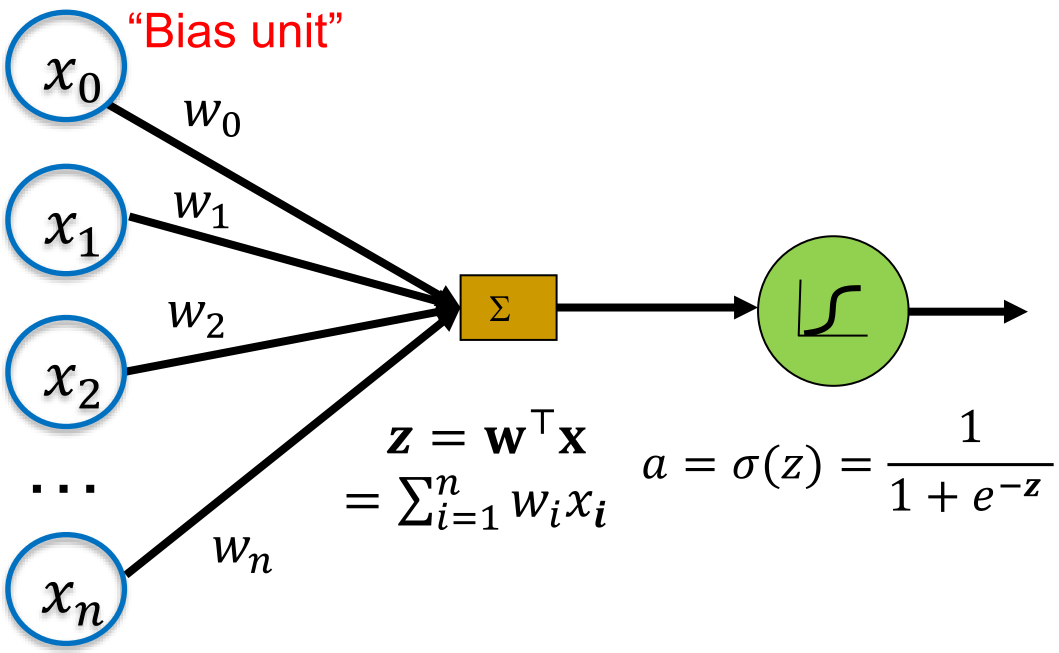
# Multiple classes

- With multiple classes in a classification problem, we will need multiple output units, one output unit per class.



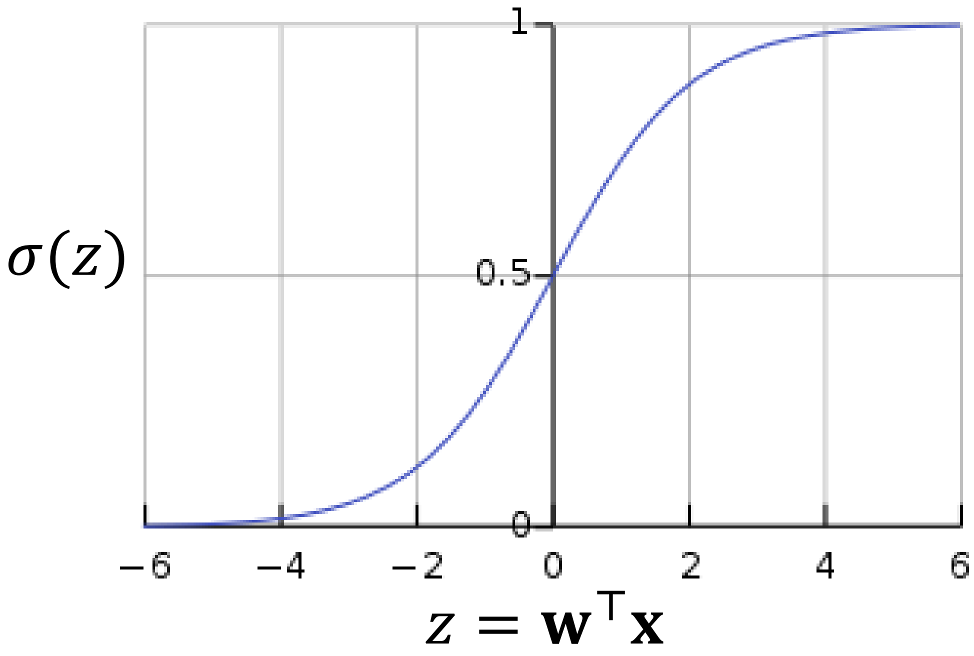
# Activation function

- So far, we've assumed that the activation function is always the sigmoid/logistic function. In fact, it is not widely used any more.



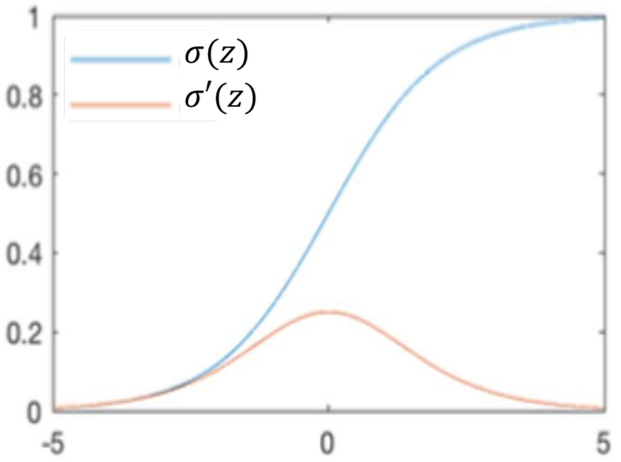

$$\boldsymbol{z} = \mathbf{w}^{\top}\mathbf{x} = \sum_{i=1}^{n} w_i x_{\boldsymbol{i}}$$

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

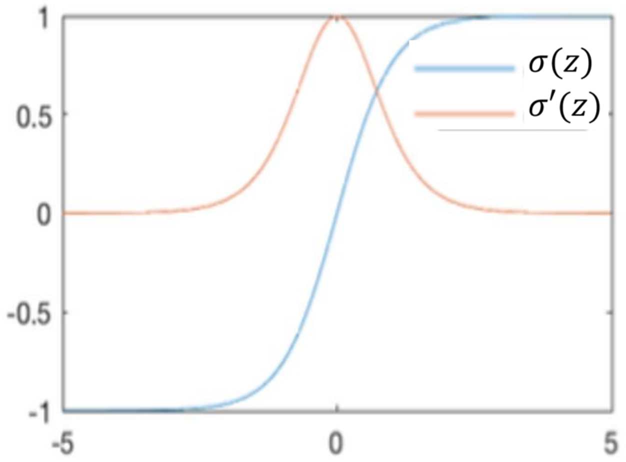


# Two more activation functions, Tanh and ReLu

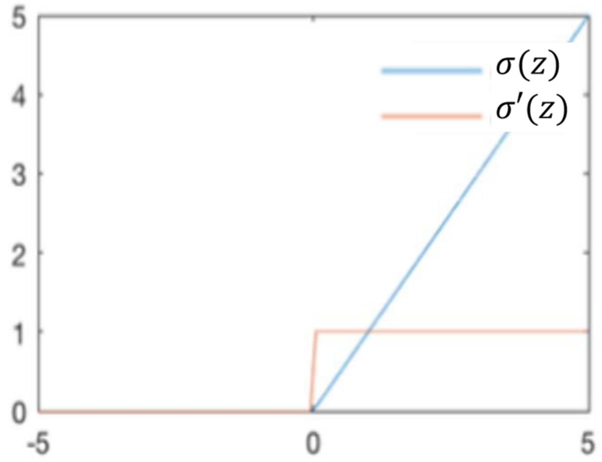
Sigmoid Function




Hyperbolic Tangent




Rectified Linear Unit (ReLU)




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

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

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

$$\sigma'(z) = 1-\sigma(z)^2$$

$$\sigma(z) = \max(0, z)$$

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

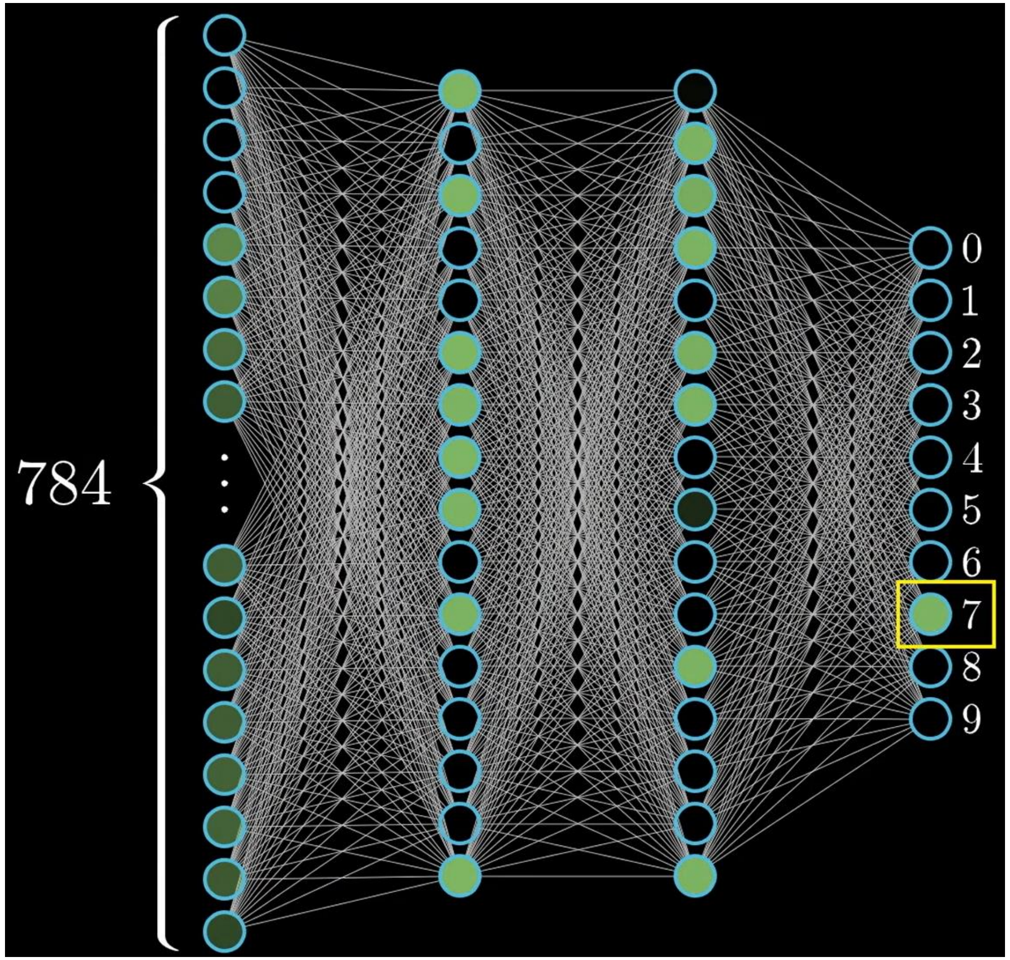
# An example: recognizing hand-written digits

- Each hand-written digit is a 28x28 = 784 image
- We want to build a neural network to recognize 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9


Subsequent slides are based on 4 videos starting from the following:
https://www.youtube.com/watch?v=aircAruvnKk

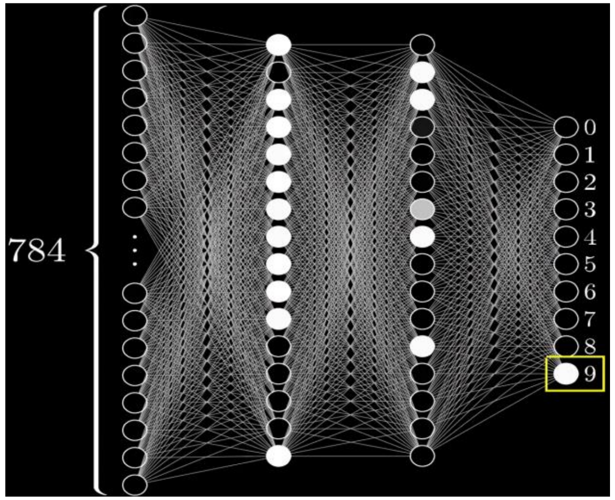
# A network for recognizing of hand-written digits

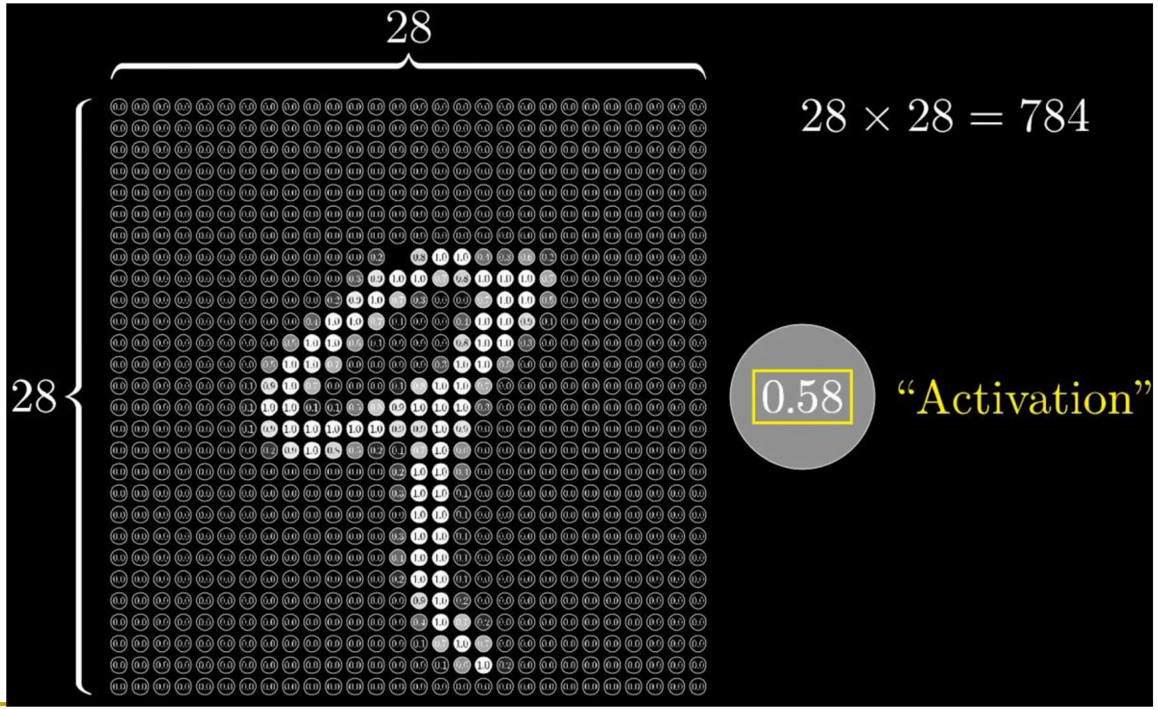
- This is the simplest network, called **Multilayer perceptron (MLP)**
- One input layer
- Two hidden layers
  - 2 is an arbitrary choice
  - Each has 16 neurons or units
- One output layer with 10 units for the 10 digits
- All units are fully connected.



# Each neuron is a function, computing an activation value based on all its inputs

- These 784 neurons form the first layer.
- The value held in each output neuron basically tells how likely the input image is each digit.
- Activations of one layer determine the activations of the next layer

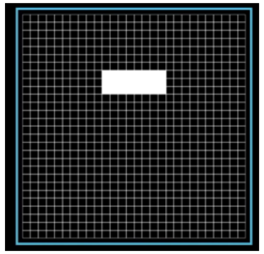


# Intuitive idea of layers

- The first layer just the gray scale value of each pixel in the image.
- The second layer may capture some low-level features, e.g., edges of different orientations.
- The third layer may capture some high-level features such as loops, strokes, and lines.
- The final layer tells which combination of the subcomponents corresponds to each digit.

