UPPSALA UNIVERSITET

Natural Computation Methods in Machine Learning (NCML)

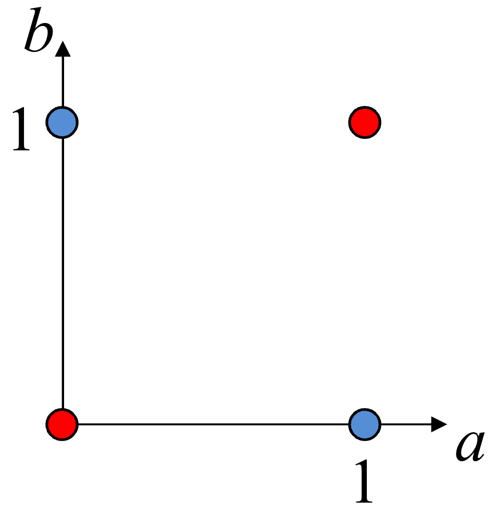
Lecture 4: Multilayer Perceptrons and Backpropagation

From previous lecture

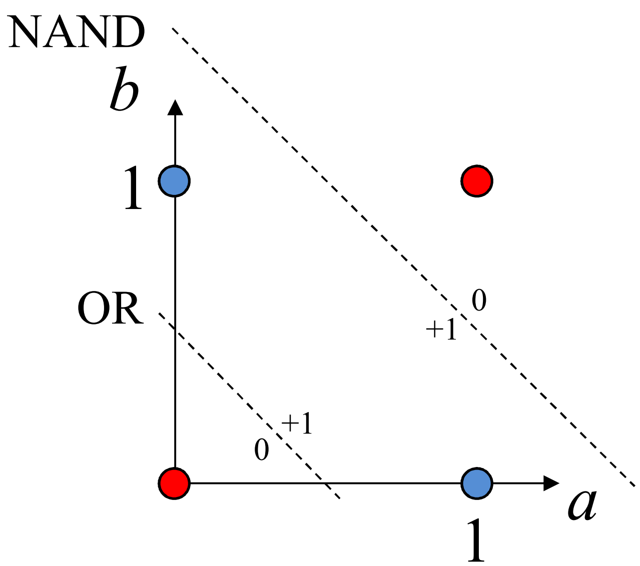
Linear (in)separability

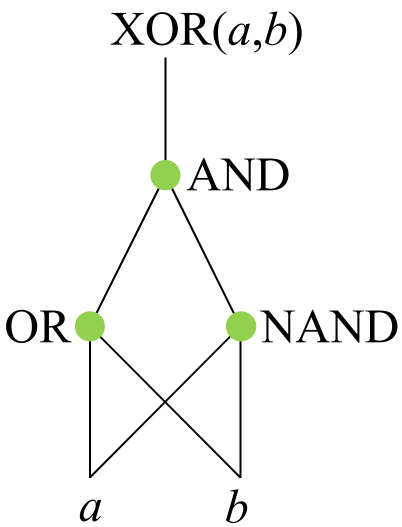
- The Perceptron Convergence Procedure converges to an optimal discriminant in a finite number of steps, ***if such a discriminant exists***
  - Problem: It seldom does! Few interesting classification problems are separable by one linear discriminant (a hyperplane)
  - Simple example: XOR

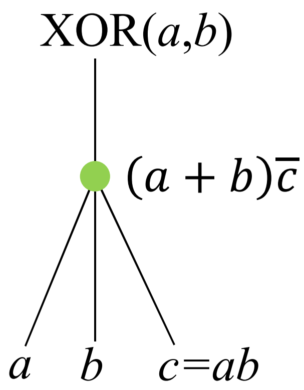
a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

