Course schedule

27.02.19 - Lecture 1. Machine Learning - an Introduction. **P**

06.03.19 - Lecture 2. Introduction to classification methods. **P**

13.03.19 - Lecture 3. Regression, Regularization, Dimensional Reduction. **P**

20.03.19 - Lecture 4. Clustering methods . **K**

27.03.19 - Lecture 5. Decision trees methods. **K**

0**3.04.19** - Lecture 6. Support Vector Machine. **K**

10.04.19 - Lecture 7. Ensemble methods 1. **K**

17.04.19 - No lecture - to be confirmed

24.04.19 - Lecture 8. Ensemble methods 2. **K**

0**8.05.19** - **Lecture 9. Neural Networks 1.** **P**

15.05.19 - Lecture 10. Neural Networks 2. **P**

22.05.19 - Lecture 11. Sequential Data models. **P**

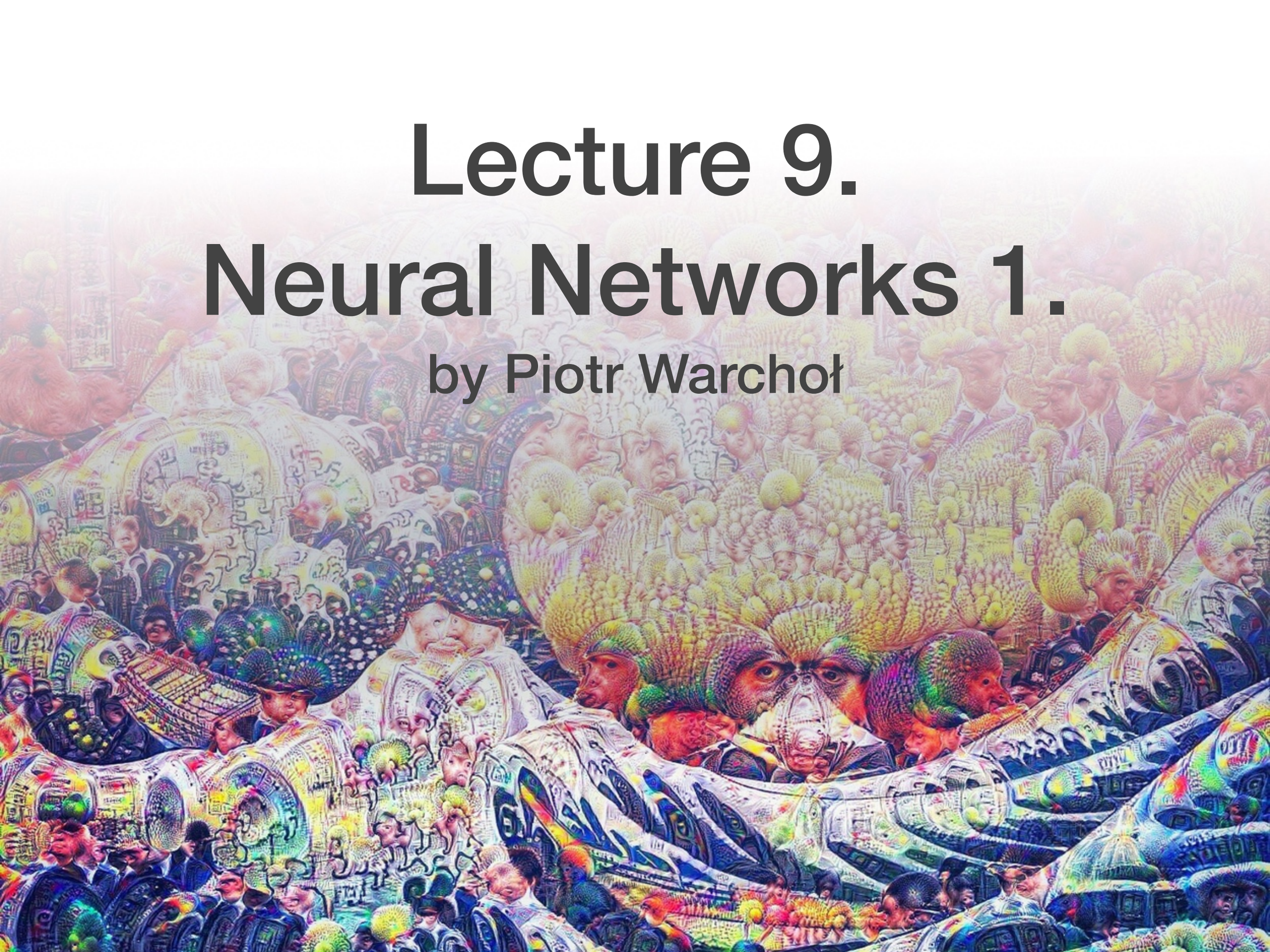
29.05.19 - Lecture 12. Recommendation systems. **P**

05.06.19 - Lecture 13. Introduction to Natural Language Processing 1. **K**

12.06.18 - Lecture 14. Introduction to Natural Language Processing 2. **K**

Lecture 9.
Neural Networks 1.

by Piotr Warchoł



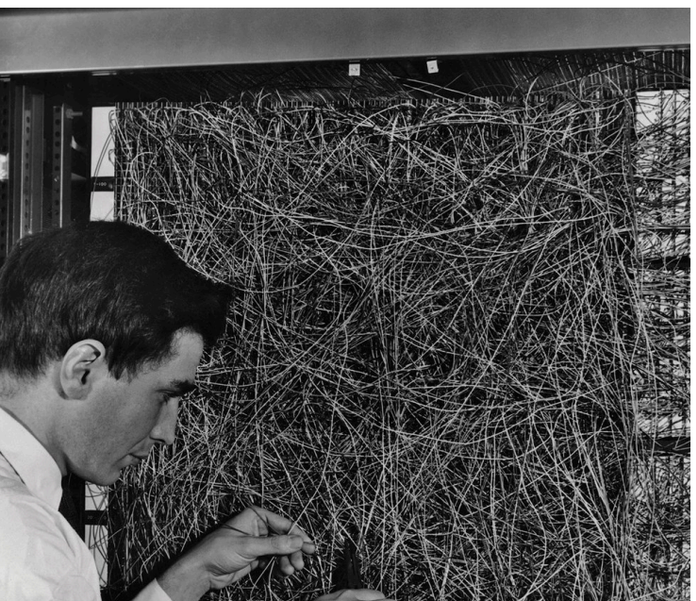
Todays lecture

- Some history
- The neuron
- Logistic regression as an artificial neuron
- The Perceptron
- Network of Perceptrons, feed-forward architecture
- Training Artificial Neural Networks
  - Stochastic Gradient Descent
  - Backpropagation
  - Cross Entropy

Some history

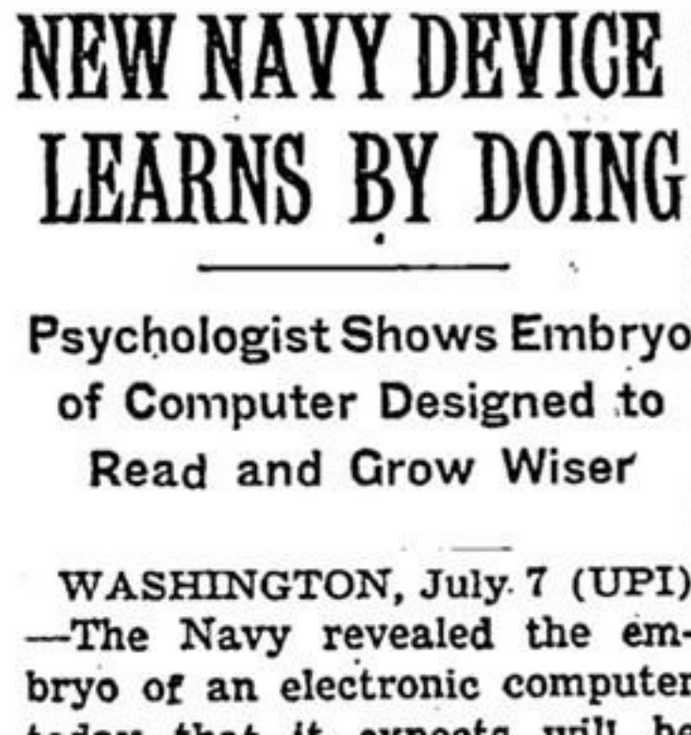
- 1940s—Hebbian theory of neuronal cell learning and the notion of a computing system that would mimic the web of neurons in the brain
- Frank Rosenblatt - the Perceptron - first simulated in IBM, 1957
- 1969 - Marvin Minsky and Seymour Papert - “Perceptrons” with a mathematical proof about the limitations of two-layer feed-forward perceptrons (incapable of processing the exclusive-or circuit) as well as unproven claims about the difficulty of training multi-layer perceptrons. **1st AI winter**

Frank Rosenblatt



Frank Rosenblatt's Mark I Perceptron at the Cornell Aeronautical Laboratory, Buffalo, New York, circa 1960

Perceptron in the NYT



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be

Some history

- Paul J. Werbose discovers backpropagation (1974 PhD thesis)
- late 80's, earlu 90's. Renewed interest - some new results. Application in character recognition, early self driving cars (Dean Pomerleau)
- not enough computing power and data. Other ("classical") methods work better on smaller data set problems. **2nd AI winter.**

Paul J. Werbose

Geoff Hinton

Jurgen Schmidhuber

Yan Lecun

Yoshua Bengio

Some history

- 2003 - Hinton approached by CIFAR (Canadian Institute for Advanced Research) to start a program on neural computations
- 2006 - G. E. Hinton and R. R. Salakhutdinov - publish “Reducing the Dimensionality of Data with Neural Networks” in Science - 6899 citations as of 19.05.18

Reducing the Dimensionality of Data with Neural Networks

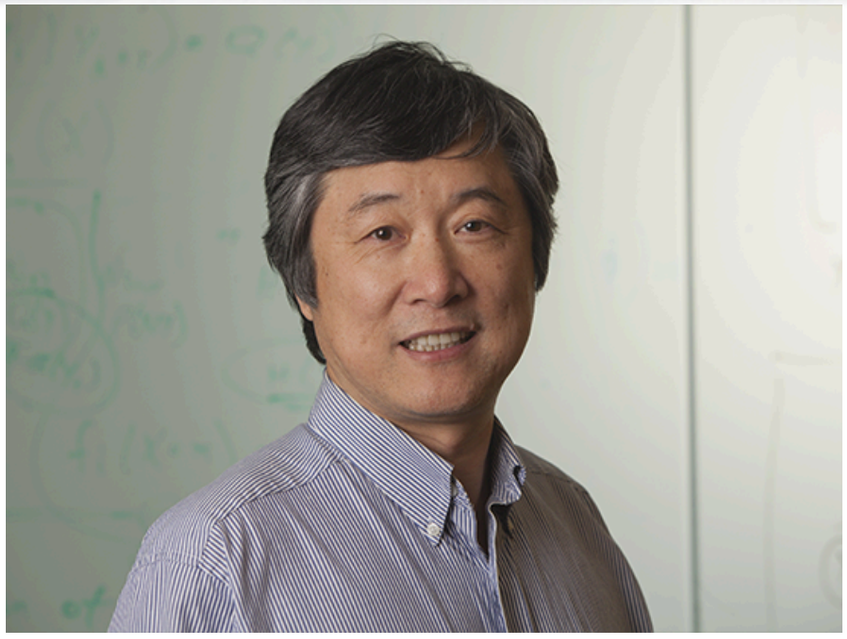
G. E. Hinton* and R. R. Salakhutdinov

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such “autoencoder” networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.