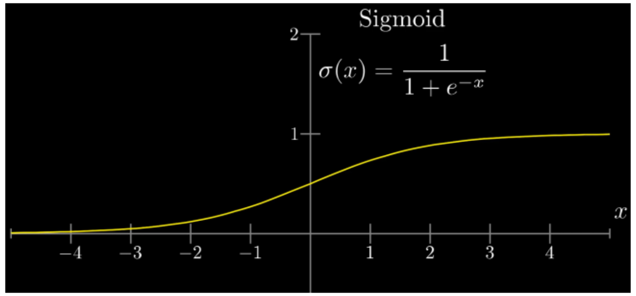
# Let us look at a particular neuron

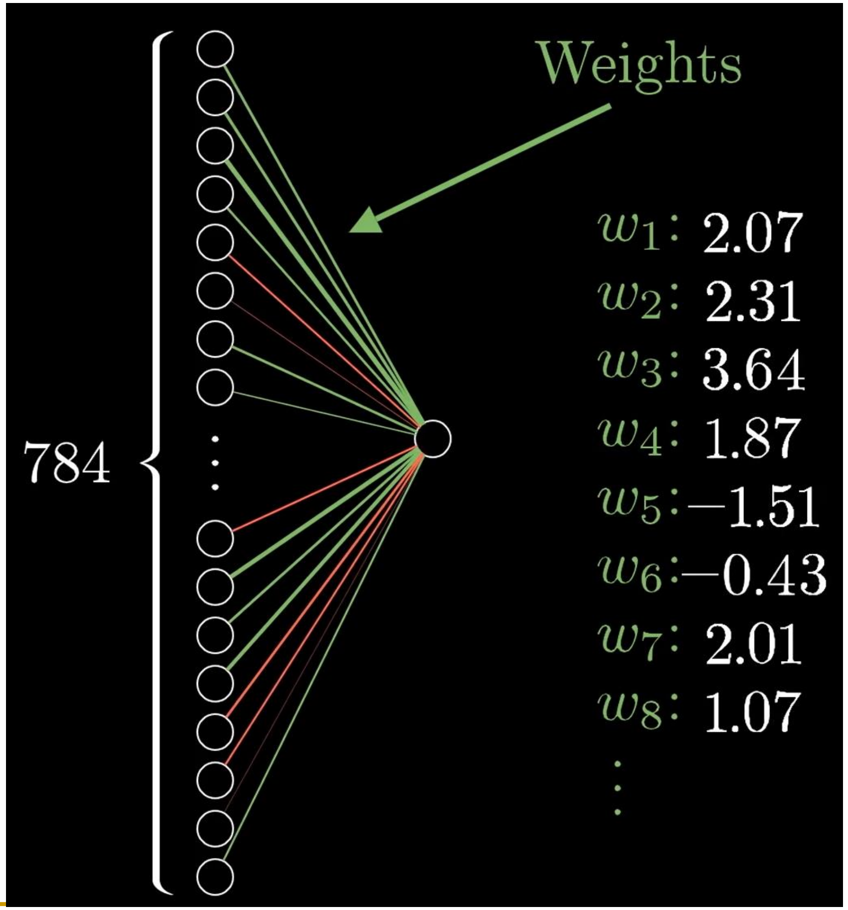


- How does it pick up a small patten?
- For the value of this neuron, we compute

  $w_1a_1+w_2a_2+ w_3a_3+ \ldots +w_na_n+ b$

  - Which may be any value. In this case, we want the values between 0 and 1, we use squash function sigmoid ($\sigma$)



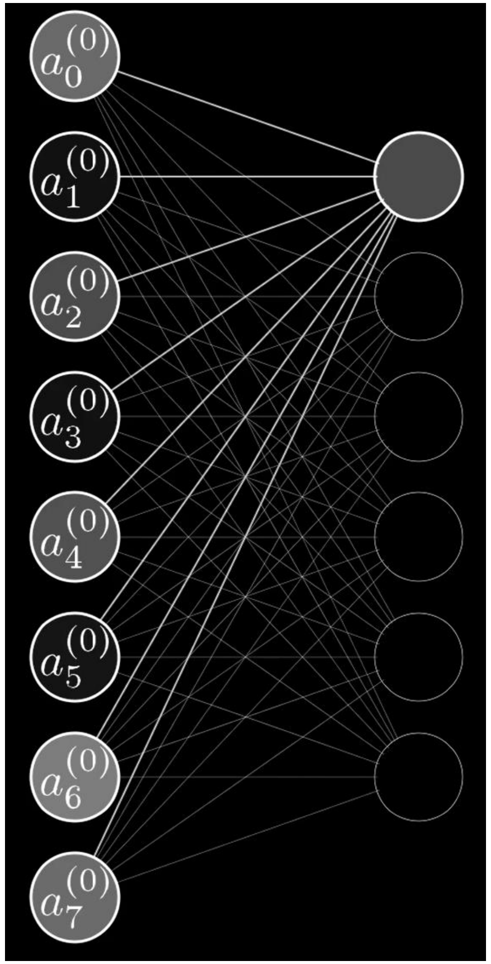




- $\sigma(w_1a_1+w_2a_2+ w_3a_3+ \ldots +w_na_n + b)$

# How many parameters?

- Each neuron in one layer is connected with every neuron in the next layer (fully connected).
- We have
  - Number of parameters (or weights): 784x16 + 16x16 + 16x10
  - Number of biases: 16 + 16 + 10
- Total number of parameters:  13,002
  - These all can be tuned and changed.
- Learning: find suitable values for all these parameters to solve the problem at hand, e.g., classifying hand-written digits.

# This network is a function with 13,002 parameters




Sigmoid ↓

$$a_0^{(1)} = \sigma\left(w_{0,0}\, a_0^{(0)} + w_{0,1}\, a_1^{(0)} + \cdots + w_{0,n}\, a_n^{(0)} + b_0\right)$$

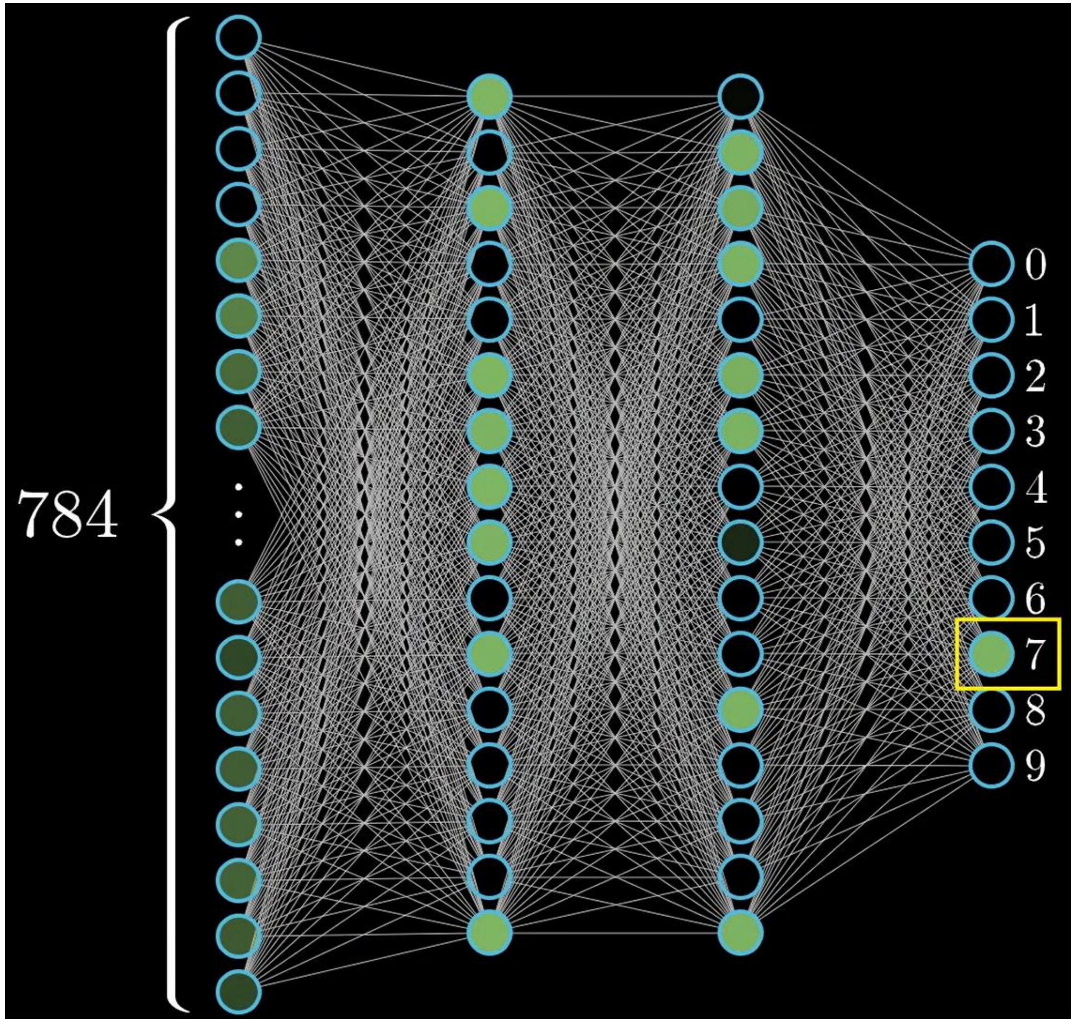
↑ Bias

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}\mathbf{a}^{(0)} + \mathbf{b}\right)$$

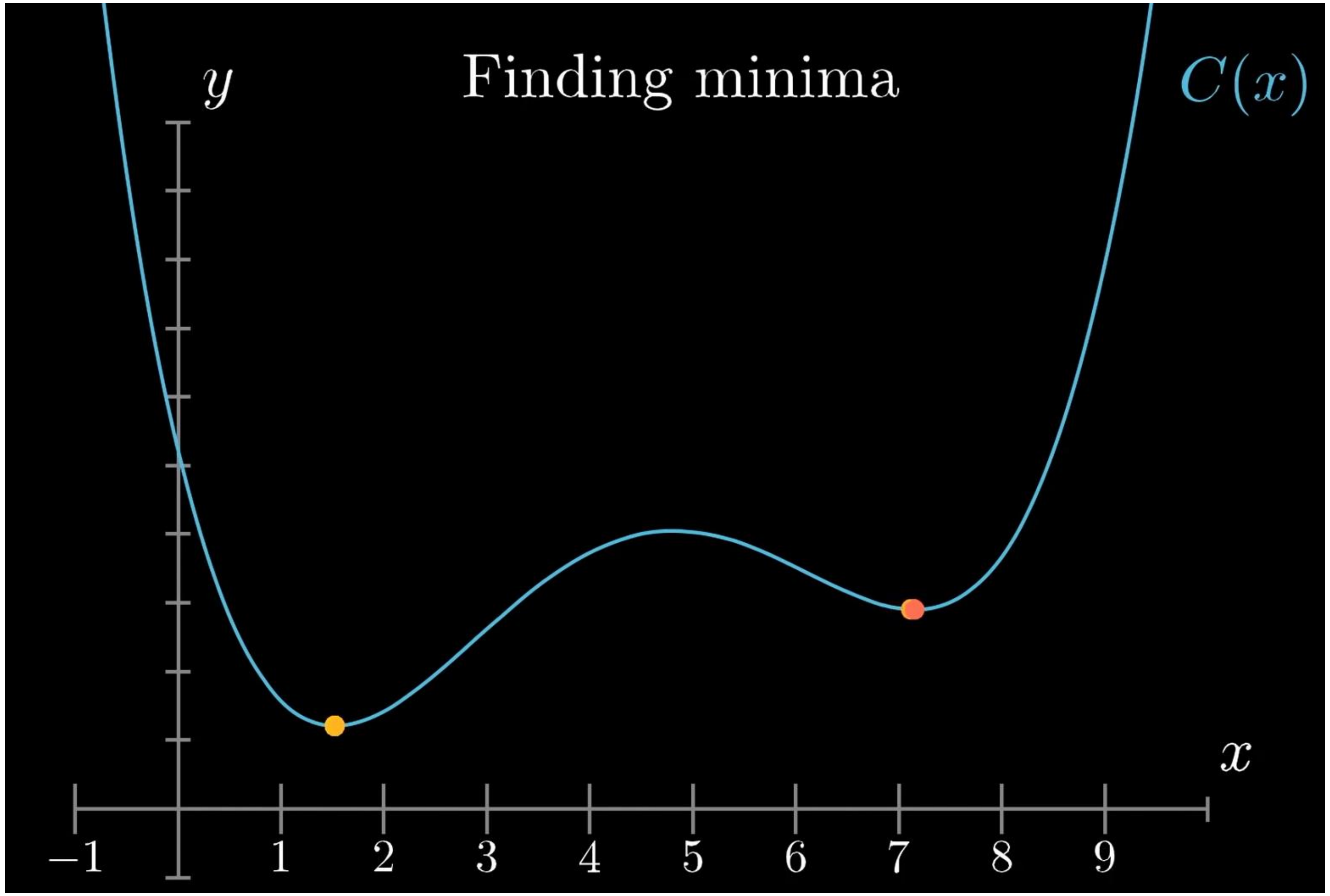
# Learning

- Use a lot of training examples
  - Images of handwritten digits with the correct labels (what numbers they correspond to)
- to adjust those 13,002 weights and biases to improve the performance on training data.
  - Hopefully, the resulting network also generalizes to test data.
- An algorithm is needed: backpropagation



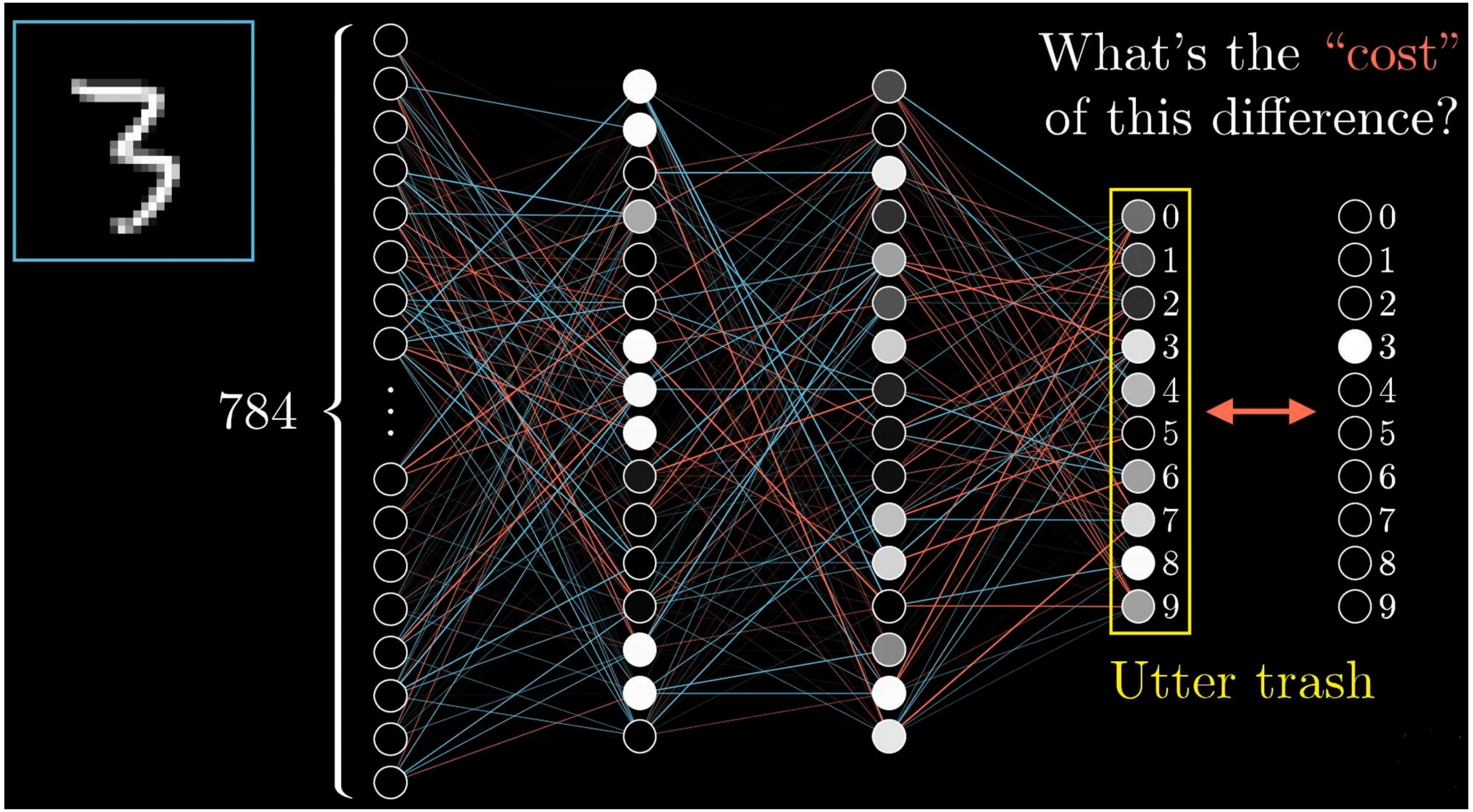
# Training is an optimization problem.