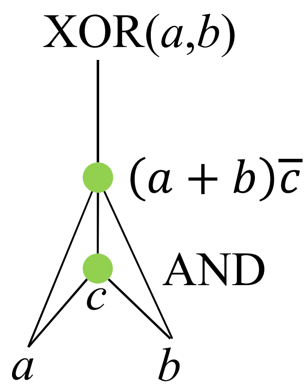


From previous lecture



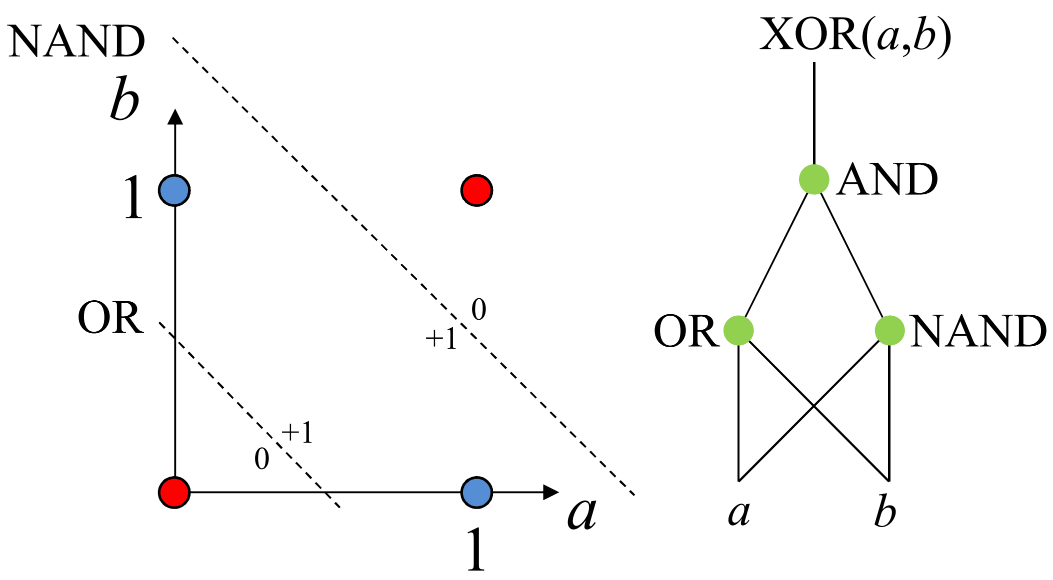






From previous lecture

Multilayer perceptrons (MLP)

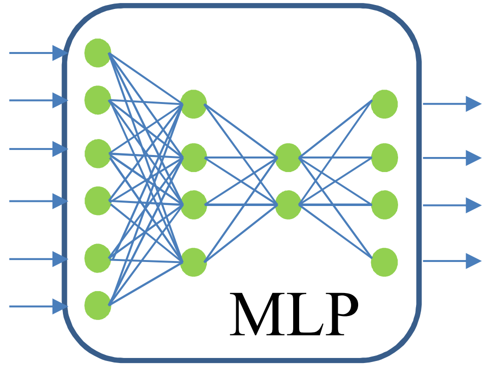


Problem: How to find the weight values automatically
(i.e. how to train the network)

The credit assignment problem

Structural. We will discuss a temporal one later

- To decide how much to blame an individual weight for the result
- We now have '*hidden layers*'
  - *No target (desired output) information*
  - What is the *error* of a hidden node?

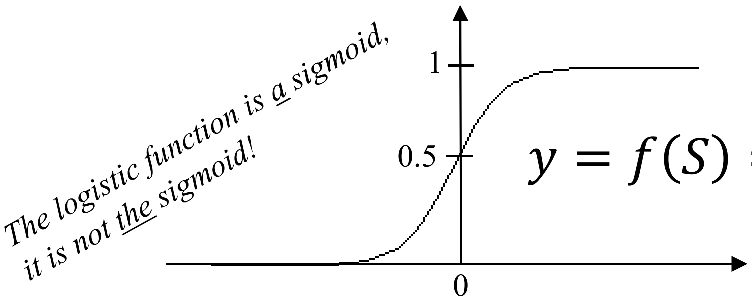


- The step function (f_h) is an obstacle
  - Can't decide, analytically from the outputs, how close to the flipping point ($S=0$) the weighted sum was
  - Removing f_h ($y=S$) does not help *(us. It helped B. Widrow)*
    - Linear nodes → The MLP can be reduced to one (output) layer
    - Back to where we started

