Foundations of Machine Learning
CentraleSupélec — Fall 2017

10. Artificial Neural Networks

Chloé-Agathe Azencott
Centre for Computational Biology, Mines ParisTech
chloe-agathe.azencott@mines-paristech.fr

Inserm
Institut national de la santé et de la recherche médicale

institut Curie

MINES ParisTech

Learning objectives

- Draw a **perceptron** and write out its **decision function.**
- Implement the **learning algorithm** for a perceptron.
- Write out the **decision function** and **weight updates** for any **multiple layer perceptron.**
- Design and train a **multiple layer perceptron.**

The human brain

- Networks of **processing units** (neurons) with **connections** (synapses) between them
- Large number of neurons: 10^{10}
- Large connectitivity: 10^4
- Parallel processing
- Distributed computation/memory
- Robust to noise, failures

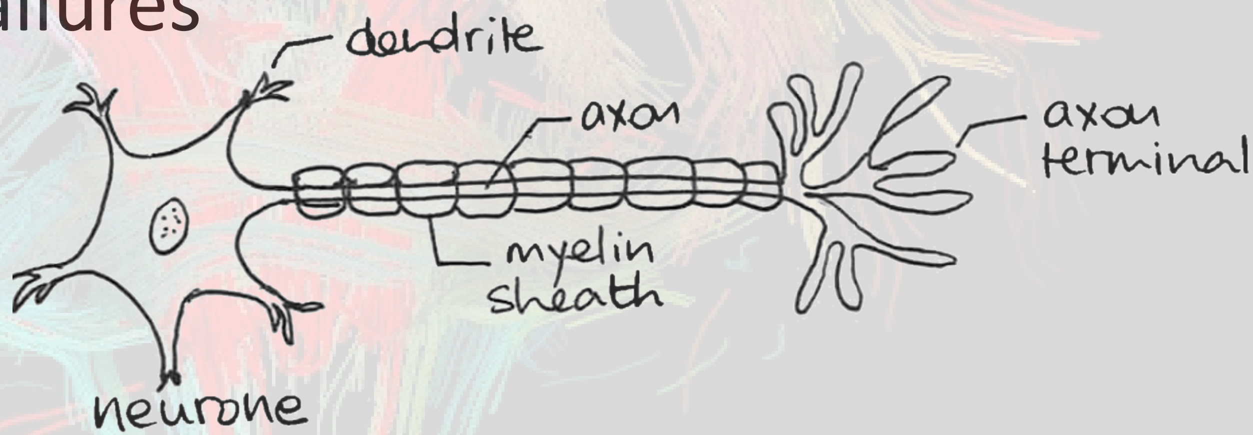
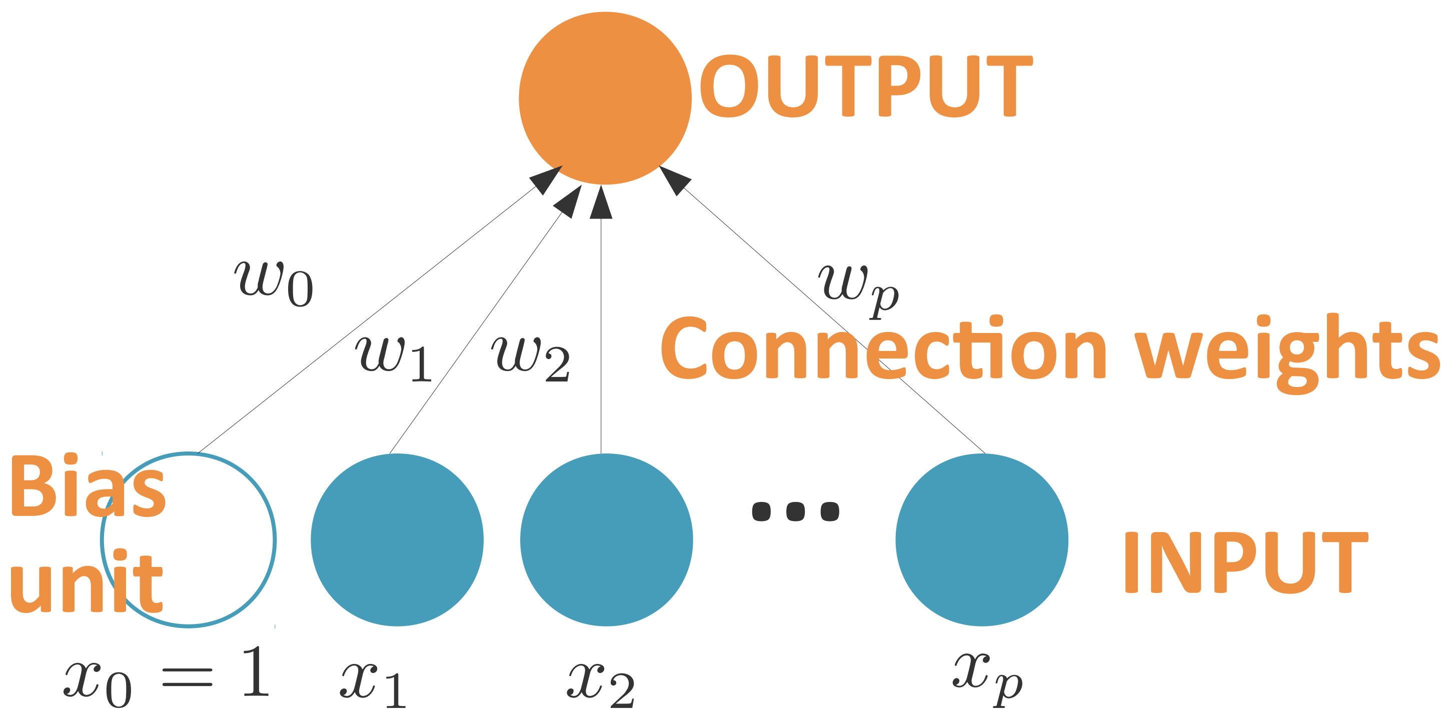


Image credit: Radu Jianu/Brown University

1950s – 1970s: The perceptron

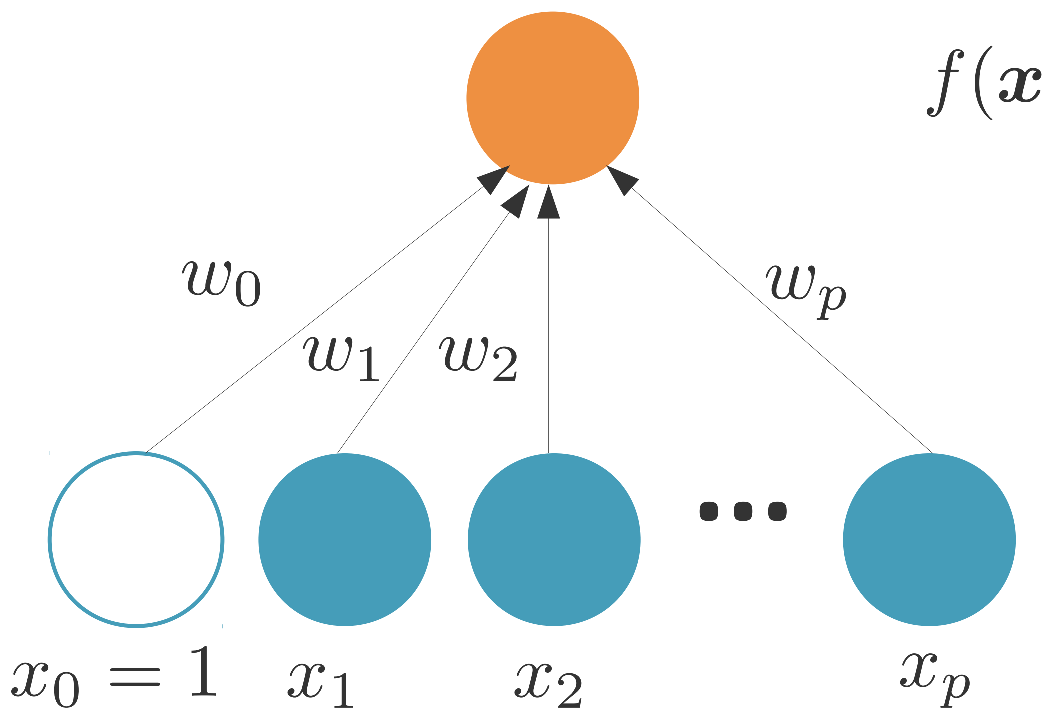
Perceptron

[Rosenblatt, 1958]



Perceptron

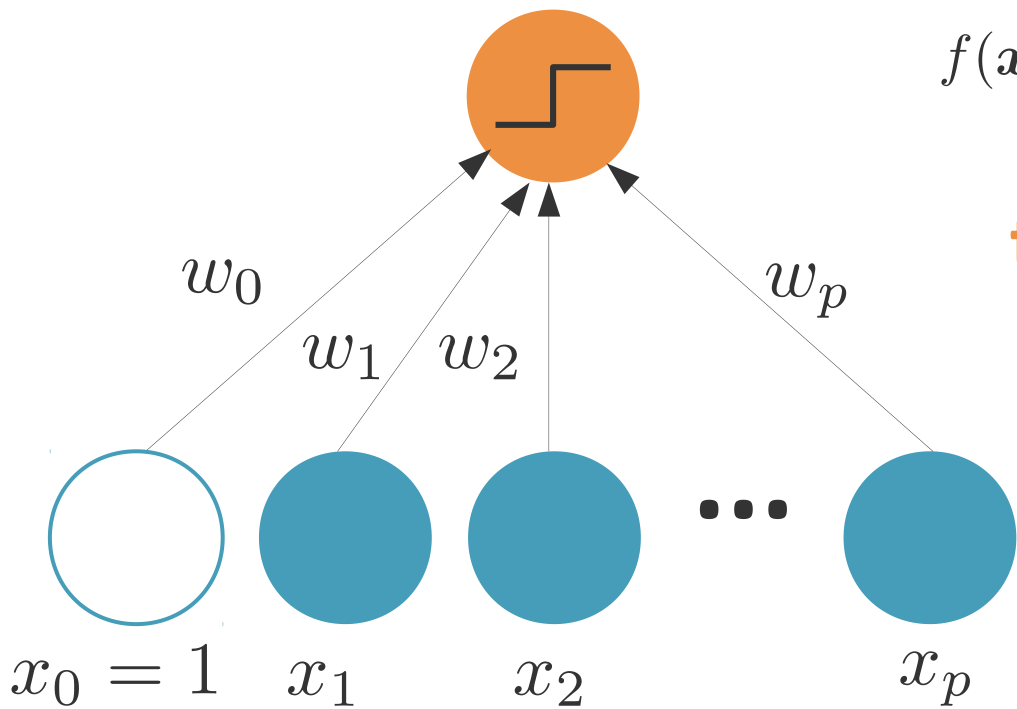
[Rosenblatt, 1958]



$$f(\boldsymbol{x}) = \sum_{j=1}^{p} w_j x_j + w_0 = \boldsymbol{w}^\top \boldsymbol{x}$$

How can we do classification?

Classification with the perceptron



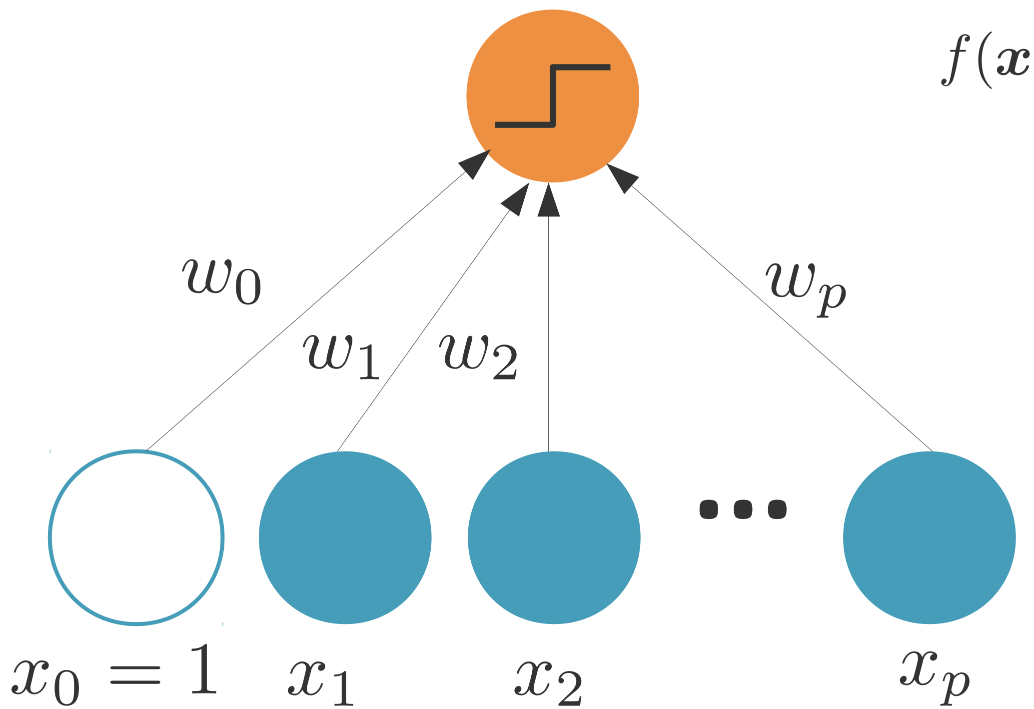
$$f(\boldsymbol{x}) = s\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = s(\boldsymbol{w}^\top \boldsymbol{x})$$

threshold function

$$s(o(\boldsymbol{x})) = \begin{cases} 1 & \text{if } o(\boldsymbol{x}) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

What is the shape of the decision boundary?

Classification with the perceptron



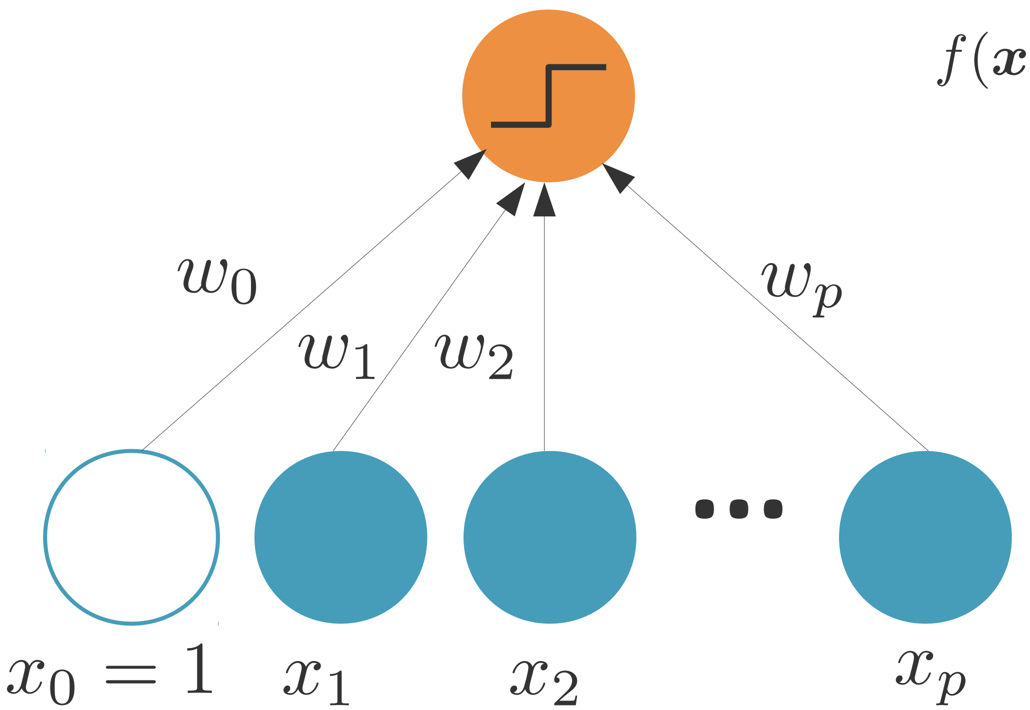
$$f(\boldsymbol{x}) = s\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = s(\boldsymbol{w}^\top \boldsymbol{x})$$

$$s(o(\boldsymbol{x})) = \begin{cases} 1 & \text{if } o(\boldsymbol{x}) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The decision boundary is a hyperplane (a line in dim 2).

Which other methods have we seen that yield decision boundaries that are lines/hyperplanes?

Classification with the perceptron

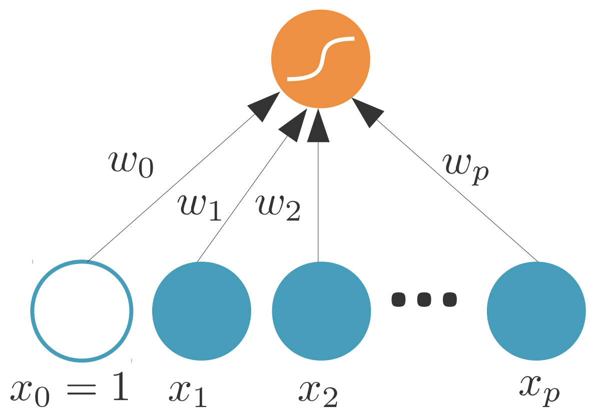


$$f(\boldsymbol{x}) = s\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = s(\boldsymbol{w}^\top \boldsymbol{x})$$

$$s(o(\boldsymbol{x})) = \begin{cases} 1 & \text{if } o(\boldsymbol{x}) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

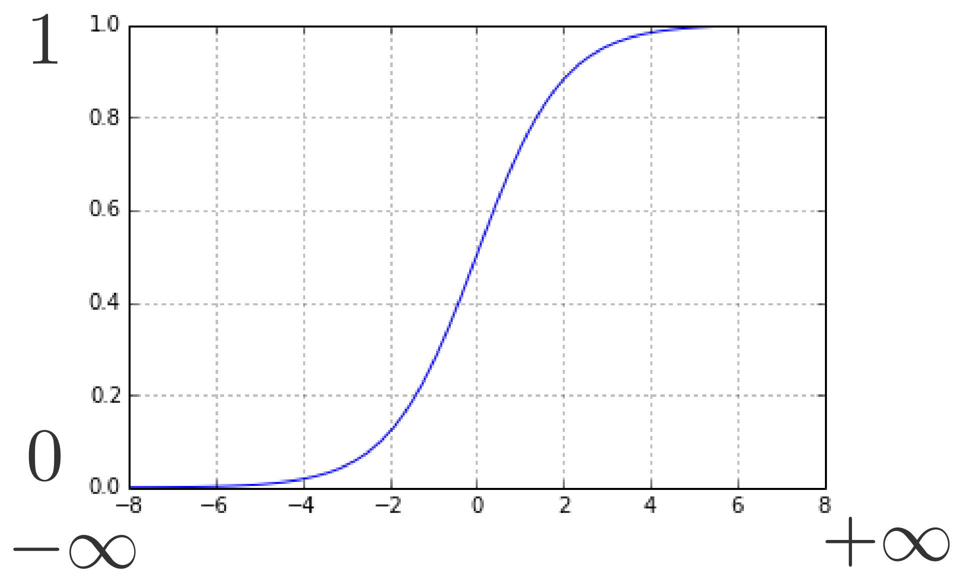
What if instead of just a decision (+/-) we want to output the probability of belonging to the positive class?