- Trying to find a minima for a cost function *C*(*x*)
- At the beginning, we just give those weights and biases some random values.
- The cost function basically shows how bad the prediction is.



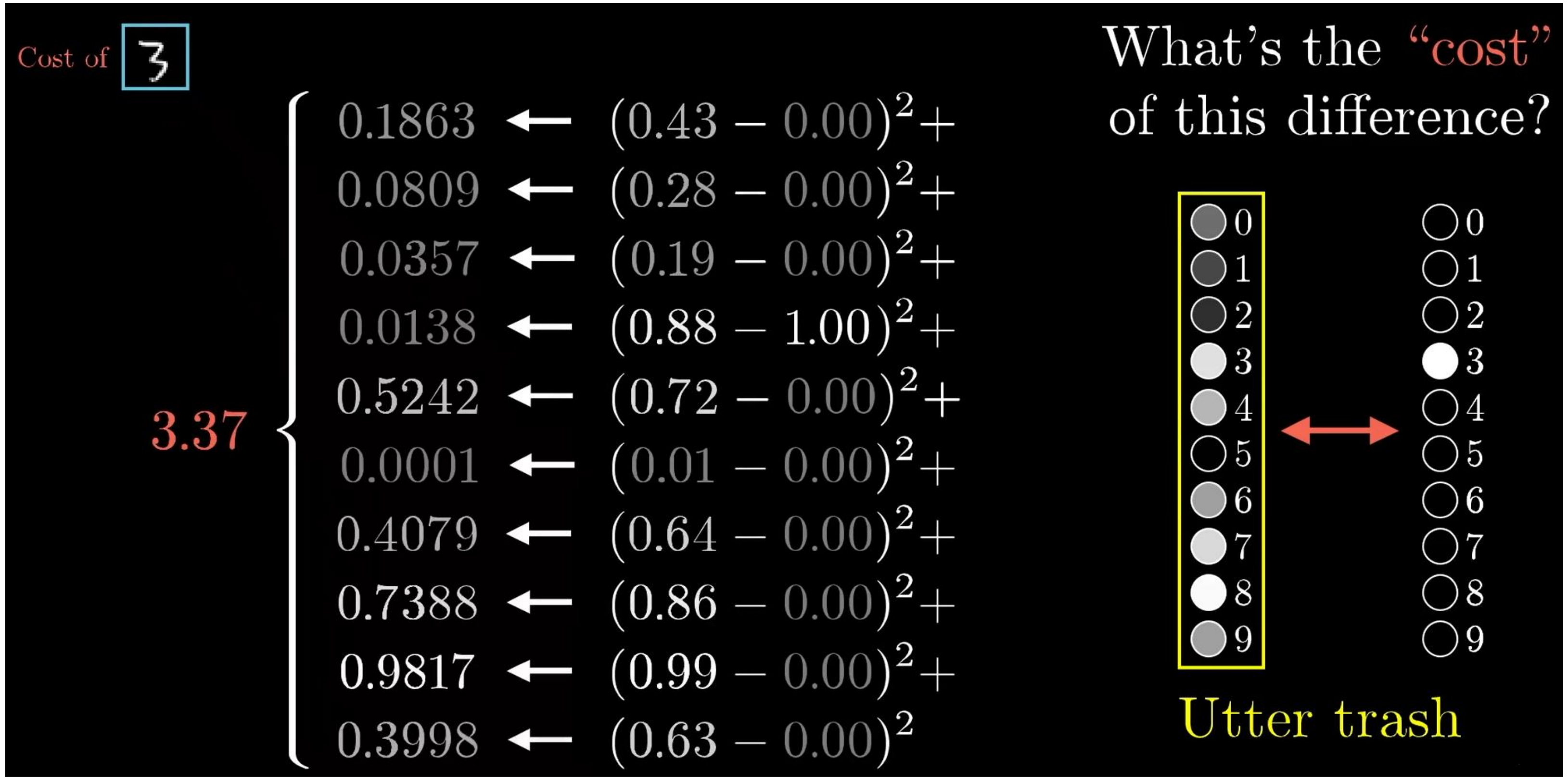
# We start with a random initialization

- Input 3 gets nonsense results at the output layer.
- Use cost function to measure the difference.

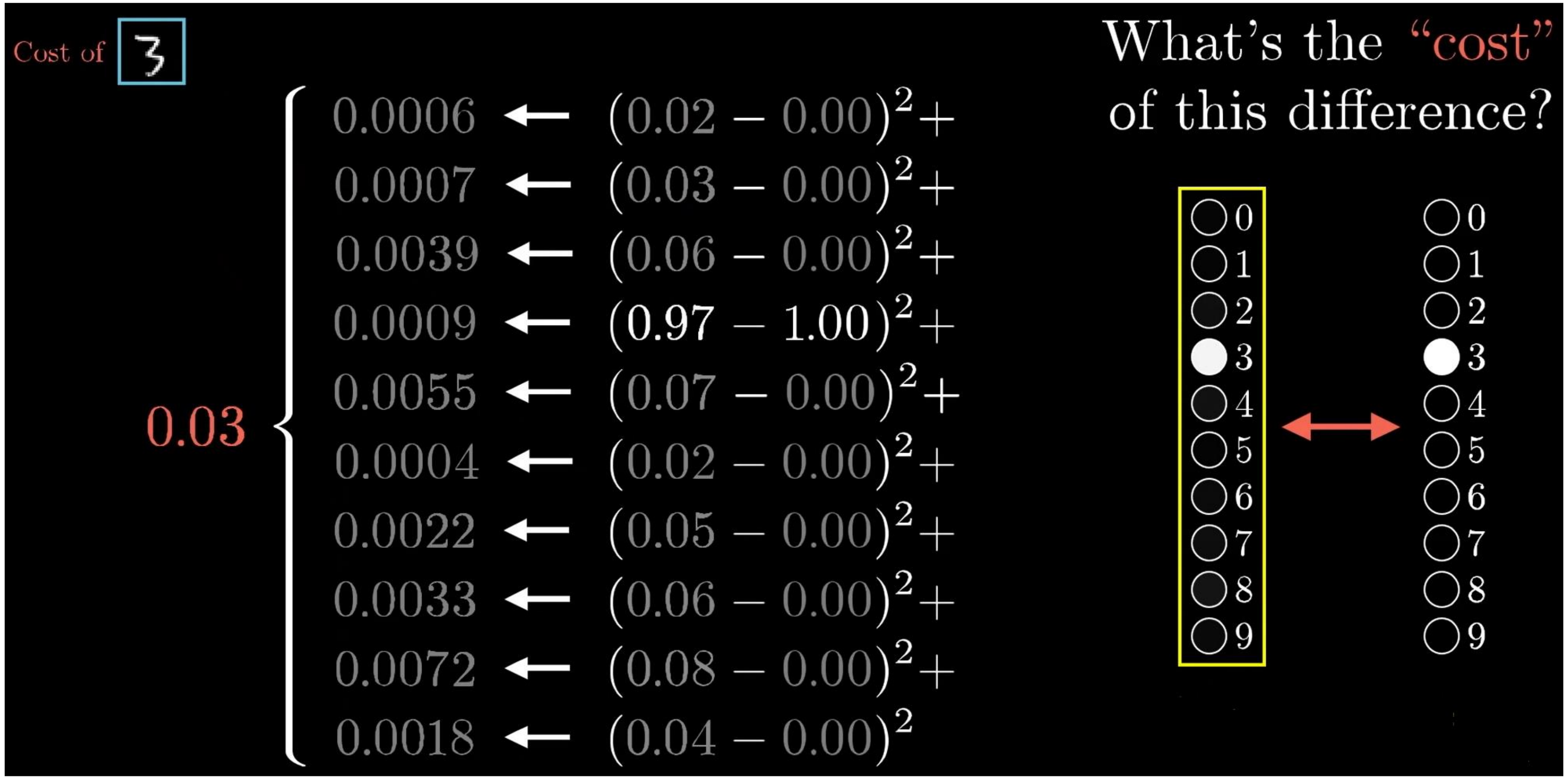


# Square loss (cost) function

- We take the squared difference of what the system gives and what is correct.



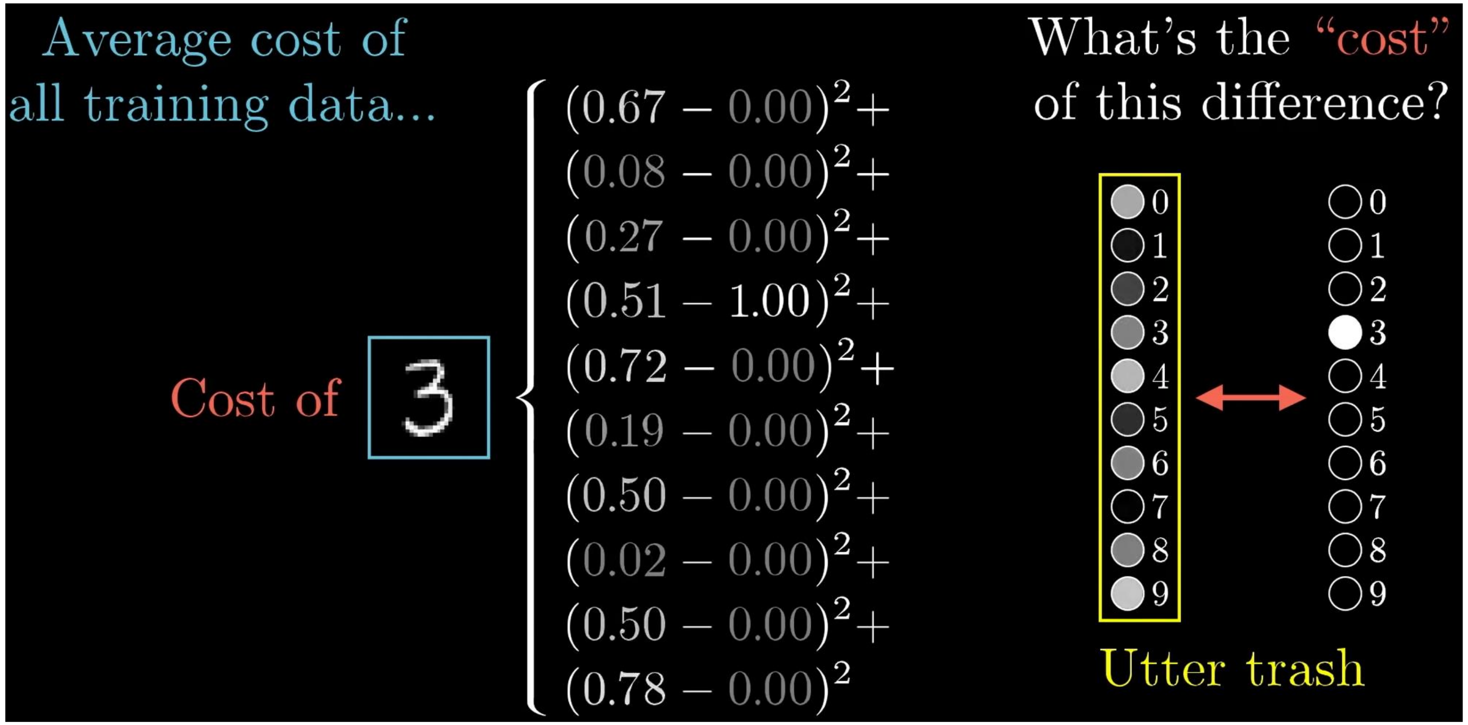
# Cost will be small if the classification is correct.



# Cost average over all training data

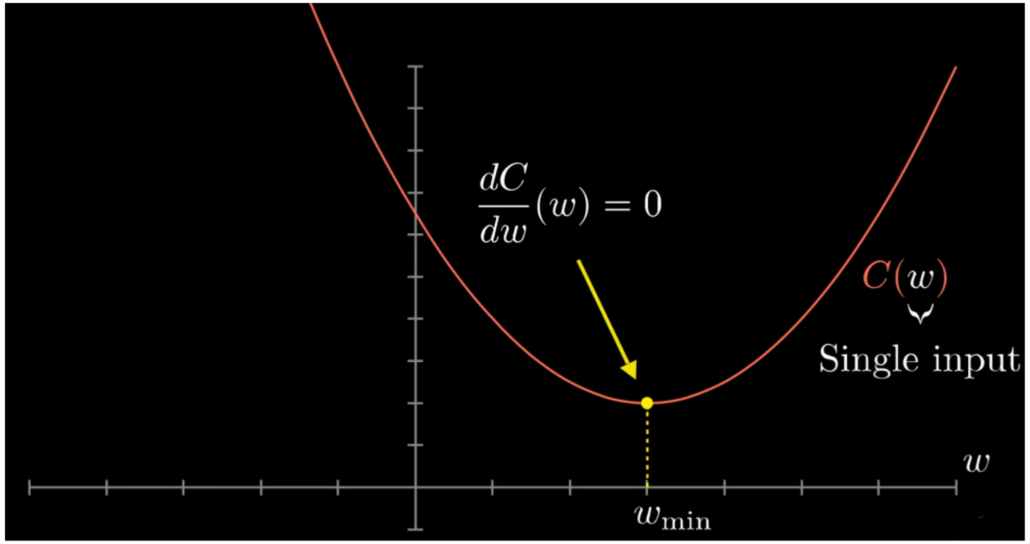
- The average cost gives an idea how good the network is in classification.
- Training algorithm basically changes all 13002 those weights and biases to get better cost.
  - How to do that?

Here we only show only one training example

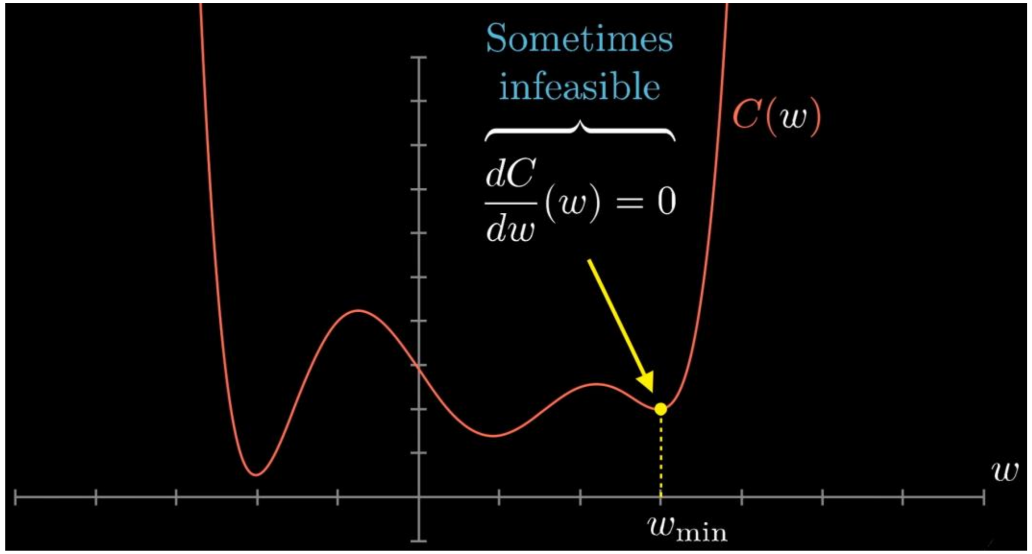


# How do we optimize? Let us consider only one weight *w* first

For a simple function



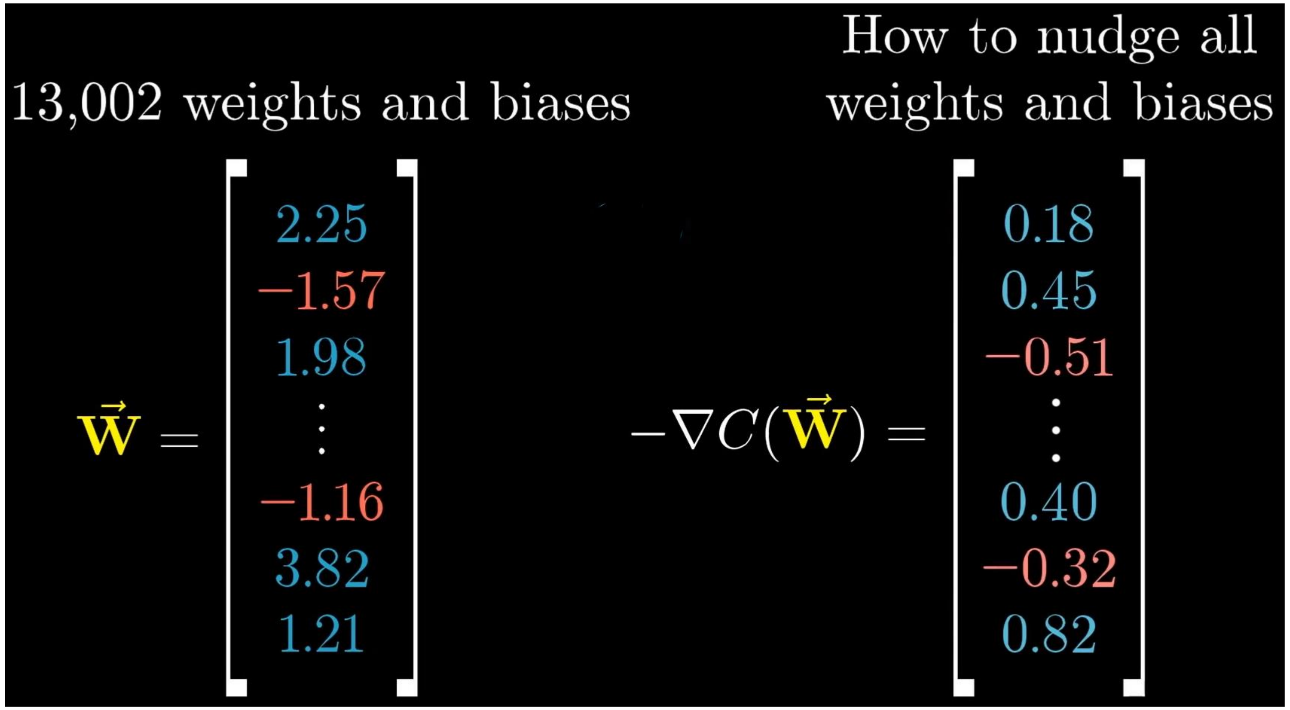

For a complex function




- Very difficult for our cost function with 13,002 variables.
  - We need gradient decent.