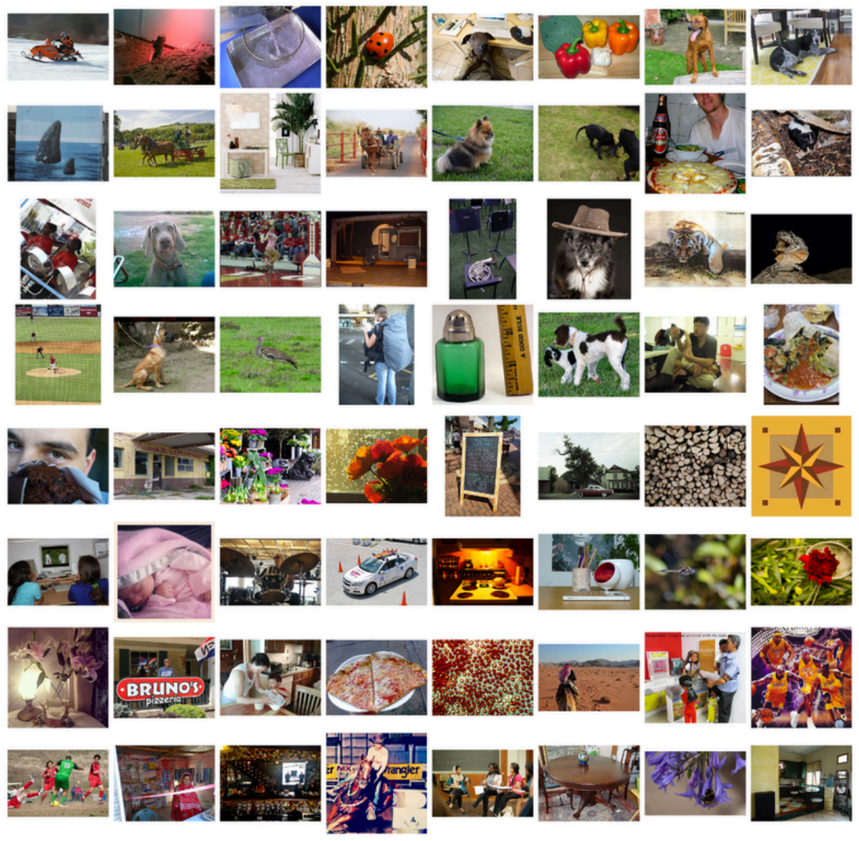
Dimensionality reduction facilitates the classification, visualization, communication, and storage of high-dimensional data. A simple and widely used method is principal components analysis (PCA), which finds the directions of greatest variance in the data set and represents each data point by its coordinates along each of these directions. We describe a nonlinear generalization of PCA that uses an adaptive, multilayer “encoder” network

Some history

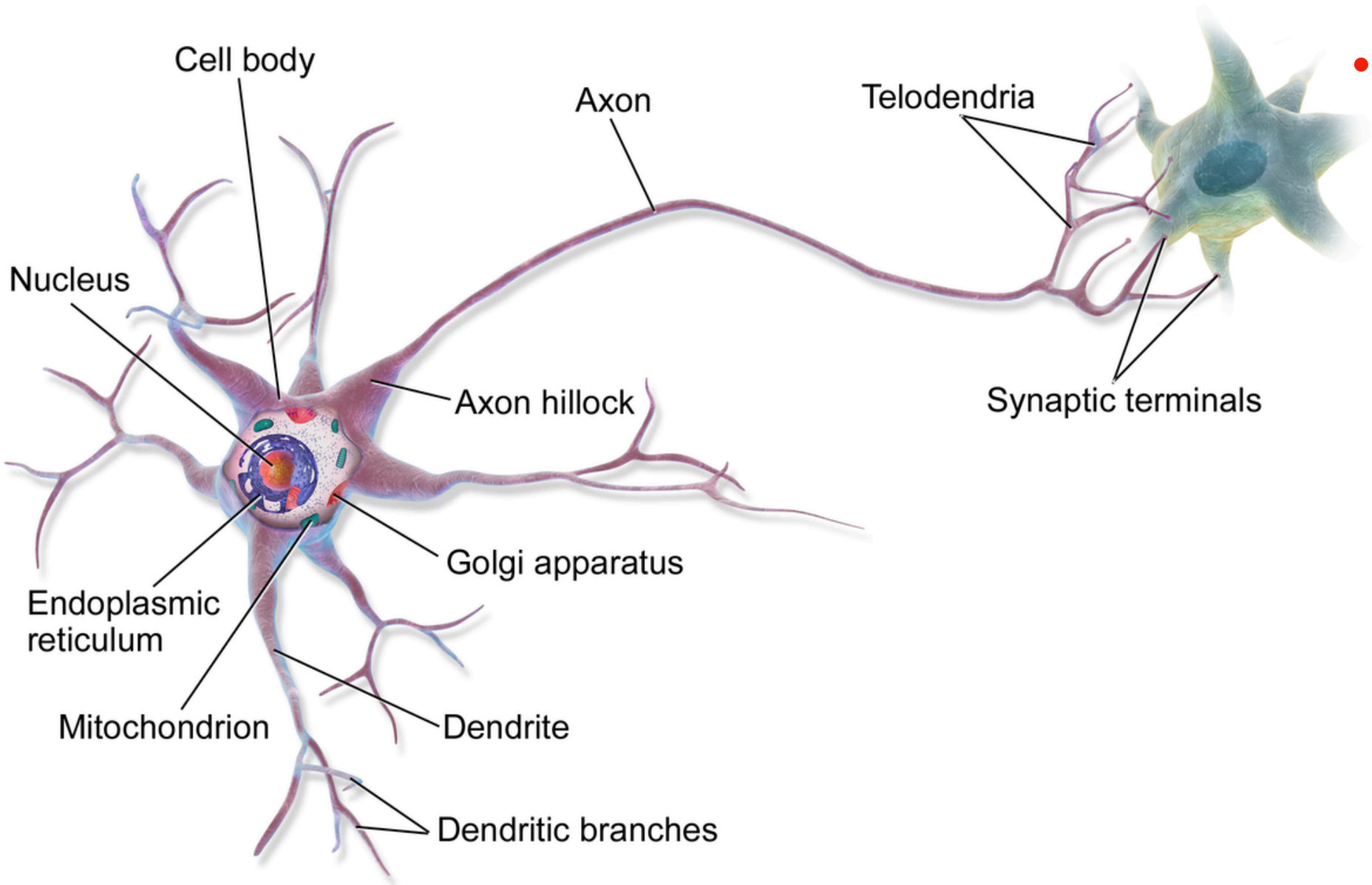
- 2009 - successful application to speech recognition in Hintons lab. Microsoft takes interest through Li Deng. (These are already deep nets)
- 2010 - huge leap in speech recognition technology at Microsoft and Google
- 2009 - 2012 deep neural nets strat to win Machine Learning competitions (mostly pattern recognition). Superhuman performance is achieved on some tasks.
- 2012 - a deep convolutional neural net achieved 16% error rate on the ImageNet challenge (drop from 25% in 2011, now about 5%)
- start of an industry-wide artificial intelligence boom



Li Deng

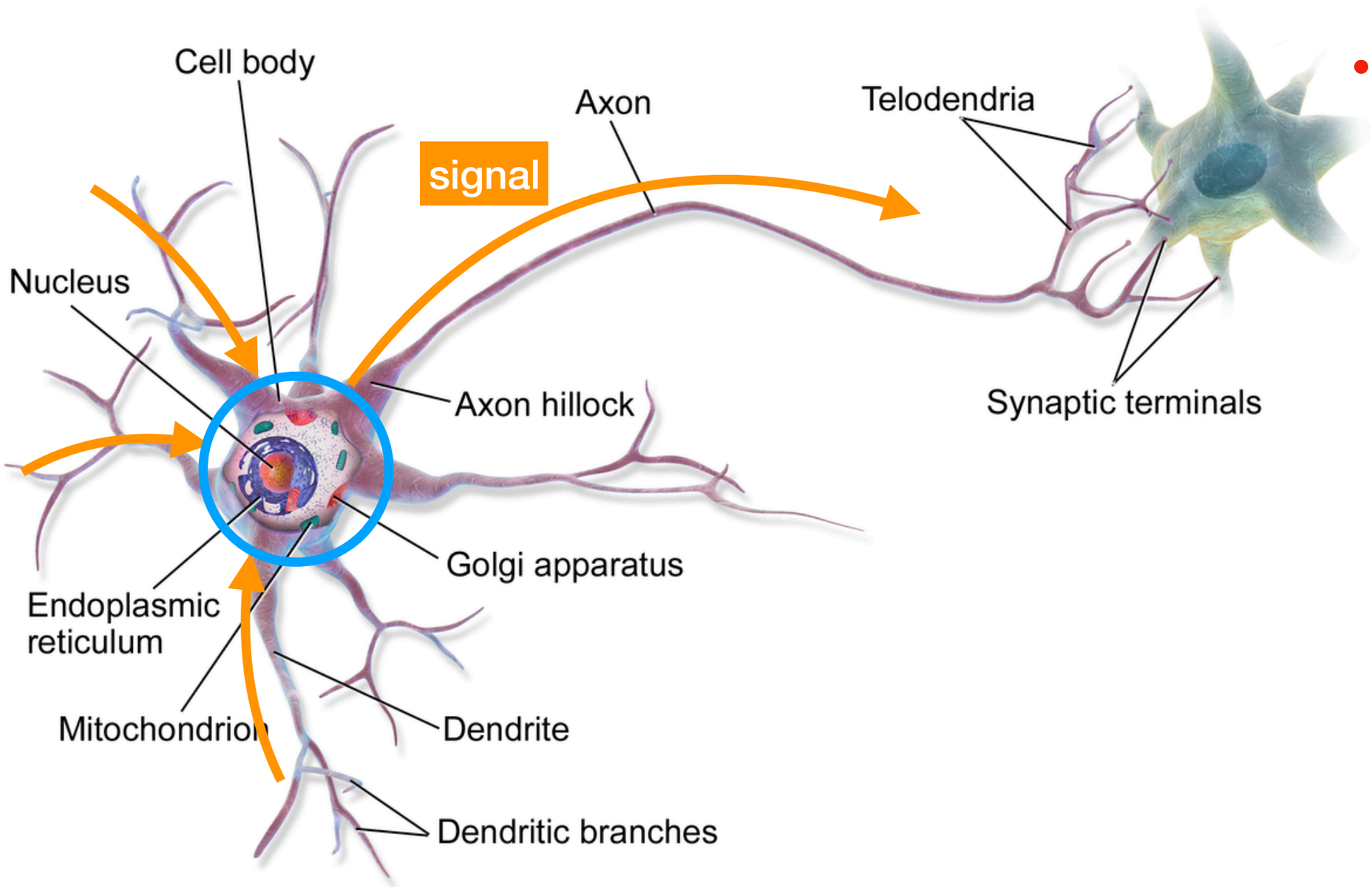


The neuron and the neural network



- neuron - an electrically excitable cell that receives, processes, and transmits information through electrical and chemical signals