Classification with the perceptron



$$f(\boldsymbol{x}) = \sigma\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = \sigma(\boldsymbol{w}^\top \boldsymbol{x})$$

$$\sigma(u) = \frac{1}{1 + \exp(-u)}$$

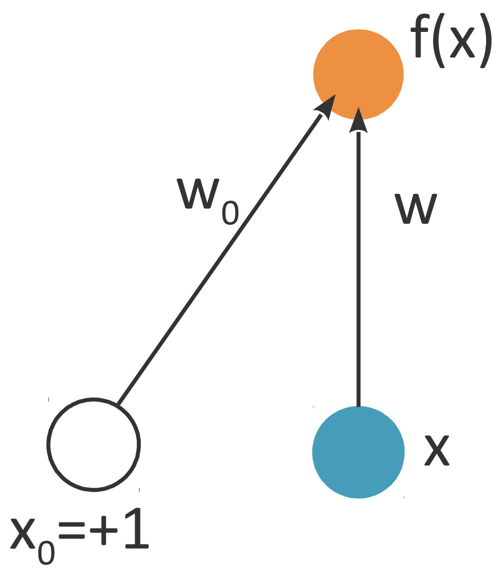


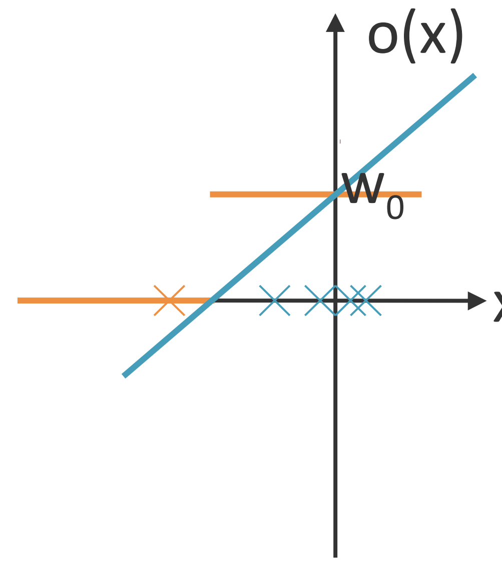
Probability of belonging to the positive class:

f(**x**) = logistic(**w**ᵀ**x**).

Perceptron: 1D summary

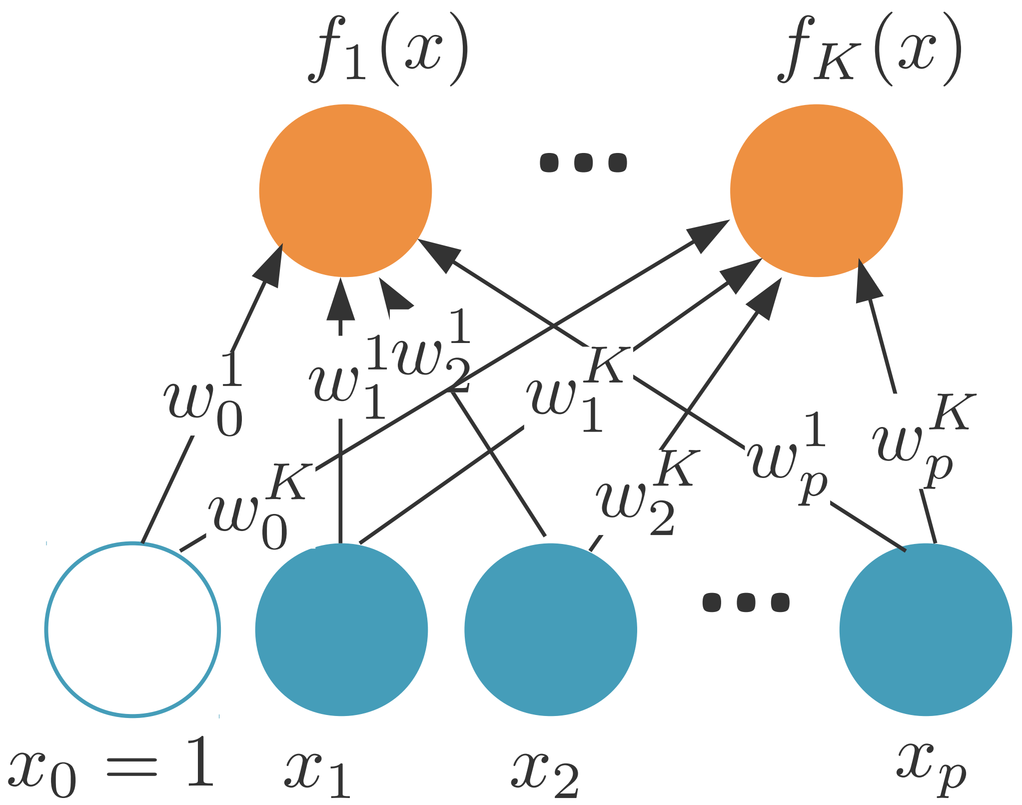
- **Regression:** $f(x) = w.x + w_0$
- **Classification:** $f(x) = \dfrac{1}{1 + \exp - (w.x + w_0)}$





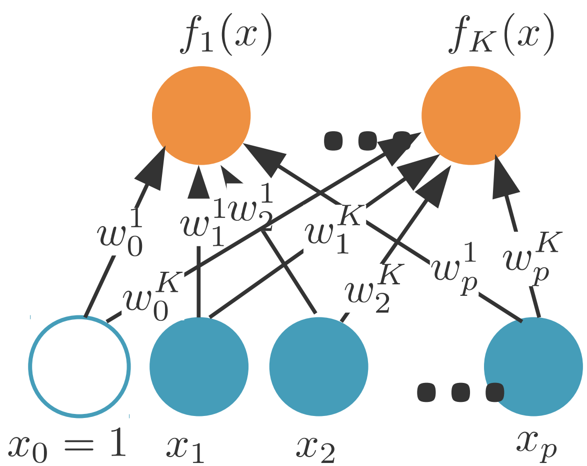
Multiclass classification

Use K output units



How do we take a final decision?

Multiclass classification



- Choose C_k if

$$f_k(\boldsymbol{x}) = \max_{l \in \{1,\ldots,K\}} f_l(\boldsymbol{x})$$

- To get probabilities, use the **softmax:**

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

$$f_k(\boldsymbol{x}) = \frac{\exp(o_k)}{\sum_{l=1}^{K} \exp(o_l)} \qquad o_k = w^{k\top}\boldsymbol{x}$$

  - If the output for one class is sufficiently larger than for the others, its softmax will be close to 1 (0 otherwise)
  - Similar to the max, but differentiable.

Training a perceptron

Training a perceptron

- **Online** (instances seen one by one) vs **batch** (whole sample) learning:
  - No need to store the whole sample
  - Problem may change in time
  - Wear and degradation in system components.

Training a perceptron

- **Online** (instances seen one by one) vs **batch** (whole sample) learning:
  - No need to store the whole sample
  - Problem may change in time
  - Wear and degradation in system components.
- **Gradient descent:**
  - Start from random weights
  - After each data point, adjust the weights to minimize the error.

Training a perceptron

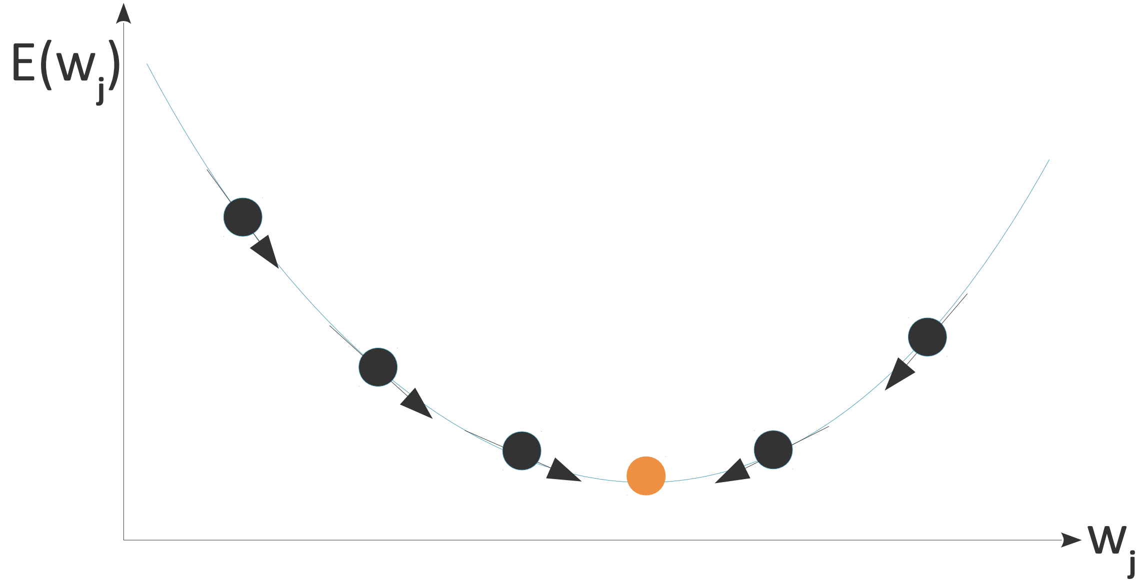
- **Generic update rule:**

$$\Delta w_j = -\eta \frac{\partial \text{Error}(f(\boldsymbol{x}^i), y^i)}{\partial w_j}$$

Learning rate

- After each training instance, for each weight:

$$w_j^{(t+1)} = w_j^{(t)} + \Delta w_j^{(t)} \qquad w_j \leftarrow w_j + \Delta w_j$$



Training a perceptron: regression

- Regression

$$\text{Error}(f(\boldsymbol{x}^i), y^i) = \frac{1}{2}(y^i - f(\boldsymbol{x}^i))^2 = \frac{1}{2}(y^i - w^\top \boldsymbol{x}^i)^2$$

- **What is the update rule?**

Remember the generic update rule:

$$w_j^{(t+1)} = w_j^{(t)} + \Delta w_j^{(t)} \qquad \Delta w_j = -\eta \frac{\partial \text{Error}(f(\boldsymbol{x}^i), y^i)}{\partial w_j}$$

Training a perceptron: regression

- **Regression**

$$\mathrm{Error}(f(\boldsymbol{x}^i), y^i) = \frac{1}{2}(y^i - f(\boldsymbol{x}^i))^2 = \frac{1}{2}(y^i - w^\top \boldsymbol{x}^i)^2$$

- The update rule for the regression is:

$$\Delta w_j = \eta(y^i - f(\boldsymbol{x}^i))x_j^i$$

Training a perceptron: classification

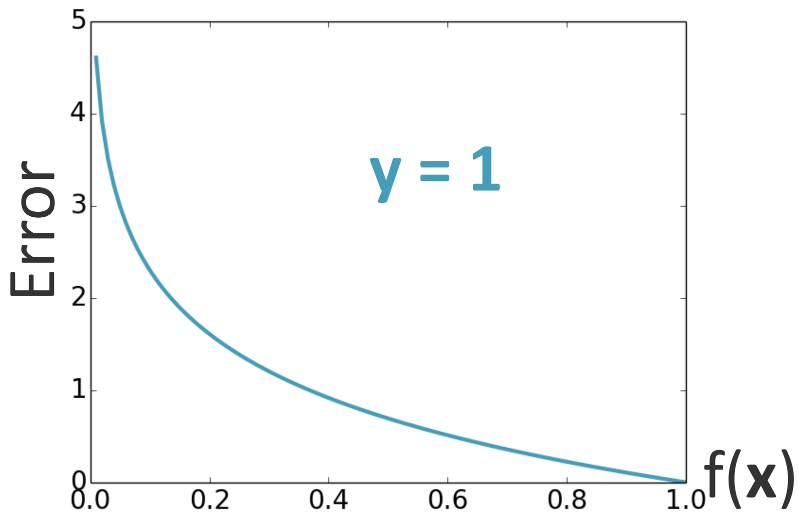
- **Sigmoid output:**

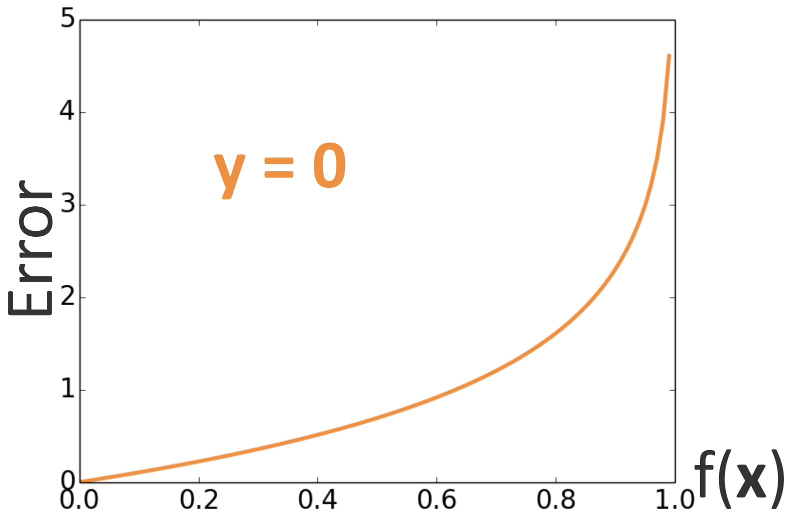
$$\sigma(u) = \frac{1}{1+\exp(-u)}$$

$$f(\boldsymbol{x}^i) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}^i)$$

- **Cross-entropy error:**

$$\mathrm{Error}(f(\boldsymbol{x}^i), y^i) = -y^i \log f(\boldsymbol{x}^i) - (1-y^i)\log(1-f(\boldsymbol{x}^i))$$





- **What is the update rule now?**

Training a perceptron: classification

- **Sigmoid output:**

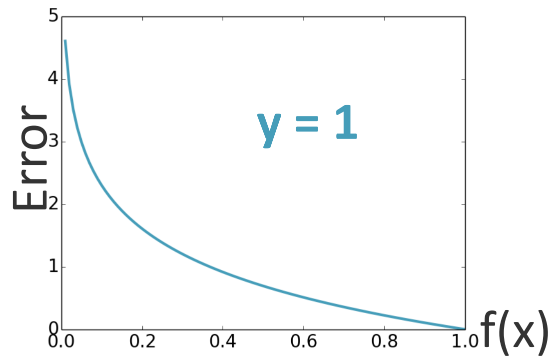
$$f(\boldsymbol{x}^i) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}^i)$$

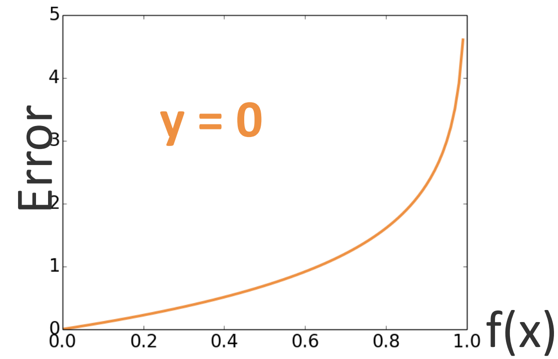
$$\sigma(u) = \frac{1}{1 + \exp(-u)}$$

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

- **Cross-entropy error:**

$$\mathrm{Error}(f(\boldsymbol{x}^i), y^i) = -y^i \log f(\boldsymbol{x}^i) - (1 - y^i) \log(1 - f(\boldsymbol{x}^i))$$





- **Update rule for binary classification:**

$$\Delta w_j = -\eta \frac{\partial \mathrm{Error}(f(\boldsymbol{x}^i), y^i)}{\partial w_j} \qquad \Delta w_j = \eta(y^i - f(\boldsymbol{x}^i))x_j^i$$

Training a perceptron: K classes

- **K > 2 softmax outputs:**

$$f_k(\boldsymbol{x}^i) = \frac{\exp(\boldsymbol{w}^{k\top}\boldsymbol{x}^i)}{\sum_{l=1}^{K}\exp(\boldsymbol{w}^{l\top}\boldsymbol{x}^i)}$$

- **Cross-entropy error:**

$$\mathrm{Error}(f(\boldsymbol{x}^i), y^i) = -\sum_{k=1}^{K} y_k^i \log f_k(\boldsymbol{x}^i)$$

- **Update rule for K-way classification:**

$$\Delta w_j^k = \eta(y^i - f_k(\boldsymbol{x}^i))x_j^i$$

Training a perceptron

- **Generic update rule:**

$$\Delta w_j = \eta(y^i - f(\boldsymbol{x}^i))x_j^i$$

Update = Learning rate.(Desired output – Actual output).Input

- After each training instance, for each weight:

$$w_i^{(t+1)} = w_j^{(t)} + \Delta w_i^{(t)} \qquad w_j \leftarrow w_j + \Delta w_j$$

- **What happens if**
  - **desired output = actual output?**
  - **desired output < actual output?**

Training a perceptron: regression

- **Generic update rule:**

$$\Delta w_j = \eta(y^i - f(\boldsymbol{x}^i))x_j^i$$

Update = Learning rate.(Desired output – Actual output).Input

- After each training instance, for each weight:

$$w_j^{(t+1)} = w_j^{(t)} + \Delta w_j^{(t)} \qquad w_j \leftarrow w_j + \Delta w_j$$

  - If desired output = actual output: no change
  - If desired output < actual output:
    - input > 0 → update < 0 → w_j smaller → prediction ↘
    - input < 0 → update > 0 → w_j bigger → prediction ↘

Learning boolean functions

Learning AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

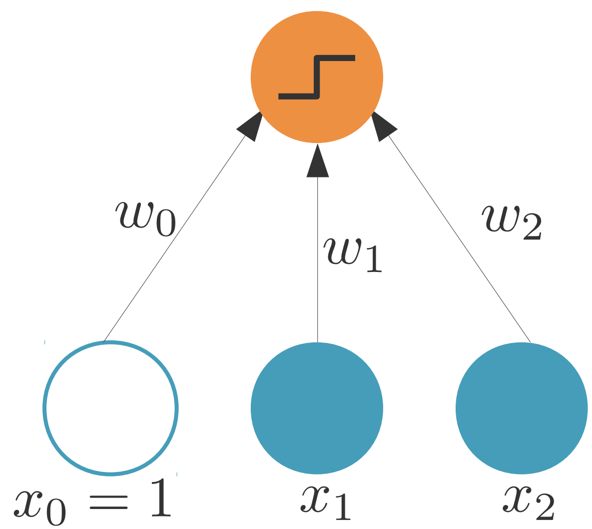
Design a perceptron that learns AND.