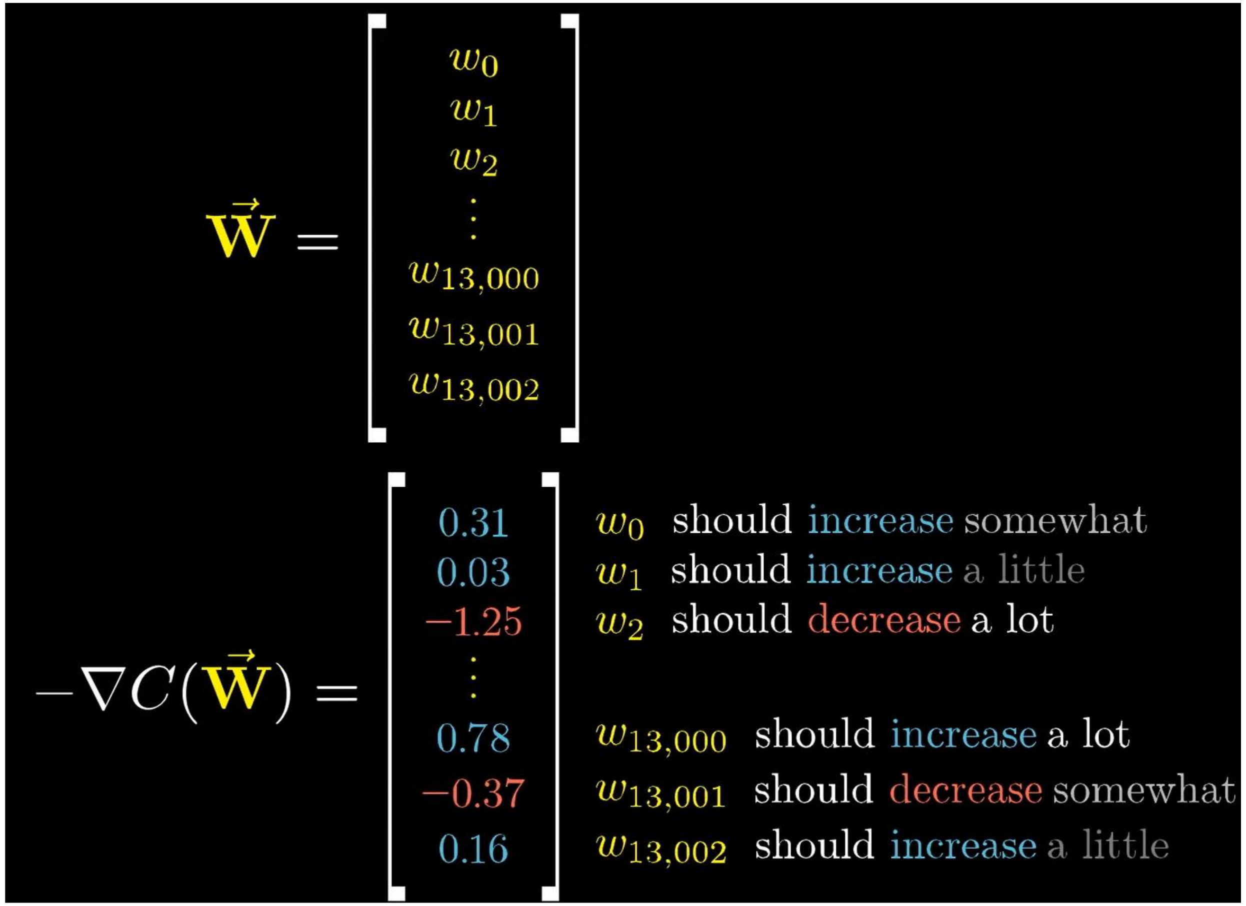
# How is gradient descent used

- Let us put all the 13,002 weights and biases in a single vector and all the negative gradients of them into another vector.
- We can nudge or change the weights and biases to reduce the cost and to minimize it.
- The algorithm doing this is backpropagation.



# Meaning of those gradient numbers

- We can see
  - what weight should increase and
  - what should decrease
  - what change means a lot



# Backpropagation

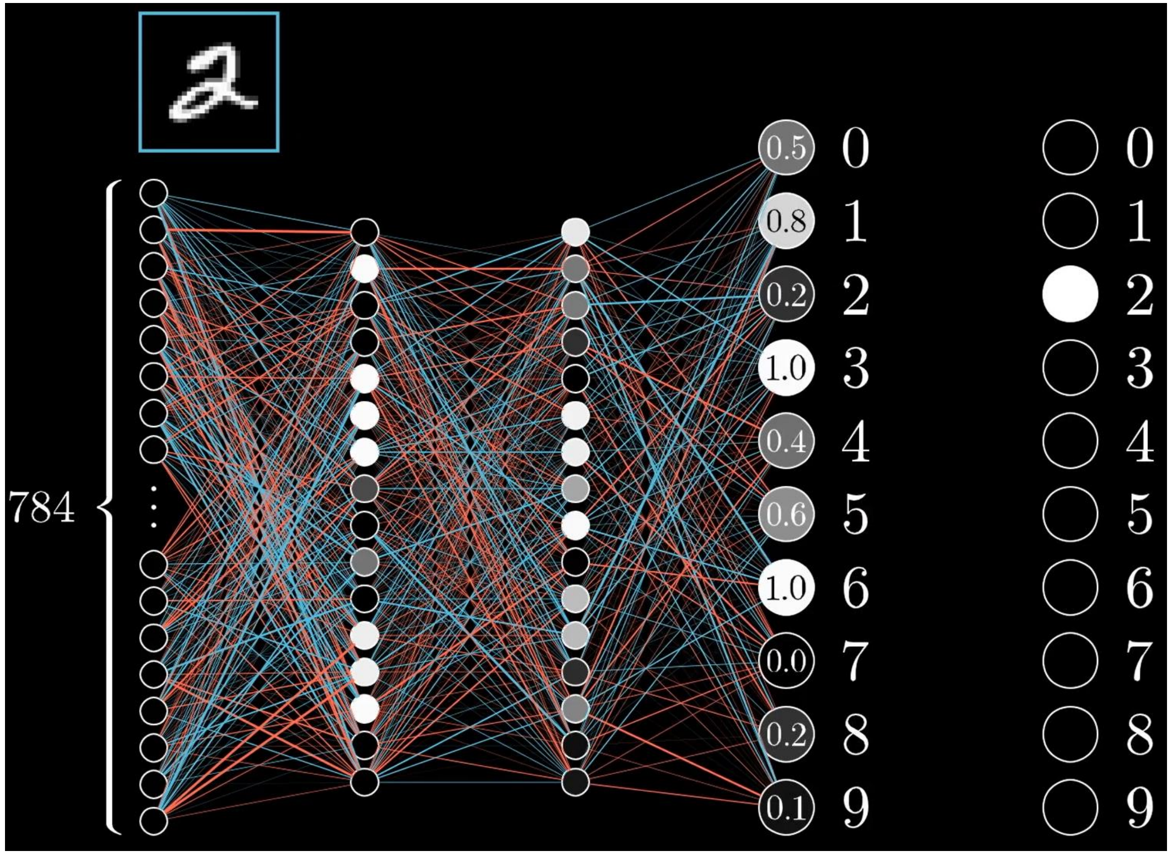
- The backpropagation algorithm was originally introduced in the 1970s,
- but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams.
- That paper describes several neural networks where backpropagation works far faster than earlier approaches,
  - making it possible to use neural nets to solve problems which had previously been insoluble.
- Today, the backpropagation algorithm is the workhorse of learning in neural networks.

# Training: backpropagation algorithm

- Step 1: initialize the weights and biases.
  - Weights in the network are initialized to random numbers from interval [-1,1]
  - Each unit has a BIAS associated with it
  - Biases are similarly initialized to random numbers from the interval [-1,1]
- Step 2: feed the training sample
- Step 3: propagate the inputs forward; we compute the net input and output of each unit in the hidden and output layers.
- Step 4: back-propagate the error.
- Step 5: update weights and biases to reflect the propagated errors.
- Step 6: terminating conditions.

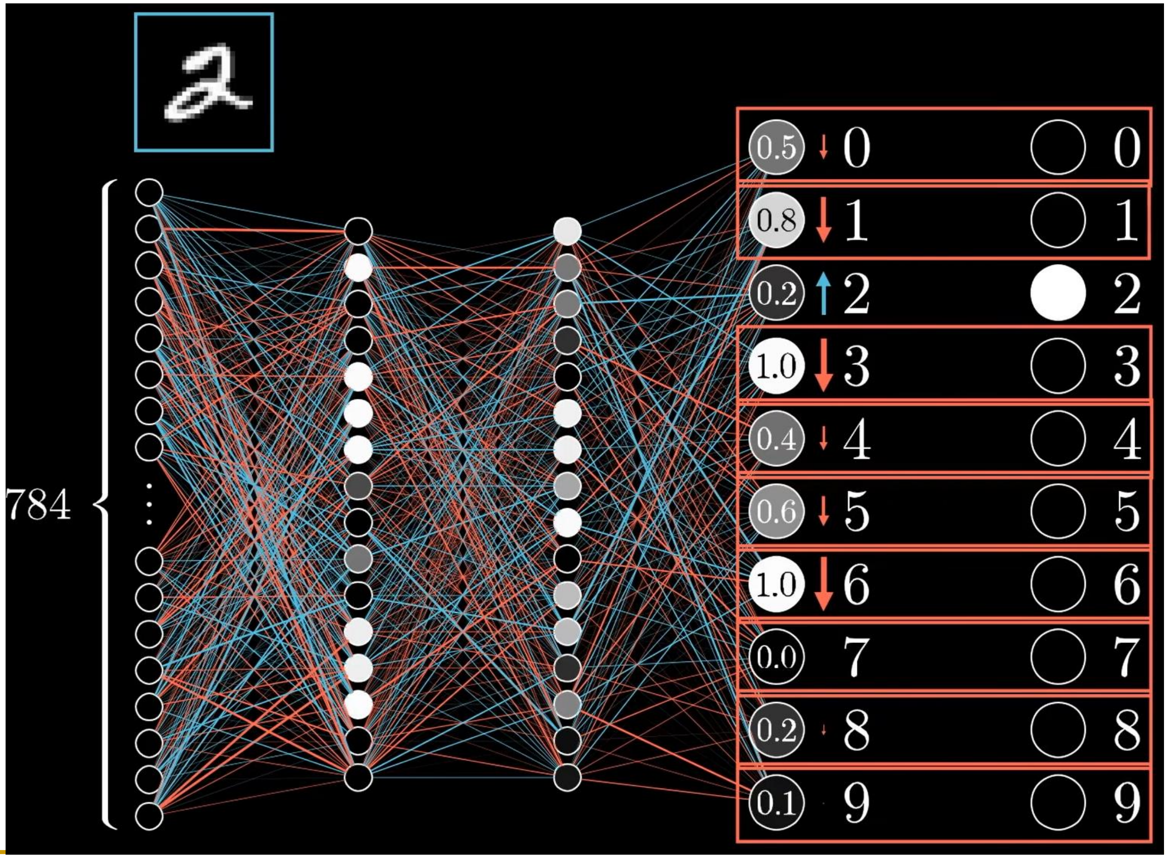
# Intuition of backpropagation

- Since in each step the cost is over all training examples, let us focus on a single example.
- The network isn't well trained, the output activations are pretty random for the input image of 2.
- So we need to adjust those weights and biases.



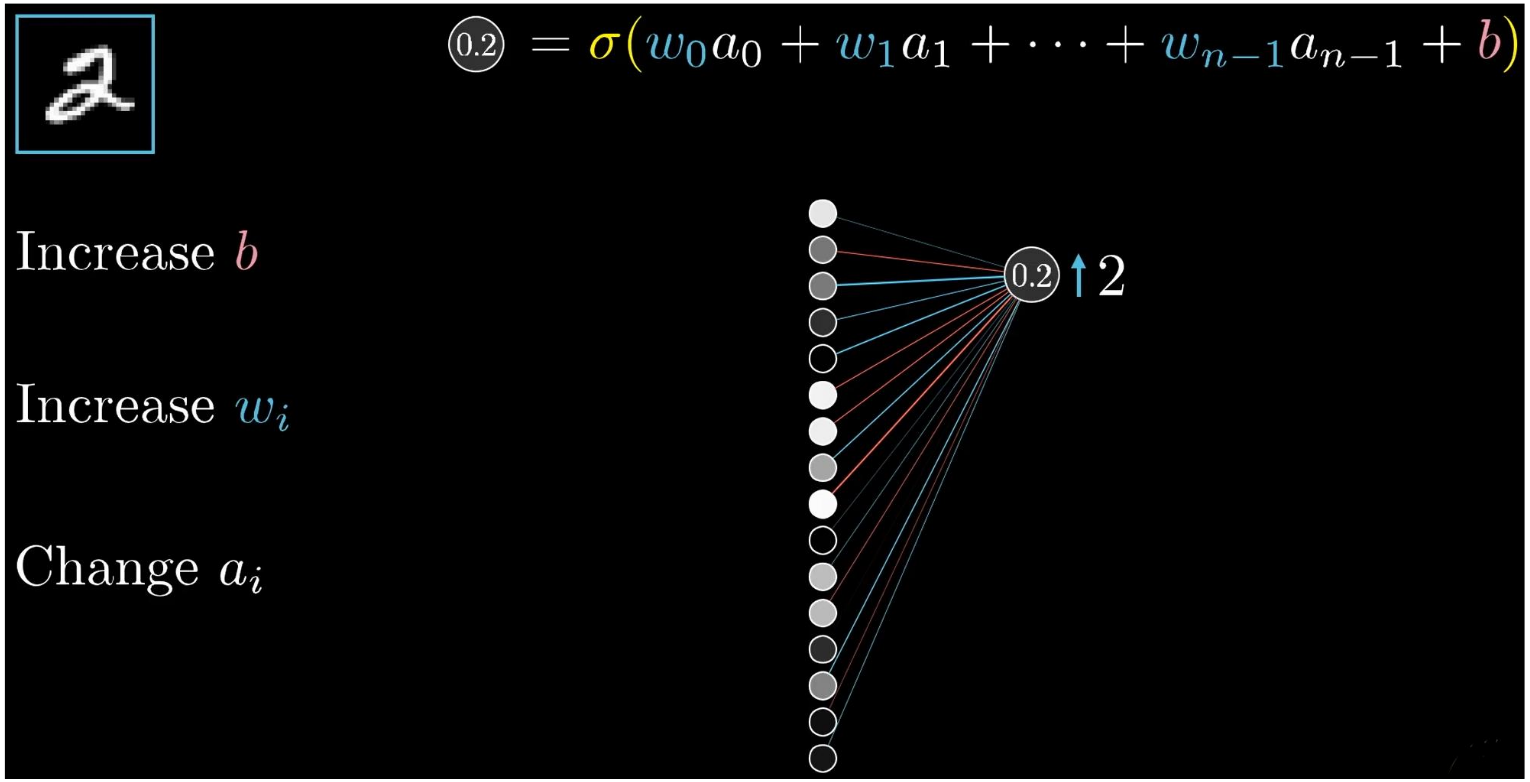
# Intuition of backpropagation (cont.)