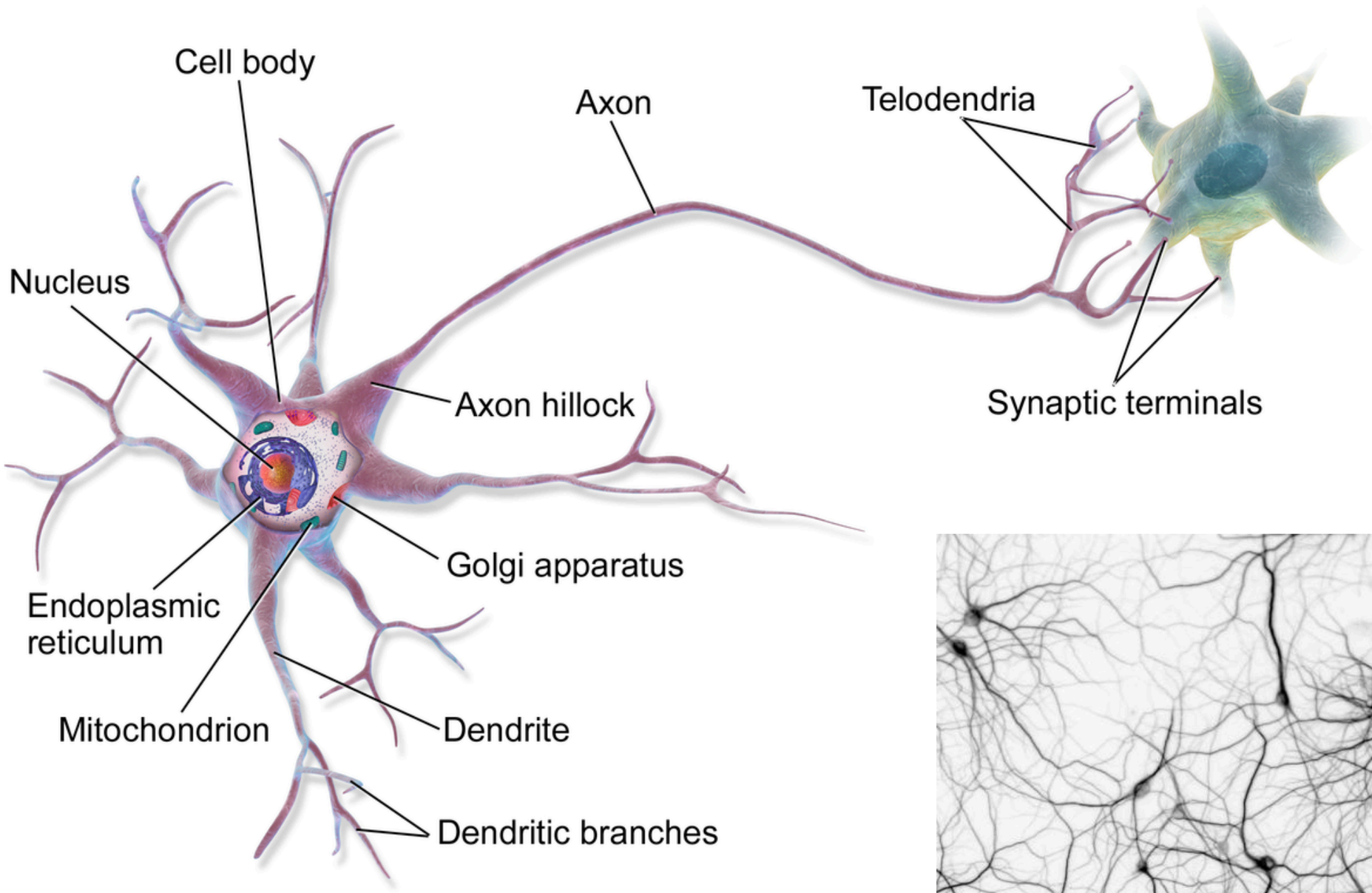
The neuron and the neural network



- neuron - an electrically excitable cell that receives, processes, and transmits information through electrical and chemical signals

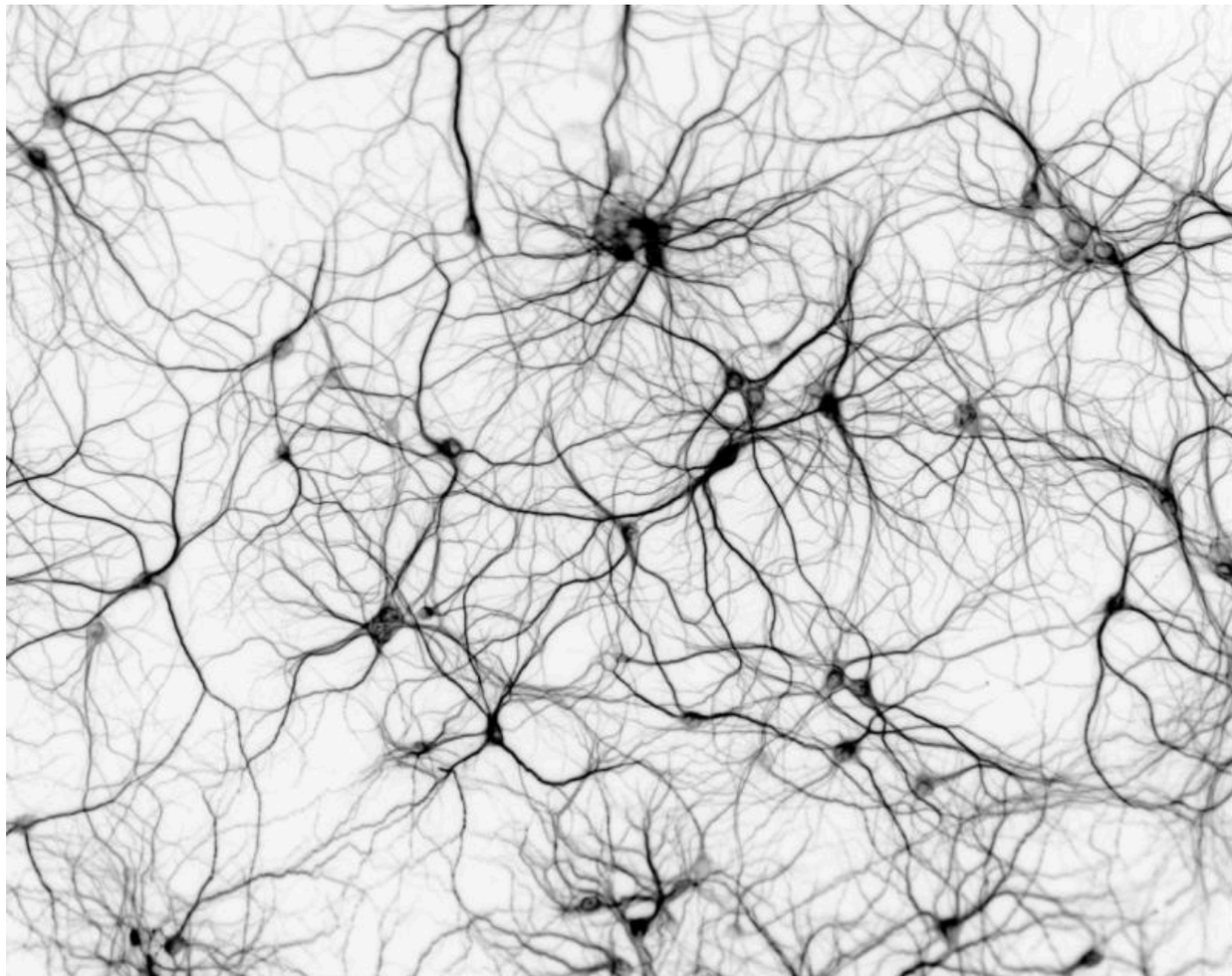
If the voltage changes by a large enough amount, an all-or-none electrochemical pulse called an action potential is generated and this change in cross-membrane potential travels rapidly along the cell's axon, and activates synaptic connections with other cells when it arrives

The neuron and the neural network

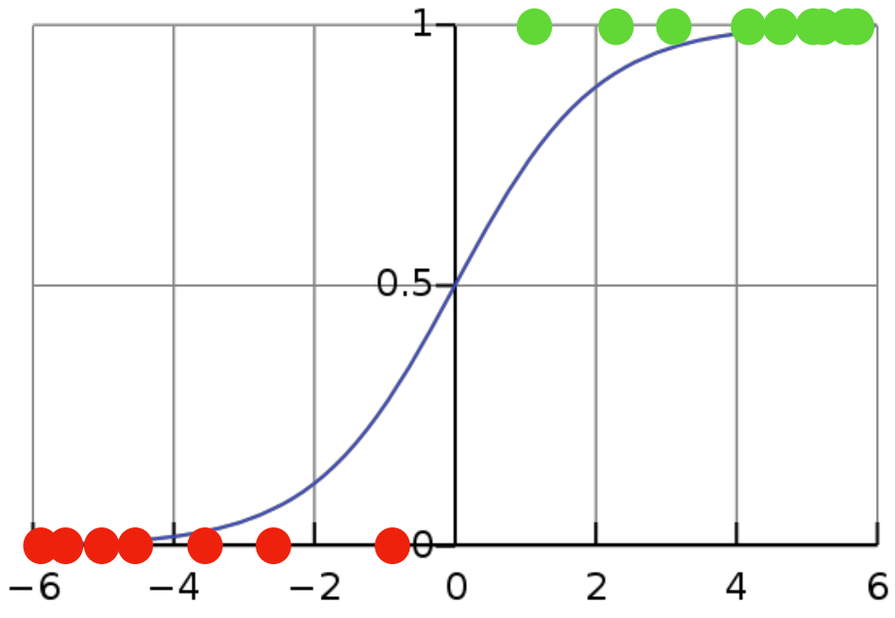


neuron - an electrically excitable cell that receives, processes, and transmits information through electrical and chemical signals

Neurons can connect to each other to form neural networks. About 85 billion neurons in the adult human brain



Back to logistic regression



Learn the probability based on categorical and continuous features

$y_j = 1$

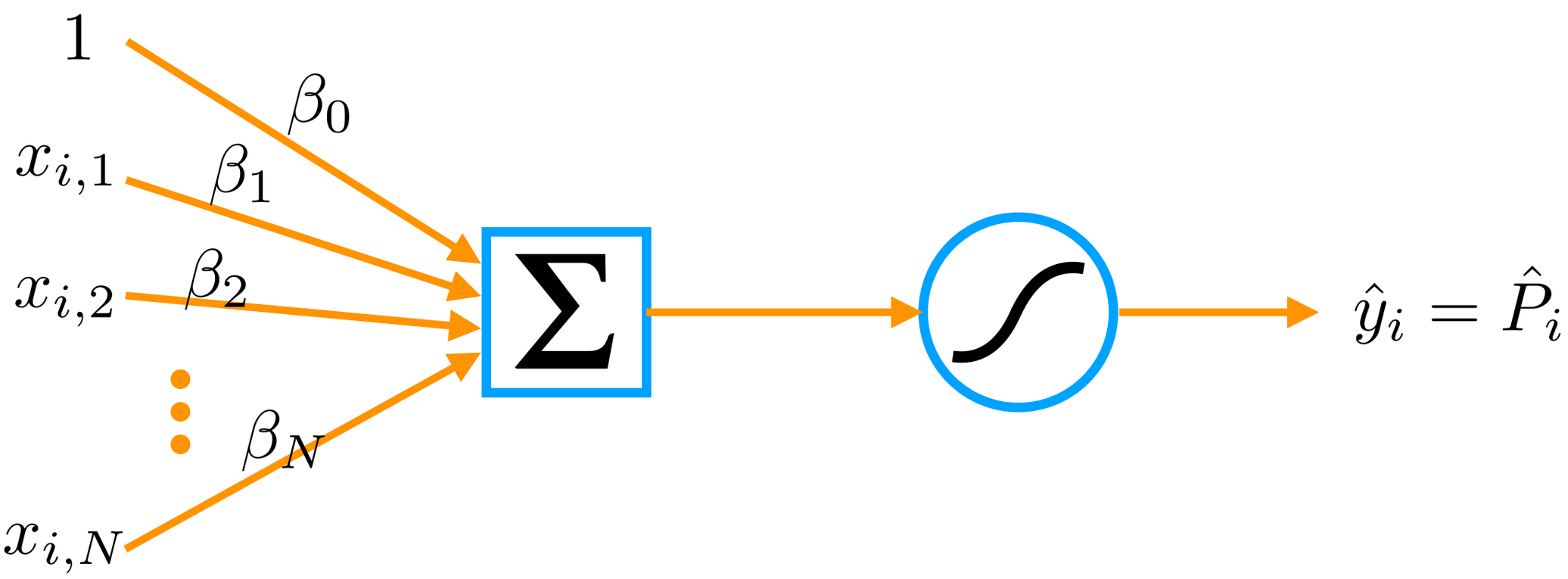
$y_i = 0$

$$\hat{P}\left(y_i = 1 | x_{i,1}, x_{i,2}, \ldots\right)$$

$$\hat{P}_i \equiv \frac{1}{1 + \exp\left(-\beta_0 - \beta_1 x_{i,1} - \beta_2 x_{i,2} - \ldots\right)}$$