- **What is its architecture?**

Learning AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

Design a perceptron that learns AND.



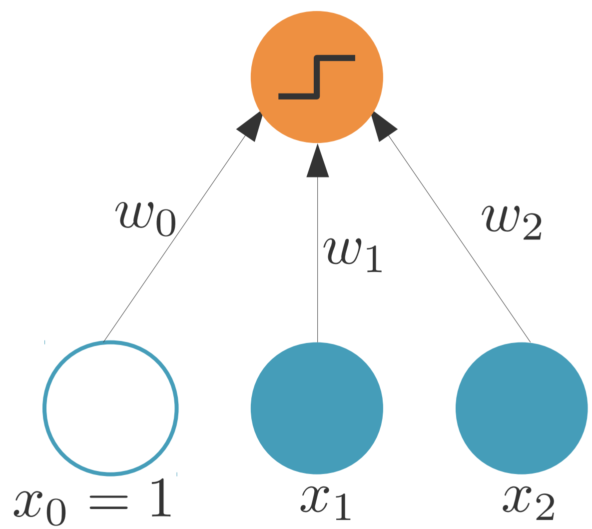
$$f(x) = s(w_0 + w_1 x_1 + w_2 x_2)$$

Draw the 4 possible points (x1, x2) and a desirable separating line. What is its equation?

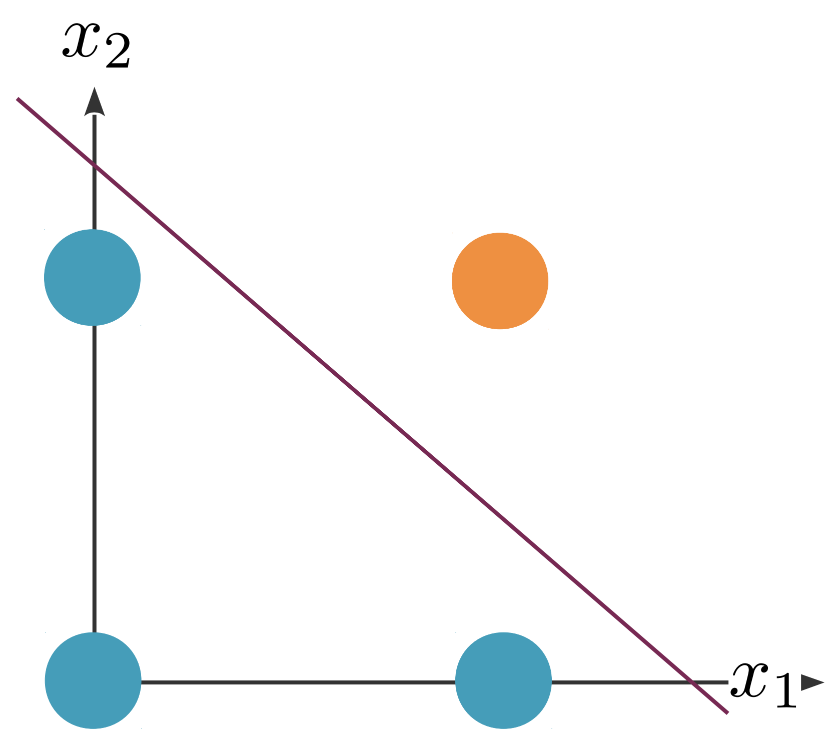
Learning AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

Design a perceptron that learns AND.



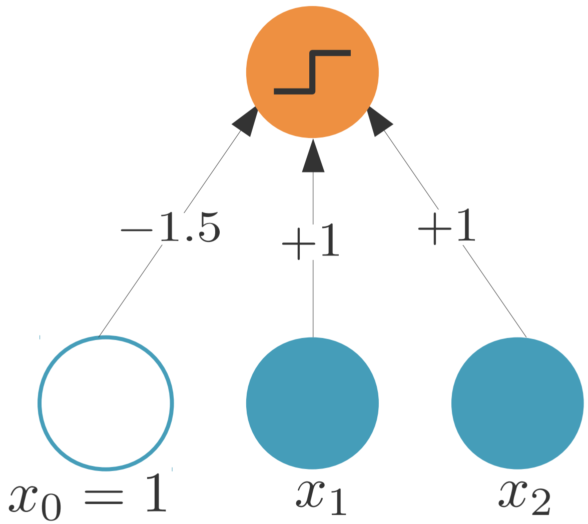
$$f(x) = s(w_0 + w_1 x_1 + w_2 x_2)$$



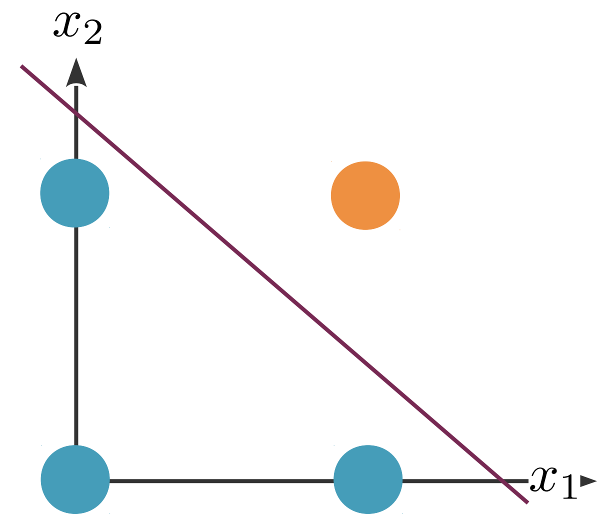
Learning AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

Design a perceptron that learns AND.



$$f(x) = s(w_0 + w_1 x_1 + w_2 x_2)$$



x1	x2	f(x)
0	0	s(-1.5 + 0 + 0) = s(-1.5) = 0
0	1	s(-1.5 + 0 + 1) = s(-0.5) = 0
1	0	s(-1.5 + 1 + 0) = s(-0.5) = 0
1	1	s(-1.5 + 1 + 1) = s(0.5) = 1

Learning XOR

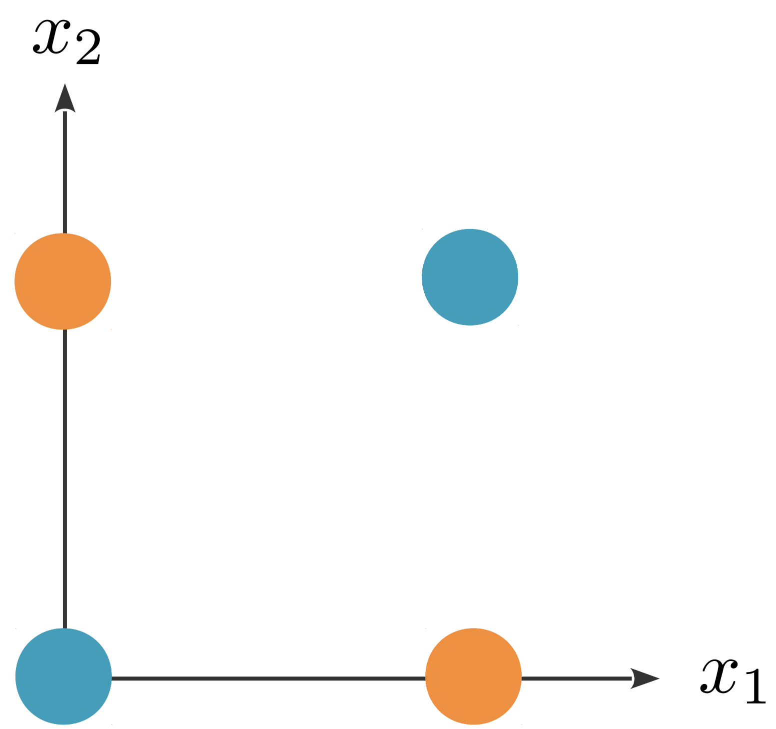
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Design a perceptron that learns XOR

Learning XOR

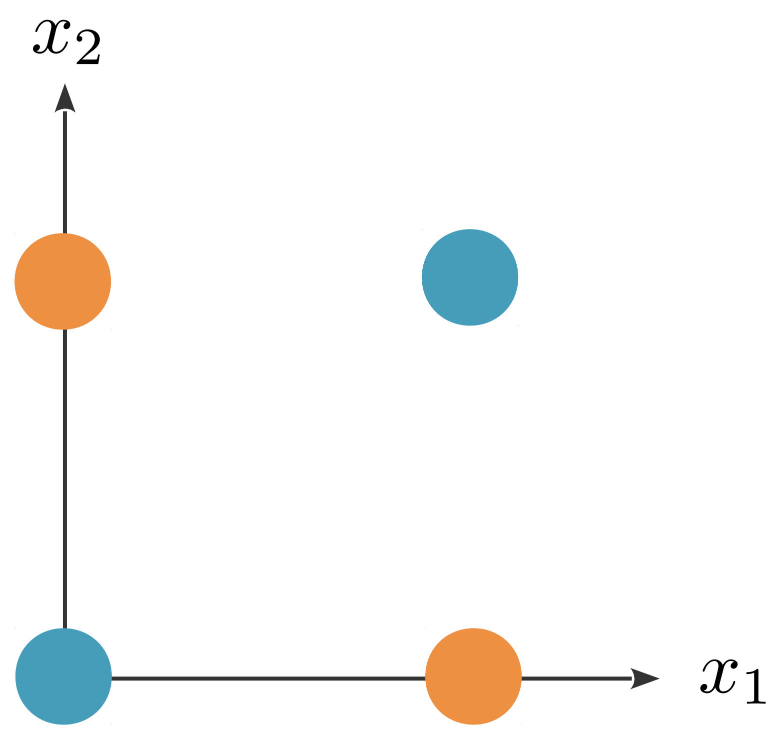
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Design a perceptron that learns XOR



Learning XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



[Minsky and Papert, 1969]

No w_0, w_1, w_2 satisfy:

$$\begin{aligned} w_0 &\leq 0 \\ w_0 + w_2 &> 0 \\ w_0 + w_1 &> 0 \\ w_0 + w_1 + w_2 &\leq 0 \end{aligned}$$

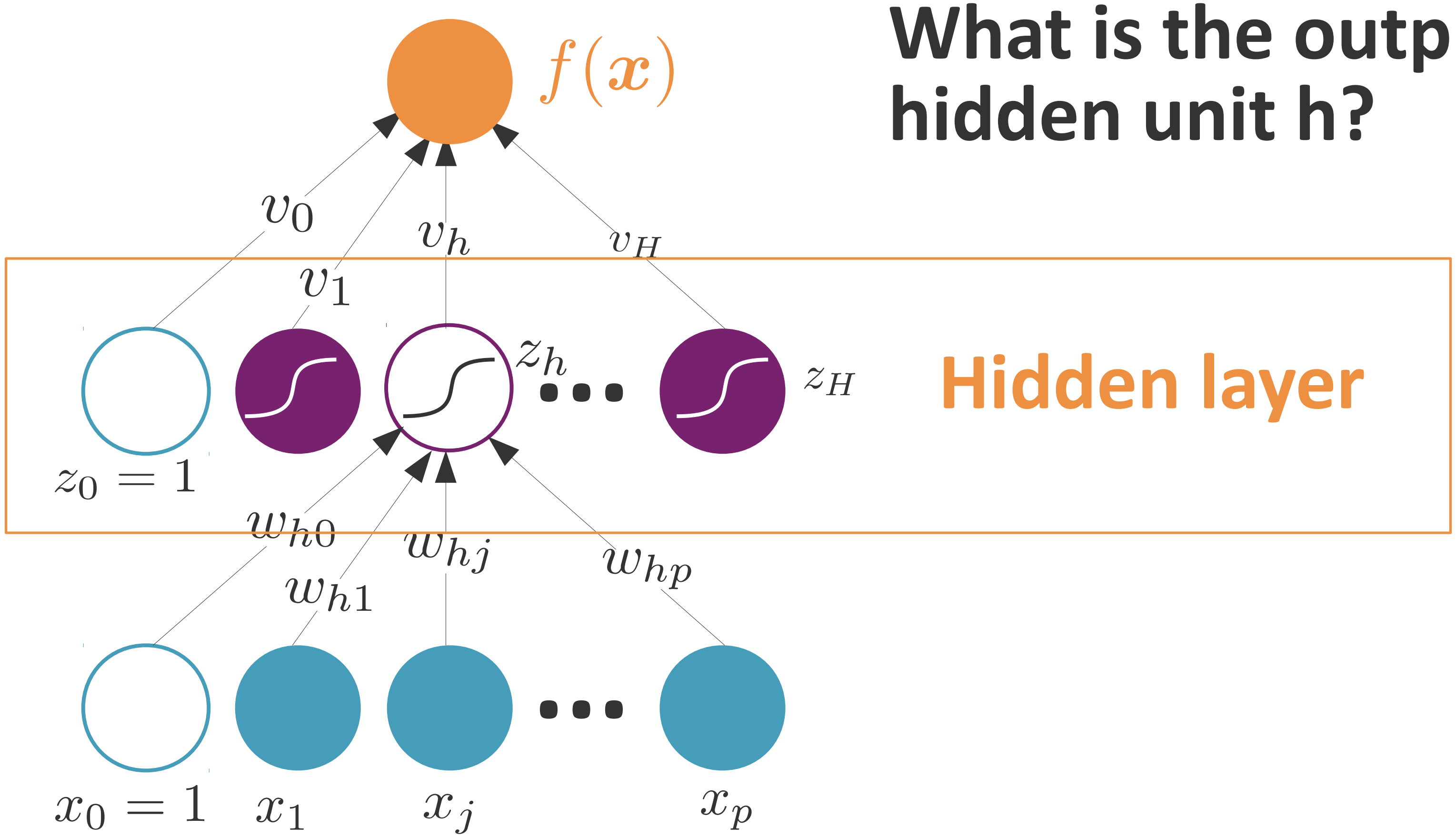
Perceptrons

M. Minsky & S. Papert, 1969

The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension to multilayer systems is sterile.

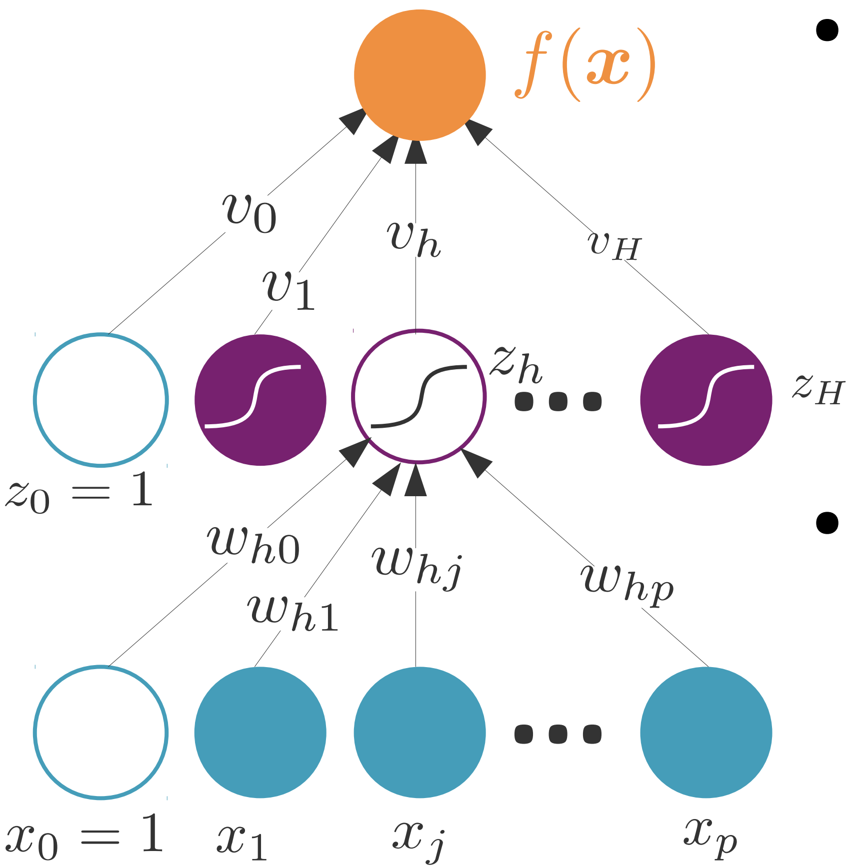
1980s – early 1990s

Multilayer perceptrons



What is the output of the hidden unit h?