∴ The activation function, $f(S)$, must be <u>non-linear</u> and <u>differentiable</u>

Sigmoid functions

- Sigmoid = any S-shaped function
- Often confused with the *logistic function*, which is actually just an example, though a very common one:



$$y = f(S) = \frac{1}{1 + e^{-\lambda S}}$$

  - λ (≥ 0) decides slope. In the extremes:
    - $\lambda = 0$ ➔ $y = 0.5$ (flat line)
    - $\lambda \rightarrow \infty$ ➔ $y = f_h(S)$ (step function)
  - Simple derivative: $y' = f'(S) = \lambda y(1 - y)$
- Another commonly used sigmoid function: $\tanh(S)$

  ≈ logistic, but in the range]-1,1[instead of]0,1[

How to make a learning rule

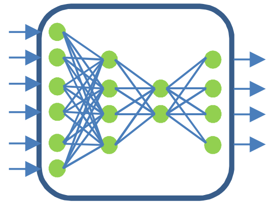
A (very) general recipe

1. State what is to be minimized as a *loss function*(*)

   For example: $$E = \frac{1}{2}(d-y)^2$$ d = desired output, y = actual output

2. How much did weight w_i, contribute to this loss?

$$\frac{\partial E}{\partial w_i}$$



3. Make an update rule which moves the weight in proportion to its contribution, but in the other direction:

$$w_i \leftarrow w_i + \Delta w_i \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

4. Can often be expressed as $\Delta w_i = \eta \delta x_i$ where x_i is the input corresponding to weight w_i

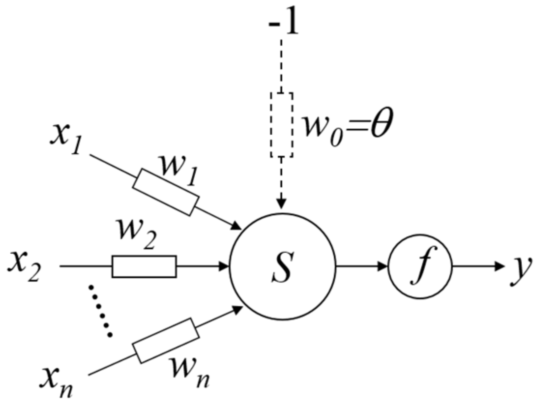
(*) *loss function = error function = cost function ≈ objective function*

Deriving the delta rule
following the recipe

- Consider a single neuron with a differentiable activation function:

$$y = f(S)$$

$$S = \sum_{i=0}^{n} w_i x_i, \text{ where } \begin{cases} x_0 = -1 \\ w_0 = \theta \end{cases}$$



- *Step 1: State what is to be minimized*
  - Let's assume that the loss function is the squared error

$$E = \frac{1}{2}(d - y)^2$$

(hidden assumption here – 'Gaussian prior'. We assume that data comes from a normal distribution)

- *Step 2: Use the chain rule to break down* $\frac{\partial E}{\partial w_i}$

Deriving the delta rule

Use the chain rule to break down $\partial E/\partial w_i$

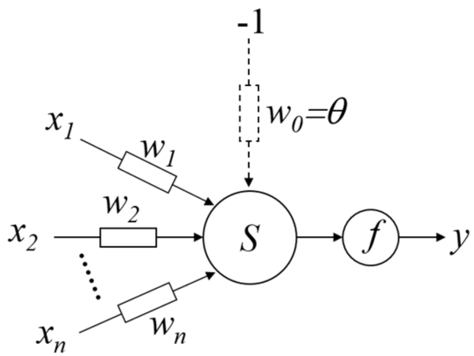
- The loss (E) depends on the output (y), which depends on the sum (S), which depends on the weight (w_i):