- We know which activation should go up and which should go down.
- In this case, the target value for 2 should 1.0 and the others should be 0.0.
- We should nudge activation value for the number ‘2’ up & the rest down.
  - For 7, 8, 9, the values are small.
  - The size of each nudge should be in proportion to its target value



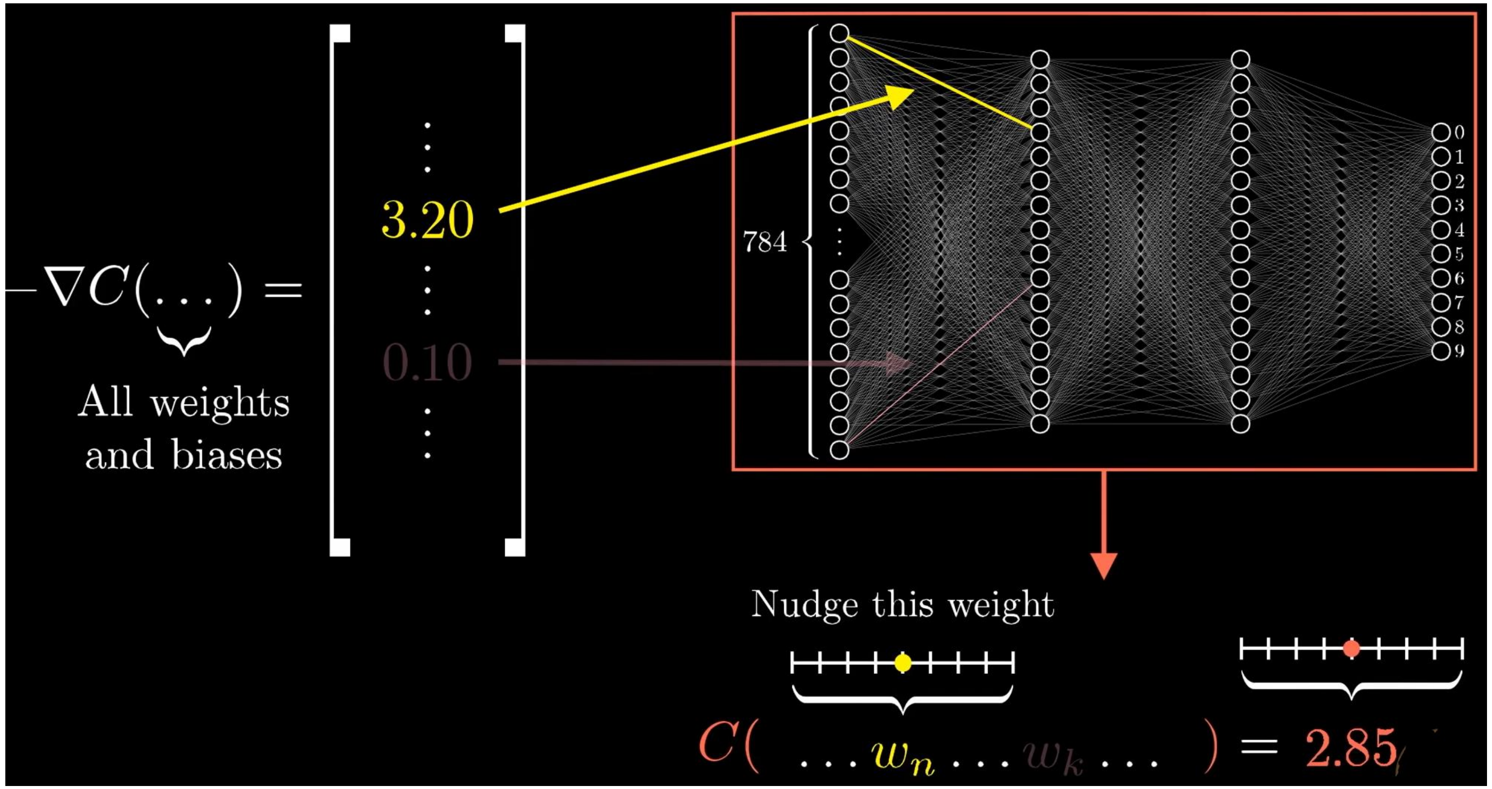
# Let us look at neuron for 2 only

- We can nudge weights, the bias and activations.
  - Note that we cannot change activations,
  - but only the weights and biases of the previous layers, which affect the activations



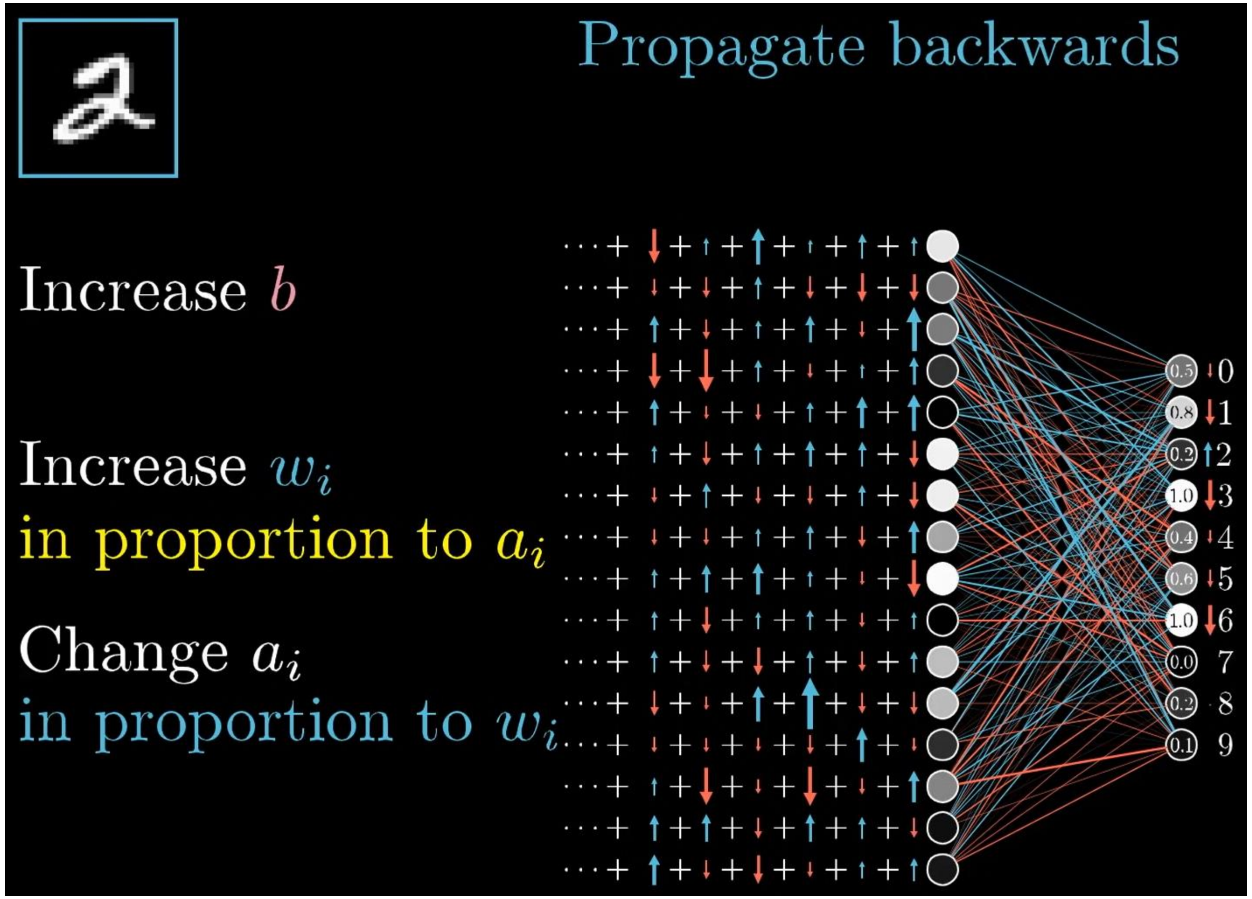
# The effect of gradient

- The gradients tell us which weight or bias should be nudged up or which down,
- but which nudge will give us the best effect "best bang for the buck".



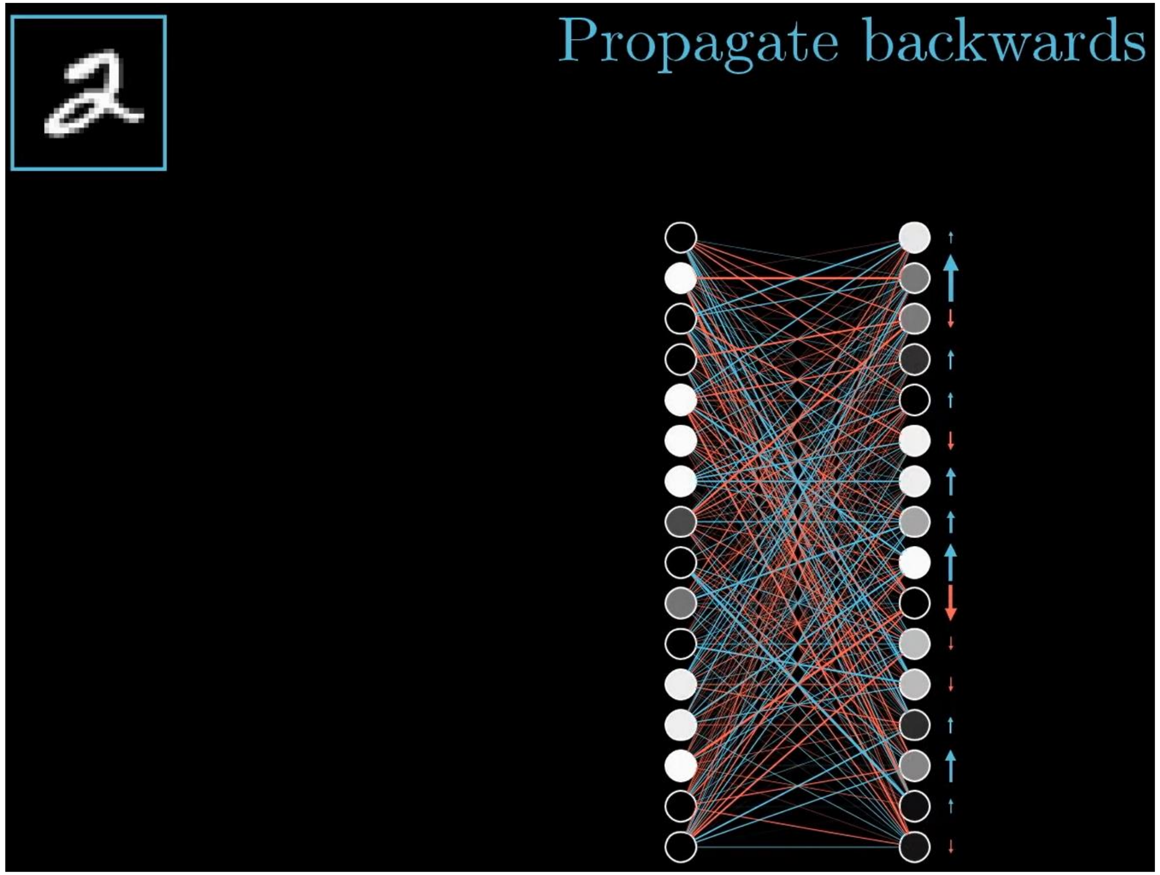
# Considering all output neurons

- We have only considered the output neuron for 2.
- We also need to consider all the output neurons and how they should be nudged and their effect on the second last layer.



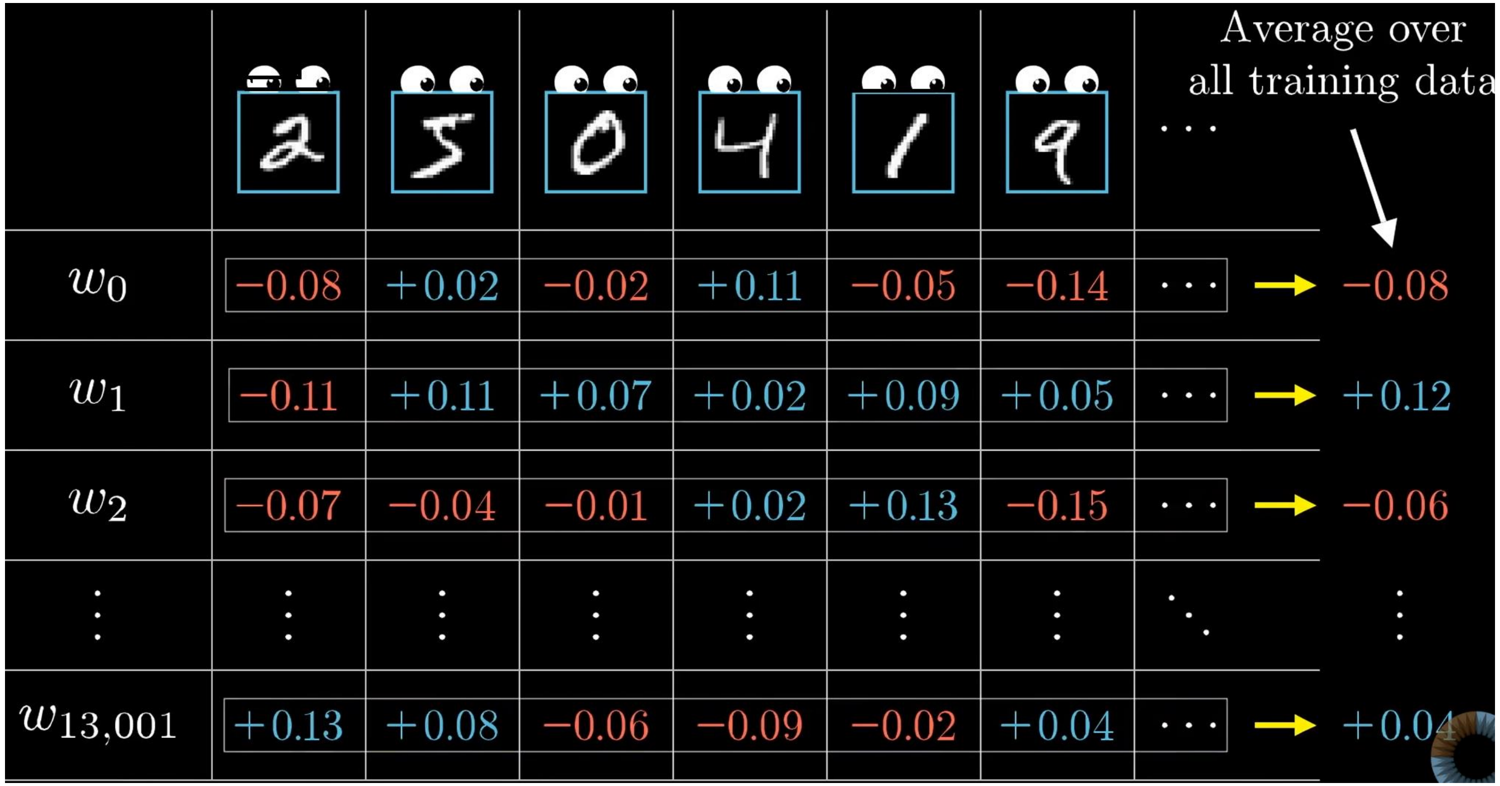
# The idea of backpropagation

- Finally, we sum up all the effects to get what should happen to the second to the last layer.
- Then we can recursively apply the same process to the previous layer and so on.
  - So that their weights and biases can be adjusted.