Logistic regression as the neuron

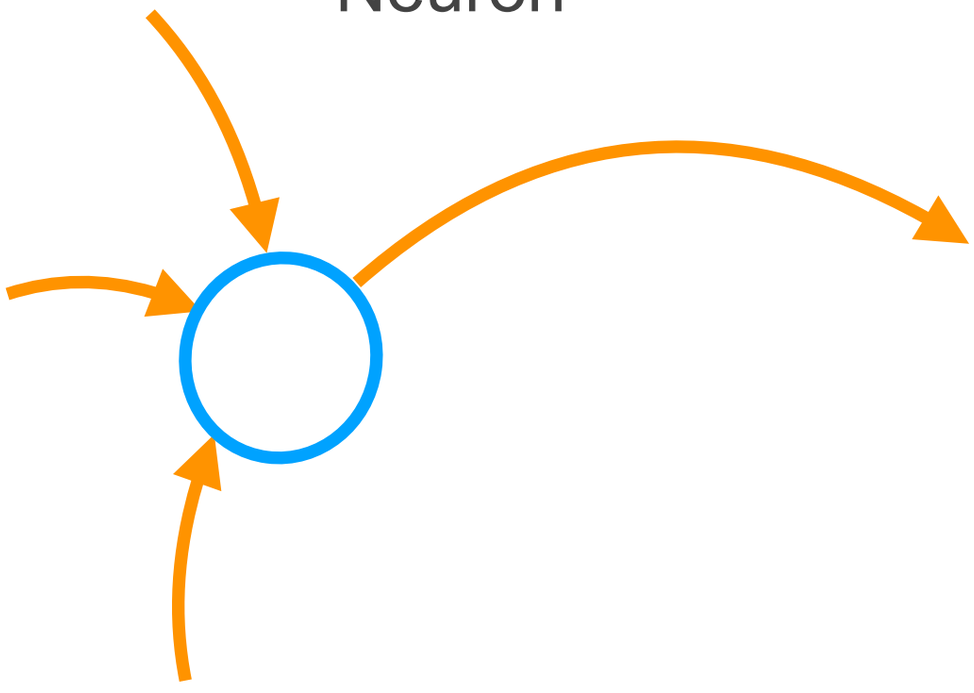
$$\hat{P}_i \equiv \frac{1}{1 + \exp\left(-\beta_0 - \beta_1 x_{i,1} - \beta_2 x_{i,2} - \ldots\right)}$$