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial S}\frac{\partial S}{\partial w_i}$$

$$\frac{\partial E}{\partial y} = \frac{\partial \frac{1}{2}(d-y)^2}{\partial y} = 2\frac{1}{2}(d-y)(-1) = -(d-y)$$

$$\frac{\partial y}{\partial S} = \frac{\partial f(S)}{\partial S} = f'(S)$$

$$\frac{\partial S}{\partial w_i} = \frac{\partial \sum_{j=0}^{n} w_j x_j}{\partial w_i} = \frac{\partial (w_i x_i)}{\partial w_i} = x_i$$



$$\therefore \frac{\partial E}{\partial w_i} = -(d-y)f'(S)x_i$$

The delta rule

Express Δw

- *Step 3:* $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$ $\qquad \frac{\partial E}{\partial w_i} = -(d-y)f'(S)x_i$

$$\Delta w_i = \eta \underbrace{(d-y)f'(S)}x_i$$

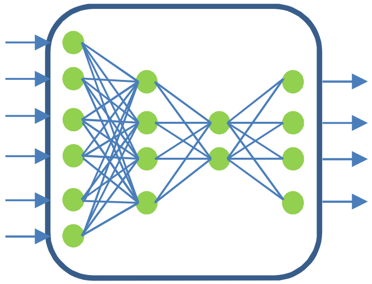
- Or, in the general form:

$$\Delta w_i = \eta \delta x_i \qquad \text{where } \delta = f'(S)(d-y)$$

- If we assume that the activation function is logistic:

$$\delta = \lambda y(1-y)(d-y)$$

- This is the **delta rule**, a.k.a. LMS (least-mean-squares) (Widrow & Hoff, 1960)
  - Compare to the Perceptron Convergence Procedure (PCP)!
  - Same. but without $f'(S)$, since the step function used in a binary perceptron is not differentiable

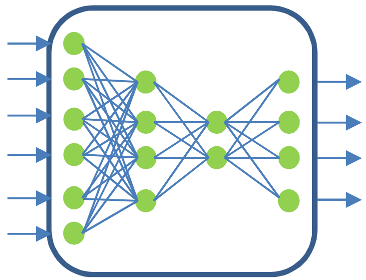


Backpropagation

The generalized delta rule

- We can extend the delta rule to cover a whole network of neurons (as long as everything is differentiable)
  - Same idea, the chain of partial derivatives just gets longer

- But, we now have several nodes. Therefore:
  - We must index the nodes, δ-values, and desired outputs d
  - Weights need a second index. Let w_{ji} denote the weight from node i to node j
  - x_i in the equations is the input to the node, i.e. it may be the value of a hidden node, not necessarily an input to the network.



Backpropagation

"of errors"

- We can still express the update rule on the general form

$$\Delta w_{ji} = \eta \delta_j x_i$$

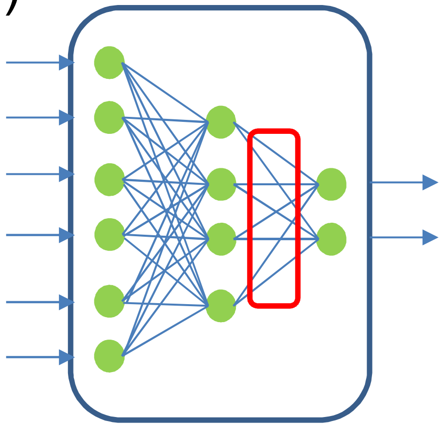
- but the definition of δ_j now depends on if node j is an output node or a hidden node

