Neural networks

AI 2025/2026

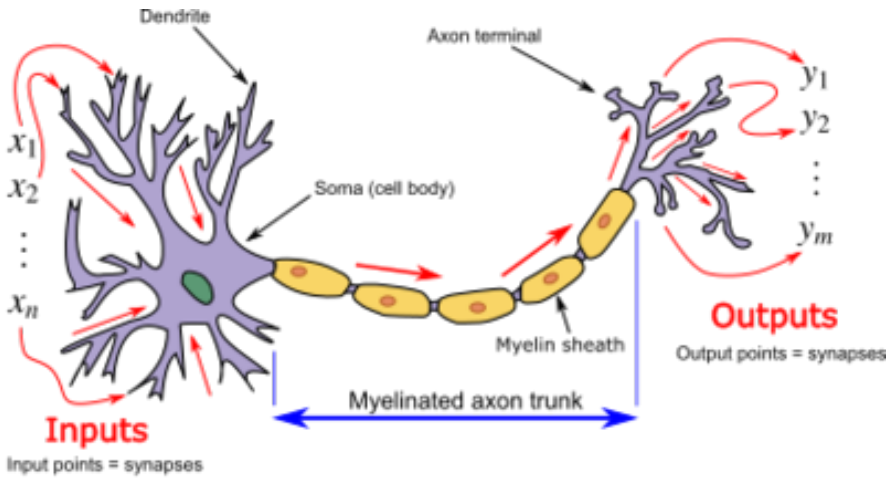
Content

History

- McCulloch&Pitts '43 propose the first mathematical model of an artificial neuron.
  It cannot learn, the parameters are set analytically.
- Minsky '51 - the first electronic circuit built as an artificial neural network (subcircuits that work like interconnected neurons)
- Rosenblatt '58 develops the perceptron, the first functional neural network
- Hinton '06 designs *Deep Neural Network*

Artificial neural networks

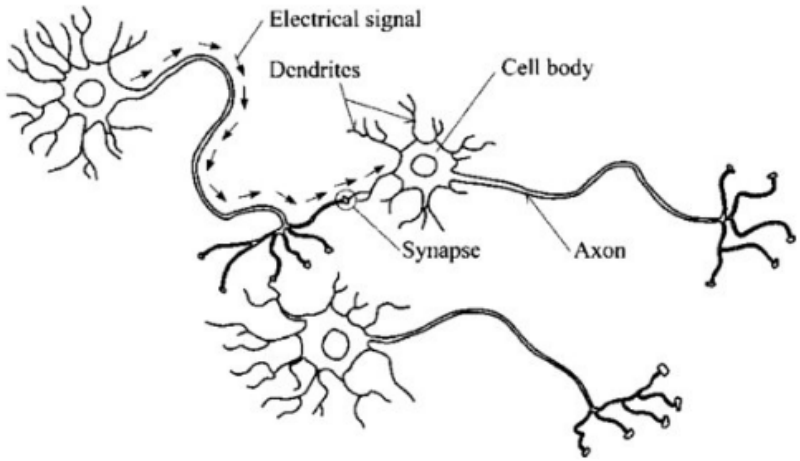
- Inspired by the way the brain is structured and the way it works



Try to reproduce the intelligence (the behavior of a biological neuron).

Artificial neural networks

A neuron connects to other neurons via dendrites. Neurons communicate with each other through synapses (excitatory or inhibitory). The neuron can activate and produce an electrical signal that is transmitted further along the axon.



The interconnection of neurons provides computing power.

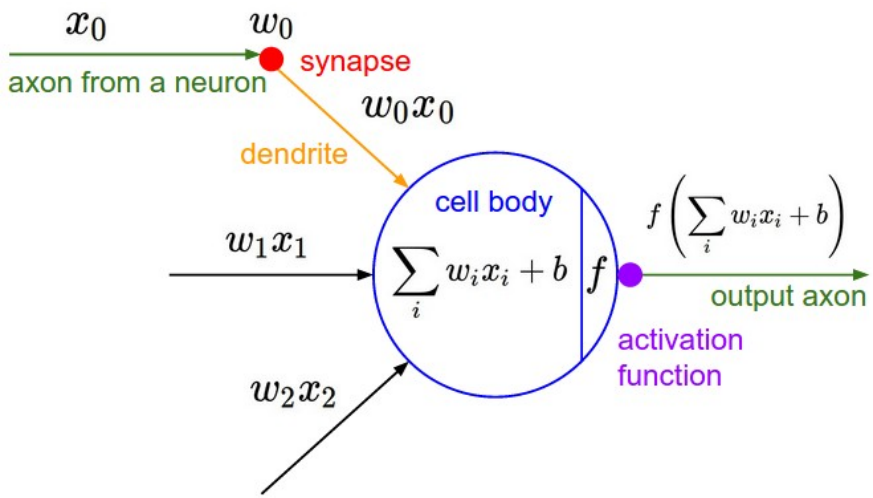
Artificial neural networks

Unit (artificial neuron): a simplified computational model of the neuron

- input signals
- weights attached to connections
- activation threshold
- output

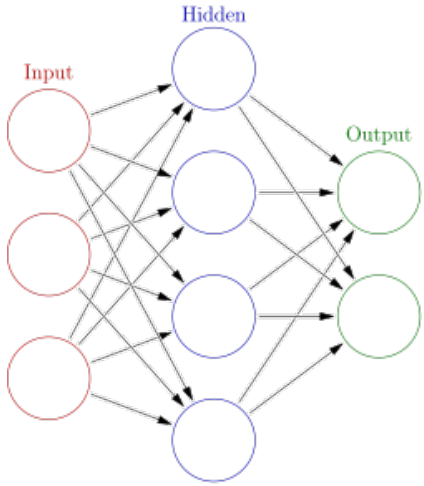
Analogy

Biological NN	Artificial NN
the cell body	neuron
dendrites	inputs
axon	output
synapse	weight



Artificial neural networks

- An ensemble of interconnected functional units (neurons).