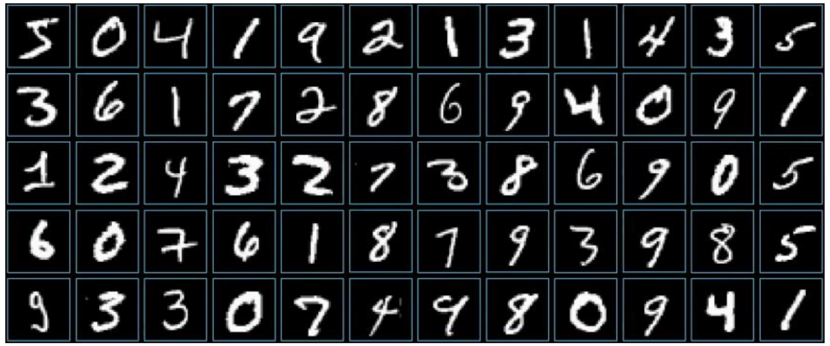


# Considering all training examples

- So far, we have only looked at one training example of 2.
  - We can get how much change should be applied to each weight and bias.
  - But we need to average over all training data to get their desired changes

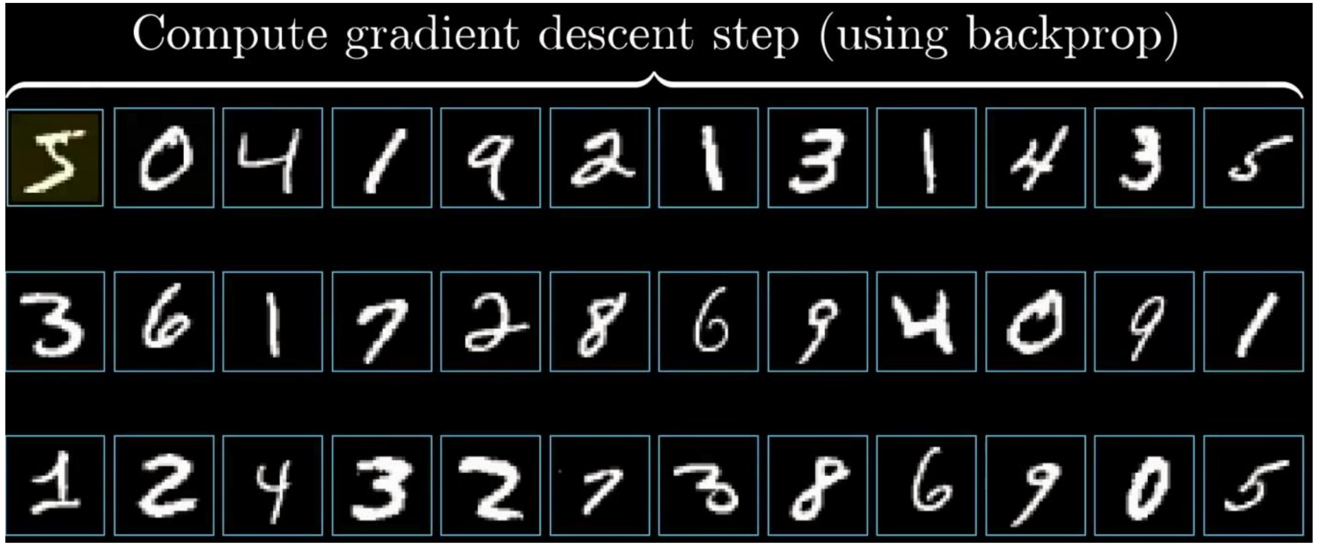


# Stochastic gradient descent



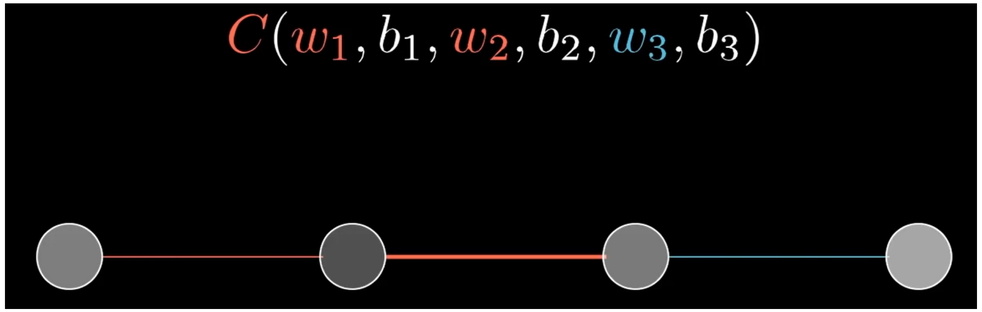
- It takes too long to go though all the training data and all those computations to calculate each nudge/change.
- In practice, we use stochastic gradient descent.
  - We shuffle the data & divide them in minibatches and
  - work on each minibatch in each step.

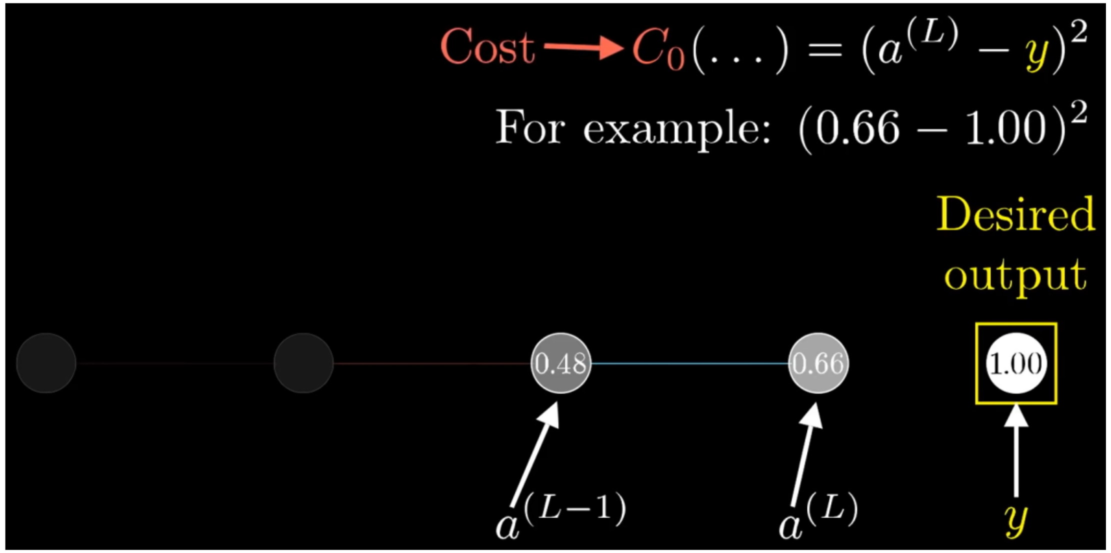
Computing based on minibatches



# Math of backpropagation

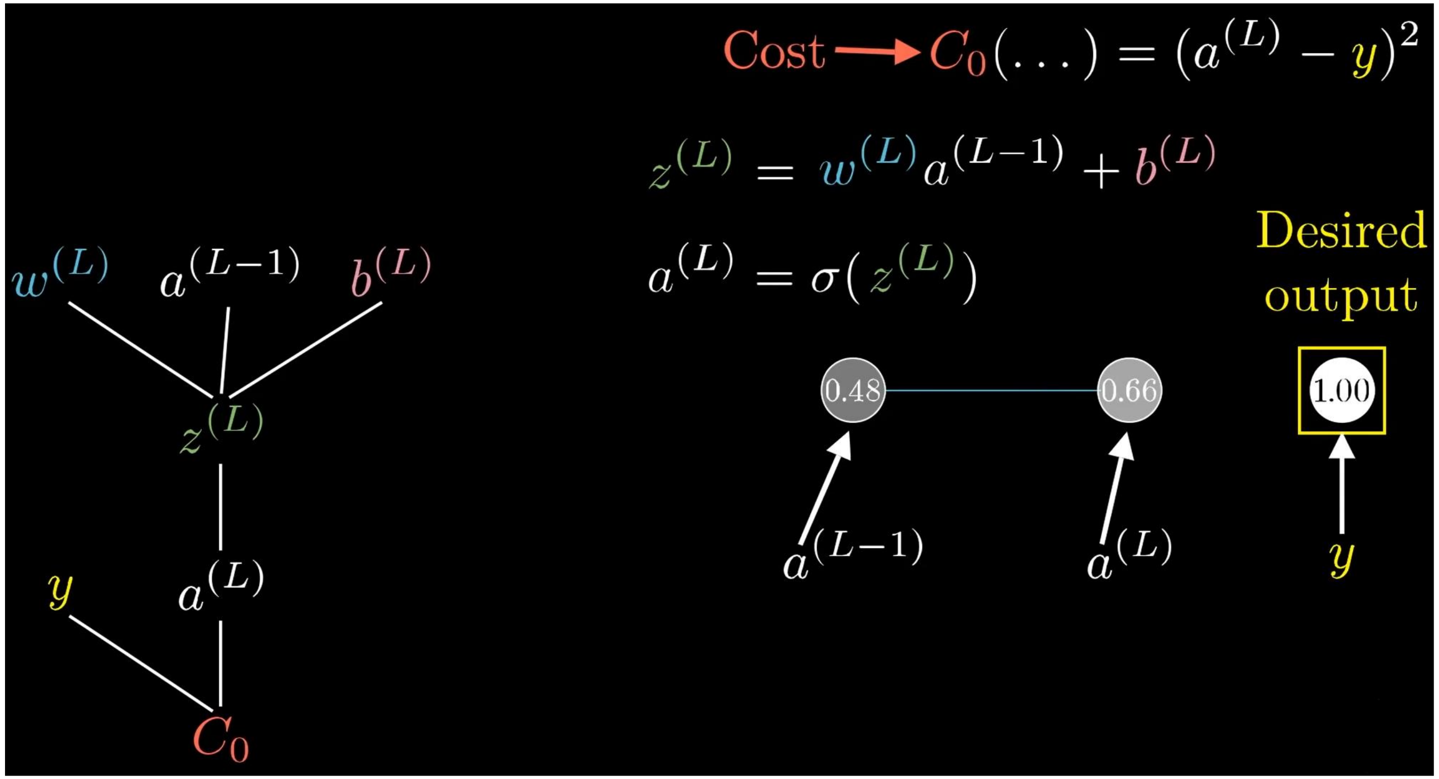
- We start with a very simple case:
  - one neuron in each layer
- Further, we will focus on the last two layers.
  - For a training example with class y, the last neuron is for the class (i.e., 1.00)
- We work on one training example first.



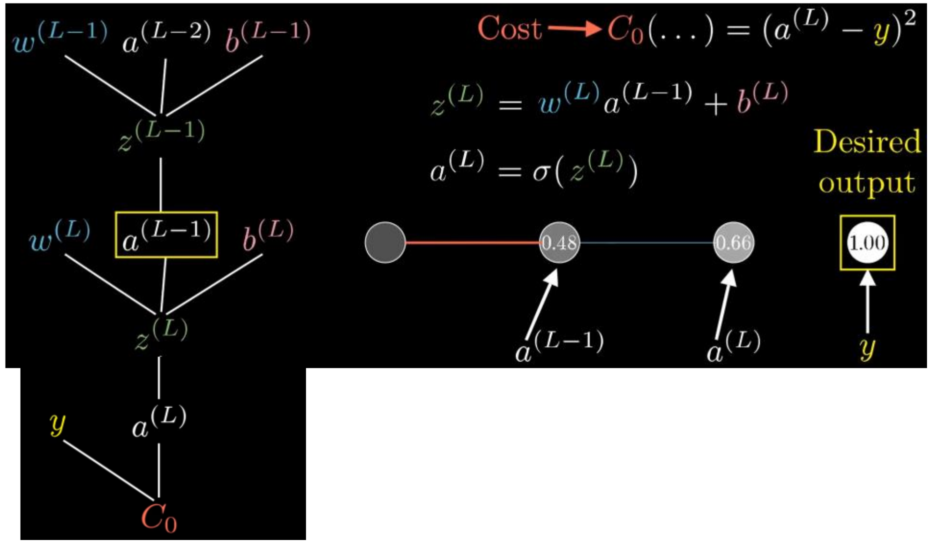



# Model the two layers

- Let us see the flow structure for 2 layers. $C_0$ is the cost of one training example

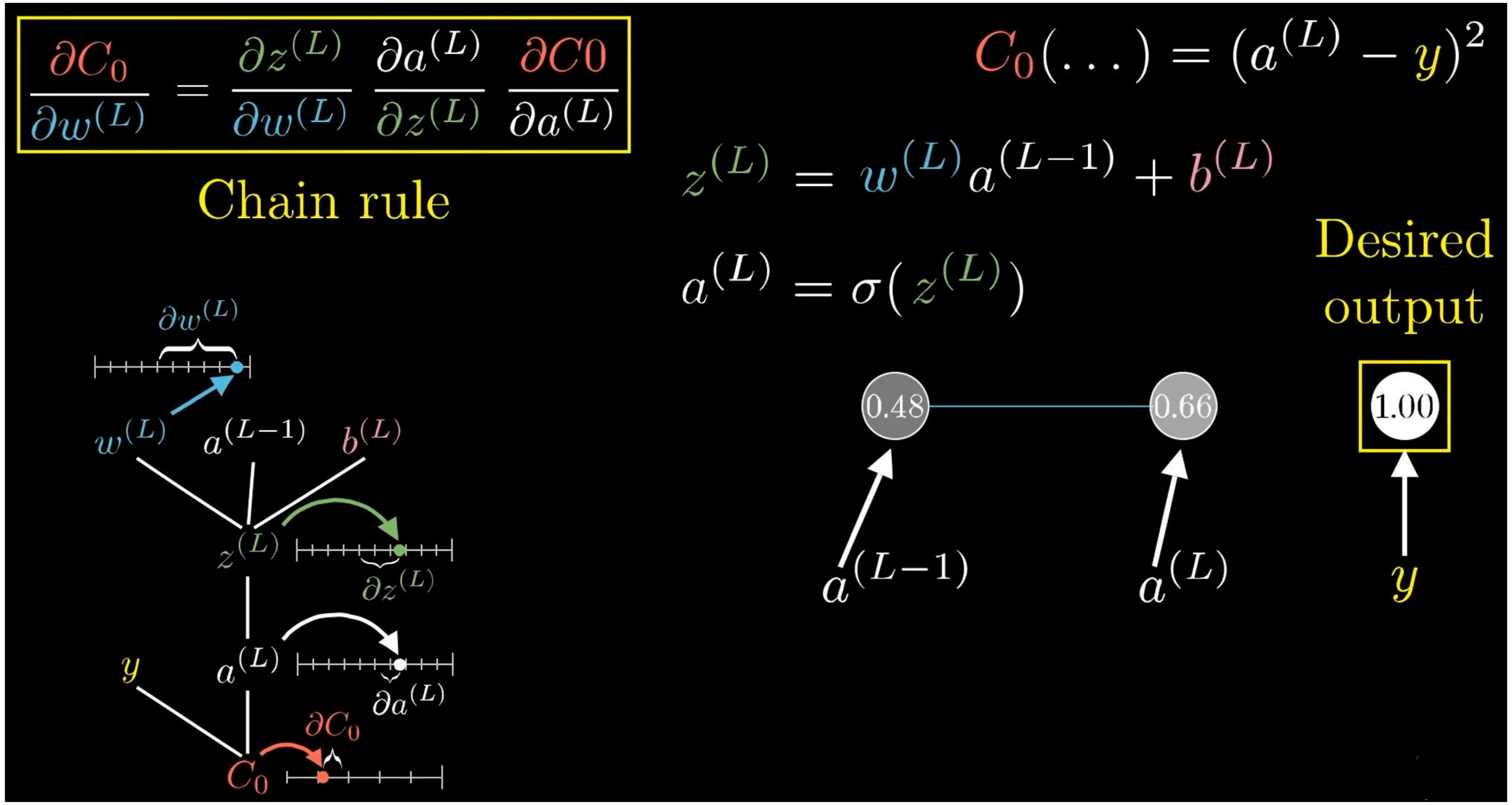



- Note that we can go to the next level too, but we will not focus on that



# How sensitive cost is to a small change in weight?

- Each term is just a numerical value with a number line.
- To get the sensitivity, we take partial derivatives
  - Chain rule