- For an output node, the delta rule is applicable as it is:

$$\delta_j = f'(S_j)(d_j - y_j) = \underbrace{\lambda y_j (1 - y_j)}_{\textit{(if logistic)}}(d_j - y_j)$$

Backpropagation
"of errors"

- Let's assume, for now, that we only have one hidden layer
- For a hidden node there is no desired output, d_j, but:
  - The hidden layer contributes to the output error through the *hidden-to-output* weights (through weighted sums)
  - The hidden layer should therefore be blamed for the error, in proportion to those same weights
  - The error of a hidden node is a weighted sum of the δ-values we just computed for the outputs
  - In other words, we backpropagate errors



$$\delta_j = f'(S_j)\sum_k w_{kj}\delta_k = \lambda y_j(1-y_j)\sum_k w_{kj}\delta_k$$

where the sum is over the nodes in the <u>*next*</u> *layer*

Backpropagation

Algorithm

1. Initialize. Set all weights to small random values with zero mean
2. Present an input vector, $\overline{x} = (x_1, x_2, \dots, x_n)$, and corresponding target vector, $\overline{d} = (d_1, d_2, \dots, d_m)$
3. Feed forward phase *(recall)*: Compute network outputs, by updating the nodes layer by layer from the first hidden layer to the outputs. The first hidden layer computes (for all nodes, y_j):

$$y_j = f\left(\sum_{i=0}^{n} w_{ji} x_i\right), \text{where } x_0 = -1$$

The next layer applies the same formula, substituting this layer's node values for x_i, etc.

Back propagation
Algorithm

4. Back propagation phase: Compute weight changes(*) iteratively, layer by layer, <u>from the outputs to the first hidden layer</u>:

$$\Delta w_{ji} = \eta \delta_j x_i$$

$$\delta_j = \begin{cases} \lambda y_j (1 - y_j)(d_j - y_j), & \text{if } y_j \text{ is an output node} \\ \lambda y_j (1 - y_j) \sum_k w_{kj} \delta_k, & \text{if } y_j \text{ is a hidden node} \end{cases}$$

   (The sum is over all k nodes in the <u>next</u> layer i.e. the layer for which δ-values were computed in the previous iteration)

5. Repeat from step 2 with a new input-target pair

(*) Note that we only compute weight changes here. It does not say when to update the weight.

Back propagation

Some notes

- λ is redundant here. It can be embedded in η.
  - λ is therefore often ignored (assumed to be 1)
- Weighted sums are just matrix-vector products.
  When computing the output:

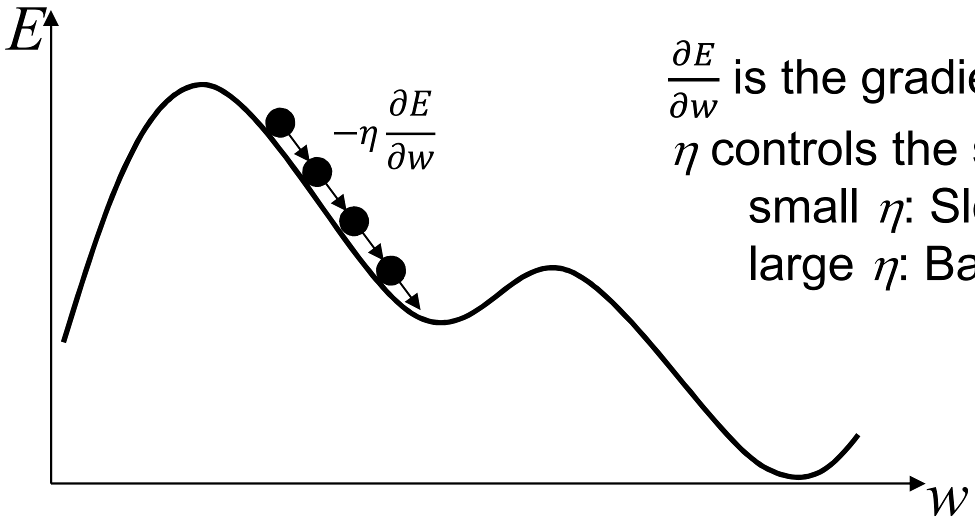
$$y_j = f\left(\sum_{i=0}^{n} w_{ji} x_i\right) = f(W\bar{x})$$