- Training involves determining the parameters of the network, given training data

- Are adaptive "black box" systems that extract a pattern through a learning process

Learning methods

- Supervised (classification, regression)
  - Labeled training examples
  - Goal: estimate parameters that minimize the error (the difference btw the correct answers and those produced by the network)
- Unsupervised (clustering, association, dimensionality reduction)
  - Unlabeled training data
  - Goal: obtaining information

Applications: Classification

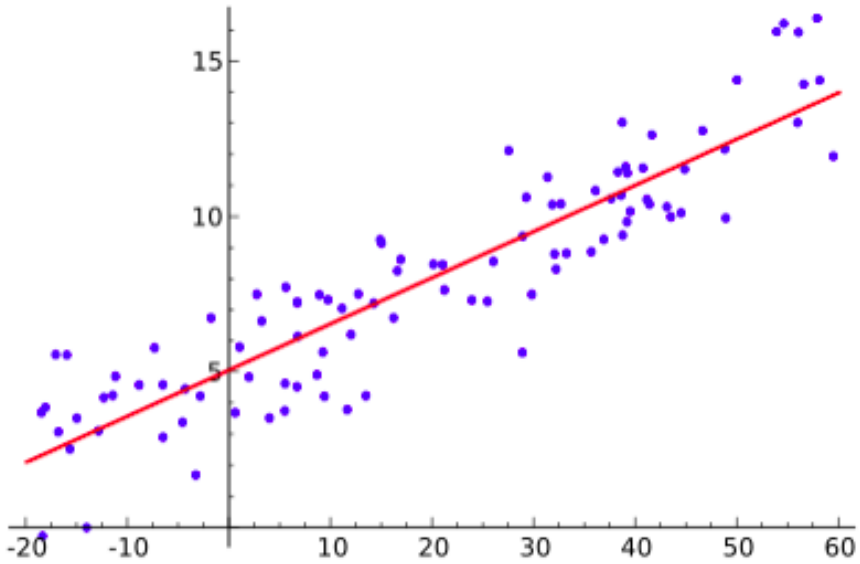
Given a set of instances (attributes, labels), identify the class of a new instance.
(supervised learning)

Example: identify the iris flower species

- attributes: sepal/petal length and width
- classes: *Iris versicolor*, *Iris setosa*, *Iris virginica*

Applications: Regression

Find the relationship between two or more variables, given a sequence of values (approximation of a function)

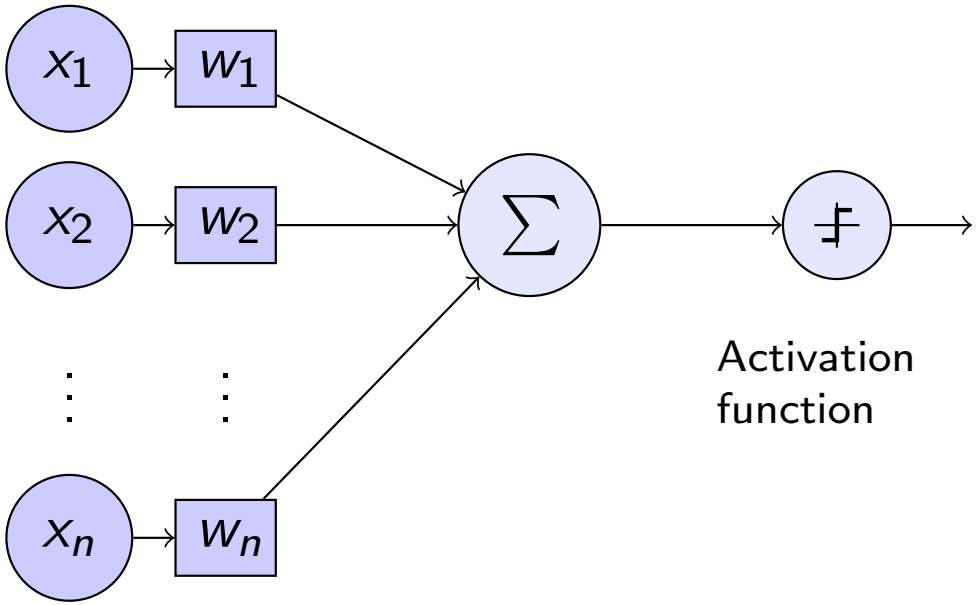


The difference btw classification and regression: the type of output (discrete vs. continuous)

Content

Perceptron (Rosenblatl, 1958)

Input: an array of real values x_i
Computes a linear combination of those values.



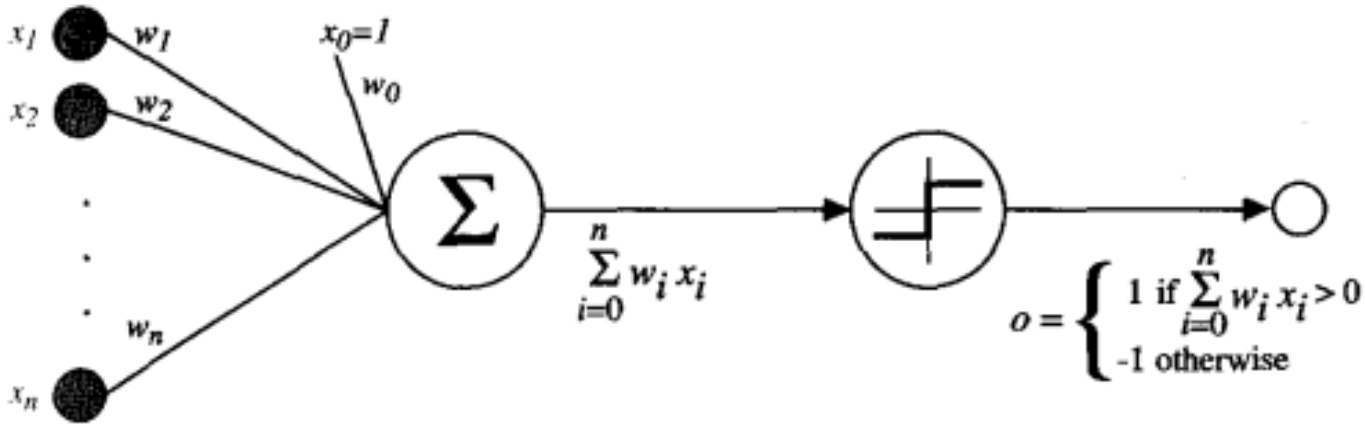
$w_1, \ldots w_n$ weights (real consts.) attached to connections; w_i the contribution of input x_i to the result.

Perceptron

Input: an array of real values x_i
Computes a linear combination of those values.
Returns 1, if the result is larger than a threshold $(-w_0)$, -1 otherwise.

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \ldots w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases} \quad (1)$$

w_0 bias



Learning a perceptron: identify weights $w_0, \ldots, w_n$.