Multilayer perceptrons

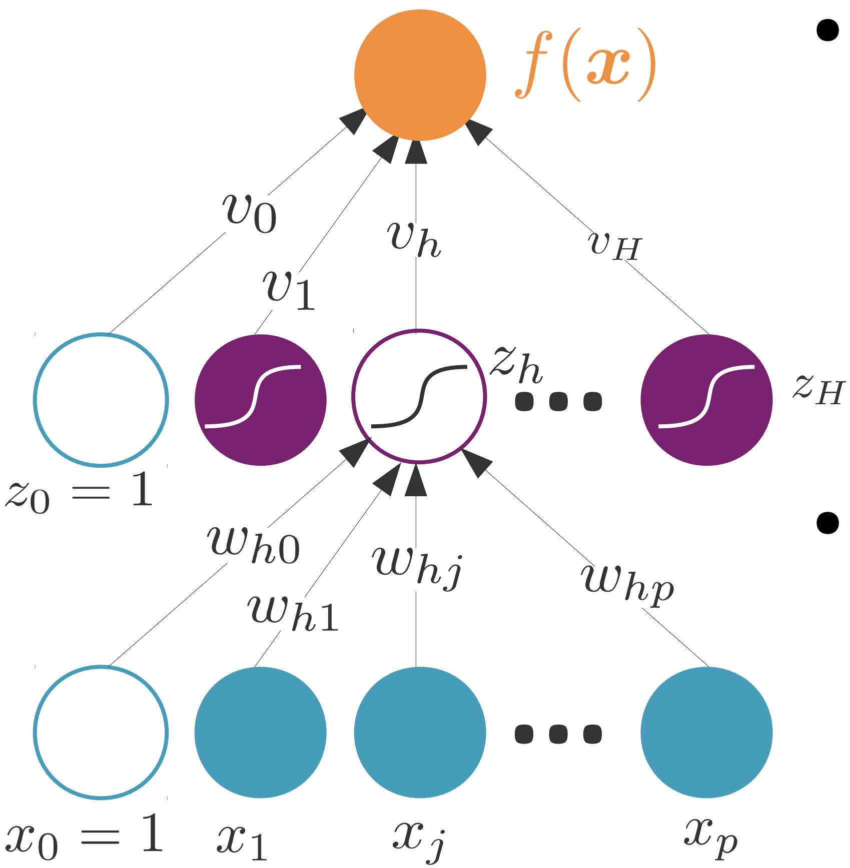


- **Output of hidden unit h:**

$$z_h = \frac{1}{1 + e^{-\boldsymbol{w}_h^\top \boldsymbol{x}}}$$

- **What is the output of the network?**

Multilayer perceptrons



- **Output of hidden unit h:**

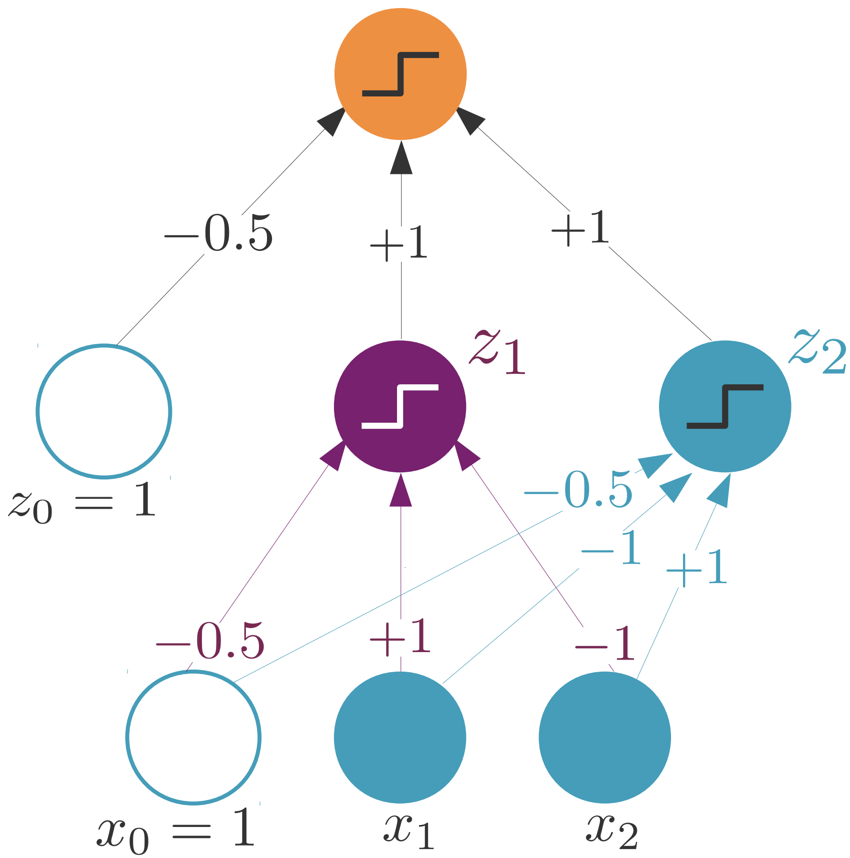
$$z_h = \frac{1}{1 + e^{-\boldsymbol{w}_h^\top \boldsymbol{x}}}$$

- **Output of the network:**

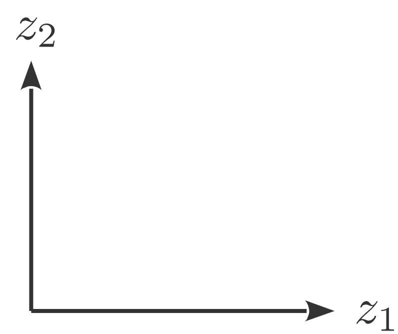
$$\begin{aligned} f(x) &= \boldsymbol{v}^\top \boldsymbol{z} \\ &= v_0 + \sum_{h=1}^{H} \frac{v_h}{1+e^{-\boldsymbol{w}_h^\top \boldsymbol{x}}} \end{aligned}$$

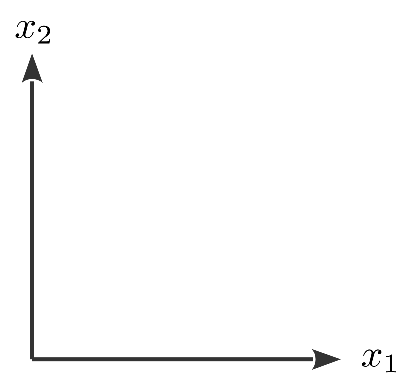
Not linear in x!

Learning XOR with an MLP

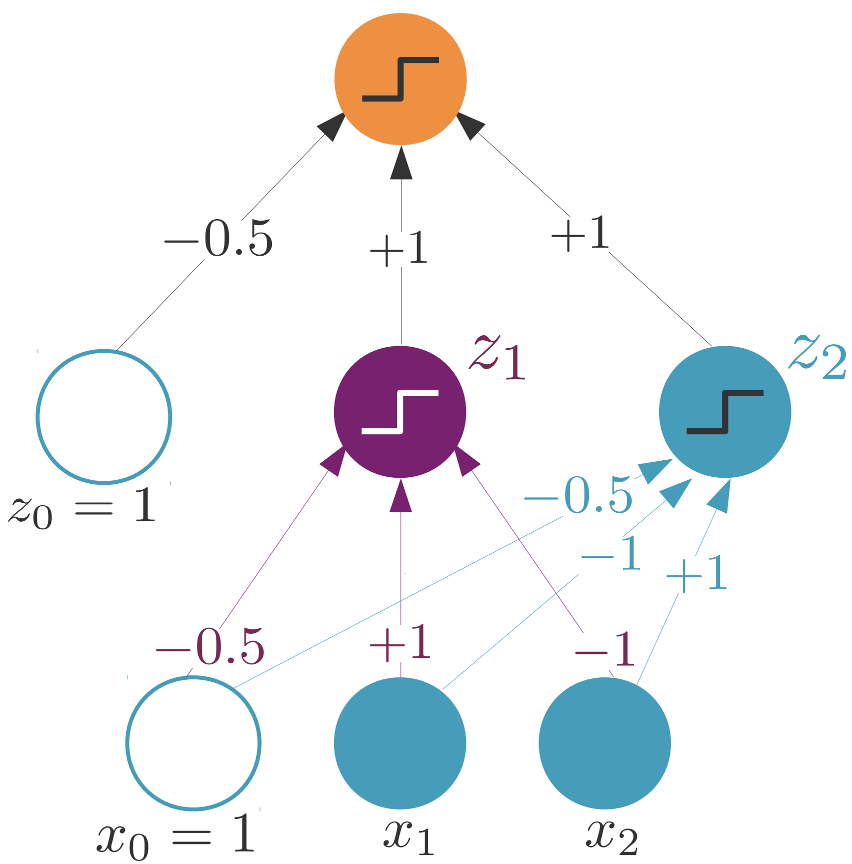


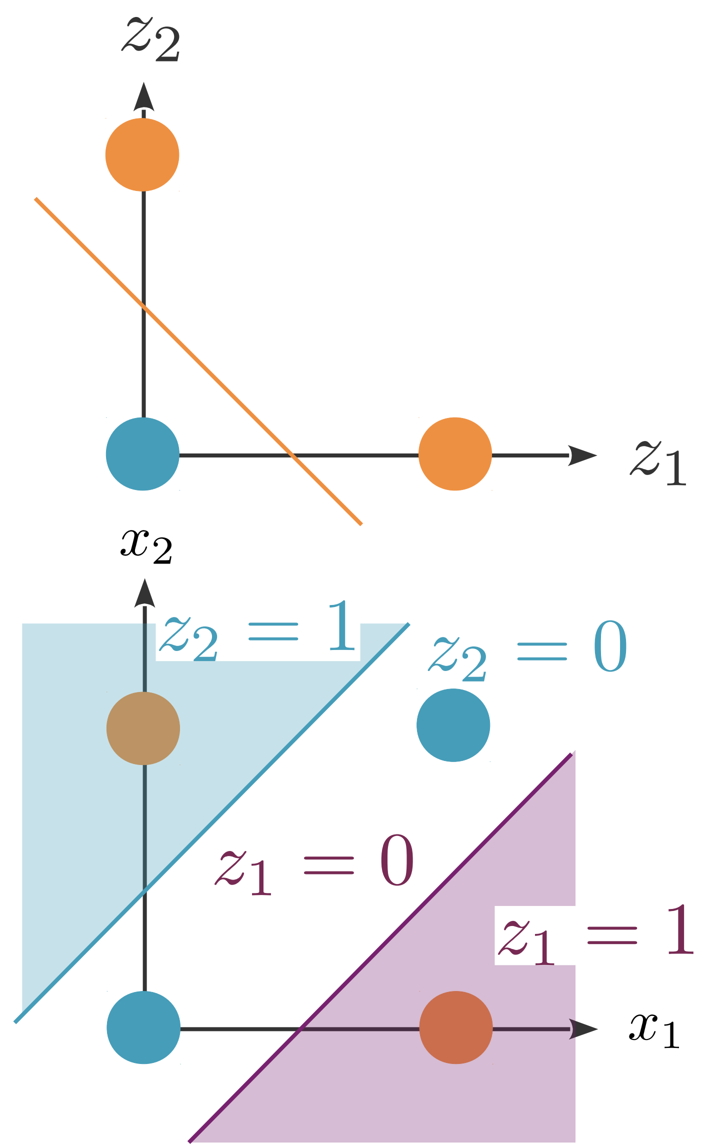
Draw the geometric interpretation of this multiple layer perceptron.





Learning XOR with an MLP





Universal approximation

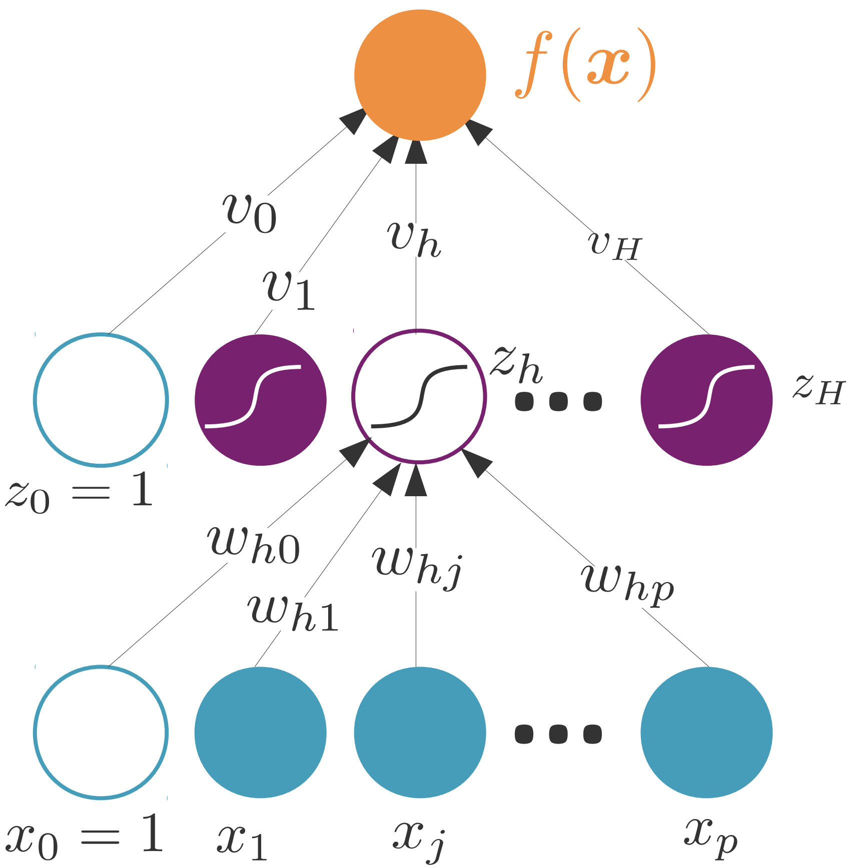
Any continuous function on a compact subset of $\mathbb{R}^n$ can be approximated to any arbitrary degree of precision by a feed-forward multi-layer perceptron with a single hidden layer containing a finite number of neurons.

Cybenko (1989), Hornik (1991)

Backpropagation

Backwards propagation of errors.

$$z_h = \frac{1}{1 + e^{-\boldsymbol{w}_h^\top \boldsymbol{x}}}$$



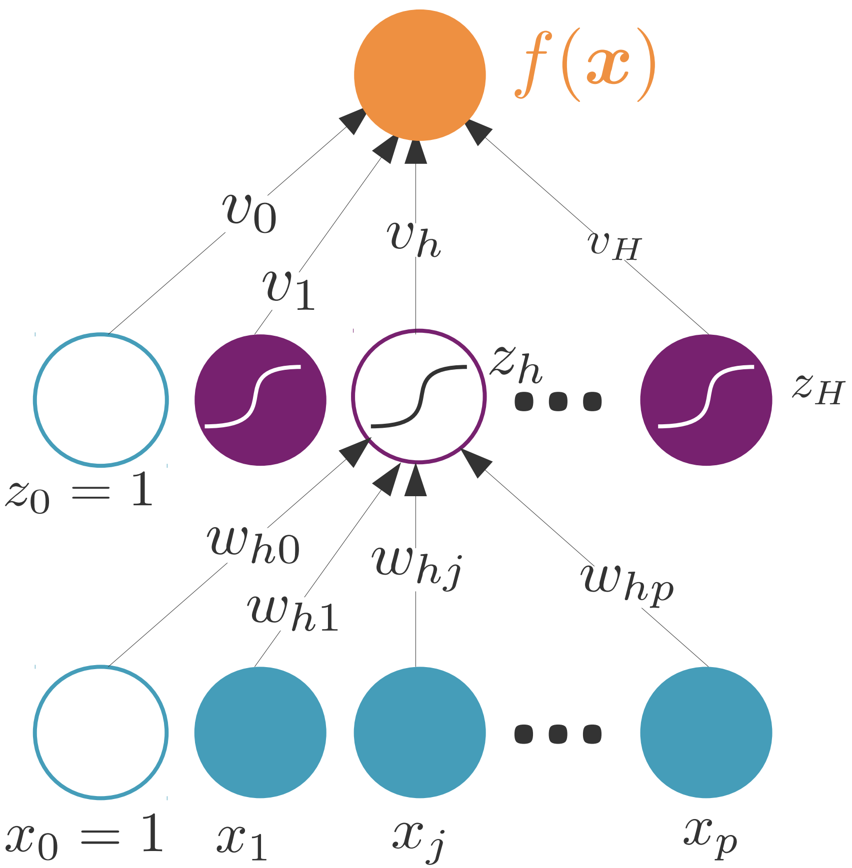
$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{v}^\top \boldsymbol{z} \\ &= v_0 + \sum_{h=1}^{H} \frac{v_h}{1+e^{-\boldsymbol{w}_h^\top \boldsymbol{x}}} \end{aligned}$$

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial f(\boldsymbol{x})} \boxed{\frac{\partial f(\boldsymbol{x})}{\partial z_h}} \boxed{\frac{\partial z_h}{\partial w_{hj}}}$$

http://people.idsia.ch/~juergen/who-invented-backpropagation.html

Backpropagation

Backwards propagation of errors.



$$z_h = \frac{1}{1 + e^{-\boldsymbol{w}_h^\top \boldsymbol{x}}}$$

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{v}^\top \boldsymbol{z} \\ &= v_0 + \textstyle\sum_{h=1}^{H} \frac{v_h}{1+e^{-\boldsymbol{w}_h^\top \boldsymbol{x}}} \end{aligned}$$

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial f(\boldsymbol{x})} \underbrace{\boxed{\frac{\partial f(\boldsymbol{x})}{\partial z_h}}}_{v_h} \underbrace{\boxed{\frac{\partial z_h}{\partial w_{hj}}}}_{z_h^i(1 - z_h^i)x_j^i}$$