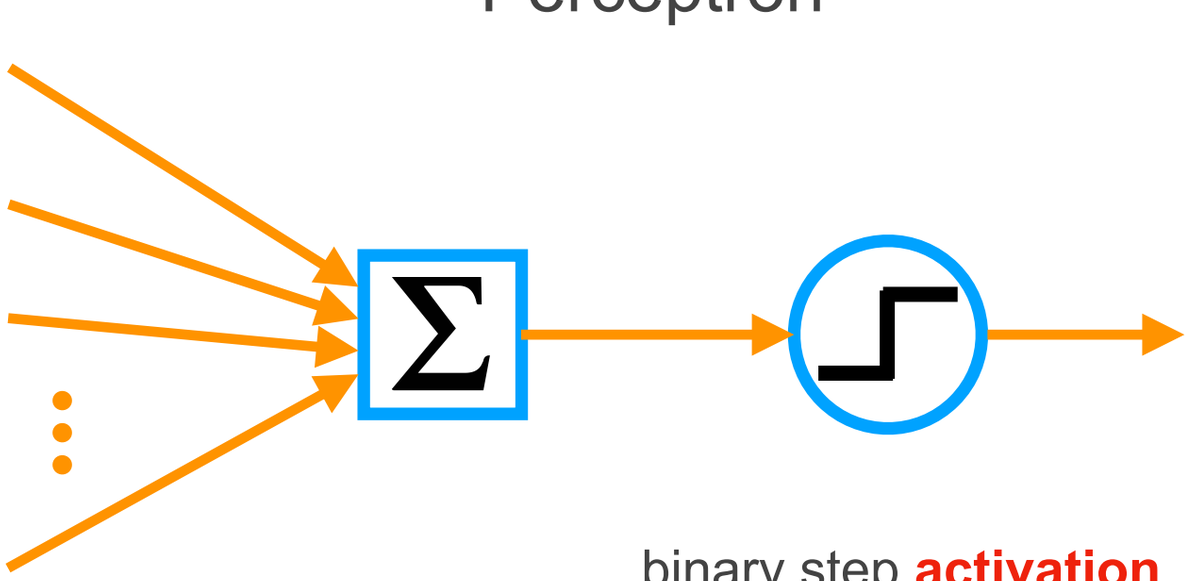


- Task: find weights that maximize likelihood given examples

$$L = \prod_i P_i^{y_i} \left(1 - P_i\right)^{1 - y_i}$$

The Perceptron

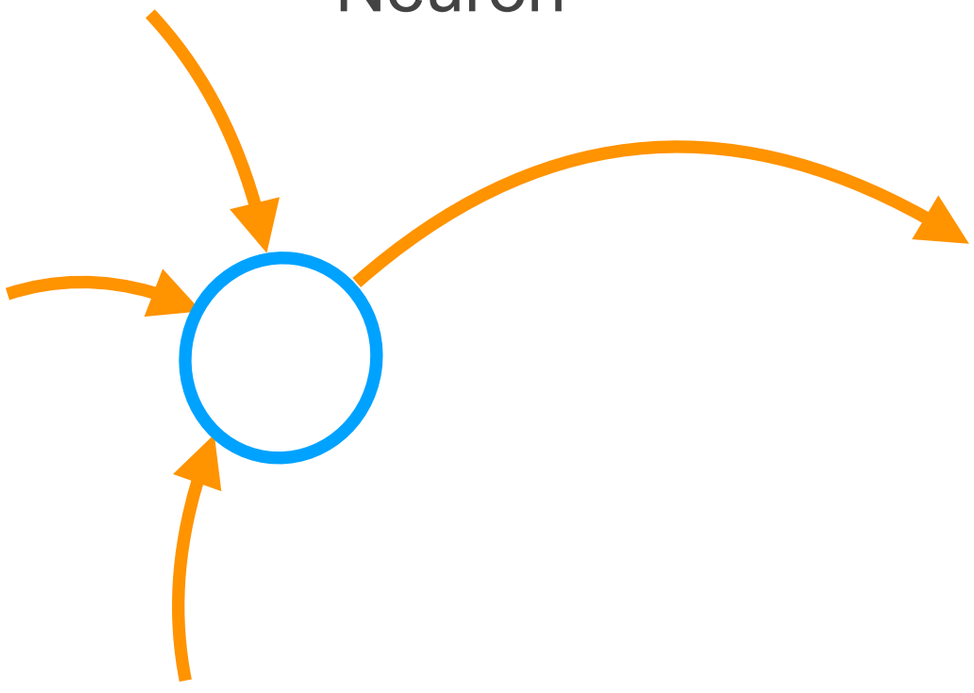




$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

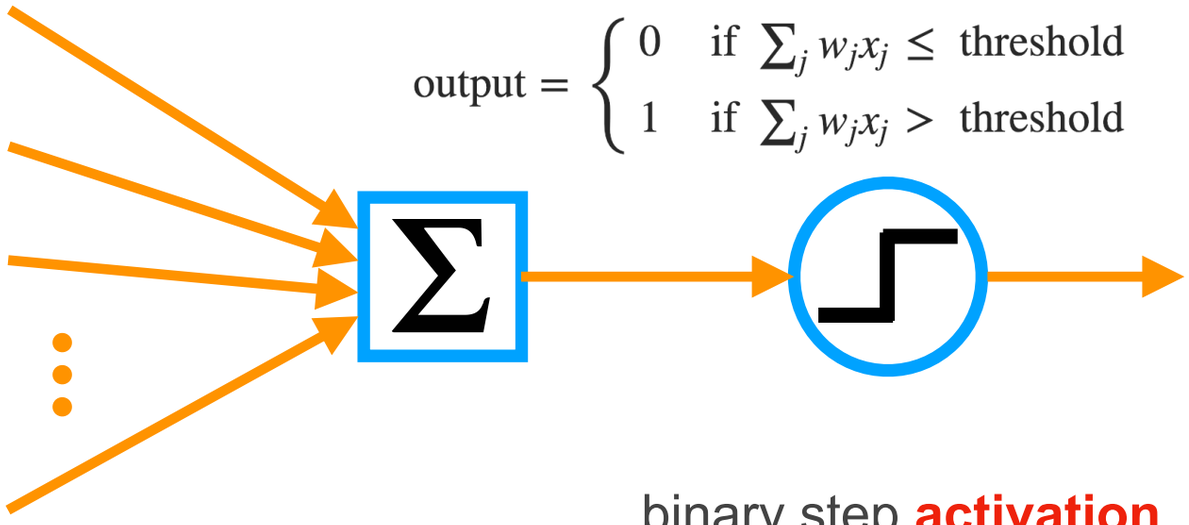
The Perceptron

Neuron



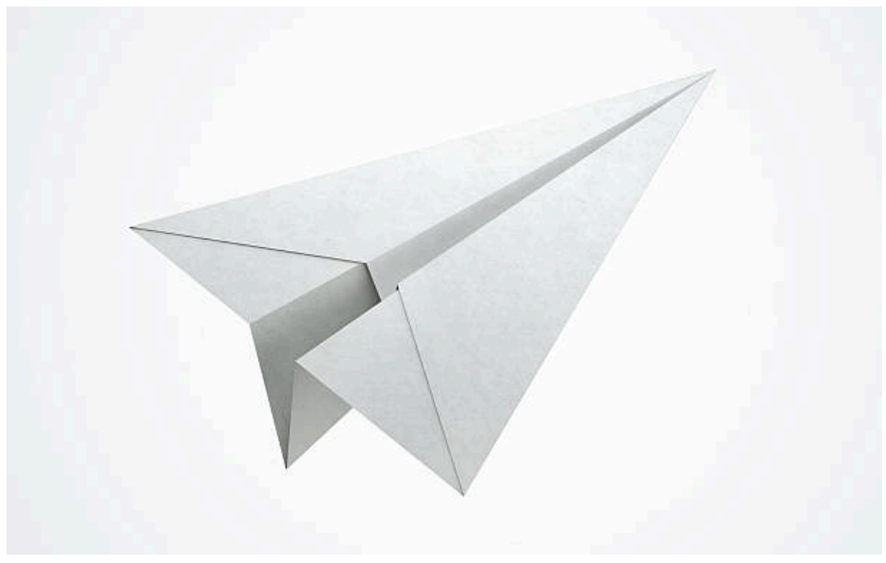
Perceptron

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$



binary step **activation function**

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$



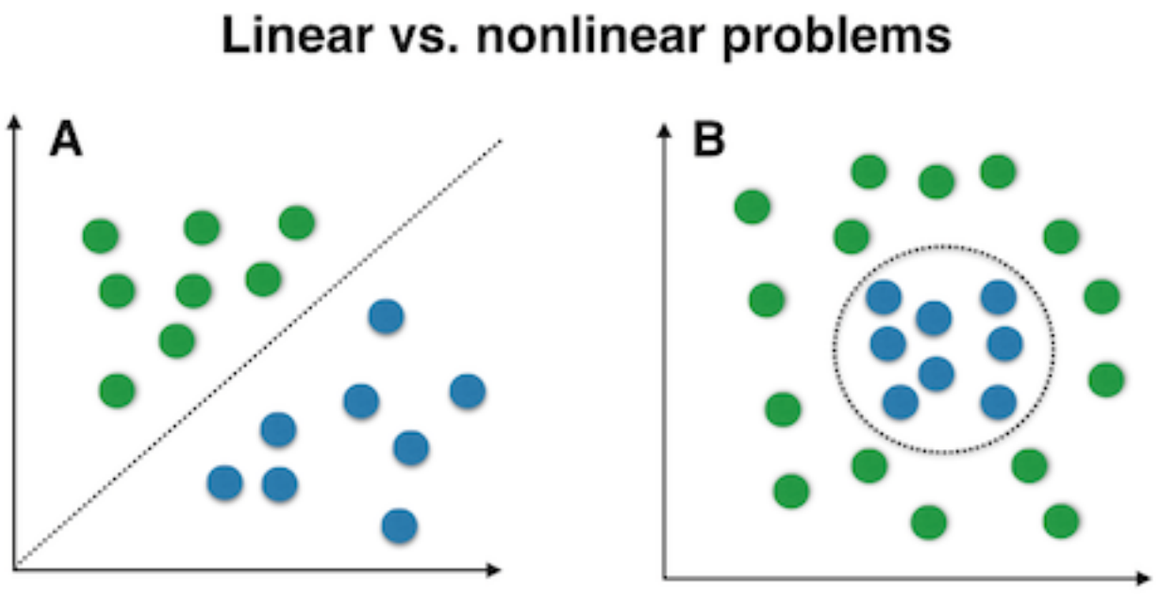
The Perceptron - training

- for each datapoint i:
  - for each j:

$$\beta_j \rightarrow \beta_j + \eta(y_i - \hat{y}_i)x_{i,j}$$

η - learning rate

y_i	$\hat{y}_i$	$(y_i - \hat{y}_i)$
0	0	0
1	1	0
1	0	1
0	1	-1



If the dataset linearly separable, then the perceptron will find the line in a finite amount of steps (not uniquely - this is done by an SVM)