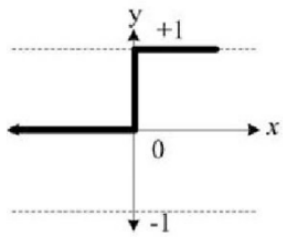
Perceptron

Simplified notation: a constant input $x_0 = 1$.
$\sum_{i=0}^{n} w_i x_i > 0$, or $\vec{w} \cdot \vec{x} > 0$.

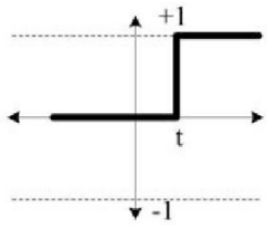
$$o(\vec{x}) = f(\vec{w} \cdot \vec{x})$$

Step activation function

$$f(y) = \begin{cases} 1 & \text{if } y \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



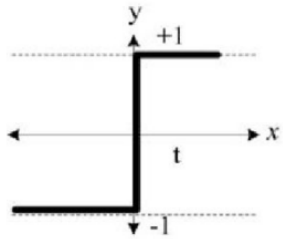
Step Function



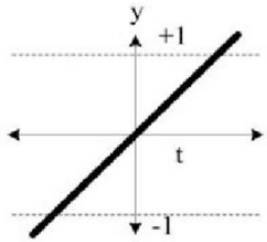
Step Function

Sign activation function sign

$$f(y) = \begin{cases} 1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$$



Sign Function



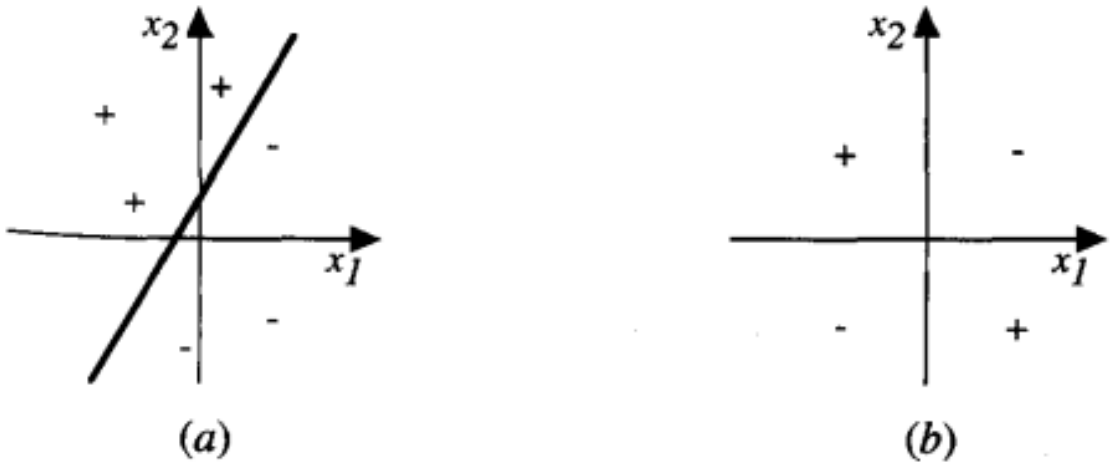
Linear Function

Perceptron: an artificial neuron using the unit step/sign function as the activation function.

The representational power of perceptrons

The goal of the perceptron: classify the inputs in 2 classes. Perceptron: a hyperplane that divides the space of input vectors (n-dimensional) in two regions: the region for which $\vec{w} \cdot \vec{x} > 0$, and the other region.

The equation of the hyperplane: $\vec{w} \cdot \vec{x} = 0$

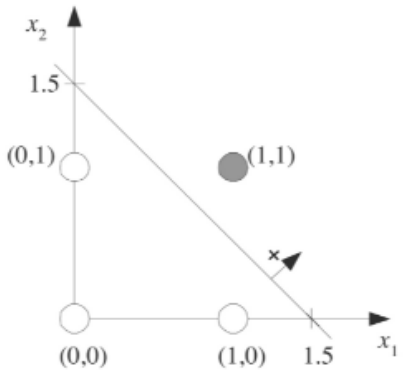


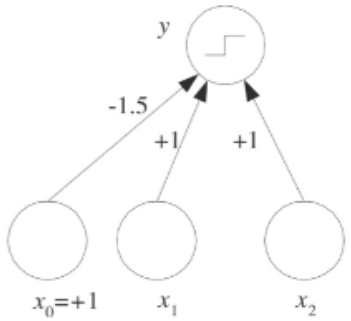
a) Linearly separable

The representational power of perceptrons

A perceptron can be used to represent boolean functions.
To represent the AND function, we set the weights, for ex.
$w_0 = -1.5, w_1 = w_2 = 1$.

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

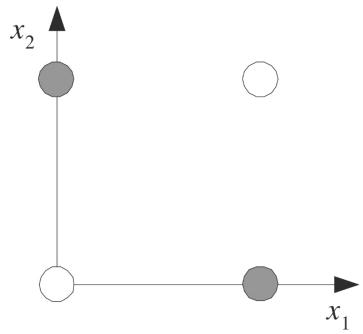




The representational power of perceptrons

The XOR function ($1 \Leftrightarrow x_1 \neq x_2$) cannot be represented by a single perceptron.

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0



Any boolean function can be represented by a network of interconnected units.

Content

Perceptron training rule

- Learning weights: identify the vector of weights s.t. the perceptron will return the correct output for each training example.

- Generate random weights.

  Compute the output for training inputs,
  modify the weights when it misclassifies an example.

  Repeat this process until the perceptron correctly classifies the training examples.

Perceptron training rule

The weights are modified according to the *perceptron training rule*:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

t the target output for the training example, o the output generated by the perceptron, η the *learning rate* (pos. const.)

Intuition for Perceptron training rule

- If the training example is correctly classified $t - o = 0; \Delta w_i = 0 \rightarrow$ no weights are updated
- If the perceptron outputs -1 when the target output is $+1$ and $\eta = 0.1, x_i = 0.8$, then $\Delta w_i = 0.1(1 - (-1))0.8 = 0.16$
- If the perceptron outputs $+1$ when the target output is -1, then the weight will decrease

Convergence

When the training examples are linearly separable and η small enough,

the perceptron training rule converges (considering a finite no. of applications of the perceptron training rule)

to a vector of weights that classifies all training examples.

Content

Delta rule

- The perceptron training rule may fail if the examples are not linearly separable.

- Delta rule: use *Gradient descent* to search in the space of weight vectors.

  Consider a linear unit for which the output is $o(\vec{x}) = \vec{w} \cdot \vec{x}$.