# Compute all derivatives

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$$

$$C_0 = (a^{(L)} - y)^2$$

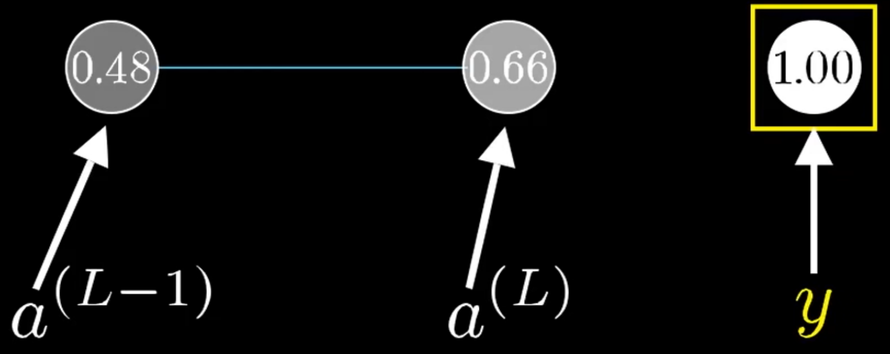
$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$\frac{\partial C0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

$$a^{(L)} = \sigma( z^{(L)})$$

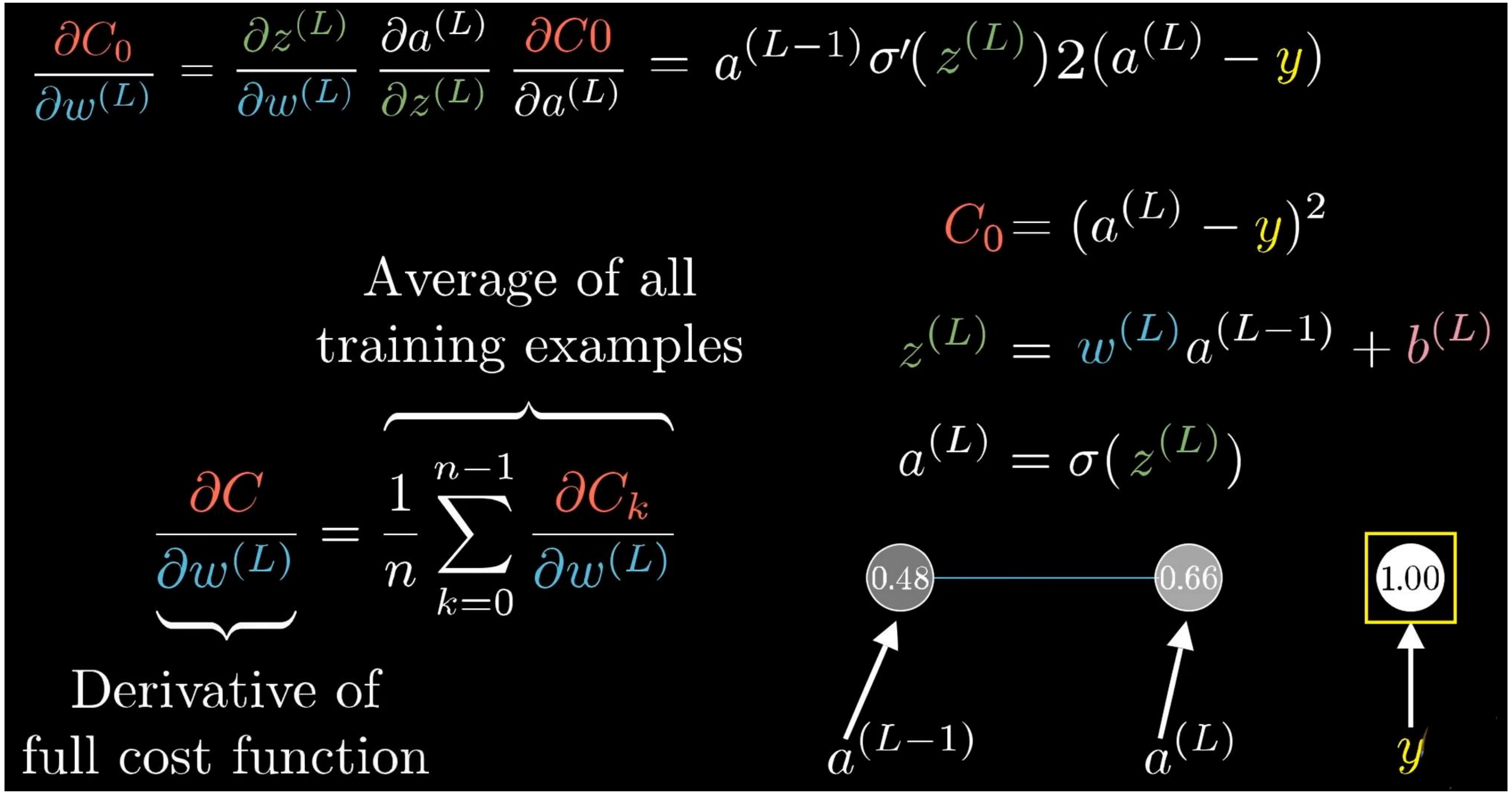
$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'( z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

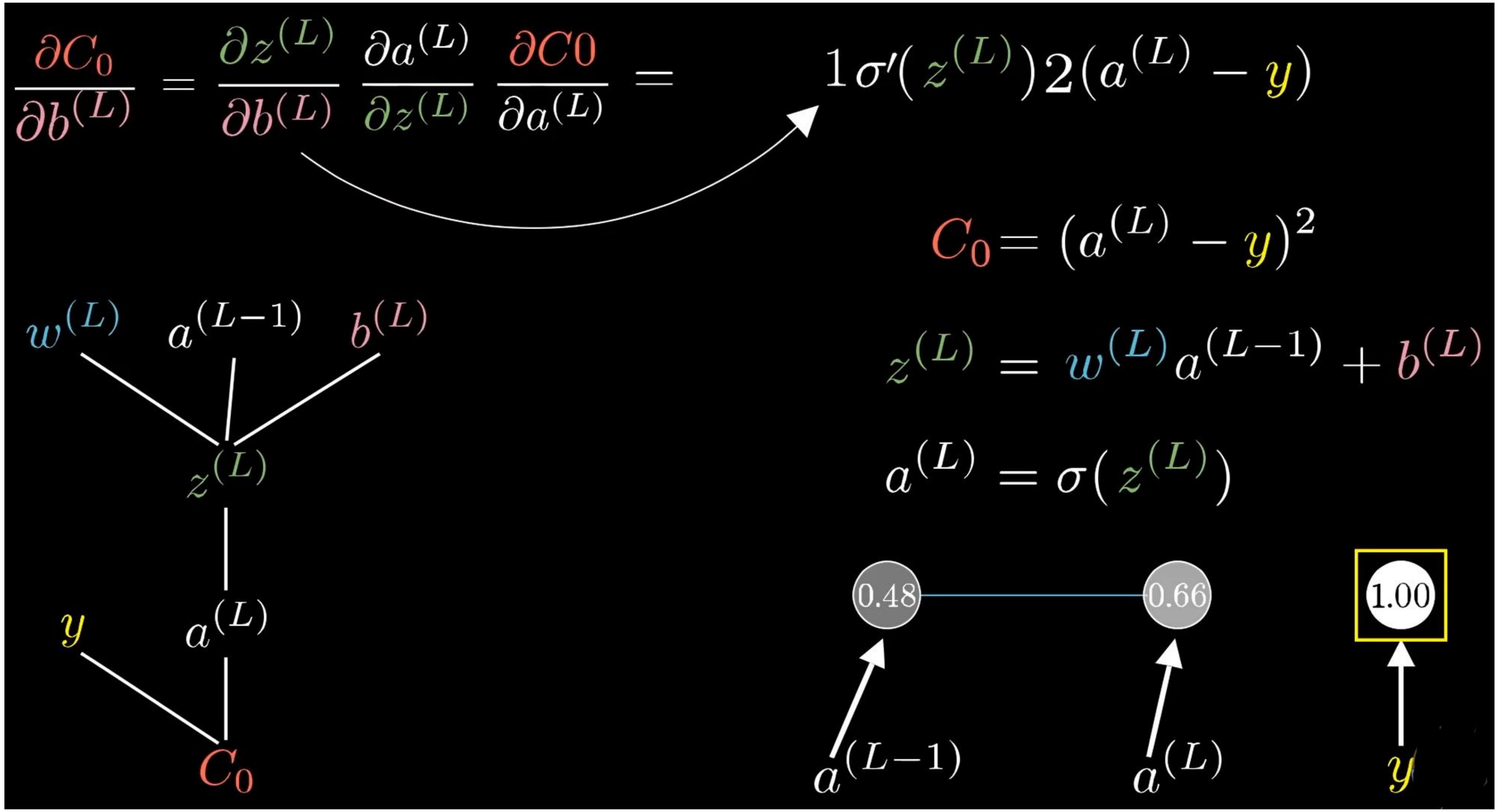


# Consider all training examples

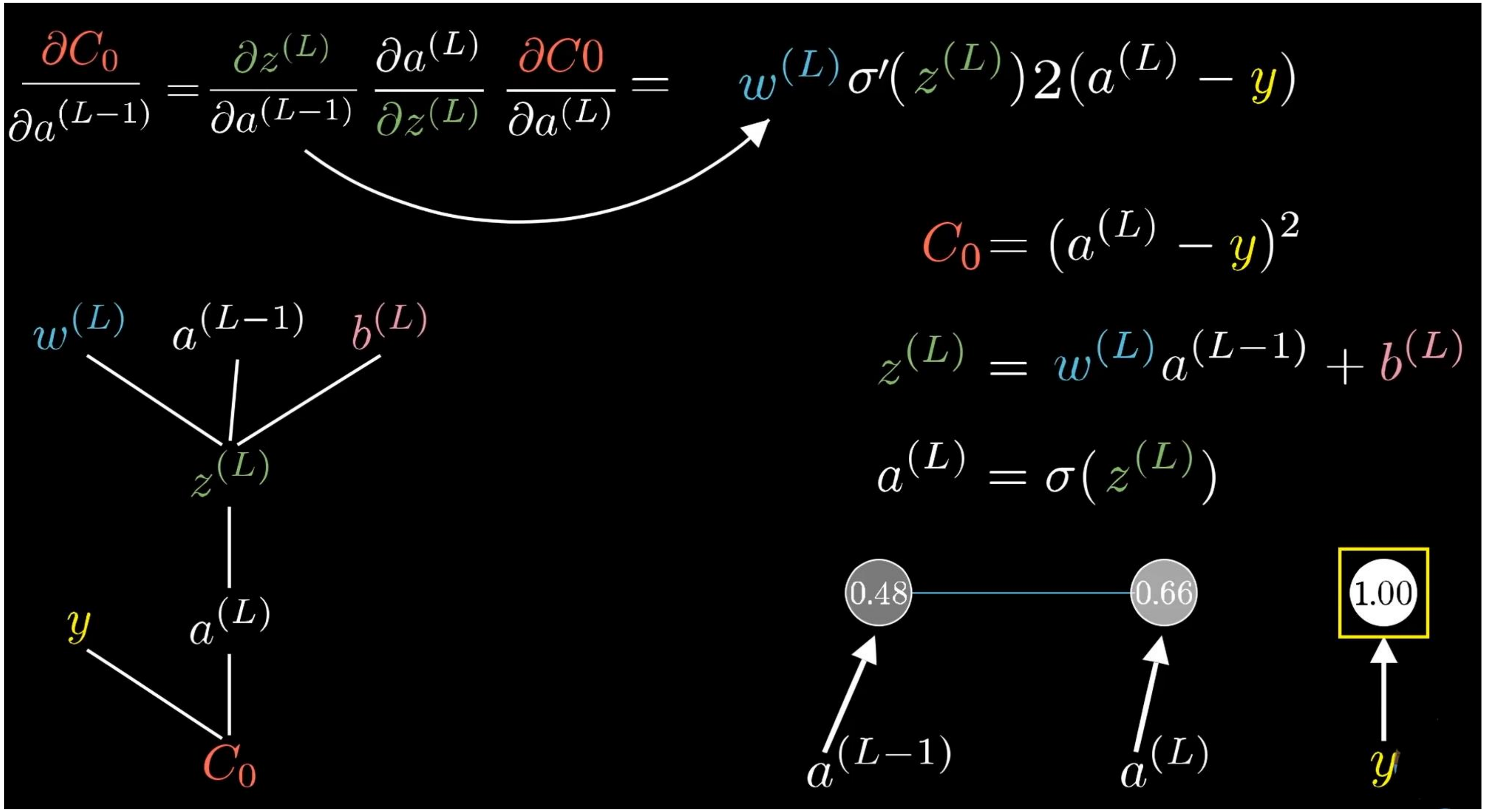
- We have only considered one example and its cost $C_0$.
- To consider all training examples, we average the gradients