The Perceptron - a problem

and

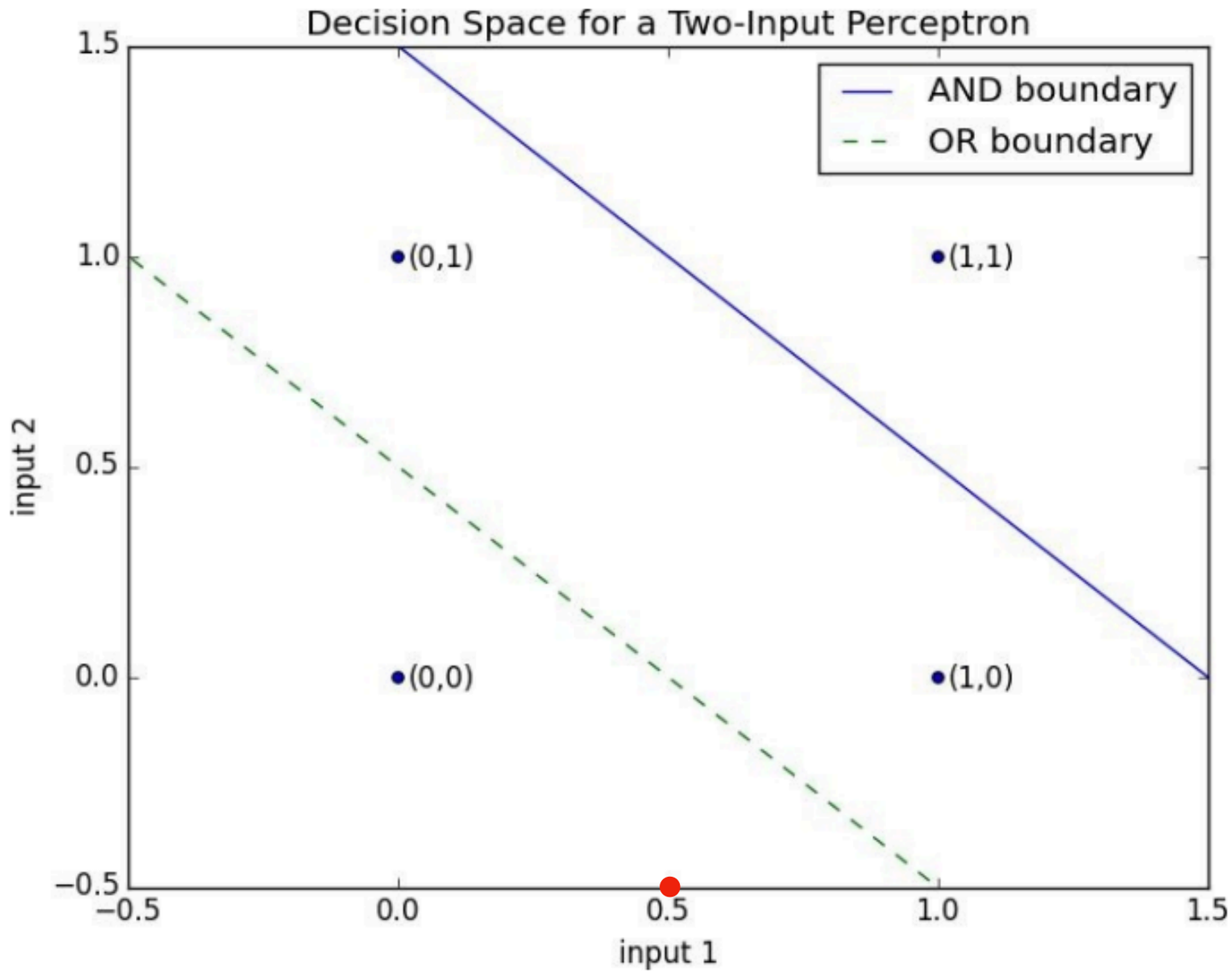
```
weights = [2, 2]
bias = -3
```

or

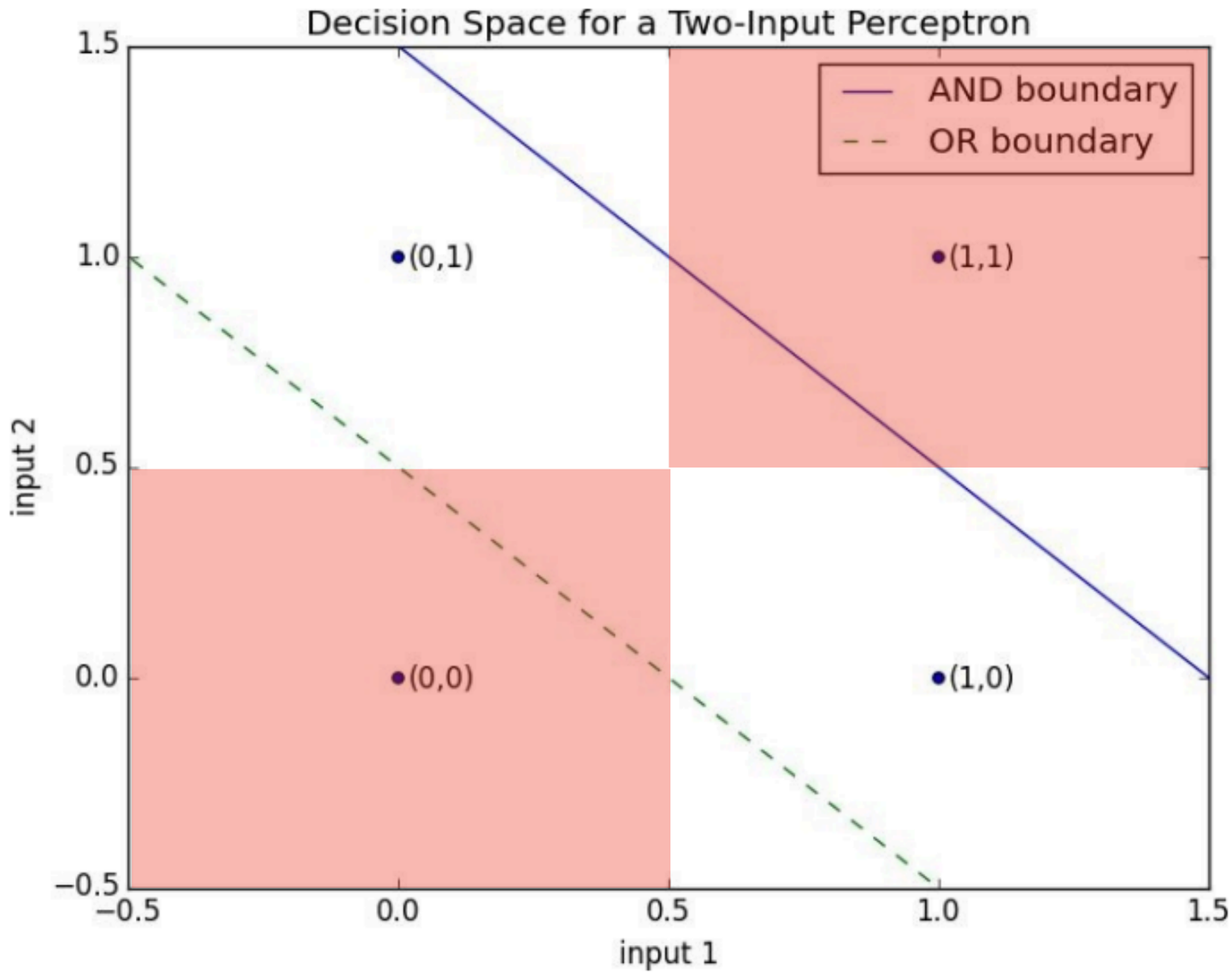
```
weights = [2, 2]
bias = -1
```

not

```
weights = [-2]
bias = 1
```

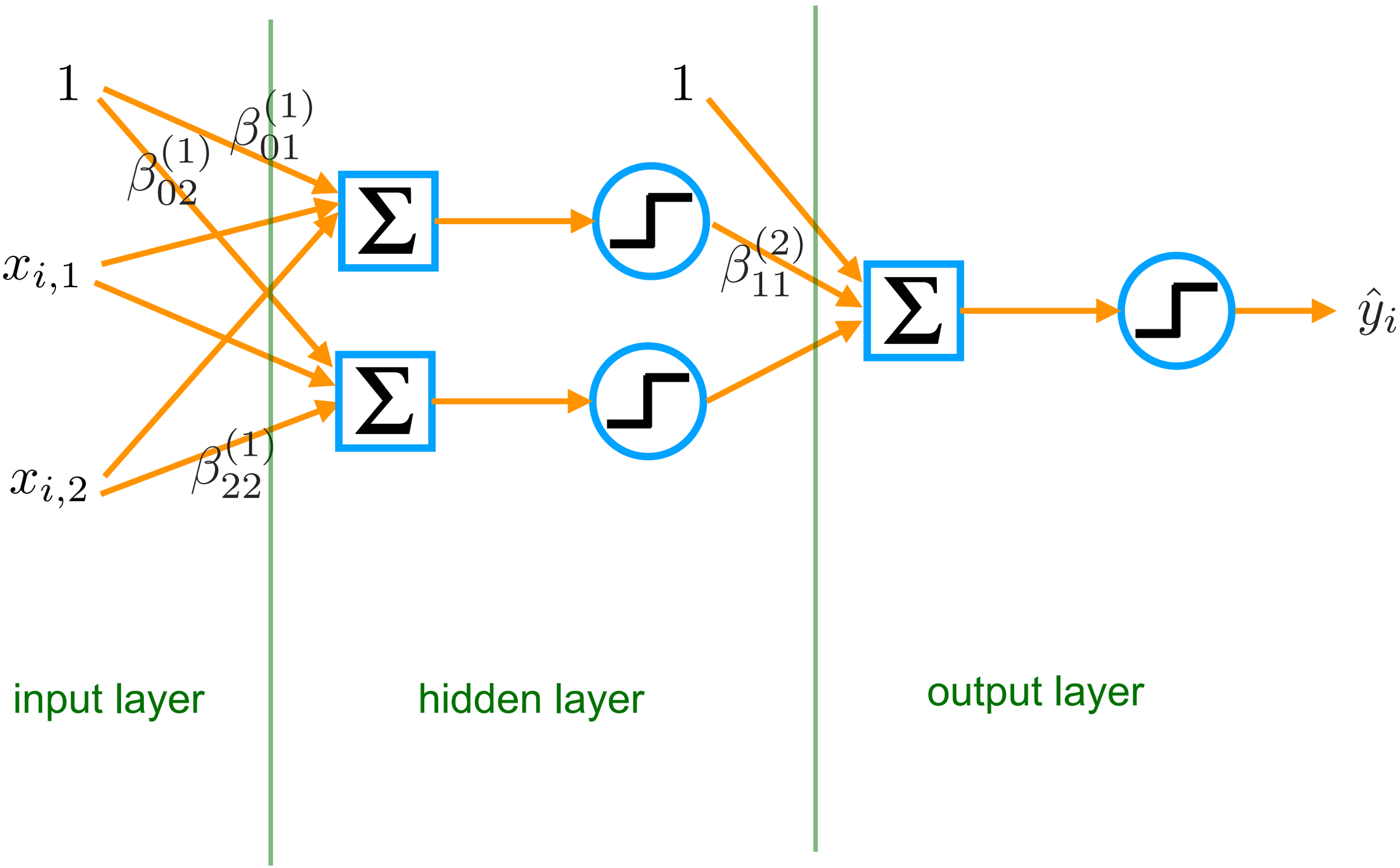


The Perceptron - a problem

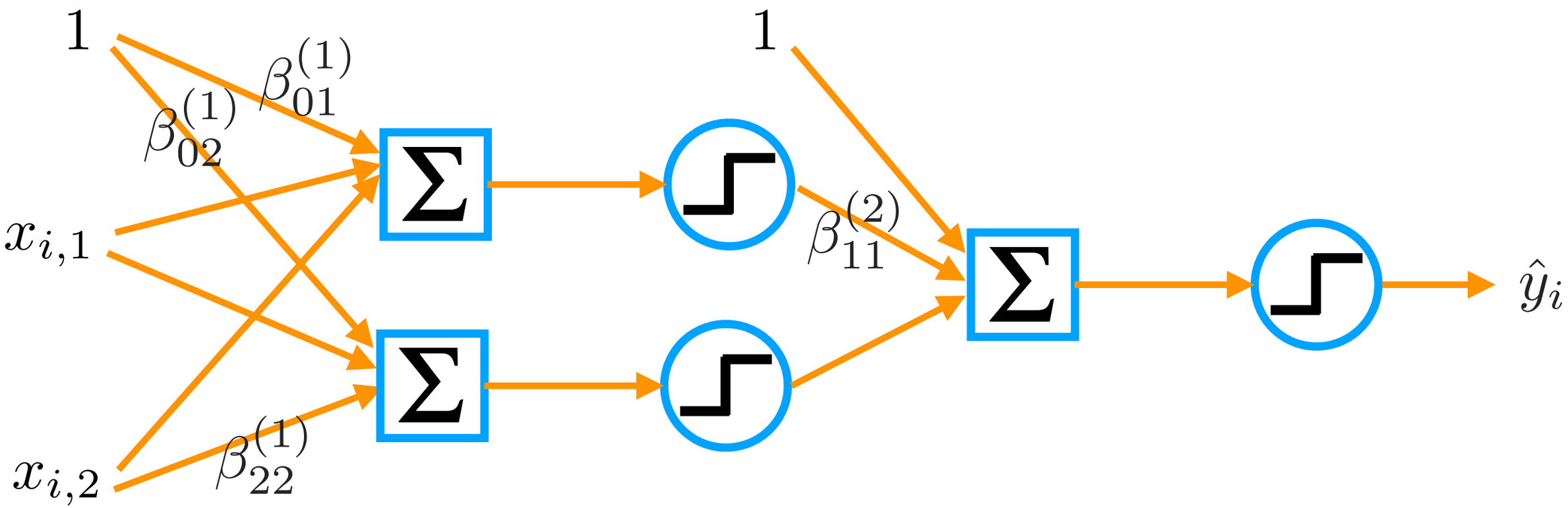


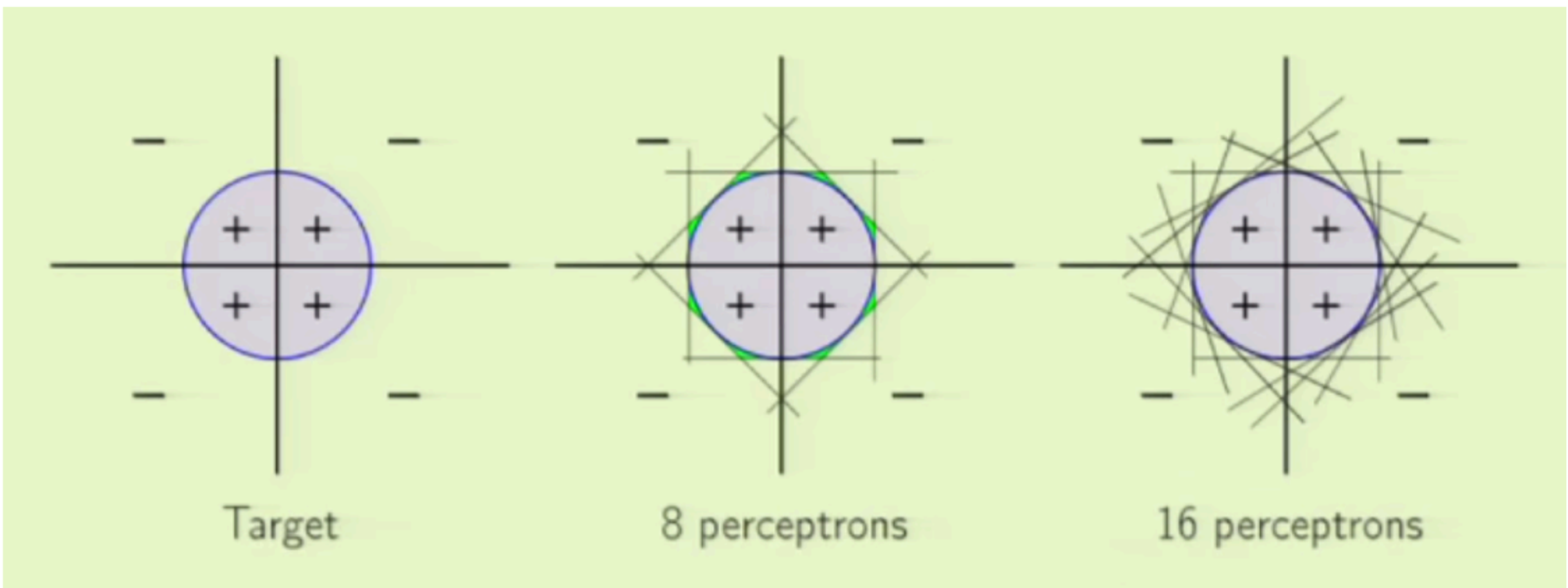
XOR not achievable with one perceptron

The Perceptron - a solution - the multilayer perceptron

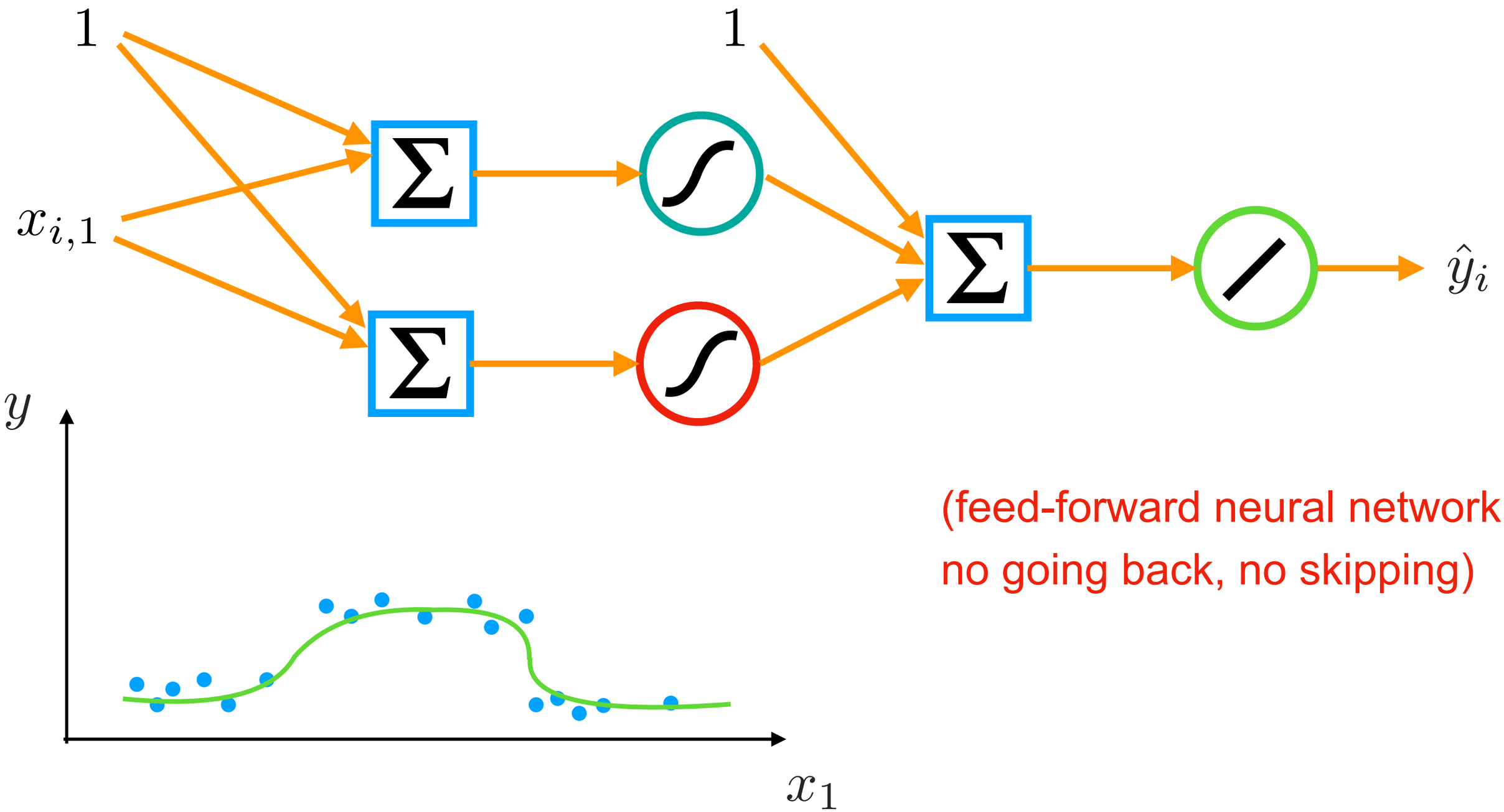


The Perceptron - a solution - the multilayer perceptron





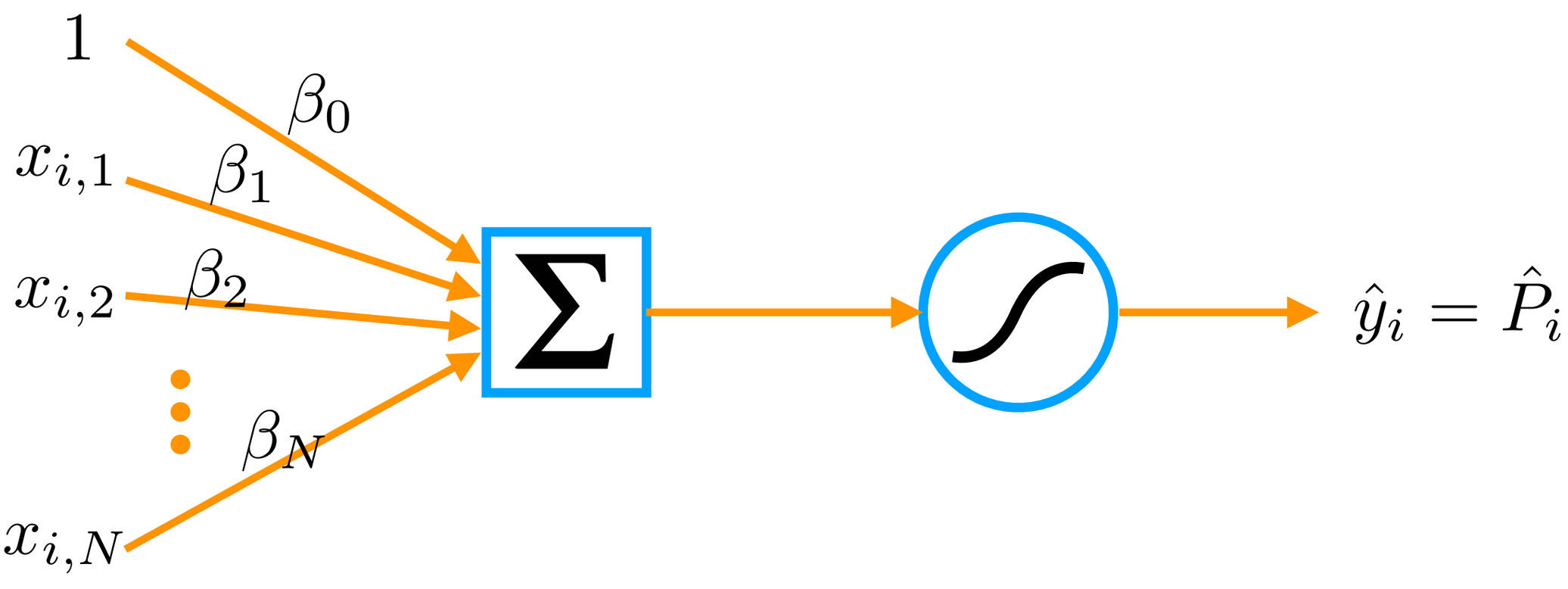
Feed-forward neural network - regression



(feed-forward neural network,
no going back, no skipping)

Logistic regression as the neuron

$$\hat{P}_i \equiv \frac{1}{1 + \exp\left(-\beta_0 - \beta_1 x_{i,1} - \beta_2 x_{i,2} - \ldots\right)}$$

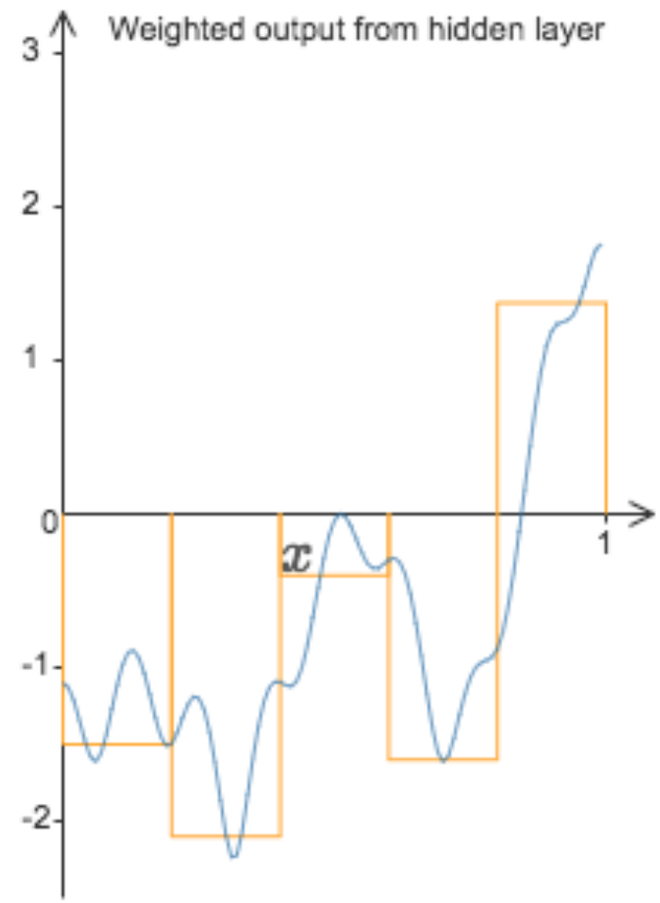


(sigmoid neurons, more control over outcome when weight change - differentiability)

Universal approximation theorem

A feed-forward network with a single hidden layer containing a finite number of neurons, can approximate continuous functions on compact subsets of $\mathbf{R}^n$, under mild assumptions on the activation function.