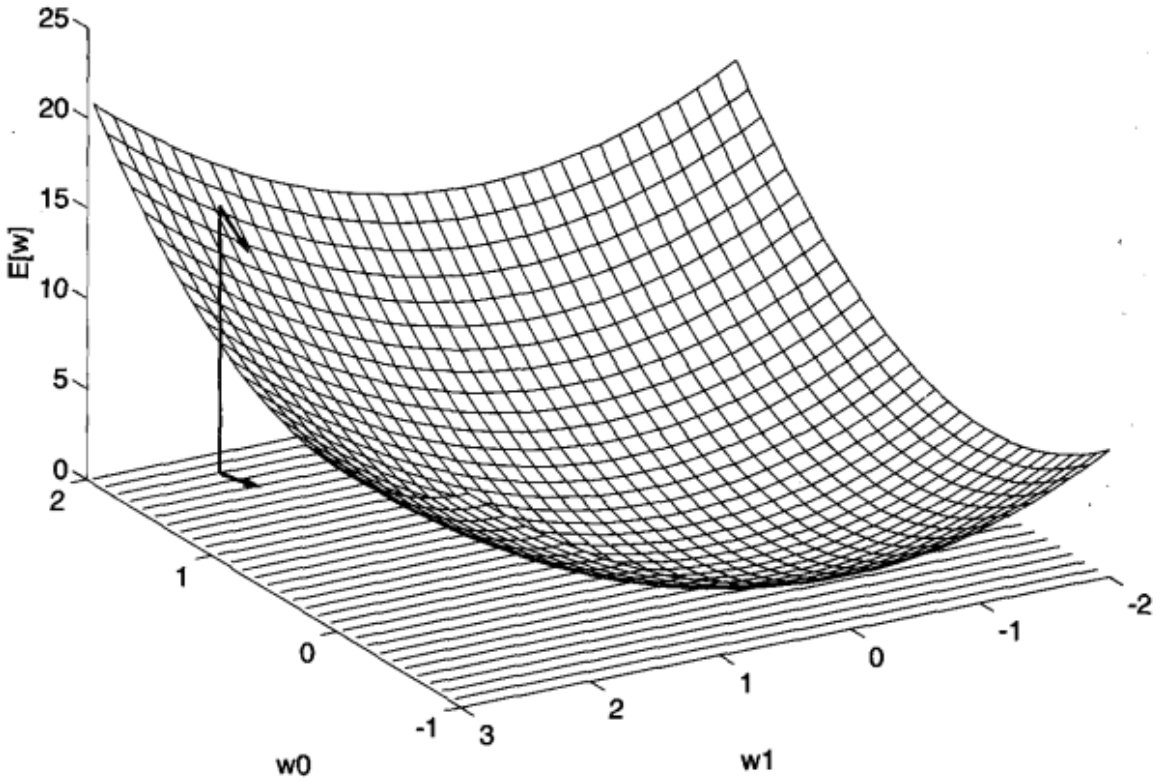
  The training error for a vector of weights w:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

  where D the training data set, t_d the target output for example d, o_d the output of the linear unit for d.

The hypothesis space visualization

The error surface has a parabolic shape, with a global minimum.



Gradient descent: iteratively modifies the vector of weights. At each step, the vector is modified in the direction that produces the steepest descent. The process continues until the global minimum error is reached.

Gradient descent

The gradient specifies the direction that produces the steepest ascent in E.

$$\nabla E(\overrightarrow{w}) = [\frac{\delta E}{\delta w_0}, \frac{\delta E}{\delta w_1}, \ldots, \frac{\delta E}{\delta w_n}]$$

The training rule for *Gradient descent*: $\overrightarrow{w} \leftarrow \overrightarrow{w} + \Delta\overrightarrow{w}$, where $\Delta\overrightarrow{w} = -\eta\nabla E(\overrightarrow{w})$, η the *learning rate* (pos. const.).

$$w_i = w_i + \Delta w_i, \quad \Delta w_i = -\eta\frac{\delta E}{\delta w_i}$$

Gradient descent

$$\frac{\delta E}{\delta w_i} = \frac{\delta}{\delta w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}\frac{\delta}{\delta w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\delta}{\delta w_i}(t_d - o_d)$$

$$= \sum_{d\in D}(t_d - o_d)\frac{\delta}{\delta w_i}(t_d - \vec{w}\cdot\vec{x_d})$$

$$\frac{\delta E}{\delta w_i} = \sum_{d\in D}(t_d - o_d)(-x_{id})$$

x_{id} the component x_i of the training example d.

Updating the weight with $\Delta w_i = \eta\sum_{d\in D}(t_d - o_d)x_{id}$.

Gradient descent

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \tag{T4.1}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

Stochastic gradient descent

Problems with the *Gradient descent* algorithm:

- slow convergence
- the existence of several local minima

Stochastic gradient descent: update the weights incrementally, by computing the error for each individual example

$$\Delta w_i = \eta(t - o)x_i$$

where t the target value, o the real output, x_i the ith input for the training example.

Equation T4.1 is replaced with $w_i \leftarrow w_i + \eta(t - o)x_i$.

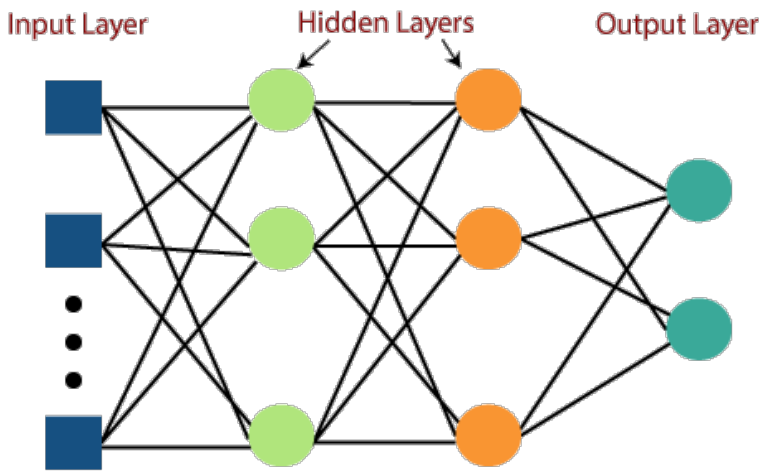
The training rule $\Delta w_i = \eta(t - o)x_i$ is also called the delta rule/LMS (*least-mean-square*) rule/Adaline rule.