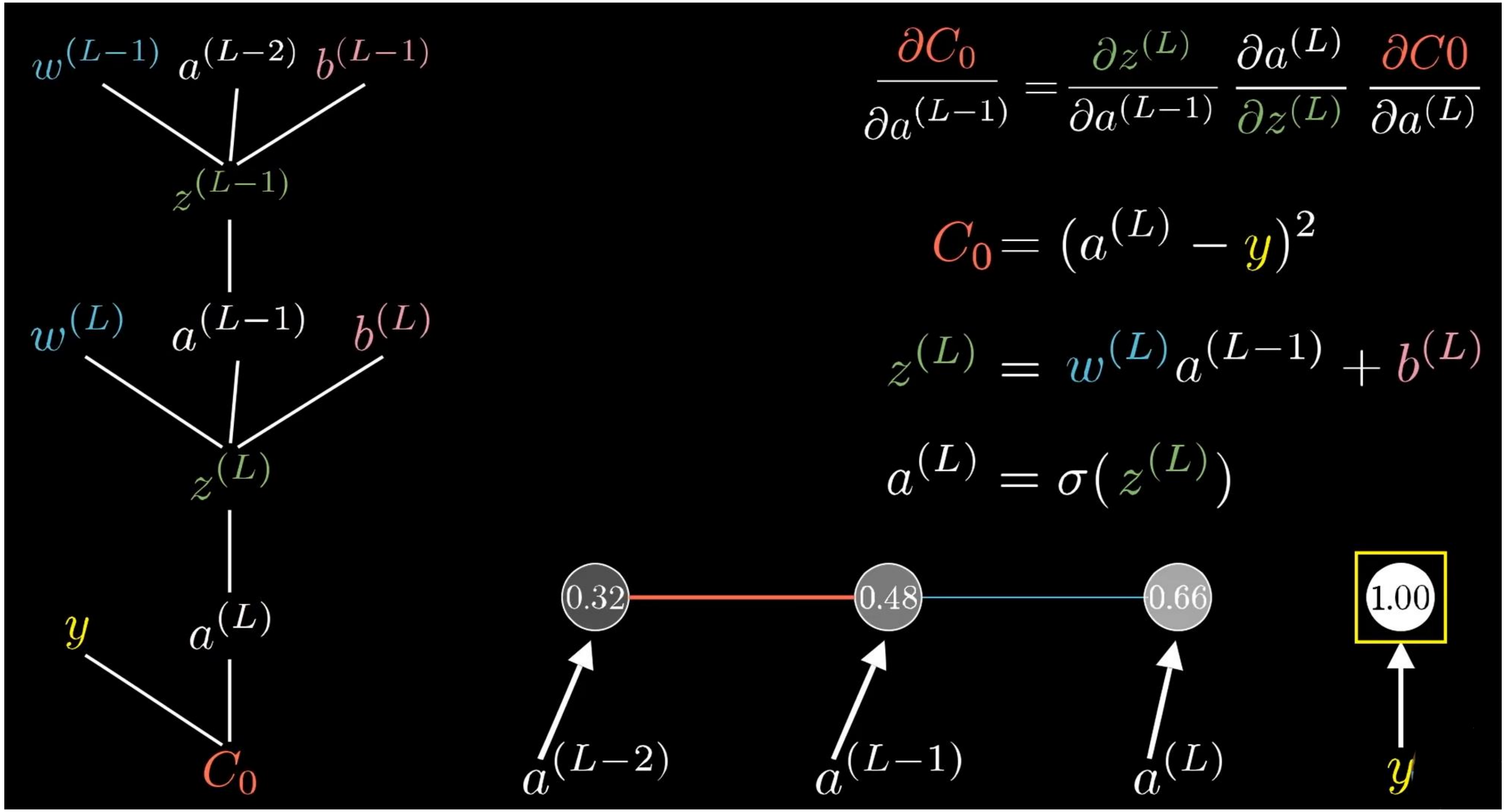
# Take derivative of the bias



# Take derivative of the activation (propagate back)

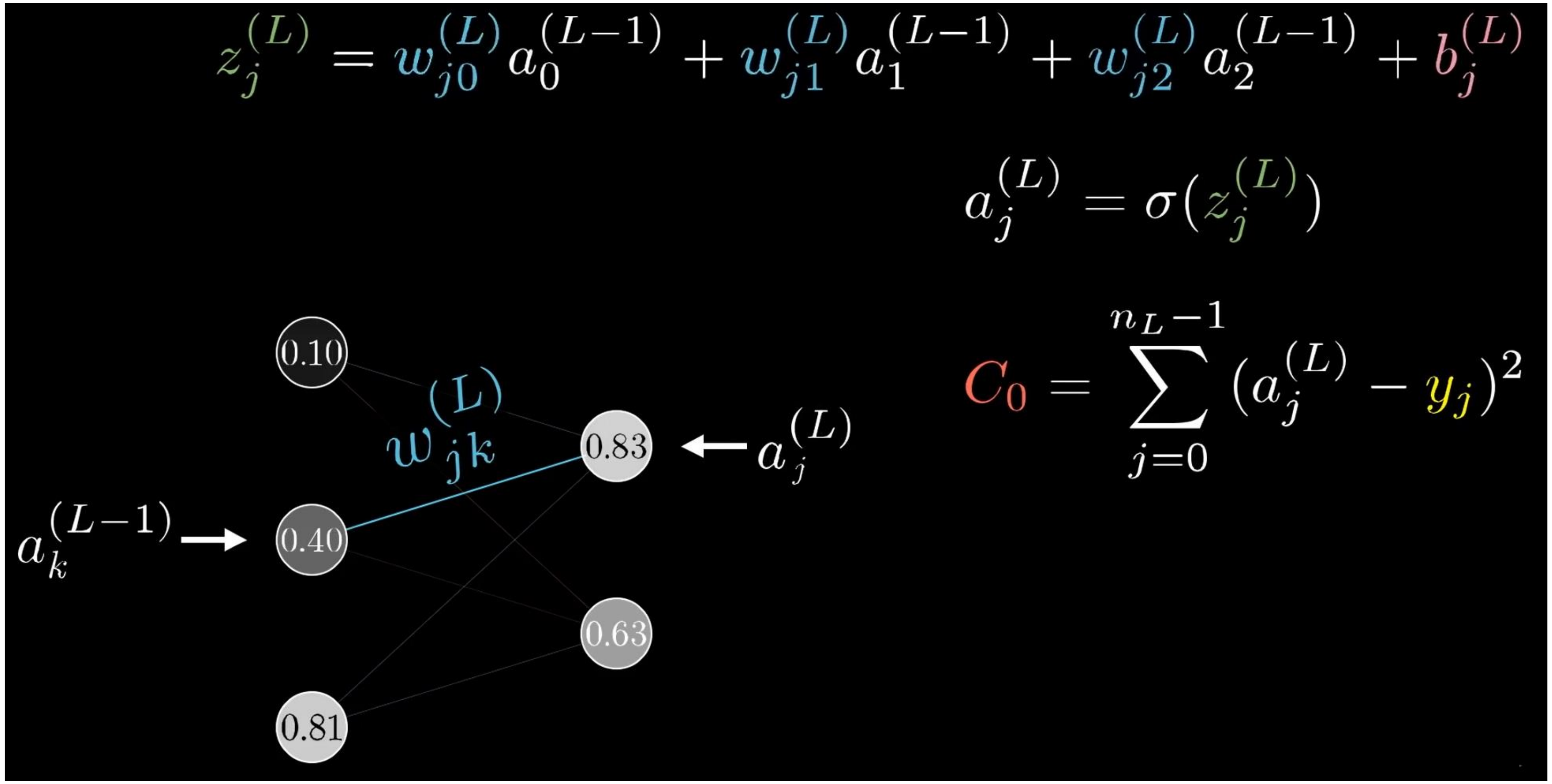


# Iterating the same chain rule idea backward to the previous layer and so on

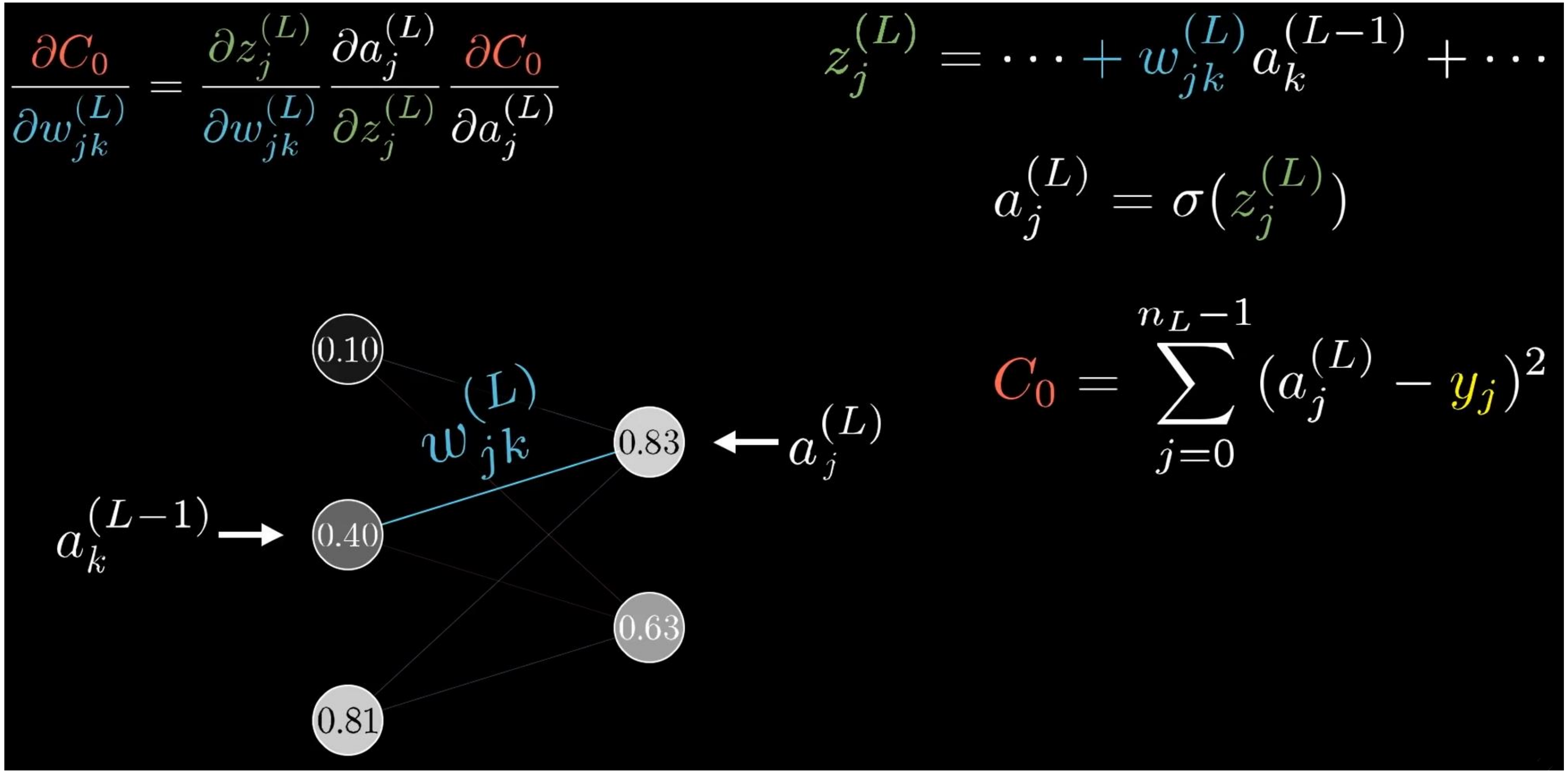


# General case: more neurons in each layer

- Need more indices and everything else is basically the same.

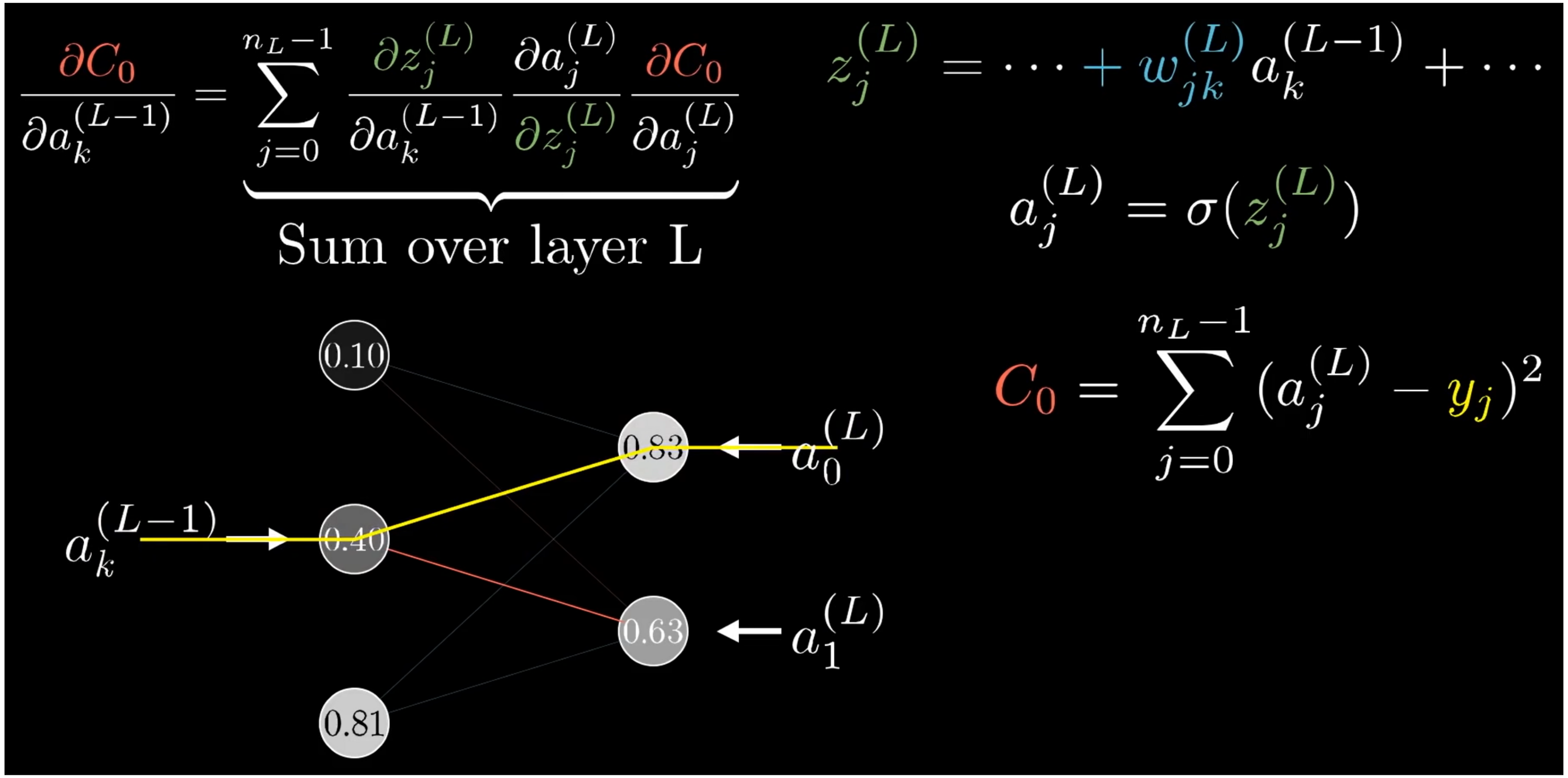


# Derivatives on weights and biases are the same



# Derivative on the activation **changes**

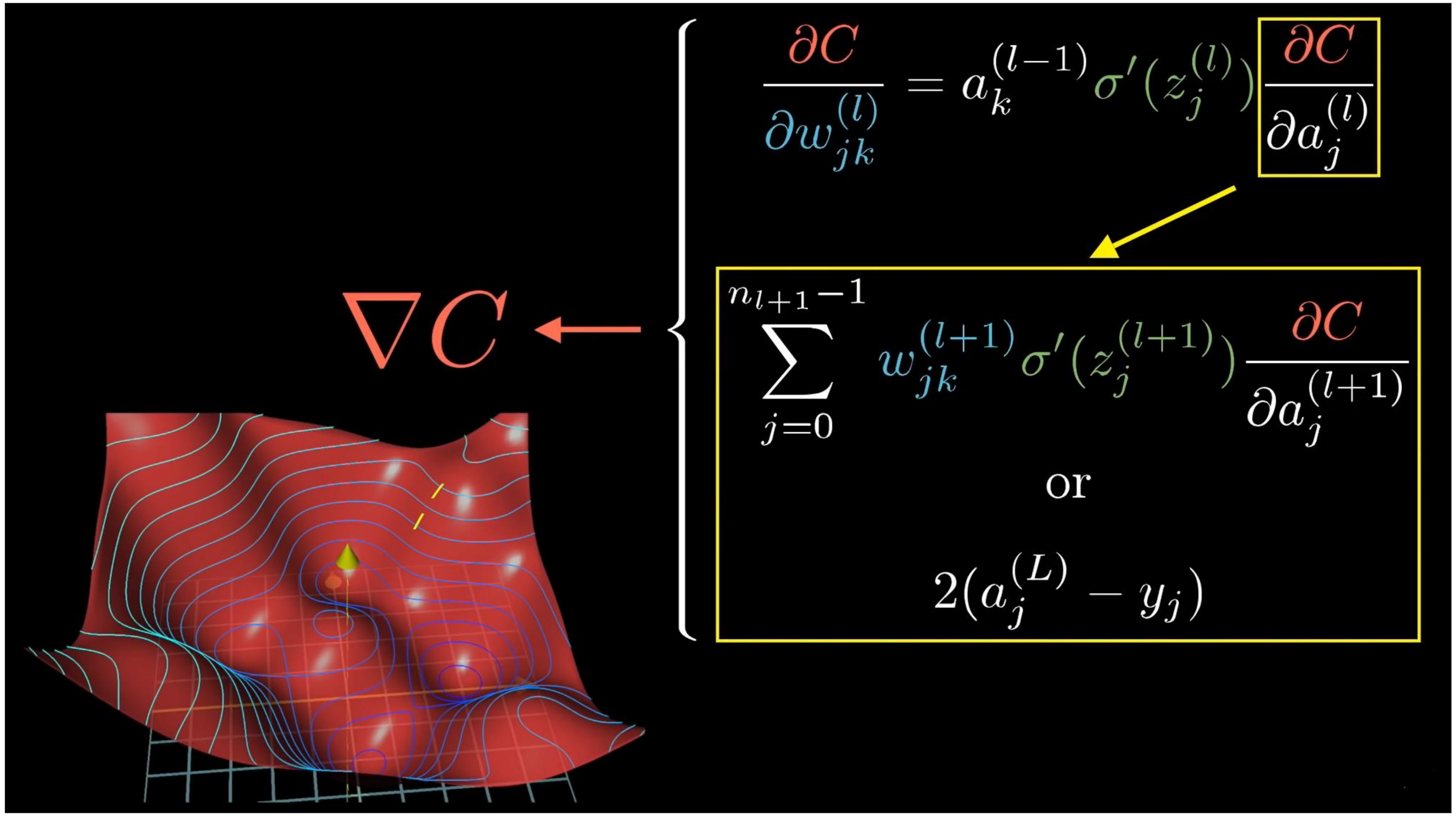
- Since the neuron ($a_k^{(L-1)}$) influences the cost function through multiple different paths (2 in this case).



# With all the gradients, we apply gradient descent

- The expression "or" means that at the last layer (which is different from other layers), we take the derivative on the cost.
- Note the typo:

  *l -> L*



# Watch these YouTube videos about neural network and backpropagation

- https://www.youtube.com/watch?v=aircAruvnKk
  - There are 4 videos introducing neural networks and backpropagation. Most of our slides are based on these videos.
- https://www.youtube.com/watch?v=IN2XmBhILt4https://www.youtube.com/watch?v=iyn2zdALii8
- https://www.youtube.com/watch?v=GKZoOHXGcLo
- A playlist:
  - https://www.youtube.com/watch?v=CqOfi41LfDw&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1