Backprop: Regression

$$\text{Error}(f(\boldsymbol{x}^i), y^i) = \frac{1}{2}(y^i - f(\boldsymbol{x}^i))^2$$

$$f(\boldsymbol{x}) = v_0 + \sum_{h=1}^{H} v_h z_h$$

$$\Delta v_h = \eta_1 (y^i - f(\boldsymbol{x}^i)) z_h^i$$

Forward

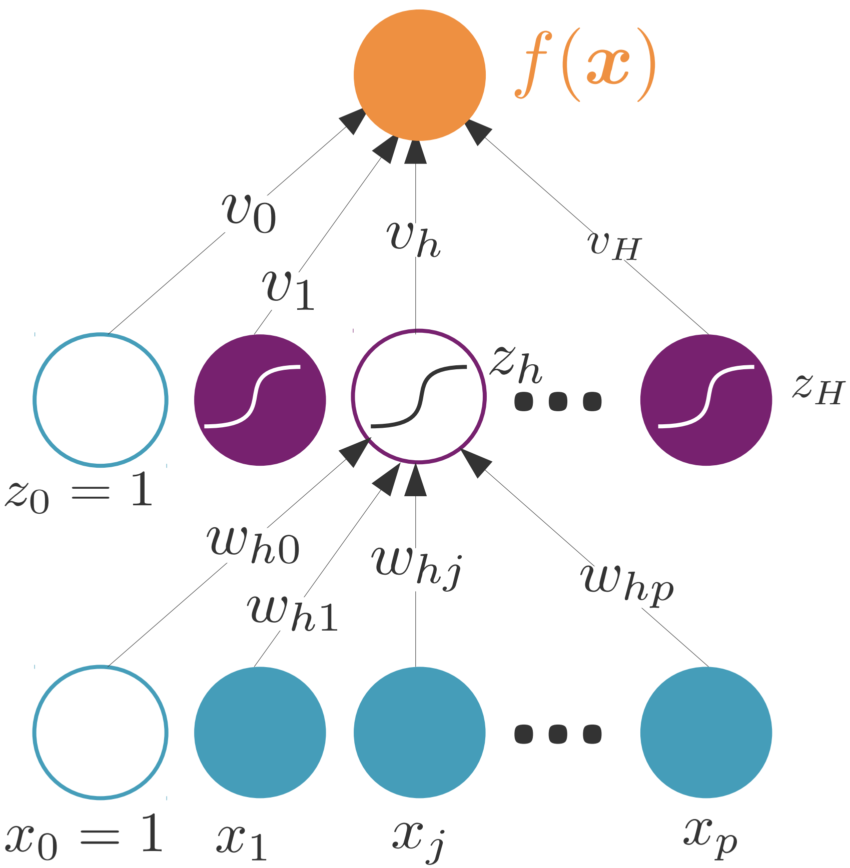
Backward

$$z_h = \sigma(w_h^\top \boldsymbol{x})$$

$$\boldsymbol{x}$$

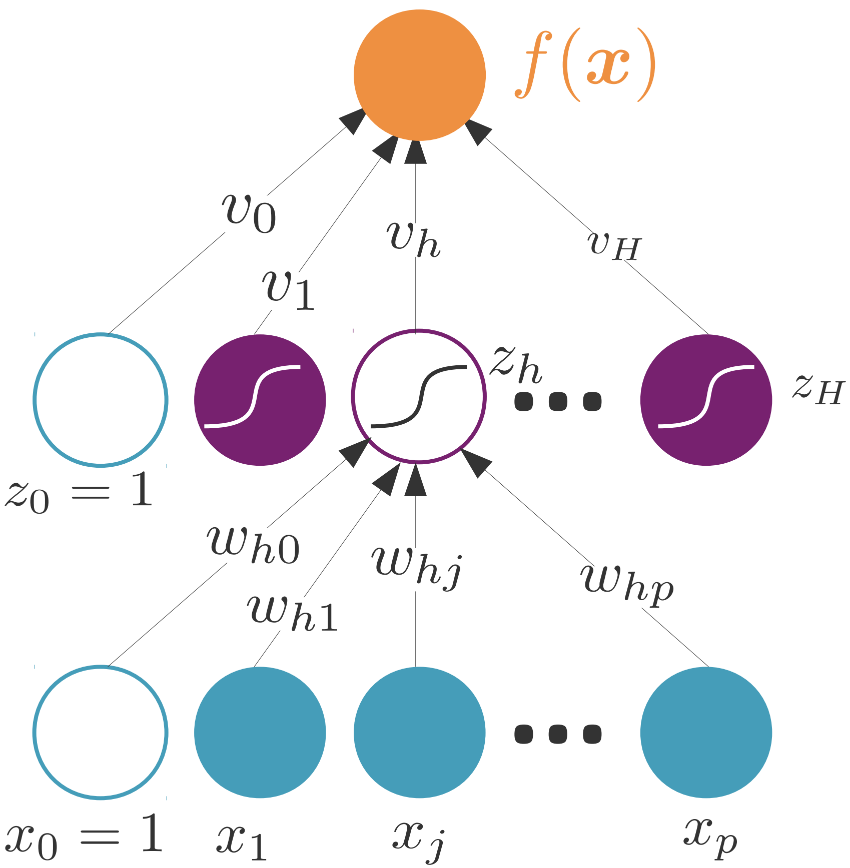
$$\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial E^i}{\partial w_{hj}} \\
&= -\eta \frac{\partial E}{\partial f(\boldsymbol{x}^i)} \frac{\partial f(\boldsymbol{x}^i)}{\partial z_h^i} \frac{\partial z_h^i}{\partial w_{hj}} \\
&= -\eta (y^i - f(\boldsymbol{x}^i)) v_h z_h^i (1 - z_h^i) x_j^i
\end{aligned}$$

Backprop: Regression



Initialize all v_h, w_{hj} to rand$(-0.01, 0.01)$
Repeat until convergence
  For $i = 1, \ldots, n$
    For $h = 1, \ldots, H$
      $z_h^i = \sigma(w_h^\top x^i)$
    $f(x^i) = v^\top z^i$
    For $h = 1, \ldots, H$
      $\Delta v_h = \eta_1(y^i - f(x^i))z_h^i$
    For $h = 1, \ldots, H$
      For $j = 1, \ldots, p$
        $\Delta w_{hj} = \eta((y_k^i - f_k(x^i)v_h)z_h^i(1 - z_h^i)x_j^i$
    For $h = 1, \ldots, H$
      $v_h \leftarrow v_h + \Delta v_h$
      For $j = 1, \ldots, p$
        $w_{hj} \leftarrow w_{hj} + \Delta w_{hj}$

Backprop: Regression

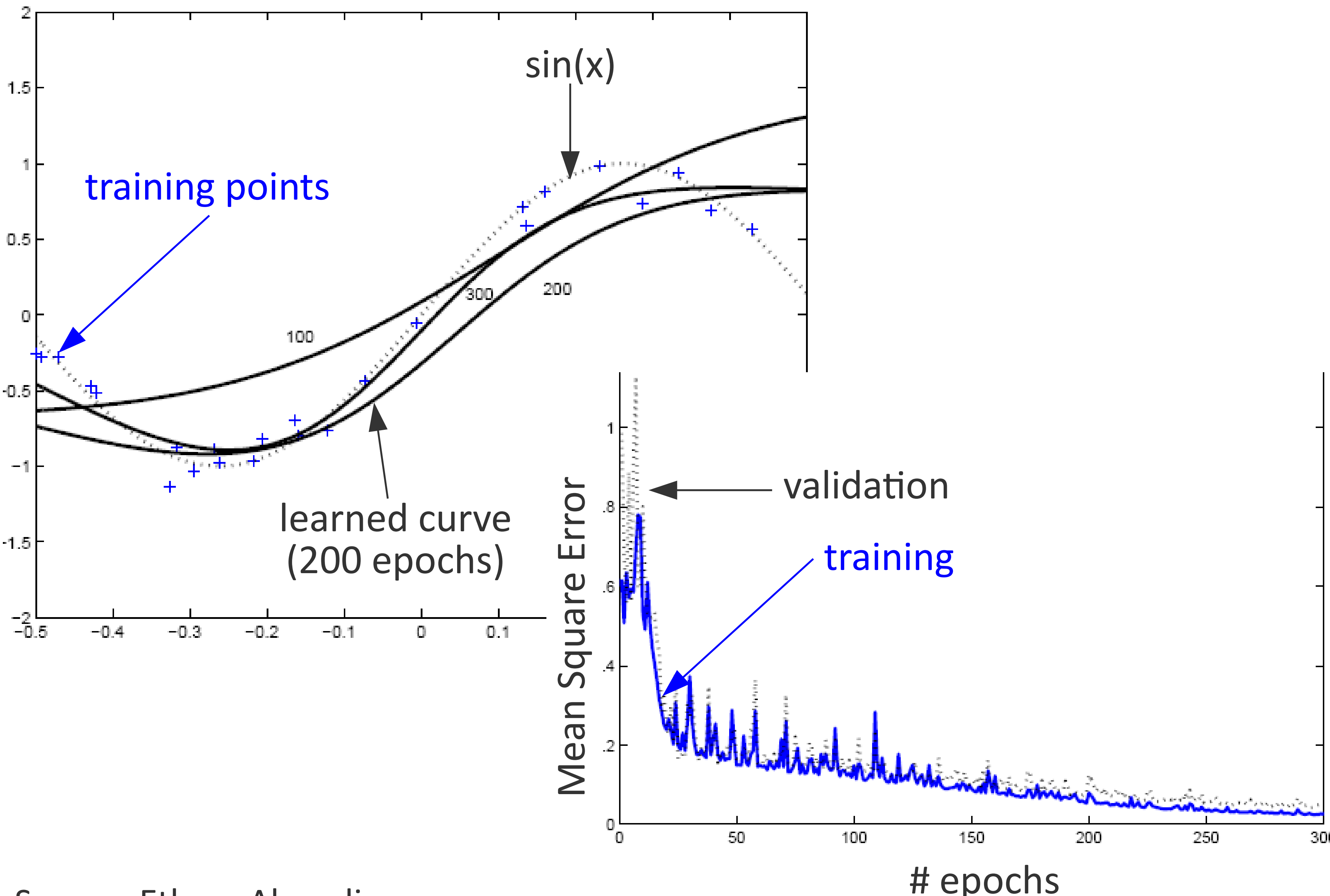


Initialize all v_h, w_{hj} to rand$(-0.01, 0.01)$
Repeat until convergence

For $i = 1, \ldots, n$
  For $h = 1, \ldots, H$
    $z_h^i = \sigma(w_h^\top x^i)$
  $f(x^i) = v^\top z^i$
  For $h = 1, \ldots, H$
    $\Delta v_h = \eta_1 (y^i - f(x^i)) z_h^i$
  For $h = 1, \ldots, H$
    For $j = 1, \ldots, p$
      $\Delta w_{hj} = \eta((y_k^i - f_k(x^i) v_h) z_h^i (1 - z_h^i) x_j^i$
  For $h = 1, \ldots, H$
    $v_h \leftarrow v_h + \Delta v_h$
    For $j = 1, \ldots, p$
      $w_{hj} \leftarrow w_{hj} + \Delta w_{hj}$

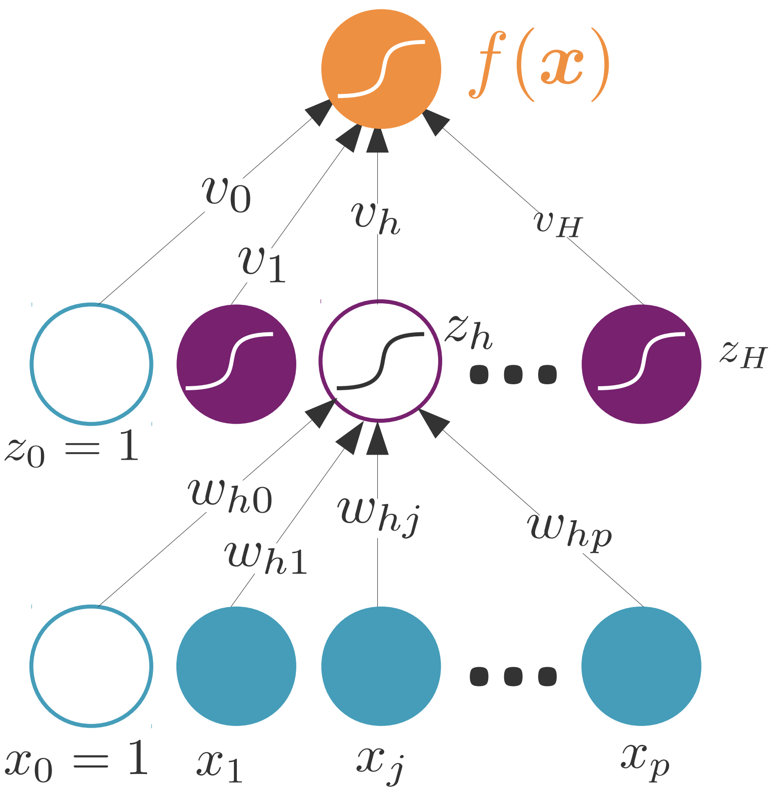
Epoch: when all the training points have been seen once

E.g.: Learning sin(x)



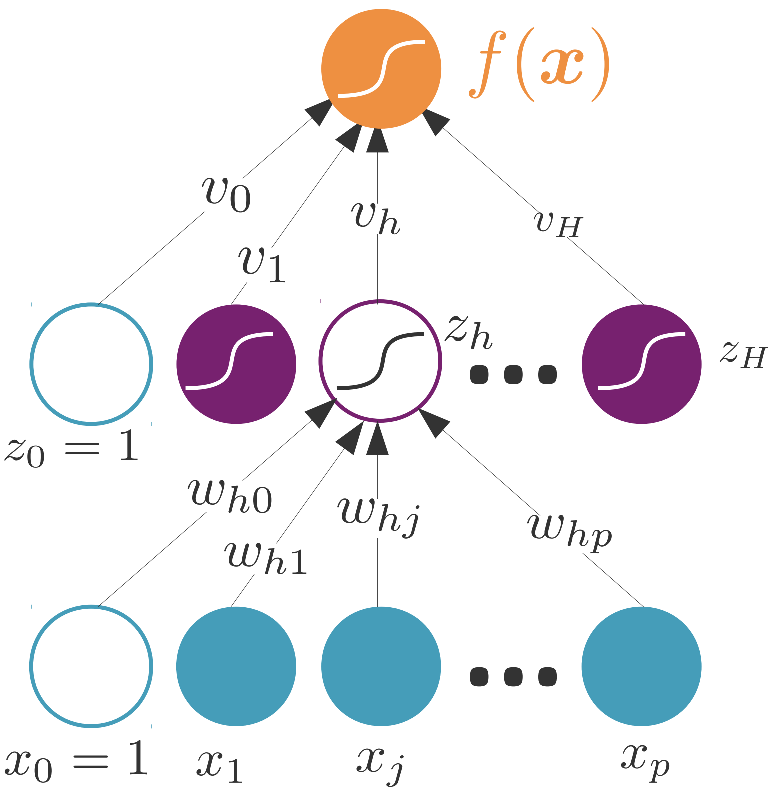
Source: Ethem Alpaydin

Backprop: Classification



- **Forward:**
  - **z_h = ?**
  - **f(x) = ?**

Backprop: Classification

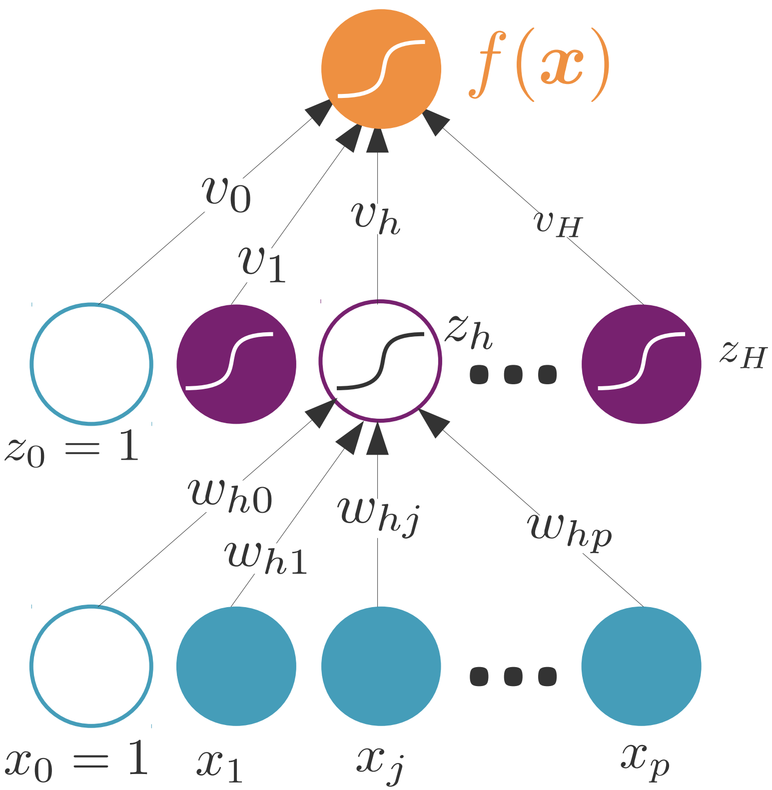


$$z_h = \sigma(w_h^\top \boldsymbol{x})$$

$$f(\boldsymbol{x}) = \sigma\left(v_0 + \sum_{h=1}^{H} v_h z_h\right)$$

- Error (cross-entropy)?