Content

Multi-layer neural networks

A *feed-forward* neural network has

- an input layer
- one or more hidden layers
- an output layer

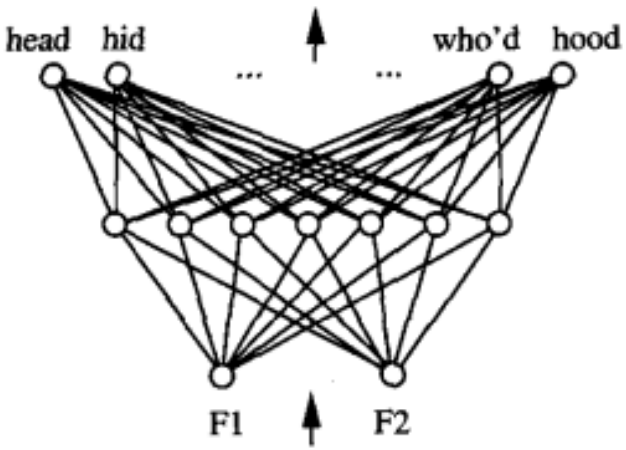


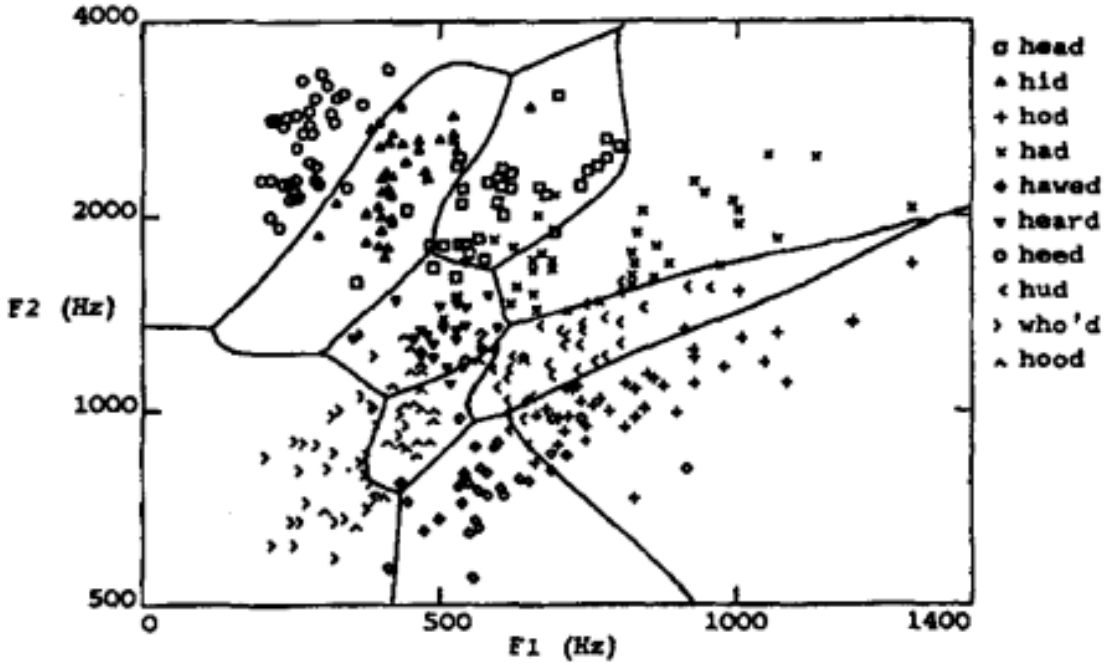
- Input signals are propagated forward through the network layers
- The calculations are performed in the neurons from hidden and output layer

Multi-layer neural networks

Can express nonlinear decision surfaces.

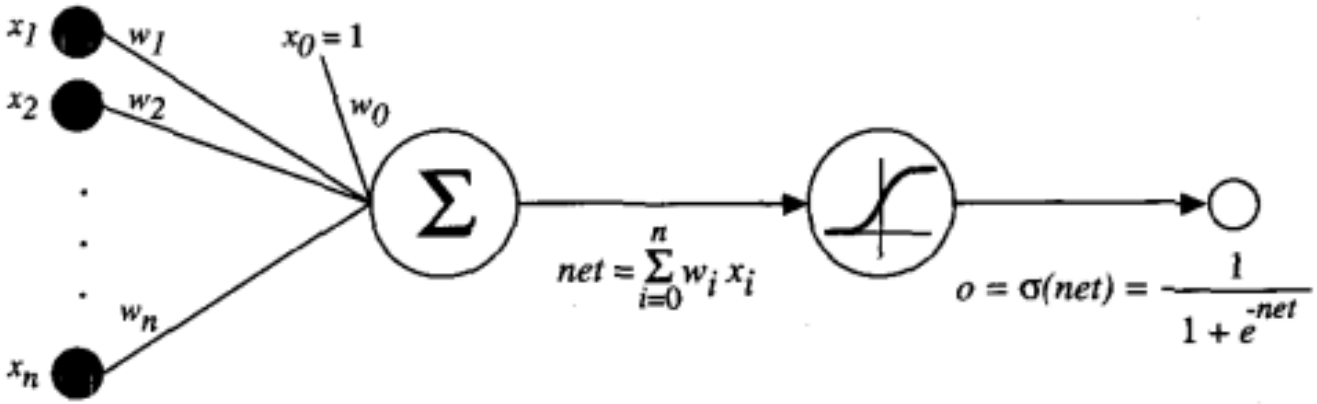
Example: Neural network trained to recognize btw 10 vowels (”h_d”).





The voice signal is represented by two numerical parameters, obtained by the spectral analysis of the sound. The points in the figure are testing data.