One of the first versions proved by George Cybenko in 1989 (for sigmoid activation functions)



nice visual proof, see:

http://neuralnetworksanddeeplearning.com/chap4.html

Training artificial neural networks -

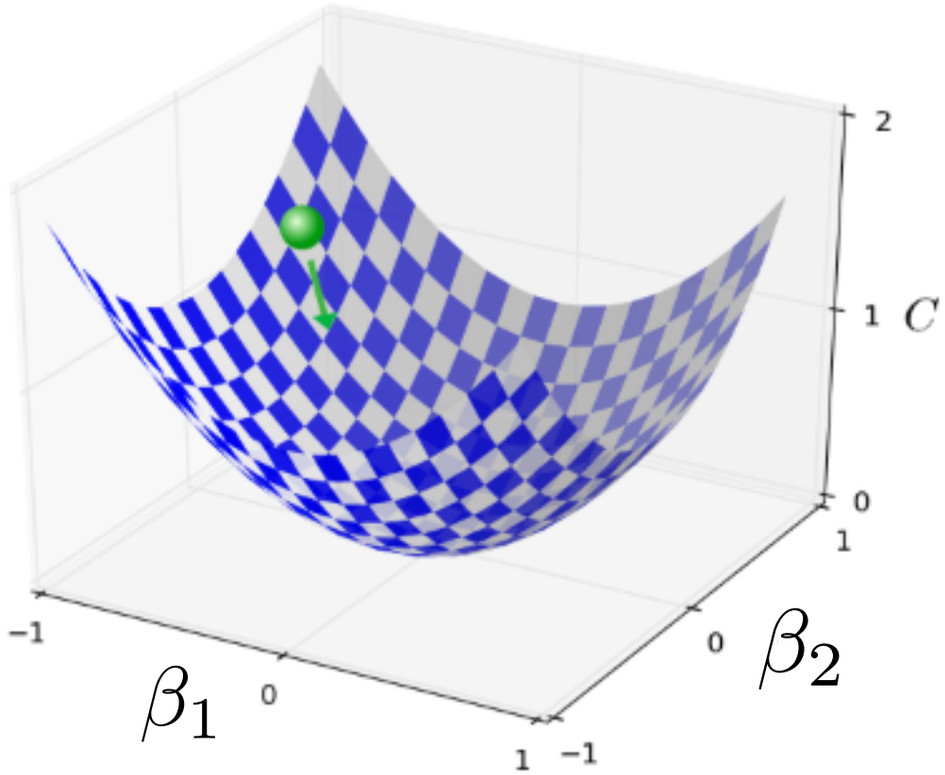
- cost/loss/objective function. Here Mean Square Error (MSE):

$$C(\beta) = \frac{1}{n}\sum_{i}^{n} ||\mathbf{y}_i - \hat{\mathbf{y}}_i(\beta)||^2$$

- we minimize it. (Others often used).

But this is very hard to do explicitly for many weights

- (batch) Gradient Descent
  - learning rate needs to be adjusted
  - can get stuck in local minimum
  - one needs to compute a derivative for every example i just for one step



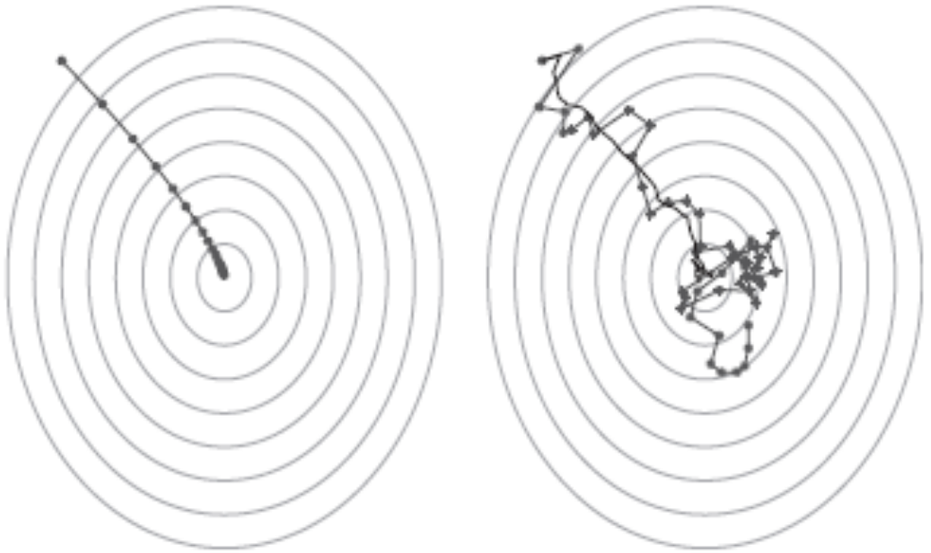
$$\nabla C = \left(\frac{\partial C}{\partial \beta_0}, \frac{\partial C}{\partial \beta_1}, \ldots, \frac{\partial C}{\partial \beta_n}\right)^T$$

$$\beta \rightarrow \beta' = \beta - \eta \nabla C$$

Training artificial neural networks -

- Gradient Descent
  - learning rate needs to be adjusted
  - can get stuck in local minimum
  - one needs to compute a derivative for every example i
- Solution: Stochastic Gradient Descent
  - instead of using all the examples, we pick a random sample (a mini-batch)

$$C(\beta) = \frac{1}{n}\sum_{i}^{n} ||\mathbf{y}_i - \hat{\mathbf{y}}_i(\beta)||^2$$