Backprop: Classification



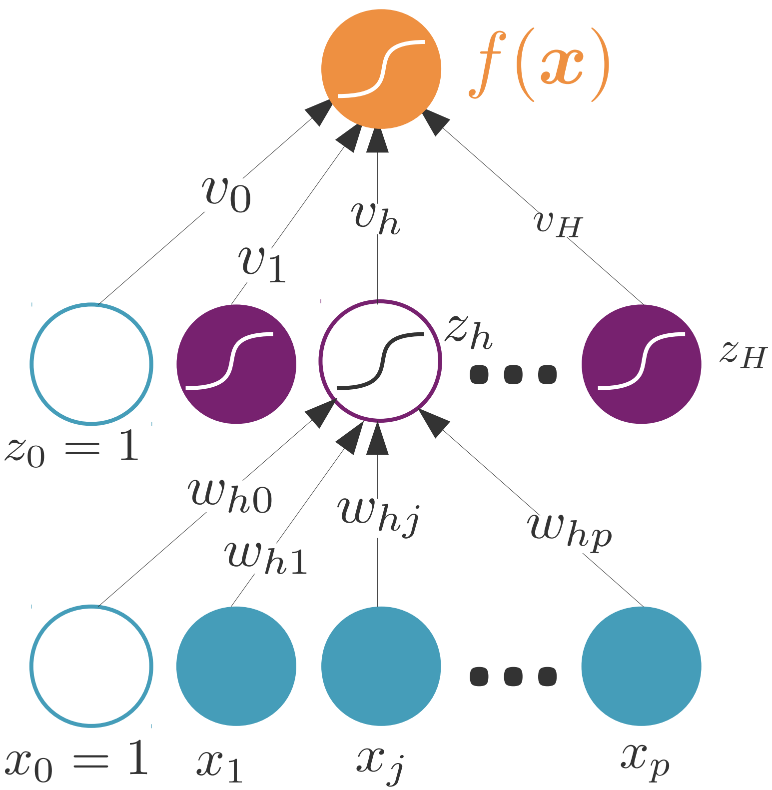
$$z_h = \sigma(w_h^\top \boldsymbol{x})$$

$$f(\boldsymbol{x}) = \sigma\left(v_0 + \sum_{h=1}^{H} v_h z_h\right)$$

$$\text{Error} = -\sum_{i=1}^{n} y^i \log(f(\boldsymbol{x}^i)) + (1 - y^i)\log(1 - f(\boldsymbol{x}^i))$$

- **Backward:**
  - **Δv_h?**
  - **Δw_{hj}?**

Backprop: Classification



$$z_h = \sigma(w_h^\top \boldsymbol{x})$$

$$f(\boldsymbol{x}) = \sigma\left(v_0 + \sum_{h=1}^{H} v_h z_h\right)$$

$$\text{Error} = -\sum_{i=1}^{n} y^i \log(f(x^i)) + (1 - y^i)\log(1 - f(x^i))$$

$$\Delta v_h = \eta_1 \sum_{i=1}^{n} (y^i - f(\boldsymbol{x}^i)) z_h^i$$

$$\Delta w_{hj} = \eta \sum_{i=1}^{n} (y^i - f(\boldsymbol{x}^i)) v_h z_h^i (1 - z_h^i) x_j^i$$

Backprop: K classes

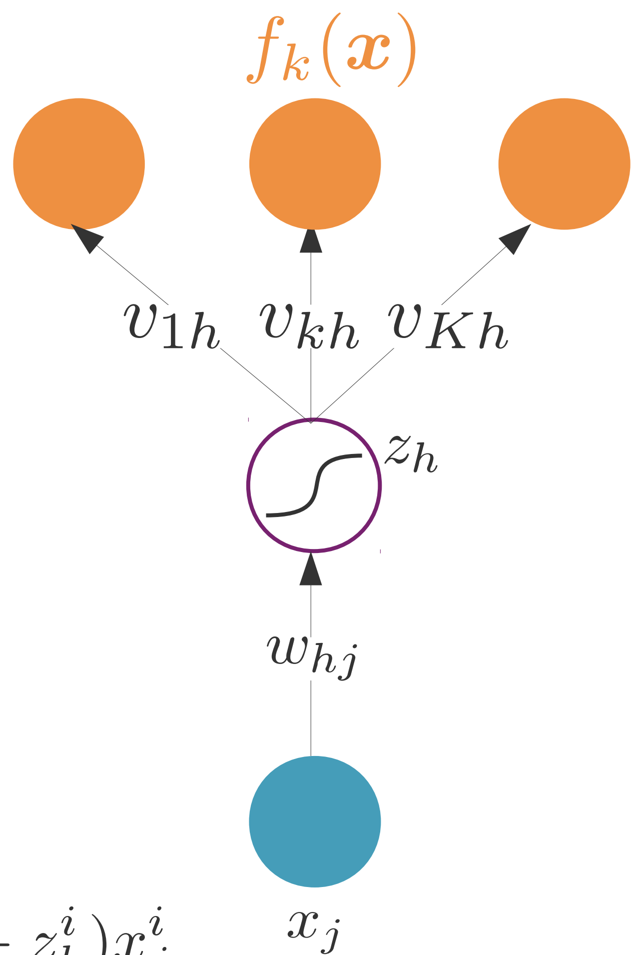
$$o_k^i = v_{k0} + \sum_{h=1}^{H} v_{kh} z_h^i$$

$$f_k(\boldsymbol{x}^i) = \frac{\exp(o_k^i)}{\sum_{l=1}^{K} \exp(o_l^i)}$$

$$\text{Error} = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_k^i \log f_k(\boldsymbol{x}^i)$$

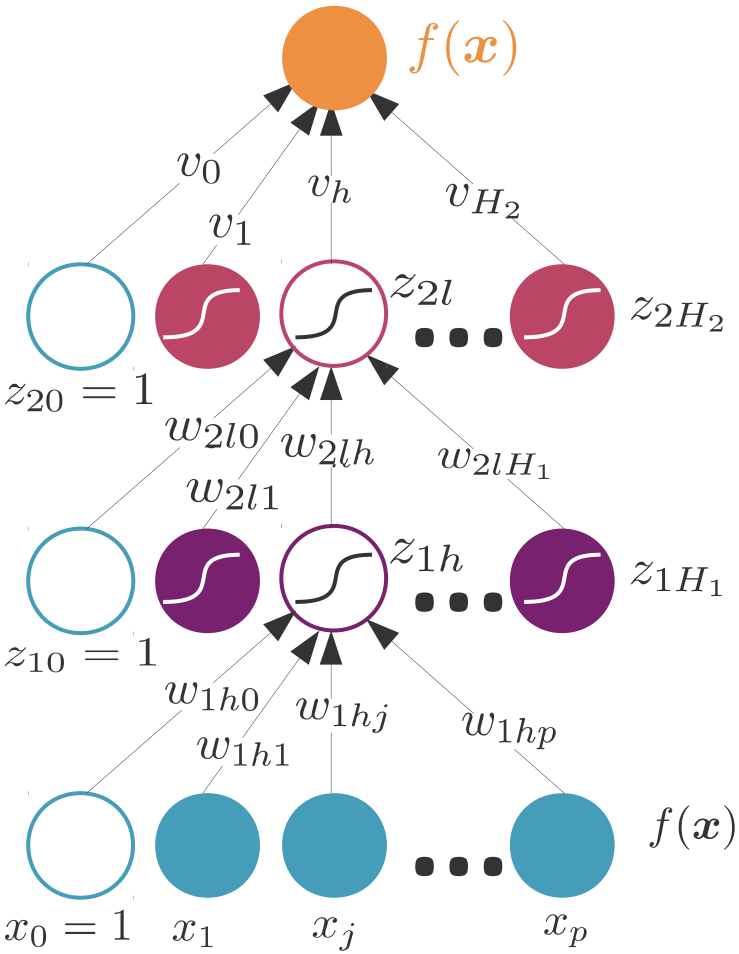
$$\Delta v_{kh} = \eta_1 \sum_{i=1}^{n} (y_k^i - f_k(\boldsymbol{x}^i)) z_h^i$$

$$\Delta w_{hj} = \eta \sum_{i=1}^{n} \left(\sum_{k=1}^{K} (y_k^i - f_k(\boldsymbol{x}^i)) v_{kh} \right) z_h^i (1 - z_h^i) x_j^i$$



Multiple hidden layers

- The MLP with one hidden layer is a **universal approximator**
- But using **multiple layers** may lead to **simpler networks.**



$$f(\boldsymbol{x}) = v_0 + \sum_{l=1}^{H_2} v_l z_{2l}$$

$$\boldsymbol{z}_{2l} = \sigma(\boldsymbol{w}_{2l}{}^\top \boldsymbol{z}_1) = \sigma\left(w_{2l0} + \sum_{h=1}^{H_1} w_{2lh} z_{1h}\right)$$

$$z_{1h} = \sigma(\boldsymbol{w}_{1h}{}^\top \boldsymbol{x}) = \sigma\left(w_{1h0} + \sum_{j=1}^{p} w_{1hj} x_j\right)$$

$$f(\boldsymbol{x}) = v_0 + \sum_{l=1}^{H_2} v_l \sigma\left(w_{2l0} + \sum_{h=1}^{H_1} w_{2lh}.\sigma\left(w_{1h0} + \sum_{j=1}^{p} w_{1hj} x_j\right)\right)$$

Deep learning

- Multi-layer perceptrons with “enough” layers are **deep feed-forward neural networks**
- Nothing more than a (possibly very complicated) **parametric model**!
- Coefficients are learned by **gradient descent**
  - **local minima**
  - **vanishing/exploding gradient**

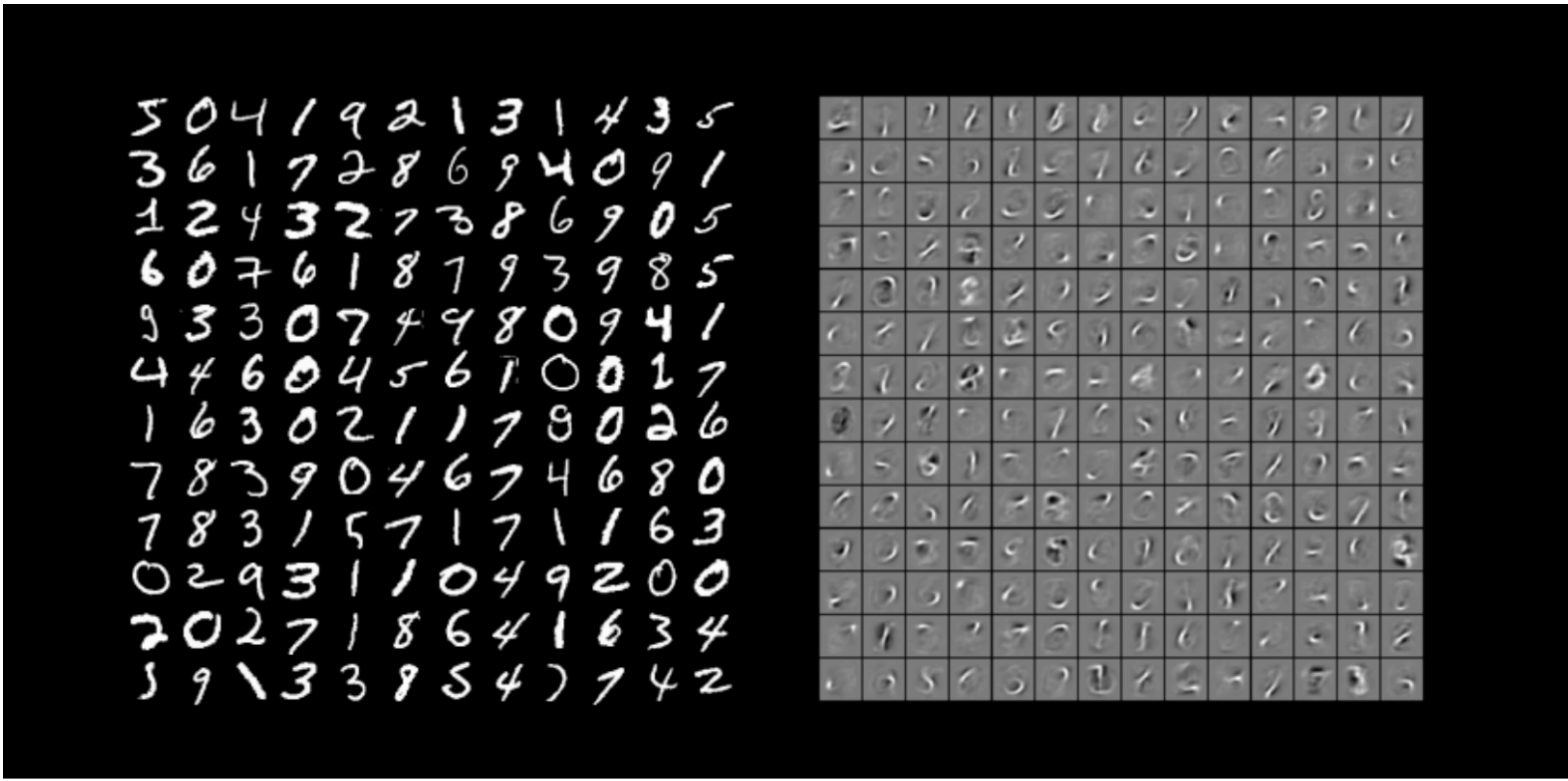
- Each layer learns a **new representation** of the data

⇨ **“representation learning”**

What makes deep networks hard to train? by Michael Nielsen
http://neuralnetworksanddeeplearning.com/chap5.html

(Deep) neural networks

Internal representation of the digits data

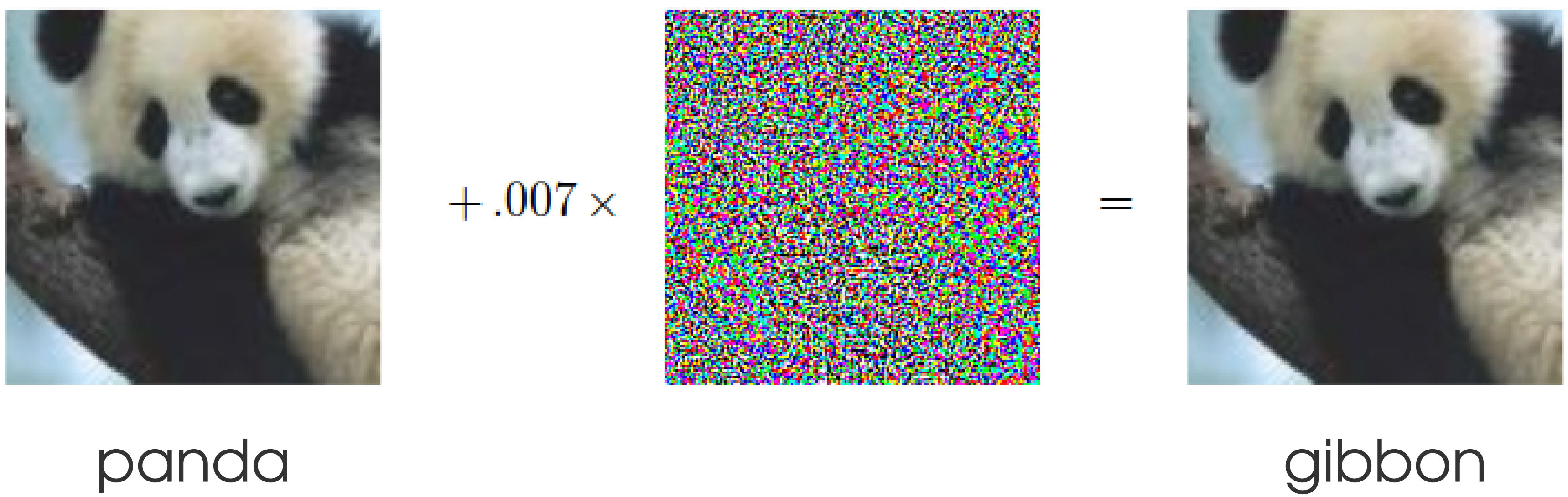


Yann Le Cun et al. (1990)

Puppy or bagel?

Photo credit: Karen Zack @teenybiscuit

Adversarial examples



Goodfellow et al. ICLR 2015
https://arxiv.org/pdf/1412.6572v3.pdf

Types of (deep) neural networks

- **Deep feed-forward** (= multilayer perceptrons)
- **Unsupervised networks**
  - **autoencoders** / **variational autoencoders** (VAE) — learn a new representation of the data
  - **deep belief networks** (DBNs) — model the distribution of the data but can add a supervised layer in the end
  - **generative adversarial networks** (GANs) — learn to separate real data from fake data they generate
- **Convolutional neural networks** (CNNs)
  - for image/audio modeling
- **Recurrent Neural Networks**
  - nodes are fed information from the previous layer and also from themselves (i.e. the past)
  - **long short-term memory networks** (LSTM) for sequence modeling.

Types of (deep) neural networks

- **Deep feed-forward** (= multilayer perceptrons)
- **Unsupervised networks**
  - **autoencoders** / **variational autoencoders** (VAE) — learn a new representation of the data
  - **deep belief networks** (DBNs) — model the distribution of the data but can add a supervised layer in the end
  - **generative adversarial networks** (GANs) — learn to separate real data from fake data they generate
- **Convolutional neural networks** (CNNs)
  - for image/audio modeling
- **Recurrent Neural Networks**
  - nodes are fed information from the previous layer and also from themselves (i.e. the past) ⇨ time series modeling
  - **long short-term memory networks** (LSTM) for sequence modeling.