Sigmoid unit



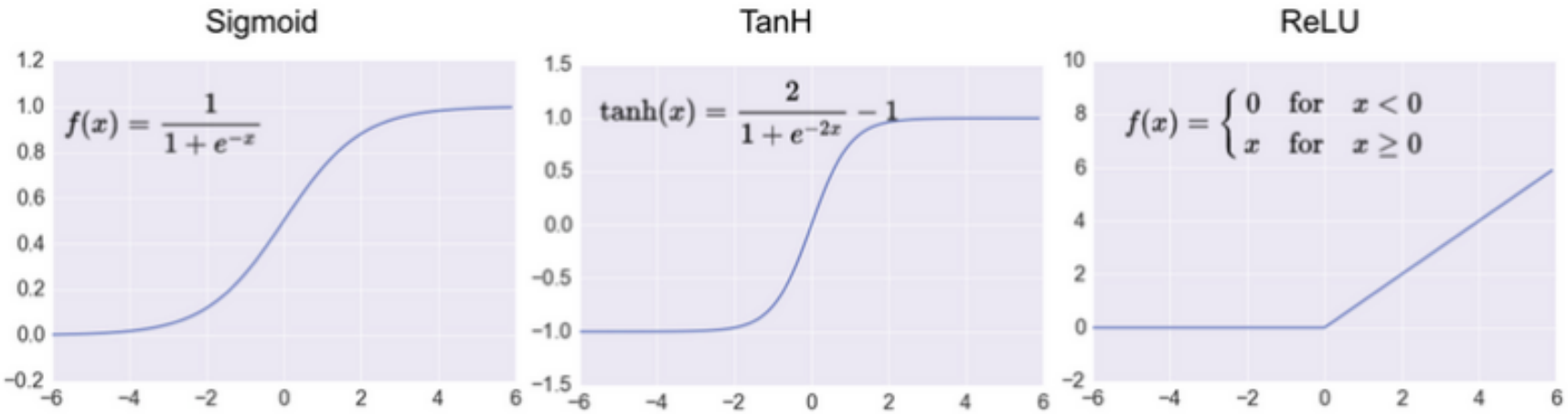
$$o = \sigma(\vec{w} \cdot \vec{x}), \quad \text{where } \sigma(y) = \frac{1}{1 + e^{-y}}$$

σ the sigmoid fct; derivative $\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$

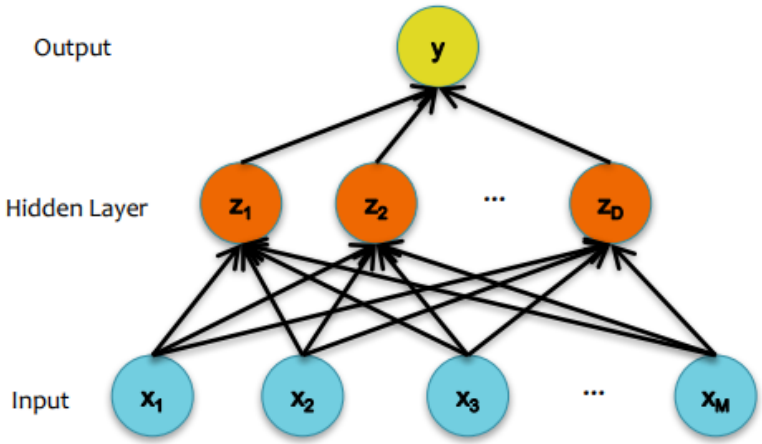
Nonlinear activation functions

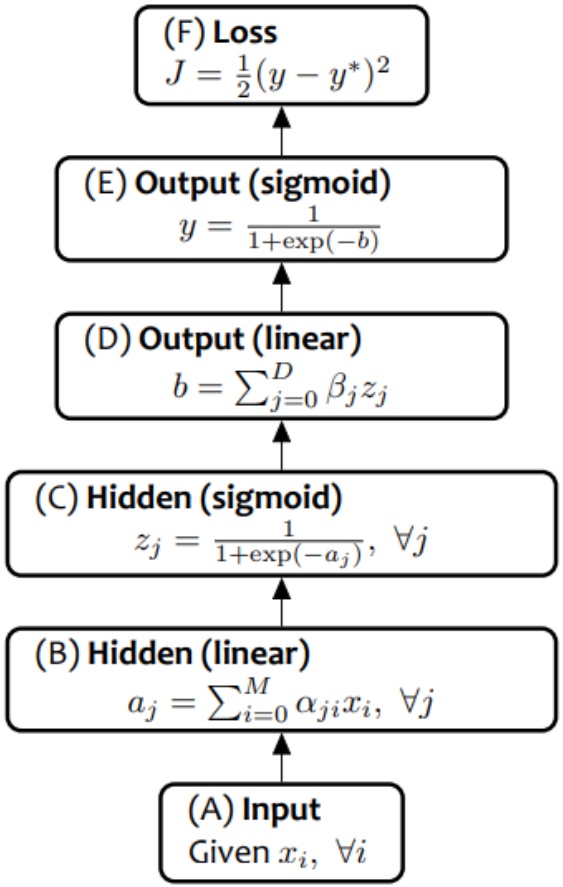
- Sigmoid function (logistic)
  $f(x) = \frac{1}{1+e^{-x}}, \quad f'(x) = f(x)(1 - f(x))$
- Bipolar sigmoid function (hyperbolic tangent)
  $f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}, \quad f'(x) = 1 - f(x)^2$
- ReLU (Rectified Linear Unit) function

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}, \quad f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



Neural network with sigmoid activation functions





Universal approximation property

- A neural network with a hidden layer, with a possible infinite number of neurons, can approximate any continuous real function
- An additional layer can reduce the number of neurons needed in the hidden layers
- A multi-layer perceptron with linear activation functions is equivalent with a single layer perceptron
  - a linear combination of linear functions is also a linear function
    ex: $f(x)=2x+1$, $g(y)=y-3$, $g(f(x))=(2x+1)-3=2x-2$

Content

Backpropagation algorithm

- Rumelhart, Hinton& Williams, '86
- Learn the weights in a multi-layer network. It uses *Gradient descent* to minimize the squared error between the network output and the target values.
- Since we have networks with multiple output units, we redefine E

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

where *outputs* the set of output units, t_{kd} and o_{kd} the target values respectively the output values associated with the output unit k and the training example d.

Backpropagation

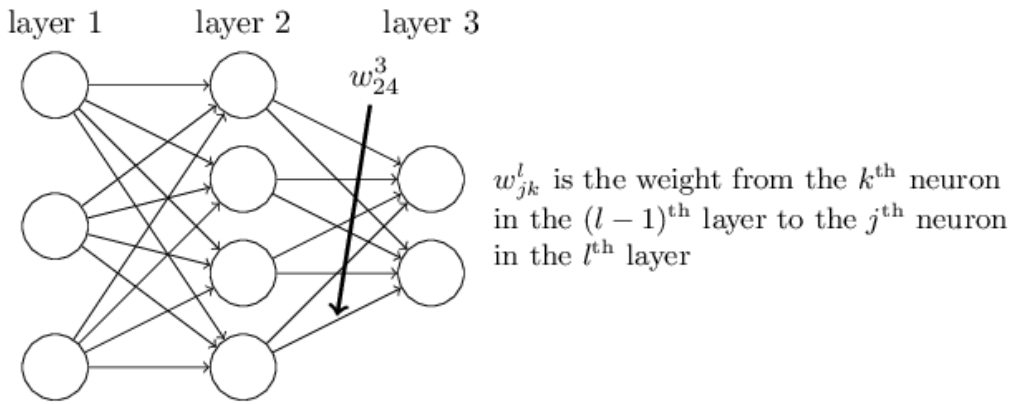
Has two phases:

- The network receives the input vector and propagates the signal forward, layer by layer, until the output is generated
- The error signal is propagated backwards, from the output layer to the input layer, by adjusting the network weights

Backpropagation

- Initialization: choose the number of inputs, hidden and output units; initialize the weights and the thresholds with small random values
  - in general, values in the range [-0.1, 0.1]
- Activation
  - the network is activated by applying the training data $\overrightarrow{x}$
  - compute the output of the neurons from the hidden layer
  - compute the output of the neurons from the output layer $o = \sigma(\overrightarrow{w} \cdot \overrightarrow{x})$

Backpropagation



- The output of the neurons from the hidden layer

$$o_h = \sigma\left(\sum_{i=0}^{n} w_{hi} x_i\right)$$

- The output of neurons from the output layer

$$o_k = \sigma\left(\sum_{i=0}^{m} w_{ki} o_i\right)$$

Backpropagation

Update the weights proportional to the learning rate η, the input value x_{ji} and the error δ_j.

- For the output neurons
  - compute the gradients of the neurons from the output layer
    For the output unit k,

$$\delta_k = (t_k - o_k)o_k(1 - o_k)$$

- For the neurons from the hidden layer
  - compute the gradients of the neurons from the hidden layer
    For the hidden unit h, sum the gradients δ_k for each output unit influenced by h, weighted by w_{kh} (the weight from the hidden layer h to the output layer k):

$$\delta_h = o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

Updating weights

For each training example d, add $w_{ji} = w_{ji} + \Delta w_{ji}$, $\Delta w_{ji} = -\eta \frac{\delta E_d}{\delta w_{ji}}$, where E_d is the error for the training example d.

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

- The weights of an output unit

$$\Delta w_{ji} = \eta \delta_j x_{ji}, \qquad \delta_j = (t_j - o_j) o_j (1 - o_j)$$

- The weights of a hidden neuron

$$\Delta w_{ji} = \eta \delta_j x_{ji}, \qquad \delta_j = o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

Backpropagation

For the *feed-forward* networks with an arbitrary number of layers,

$$\delta_r = o_r(1 - o_r) \sum_{s \in layer\ m+1} w_{sr}\delta_s$$

δ_r for the unit r from layer m is computed from the δ values from the next layer $m + 1$

Chain Rule

Backpropagation: a simple example

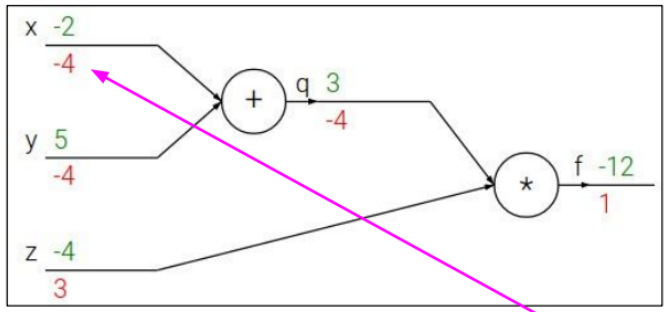
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial x}$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Derivation of the Backpropagation rule

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

Use chain rule:

$$\begin{aligned} \frac{\delta E_d}{\delta w_{ji}} &= \frac{\delta E_d}{\delta net_j} \frac{\delta net_j}{\delta w_{ji}} \\ &= \frac{\delta E_d}{\delta net_j} x_{ji} \end{aligned} \tag{2}$$

Derivation of the Backpropagation rule

Case 1: Training Rule for Output Unit Weights. Just as w_{ji} can influence the rest of the network only through net_j, net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \tag{4.23}$$

To begin, consider just the first term in Equation (4.23)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. We therefore drop the summation over output units and simply set $k = j$.

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j)\end{aligned} \tag{4.24}$$

Next consider the second term in Equation (4.23). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j)\end{aligned} \tag{4.25}$$

Substituting expressions (4.24) and (4.25) into (4.23), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j) \tag{4.26}$$

Derivation of the Backpropagation rule

Case 2: Training Rule for Hidden Unit Weights. In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by $Downstream(j)$. Notice that net_j can influence the network outputs (and therefore E_d) only through the units in $Downstream(j)$. Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)
\end{aligned} \tag{4.28}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

and

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

Stochastic Gradient Descent

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

Backpropagation

- Iterate over all training examples (vectors) (an epoch)
- The training of the network continues until the error falls below an acceptable threshold or until a predetermined maximum no. of training epochs is reached

Example: from *Artificial Intelligence. A Guide to Intelligent Systems*.

Backpropagation

- Convergence: the algorithm *Backpropagation converges to a local minimum*
- Incremental learning vs. *batch learning*
  - *batch learning*: the weights are updated once, after considering all training vectors from batch

    Advantage: the training results no longer depend on the order the training vectors are given

The "momentum" variant of *Backpropagation* algorithm

- The adjustment term for the weight from the current epoch is computed based on the error signal and the adjustments from the previous epoch

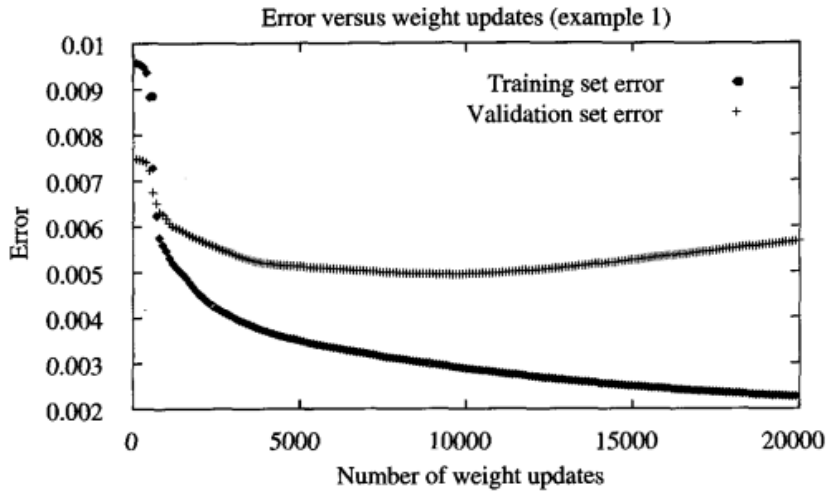
$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

where $\Delta w_{ji}(n)$ the updating weight in epoch n, $0 \leq \alpha < 1$ const. *momentum* (inertia)