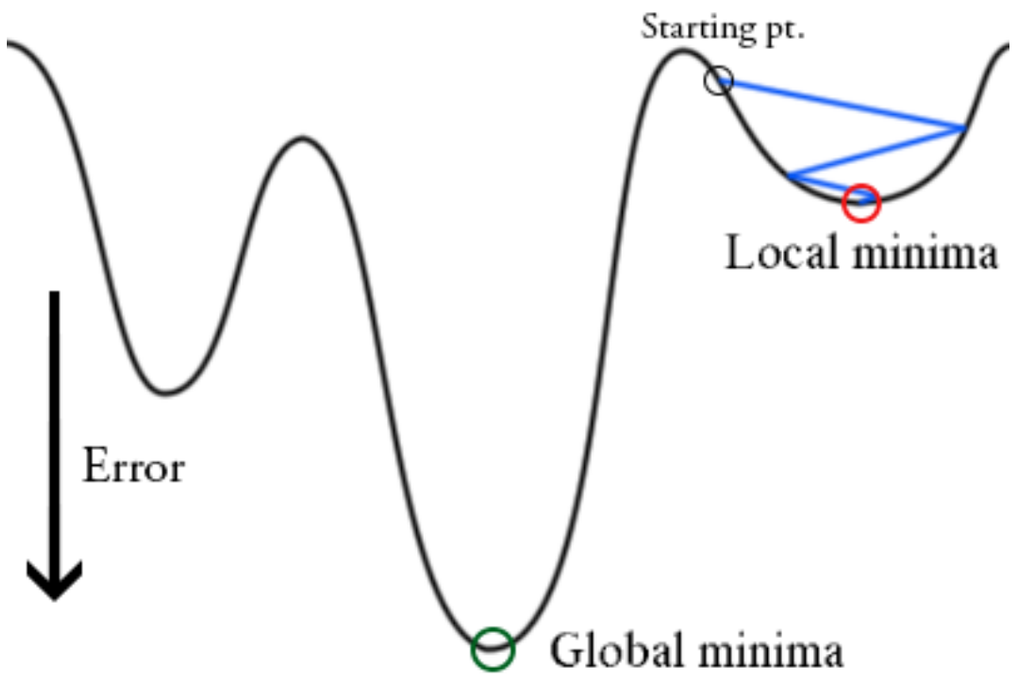
$$C_m(\beta) = \frac{1}{m}\sum_{i}^{m} ||\mathbf{y}_i - \hat{\mathbf{y}}_i(\beta)||^2$$

here, i counts the examples in a random sample

$$\beta \rightarrow \beta' = \beta - \eta \nabla C_m$$

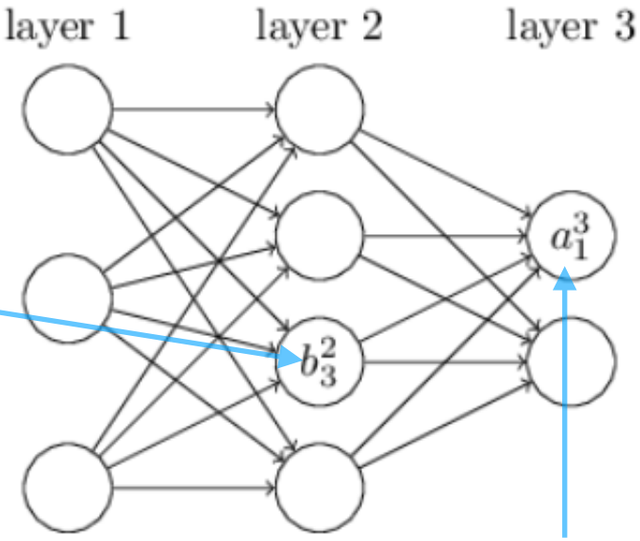
on average: $\nabla C \approx \nabla C_m$

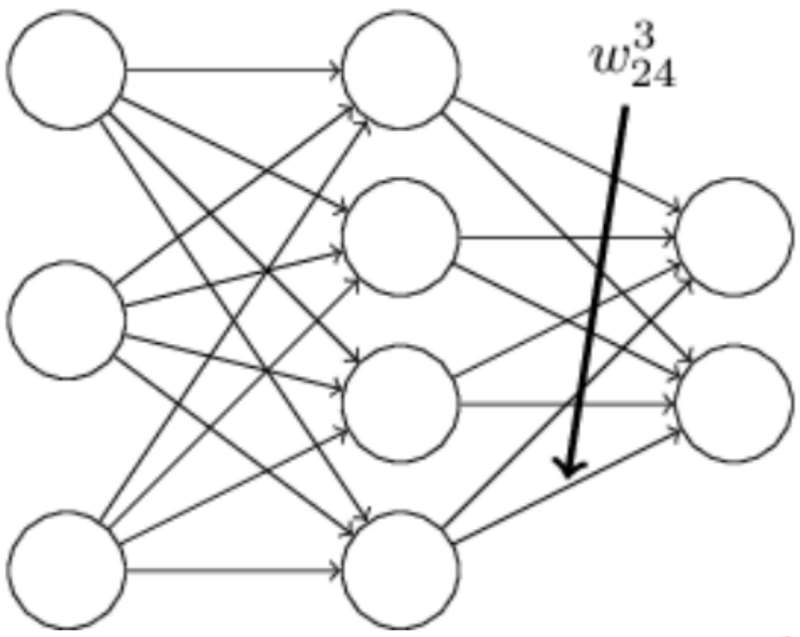
but there is a stochastic addition to that



Training artificial neural networks - backpropagation

- How to efficiently perform stochastic gradient descent for a large, multilayer network? There is to many derivatives to calculate.
- Here comes the backpropagation algorithm.
- First lets switch to a more convenient notation:





w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

Training artificial neural networks - backpropagation

- vectorization

$$\sigma(v)_j = \sigma(v_j)$$

$$f(x) = x^2 \qquad f\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}f(2)\\f(3)\end{bmatrix} = \begin{bmatrix}4\\9\end{bmatrix}$$

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

- Hadamard product

we use $s \odot t$ to denote the *elementwise* product of the two vectors. Thus the components of $s \odot t$ are just $(s \odot t)_j = s_j t_j$. As an example,

$$\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1 * 3\\2 * 4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix}.$$

- weighted input

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

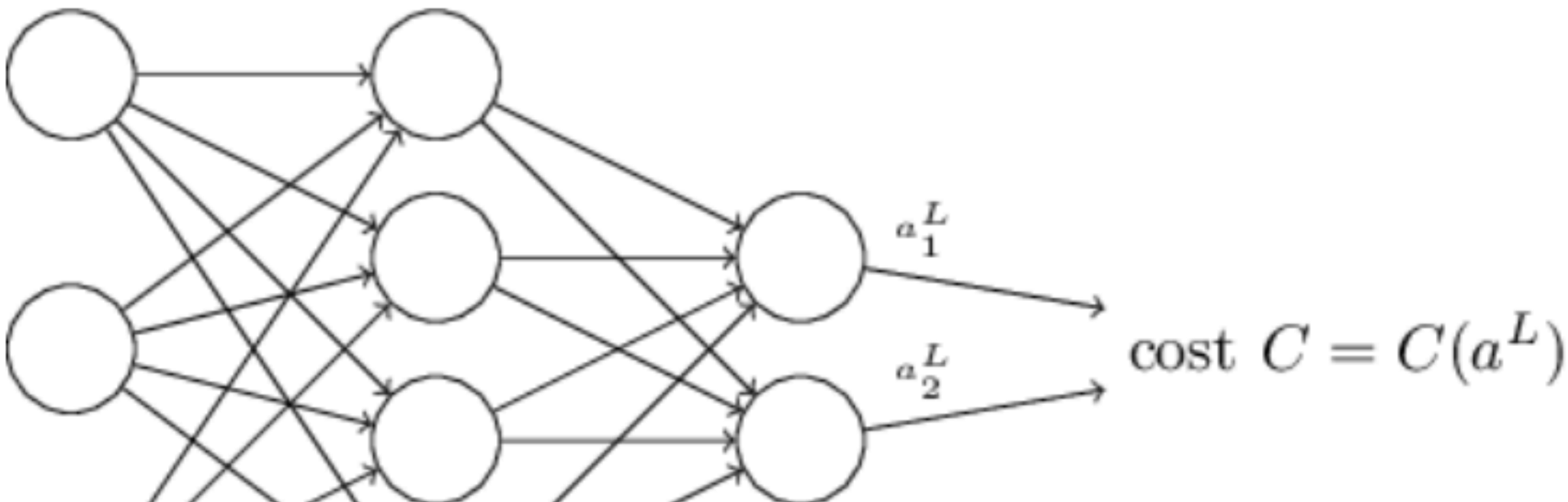
$$a^l = \sigma(z^l)$$

Training artificial neural networks - backpropagation

- assumptions on the cost function (think about MSE, $C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$ needed for others)

the cost function can be written as an average $C = \frac{1}{n} \sum_x C_x$ over cost functions C_x for individual training examples, x. This is the case for the quadratic cost function, where the cost for a single training example is $C_x = \frac{1}{2} \|y - a^L\|^2$.

The second assumption we make about the cost is that it can be written as a function of the outputs from the neural network:



Training artificial neural networks - backpropagation

we have for each l: $z^l \equiv w^l a^{l-1} + b^l$, $a^l = \sigma(z^l)$

we want: $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$?

define: $\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}$ for l=L: $\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j)$ (chain rule)

for example: $C = \frac{1}{2}\sum_j (y_j - a^L_j)^2$, and so $\partial C/\partial a^L_j = (a^L_j - y_j)$

in matrix form: $\delta^L = \nabla_a C \odot \sigma'(z^L)$ with $\nabla_a C = (a^L - y)$

Thus: $\delta^L = (a^L - y) \odot \sigma'(z^L)$ easily computable.

Training artificial neural networks - backpropagation

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

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

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

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

$$a^l = \sigma(z^l)$$

$\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$?

$$z_k^{l+1} = \sum_i w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_i w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$ in matrix form: $$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Training artificial neural networks - backpropagation

The differential, by definition (as w independent of b):

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

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

$$a^l = \sigma(z^l)$$

$\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$?

All together, the backpropagation equations:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Training artificial neural networks - backpropagation

The algorithm:

1. **Input** x**:** Set the corresponding activation a^1 for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error** δ^L**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Training artificial neural networks - backpropagation + SGD

1. **Input a set of training examples**
2. **For each training example** x**:** Set the corresponding input activation $a^{x,1}$, and perform the following steps:
   - **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.
   - **Output error** $\delta^{x,L}$**:** Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
   - **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.
3. **Gradient descent:** For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Loop in mini-batches and loop in epochs of training

also, look at https://playground.tensorflow.org/