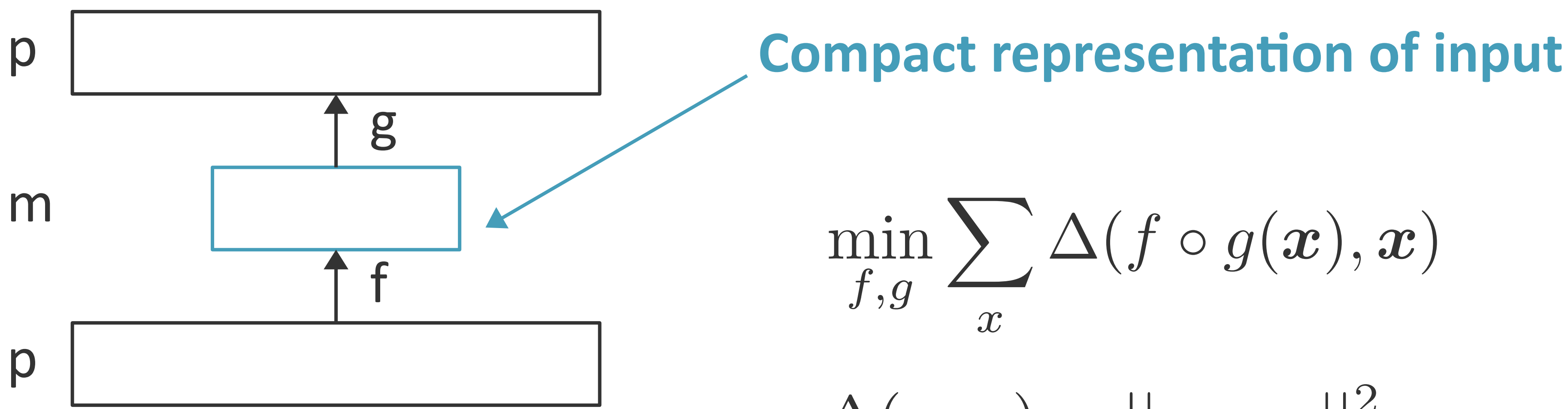
Feature extraction: Autoencoders

Autoencoders

- Dimensionality reduction with neural networks

  Rumelhart, Hinton & Williams (1986)

- **Goal:** output matches input

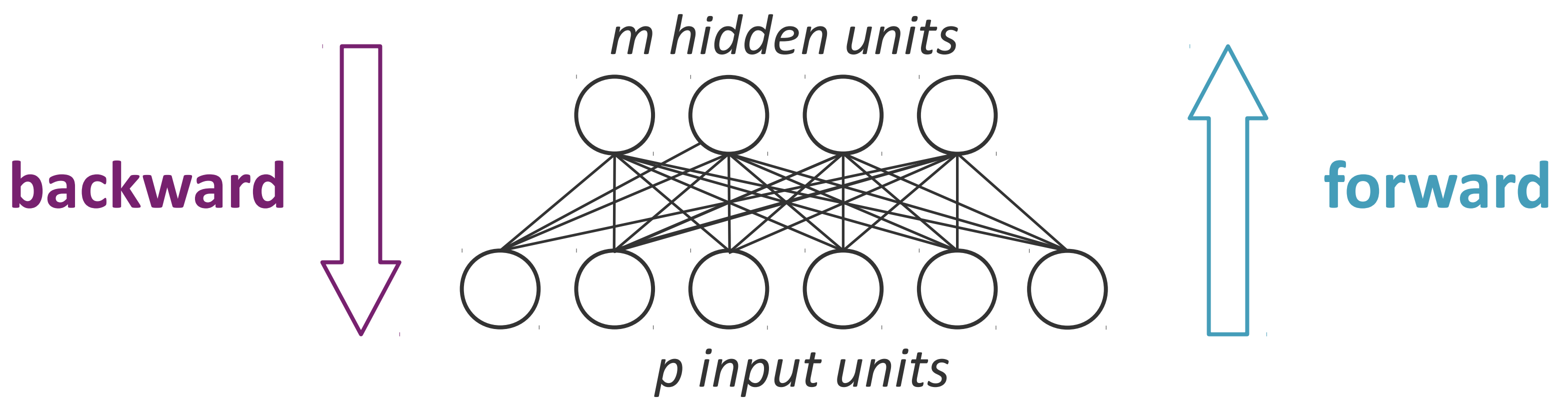


$$\min_{f,g} \sum_{x} \Delta(f \circ g(\boldsymbol{x}), \boldsymbol{x})$$

$$\Delta(\boldsymbol{y}, \boldsymbol{x}) = ||\boldsymbol{y} - \boldsymbol{x}||_2^2$$

Restricted Boltzmann Machines

- Boltzmann Machines Hinton & Sejnowsky (1985)
- RBM Smolensky (1986)



- **binary units** $x_j \in \{0,1\}, j=1,\ldots,p \quad z_h \in \{0,1\}, h=1,\ldots,H$
  (e.g. pixels in an image)
- **stochastic activation**

offset for visible unit j

$$P(x_j = 1|\boldsymbol{z}) = \sigma(a_j + \sum_{h=1}^{H} w_{jh} z_h)$$

offset for hidden unit h

connection weights

$$P(z_h = 1|\boldsymbol{x}) = \sigma(b_h + \sum_{j=1}^{p} w_{jh} x_j)$$

Restricted Boltzmann Machines

- **Restricted:**

  Boltzmann Machines are fully connected, here there are no connections between units of the same layer.

- **Boltzmann: energy-based probabilistic models**

  – **Energy** of the network:

$$E(\boldsymbol{x}, \boldsymbol{z}) = -\sum_{j=1}^{p} a_j x_j - \sum_{h=1}^{H} b_h z_h - \sum_{j=1}^{p} \sum_{h=1}^{H} x_j w_{jh} z_h$$

  Ising model (statistical physics):

  - nodes = sites
  - edges = adjacence
  - the network is a lattice
  - variables = magnetic spin (-1 or +1)

Restricted Boltzmann Machines

- **Restricted:**

  Boltzmann Machines are fully connected, here there are no connections between units of the same layer.

- **Boltzmann: energy-based probabilistic models**
  - **Energy** of the network:

$$E(\boldsymbol{x}, \boldsymbol{z}) = -\sum_{j=1}^{p} a_j x_j - \sum_{h=1}^{H} b_h z_h - \sum_{j=1}^{p} \sum_{h=1}^{H} x_j w_{jh} z_h$$

  - **Probability distribution**

$$P(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{Z} \boxed{e^{(-E(\boldsymbol{x}, \boldsymbol{z}))}}$$

Boltzmann factor

partition function = sum over all x and z of P(x, z)

$$Z = \sum_{\boldsymbol{x}, \boldsymbol{z}} e^{-E(\boldsymbol{x}, \boldsymbol{z})}$$

Restricted Boltzmann Machines

- **Restricted:**

  Boltzmann Machines are fully connected, here there are no connections between units of the same layer.

- **Boltzmann: energy-based probabilistic models**
  - **Energy** of the network:

$$E(\boldsymbol{x}, \boldsymbol{z}) = -\sum_{j=1}^{p} a_j x_j - \sum_{h=1}^{H} b_h z_h - \sum_{j=1}^{p} \sum_{h=1}^{H} x_j w_{jh} z_h$$

  - **Probability distribution**

$$P(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{Z} \boxed{e^{(-E(\boldsymbol{x},\boldsymbol{z}))}}$$

Boltzmann factor

partition function = sum over all x and z of P(x, z)

$$P(\boldsymbol{x}|\boldsymbol{z}) = \prod_{j=1}^{p} P(x_j|\boldsymbol{z})$$

$$P(x_j = 1|\boldsymbol{z}) = \frac{P(x_j = 1, \boldsymbol{z})}{P(x_j = 0, \boldsymbol{z}) + P(x_j = 1, \boldsymbol{z})} = \sigma(a_j + \sum_{h=1}^{H} w_{jh} z_h)$$

Restricted Boltzmann Machines

- **Restricted:**

  Boltzmann Machines are fully connected, here there are no connections between units of the same layer.

- **Boltzmann: energy-based probabilistic models**
  - **Energy** of the network:

$$E(\boldsymbol{x}, \boldsymbol{z}) = -\sum_{j=1}^{p} a_j x_j - \sum_{h=1}^{H} b_h z_h - \sum_{j=1}^{p} \sum_{h=1}^{H} x_j w_{jh} z_h$$

  - **Probability distribution**

$$P(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{Z} e^{(-E(\boldsymbol{x}, \boldsymbol{z}))}$$

  - Minimizing the energy of the network = minimizing the negative log likelihood of the observed data.
  - Connection to **Markov Random Fields.**

Restricted Boltzmann Machines

$$P(\boldsymbol{x},\boldsymbol{z}) = \frac{1}{Z}e^{(-E(\boldsymbol{x},\boldsymbol{z}))} \qquad Z = \sum_{\boldsymbol{x},\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})} \qquad P(\boldsymbol{z}|\boldsymbol{x}) = \frac{P(\boldsymbol{x},\boldsymbol{z})}{P(\boldsymbol{x})} = \frac{e^{-E(\boldsymbol{x},\boldsymbol{z})}}{\sum_{\boldsymbol{z}'} e^{-E(\boldsymbol{x},\boldsymbol{z}')}}$$

$$E(\boldsymbol{x},\boldsymbol{z}) = -\sum_{j=1}^{p} a_j x_j - \sum_{h=1}^{H} b_h z_h - \sum_{j=1}^{p}\sum_{h=1}^{H} x_j w_{jh} z_h$$

Gradient of the negative log likelihood:

$$P(\boldsymbol{x}) = \sum_{\boldsymbol{z}} P(\boldsymbol{x},\boldsymbol{z})$$

$$-\log P(\boldsymbol{x}) = \log Z - \log \sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})} = \log \sum_{\boldsymbol{x}',\boldsymbol{z}} e^{-E(\boldsymbol{x}',\boldsymbol{z})} - \log \sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}$$

$$\frac{\partial - \log P(\boldsymbol{x})}{\partial \theta} = -\frac{1}{\sum_{\boldsymbol{x},\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}} \sum_{\boldsymbol{x}',\boldsymbol{z}} \frac{\partial E(\boldsymbol{x}',\boldsymbol{z})}{\partial \theta} e^{-E(\boldsymbol{x}',\boldsymbol{z})} + \frac{1}{\sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}} \sum_{\boldsymbol{z}} \frac{\partial E(\boldsymbol{x},\boldsymbol{z})}{\partial \theta} e^{-E(\boldsymbol{x},\boldsymbol{z})}$$

$$= \underbrace{-\sum_{\boldsymbol{x}',\boldsymbol{z}} P(\boldsymbol{x}',\boldsymbol{z}) \frac{\partial E(\boldsymbol{x}',\boldsymbol{z})}{\partial \theta}}_{\text{negative gradient}} + \underbrace{\sum_{\boldsymbol{z}} P(\boldsymbol{z}|\boldsymbol{x}) \frac{\partial E(\boldsymbol{x},\boldsymbol{z})}{\partial \theta}}_{\text{positive gradient}}$$

Restricted Boltzmann Machines

$$P(\boldsymbol{x},\boldsymbol{z}) = \frac{1}{Z} e^{(-E(\boldsymbol{x},\boldsymbol{z}))} \qquad Z = \sum_{\boldsymbol{x},\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})} \qquad P(\boldsymbol{z}|\boldsymbol{x}) = \frac{P(\boldsymbol{x},\boldsymbol{z})}{P(\boldsymbol{x})} = \frac{e^{-E(\boldsymbol{x},\boldsymbol{z})}}{\sum_{\boldsymbol{z}'} e^{-E(\boldsymbol{x},\boldsymbol{z}')}}$$

$$\boxed{E(\boldsymbol{x},\boldsymbol{z}) = -\sum_{j=1}^{p} a_j x_j - \sum_{h=1}^{H} b_h z_h - \sum_{j=1}^{p}\sum_{h=1}^{H} x_j w_{jh} z_h}$$

Gradient of the negative log likelihood:

$$P(\boldsymbol{x}) = \sum_{\boldsymbol{z}} P(\boldsymbol{x},\boldsymbol{z})$$

$$-\log P(\boldsymbol{x}) = \log Z - \log \sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})} = \log \sum_{\boldsymbol{x}',\boldsymbol{z}} e^{-E(\boldsymbol{x}',\boldsymbol{z})} - \log \sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}$$

$$\frac{\partial - \log P(\boldsymbol{x})}{\partial \theta} = -\frac{1}{\sum_{\boldsymbol{x},\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}} \sum_{\boldsymbol{x}',\boldsymbol{z}} \frac{\partial E(\boldsymbol{x}',\boldsymbol{z})}{\partial \theta} e^{-E(\boldsymbol{x}',\boldsymbol{z})} + \frac{1}{\sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}} \sum_{\boldsymbol{z}} \frac{\partial E(\boldsymbol{x},\boldsymbol{z})}{\partial \theta} e^{-E(\boldsymbol{x},\boldsymbol{z})}$$

$$= -\sum_{\boldsymbol{x}',\boldsymbol{z}} P(\boldsymbol{x}',\boldsymbol{z}) \boxed{\frac{\partial E(\boldsymbol{x}',\boldsymbol{z})}{\partial \theta}} + \sum_{\boldsymbol{z}} P(\boldsymbol{z}|\boldsymbol{x}) \boxed{\frac{\partial E(\boldsymbol{x},\boldsymbol{z})}{\partial \theta}}$$

easy to compute

Restricted Boltzmann Machines

$$P(\boldsymbol{x},\boldsymbol{z}) = \frac{1}{Z} e^{(-E(\boldsymbol{x},\boldsymbol{z}))} \qquad Z = \sum_{\boldsymbol{x},\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})} \qquad P(\boldsymbol{z}|\boldsymbol{x}) = \frac{P(\boldsymbol{x},\boldsymbol{z})}{P(\boldsymbol{x})} = \frac{e^{-E(\boldsymbol{x},\boldsymbol{z})}}{\sum_{\boldsymbol{z}'} e^{-E(\boldsymbol{x},\boldsymbol{z}')}}$$

$$E(\boldsymbol{x},\boldsymbol{z}) = -\sum_{j=1}^{p} a_j x_j - \sum_{h=1}^{H} b_h z_h - \sum_{j=1}^{p}\sum_{h=1}^{H} x_j w_{jh} z_h$$

Gradient of the negative log likelihood:

$$P(\boldsymbol{x}) = \sum_{\boldsymbol{z}} P(\boldsymbol{x},\boldsymbol{z})$$

$$-\log P(\boldsymbol{x}) = \log Z - \log \sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})} = \log \sum_{\boldsymbol{x}',\boldsymbol{z}} e^{-E(\boldsymbol{x}',\boldsymbol{z})} - \log \sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}$$

$$\frac{\partial -\log P(\boldsymbol{x})}{\partial \theta} = -\frac{1}{\sum_{\boldsymbol{x},\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}} \sum_{\boldsymbol{x}',\boldsymbol{z}} \frac{\partial E(\boldsymbol{x}',\boldsymbol{z})}{\partial \theta} e^{-E(\boldsymbol{x}',\boldsymbol{z})} + \frac{1}{\sum_{\boldsymbol{z}} e^{-E(\boldsymbol{x},\boldsymbol{z})}} \sum_{\boldsymbol{z}} \frac{\partial E(\boldsymbol{x},\boldsymbol{z})}{\partial \theta} e^{-E(\boldsymbol{x},\boldsymbol{z})}$$

$$= -\sum_{\boldsymbol{x}',\boldsymbol{z}} P(\boldsymbol{x}',\boldsymbol{z}) \frac{\partial E(\boldsymbol{x}',\boldsymbol{z})}{\partial \theta} + \sum_{\boldsymbol{z}} P(\boldsymbol{z}|\boldsymbol{x}) \frac{\partial E(\boldsymbol{x},\boldsymbol{z})}{\partial \theta}$$

Gibbs sampling

approximation: replace
expectation with a single sample! $\approx \frac{\partial E(\boldsymbol{x},\boldsymbol{z})}{\partial \theta} - \frac{\partial E(\boldsymbol{x}',\boldsymbol{z}')}{\partial \theta}$

Restricted Boltzmann Machines

- Training procedure: **Contrastive Divergence**

  For a training sample $\boldsymbol{x}^i$

  – Compute $P(\boldsymbol{z}|\boldsymbol{x}^i)$

  – Sample a hidden activation vector $\boldsymbol{z}^i$

  – **positive gradient** = $\left(x_j^i z_h^i\right)_{j,h}$

  – Compute $P(\boldsymbol{x}|\boldsymbol{z}^i)$

  – Sample a reconstruction vector $\boldsymbol{x}'$

  – Compute $P(\boldsymbol{z}|\boldsymbol{x}')$ and sample a hidden activation vector $\boldsymbol{z}'$

  – **negative gradient** = $\left(x_j' z_h'\right)_{j,h}$

  – update weights: $\boxed{w_{jh} \leftarrow w_{jh} + \eta(x_j^i z_h^i - x_j' z_h')}$

  $a_j \leftarrow a_j - \eta(x_j^i - x_j')$ $\qquad$ $b_h \leftarrow b_h - \eta(z_h^i - z_j')$

Deep Belief Networks

- Stack multiple layers of RBM

G. E. Hinton & R. R. Salakhutdinov. *Reducing the dimensionality of data with neural networks.* (2006).

Neural network magic: How to train your (feed-forward) neural network

Architecture

- Start with one hidden layer.
- Stop adding layers when you **overfit.**
- Never use more weights than training samples.
- **Weight sharing:**

  Different units have connections to different inputs but sharing the same weights.

  E.g. Image analyses, looking for edges in different regions of space

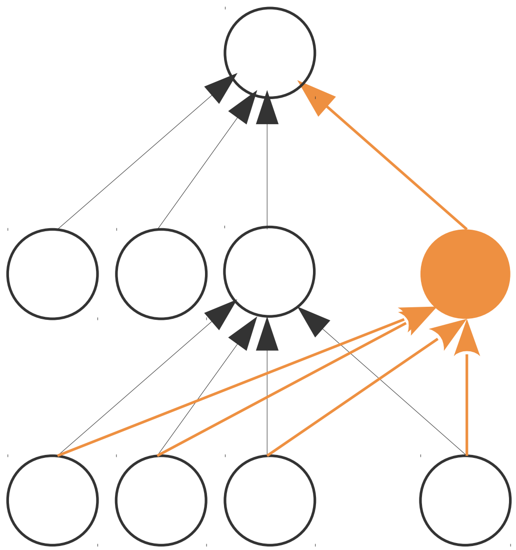
Tuning the network size

- **Destructive:**

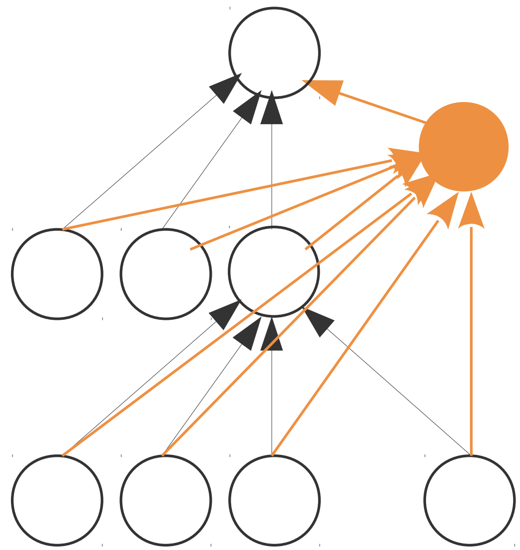
  weight decay $$\Delta w_j = -\eta \frac{\partial E}{\partial w_j} - \lambda w_j \qquad E' = E + \frac{\lambda}{2} \sum_j w_j^2$$

- **Constructive:**

  Growing networks until satisfactory error rate is reached.