- Stabilize the search

Why Momentum Really Works, `https://distill.pub/2017/momentum/`

Overfitting



Solutions

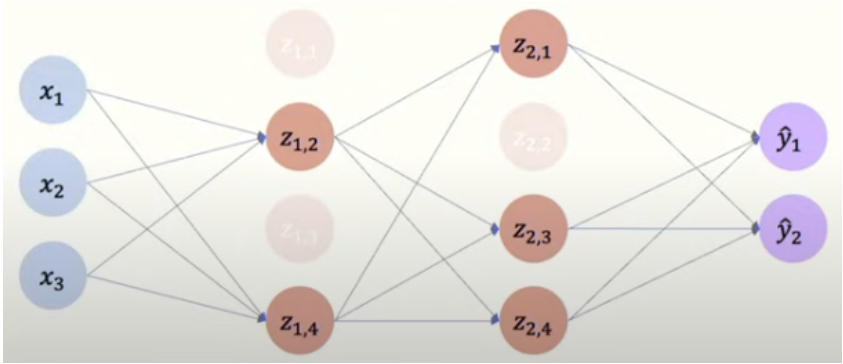
- *weight decay*: include a penalty term in the error function

$$E = E + \lambda \sum_{i,j} w_{ji}^2$$

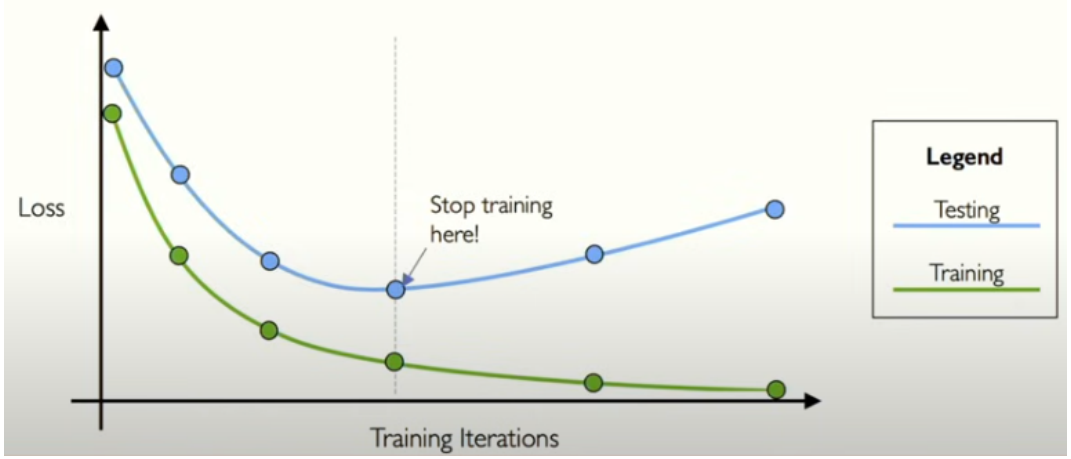
- using a validation set
  k-fold cross-validation

Regularization

- *Dropout*: during training, we randomly set activation functions to 0

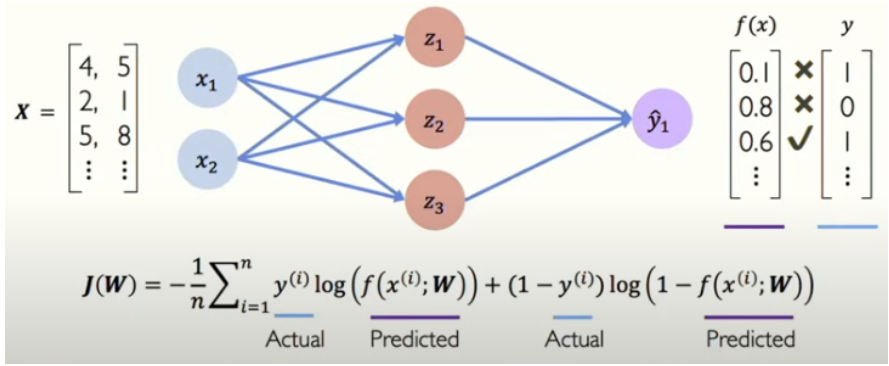


- *Early stopping*: stop training

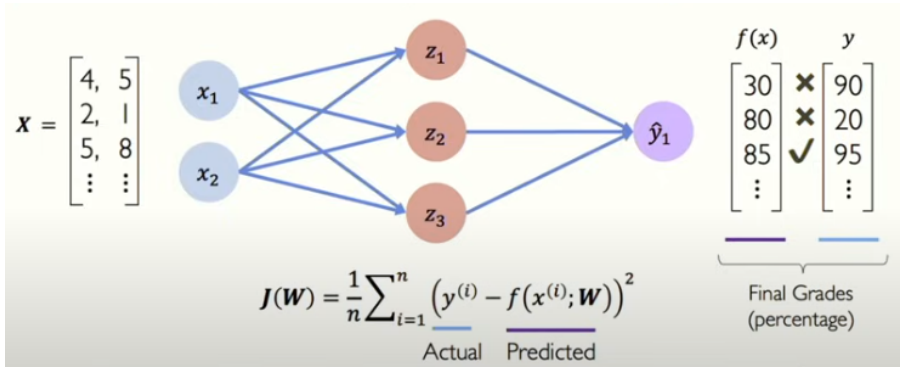


Loss

- *Cross entropy*: for models that return a probability



- *Mean squared error*: for regression



Designing neural networks: steps

- Architecture: the no. of levels and units per level, topology (interconnection method), activation functions
  Architectures: unidirectional vs. recurrent
- Training: finding the values of weights
- Testing: testing the model on test data

`https://playground.tensorflow.org/`

Bibliography

- T. M. Mitchell, *Machine Learning*, Ch. 4 Artificial Neural Networks, McGraw-Hill Science, 1997
- S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, Ch. 18.7 Artificial Neural Networks, Prentice Hall, 1995
- M. Neqnevitsky. *Artificial Intelligence. A Guide to Intelligent Systems*, Ch. 6. Multilayer neural networks, 2005