Dynamic node creation [Ash 1989]:
Add new hidden units

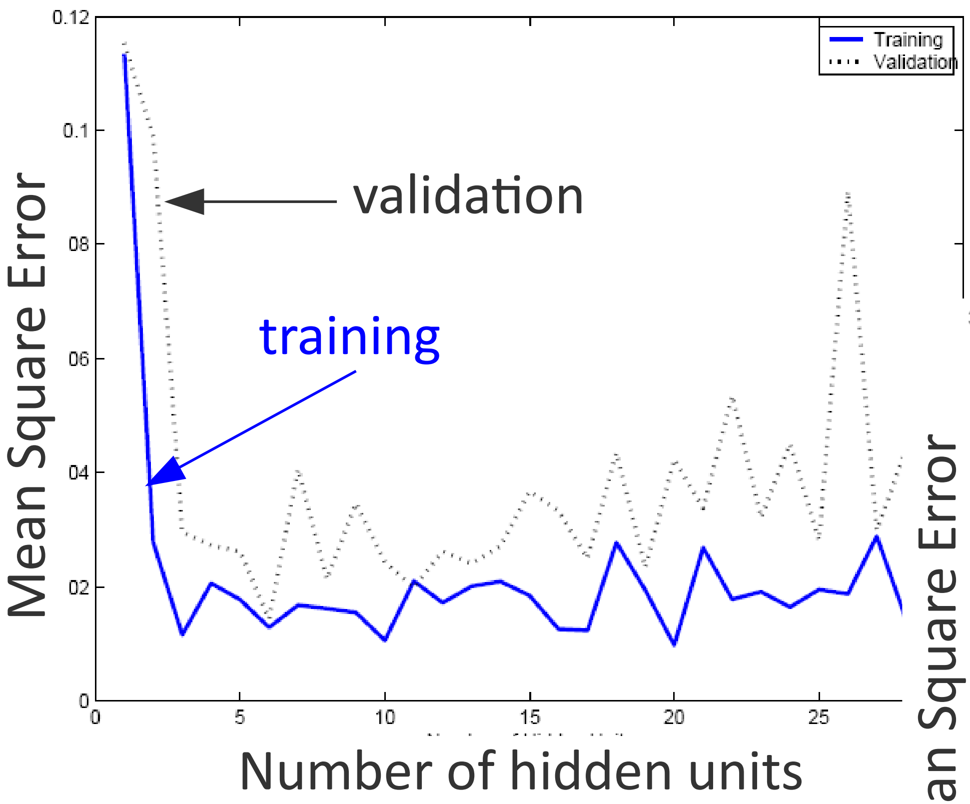


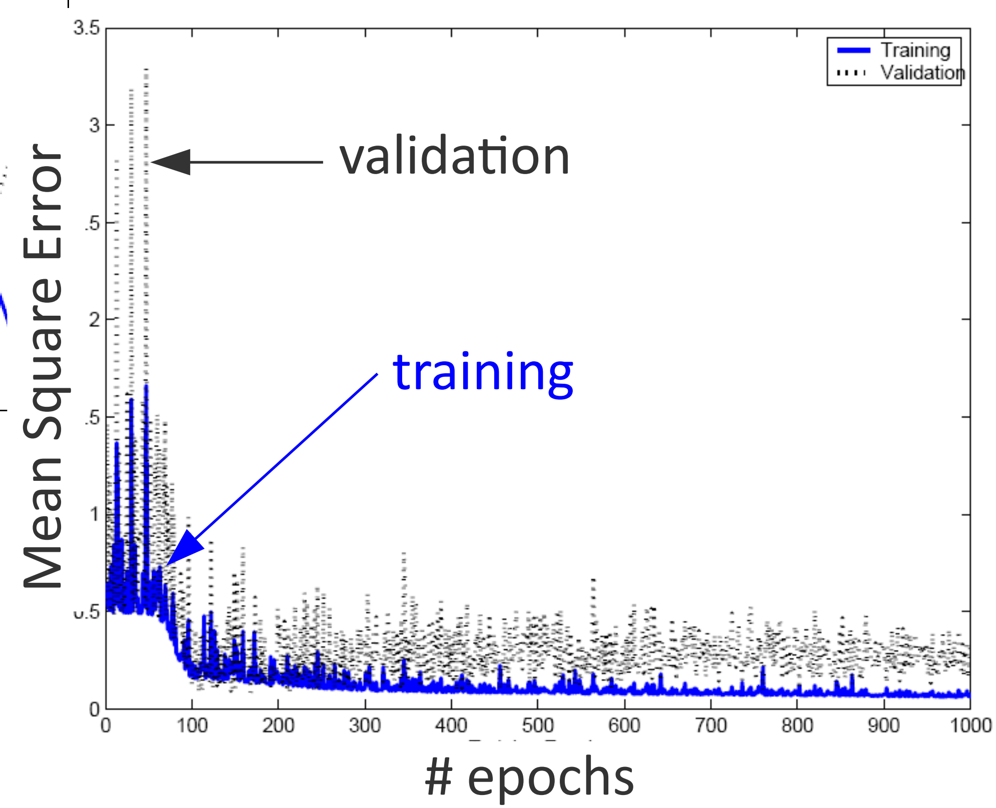
Cascade correlation
[Fahlman & Lebiere 1989]:
Add new hidden layers with one unit.

Overtraining

Number of weights: H(p+1) + (H+1).K

hidden units





Source: Ethem Alpaydin

Optimization algorithm

- **Batch learning:**

  Update the weights after a complete pass over the training set.

- **Mini-batch learning:**
  - Update the weights after a pass over a set of training points of fixed size.

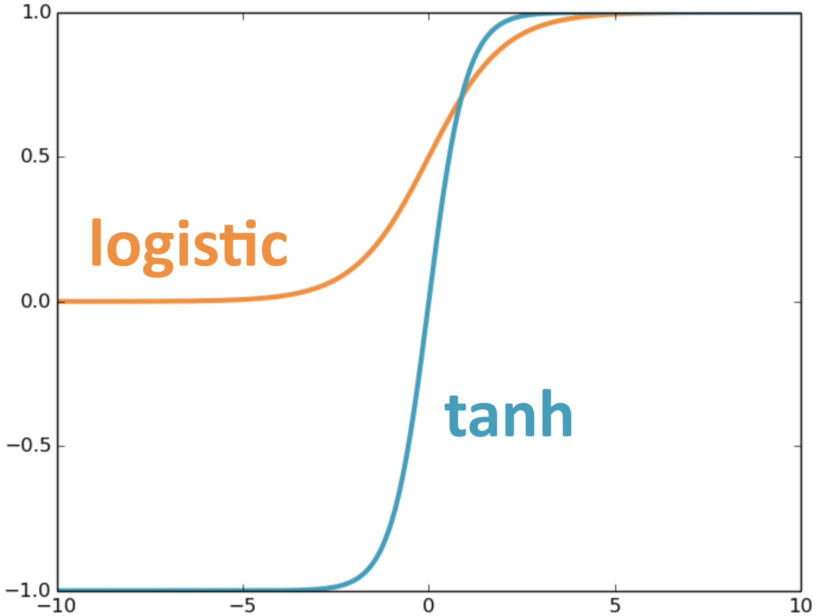
- Use **(quasi-)Newton methods** for small number of weights.

  Prefer Levenberg-Marquardt.

- Use **conjugate gradient descent** for large number of weights.

Preconditionning

- An **ill-conditionned** network cannot learn.
- The best **learning rate** is typically **different for each weight.**
- Hence
  - **Standardize** inputs and targets
  - **Initialize** weights carefully
  - **Local learning rates**



  - Use **tanh** rather than a logistic sigmoid for hidden layers so as to avoid low coefficients of variation (stdev/mean).

Standardization

- Remove **outliers**
- **Features**
  - Mean 0, standard deviation 1
  - Midrange 0, range 2
  - Orthonormalize (SVD, PCs...)
- **Targets**
  - Mean 0, standard deviation 1
  - Midrange 0, range 2

    Use lower/upper bounds rather than min/max

Escaping saturation

- Large weights ⇒ saturation
- **Weight initialization**

$$w_{ij} = [-r, r] \quad r = \frac{1}{\sqrt{|\mathcal{N}(i)|}}$$

- **Weight decay** ≡ regularization

  E → E + weight decay

$$E' = E + \frac{\lambda}{2} \sum_j w_j^2 \qquad \Delta w_j = -\eta \frac{\partial E}{\partial w_j} - \lambda w_j$$

$$E' = E + \sum_j \frac{w_j^2}{w_j^2 + \mathrm{Cte}}$$

Escaping local minima

- **Online learning or mini-batch**
- **Momentum**

$$w_j \leftarrow w_j + \eta \Delta w_j + m \Delta^{(t+1)} w_j$$

- **Adaptive learning rate**

$$w_j \leftarrow w_j + \mu_j \Delta w_j$$

  - μ↗ while the gradient keeps pointing in the same direction

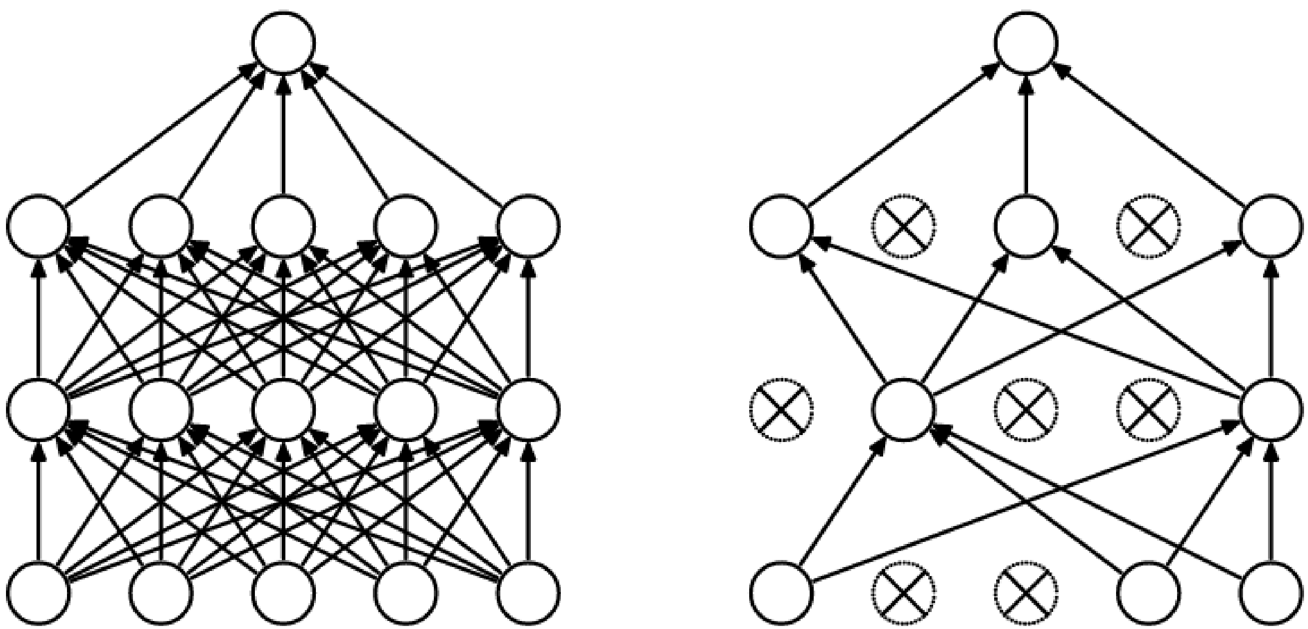
$$\mu_j \leftarrow \mu_j + q \Delta^{(t)} w_j \Delta^{(t-1)} w_j$$

  - Prevent $\mu_j < 0$: apply to $\log(\mu_j)$ instead
  - Approximate to avoid computing the exp and avoid too small values for μ_j

$$\mu_j \leftarrow \mu_j \times \max(0.5, 1 + q \Delta^{(t)} w_j \Delta^{(t-1)} w_j)$$

Dropout

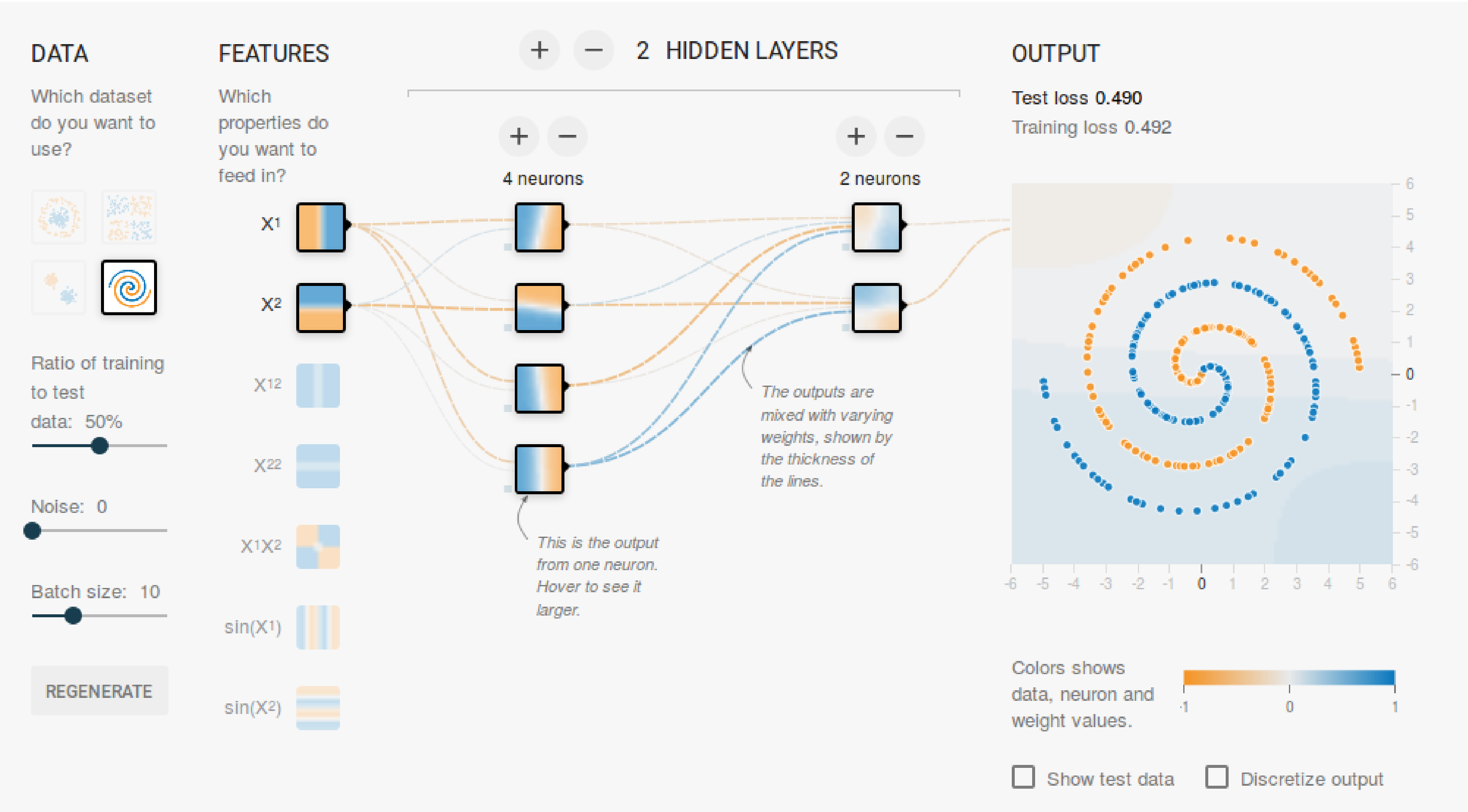
- At each iteration, **set half the units (randomly) to 0.**
- Avoid **overfitting**
- Helps focusing on **informative features**



(Srivastava, Hinton, Krizhevsky, Sutskever & Salakhutdinov 2012)

Playing with a neural network

http://playground.tensorflow.org/



Neural networks packages

http://deeplearning.net/software_links/

- Python

  Theano, **TensorFlow**, Caffe, Keras...

- Java

  Deeplearning4j, TensorFlow for Java

- Matlab

  NeuralNetwork toolbox

- R

  deepnet, H2O, MXNetR

References

- *A Course in Machine Learning.* `http://ciml.info/dl/v0_99/ciml-v0_99-all.pdf`
  - **Perceptron:** Chap 4
  - **Multi-layer perceptron:** Chap 10.1 – 10.4
- Deep learning references
  - Le Cun, Y., Bengio, Y. and Hinton, G. (2015). **Deep learning**. *Nature* 521, 436-444.
  - `http://neuralnetworksanddeeplearning.com`
  - `http://deeplearning.net/`
- Playing with a (deep) neural network
  - `http://playground.tensorflow.org/`

Summary

- **Perceptrons** learn linear discriminants.
- Learning is done by **weight update.**
- **Multiple layer perceptrons** with one hidden unit are **universal approximators.**
- Learning is done by **backpropagation.**

- Neural networks are **hard to train**, caution must be applied.
- (Deep) neural networks can be very powerful!

Exam: Fri, Dec 22 8:30am–11:30am

- **No documents, no calculators, no computer.**
- **Theoretical, technical,** and **practical** questions
- **Short answers!**
- **How to study**
  - Homework + previous year's exams
  - Labs
  - Answer the questions on the slides
  - **Review session Dec. 15 (15:30-17:00):** ask your questions!
- **Formulas**
  - To know: **Bayes,** how to compute derivatives.
  - Everything else will be given. **Interpretation** is key.

kaggle challenge project

- **Detailed instructions on the course website**
  http://cazencott.info/dotclear/public/lectures/ma2823_2017/kaggle-project.pdf
- **Deadline for submissions & for the report:**

  Sat, Dec 23 at 23:59 http://tinyurl.com/ma2823-2017-hw/
- **Report: Practical instructions:**

– PDF document
  - No more than 2 pages
  - Project_<LastName1><Initial>_<Lastname2><Initial>_<Lastname3><Initial>.pdf

– Starts with
  - Full names
  - Kaggle user names
  - Kaggle team name(s)