  - so if you implement this using a matrix library, you can compute the outputs of all nodes in a layer in one shot
  - If so, when backpropagating δ-values, just transpose the matrix!

$$\sum_{k} w_{kj}\delta_k = W^T\bar{\delta}$$

Gradient descent

- Backprop implements *gradient descent* (Cauchy, 1847)



$\frac{\partial E}{\partial w}$ is the gradient (steepest slope),
η controls the step length
small η: Slow learning
large η: Bad precision (zig-zag)

- The delta rule was invented in 1960 (by Widrow & Hoff)
- Why didn't Minsky&Papert see this solution in 1969?
  - Can get stuck in closest local minimum (almost certainly will)
  - Would have been considered a <u>big</u> problem in the 1960's
  - The cost of re-starting an experiment was very high

Momentum

- Common improvement: Add a *momentum term* to the weight update

$$\Delta w_{ji}(t+1) = \eta \delta_j x_i + \alpha \Delta w_{ji}(t)$$

  - Smoothing out weight changes over time
  - Gives the 'ball' a momentum, i.e. tends to continue in the direction as before
- Similar effects if we adapt step length over time
  - next lecture

When to update the weights
Epoch learning v.s. Pattern learning

- Epoch learning
  - Accumulate Δw until all patterns(*) have been presented once (= 1 epoch). Then update the weight and clear Δw
  - Special case of Batch Learning (batches can be smaller than the whole training set)
- Pattern learning (stochastic)
  - Update w after each pattern presentation
    - as a new step 4.5: $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
    - Requires random order of presentation (hence 'stochastic')
  - In this special case = *stochastic gradient descent*

(*) *Patterns = Input vectors*

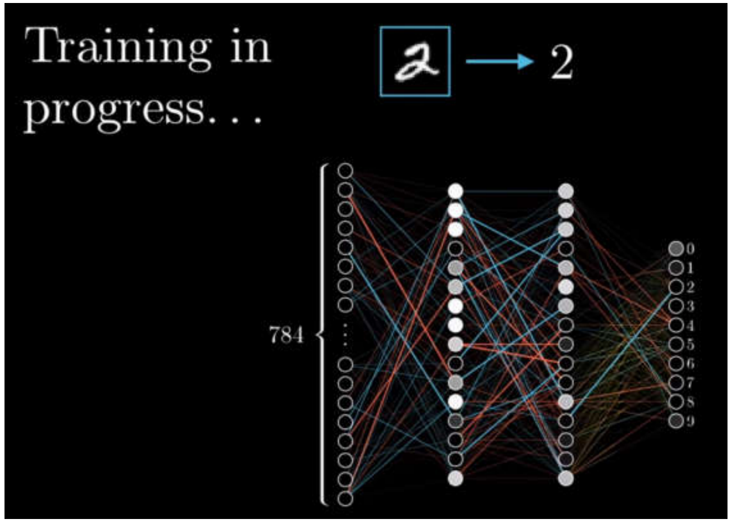
When to update the weights

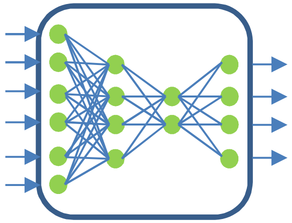
Epoch learning v.s. Pattern learning

- Epoch learning
  - This is what makes Backprop = gradient descent
  - Theorems and algorithm variants, often require this
- Pattern learning (stochastic)
  - Often better in practice (if the algorithm allows is)
  - The non-determinism reduces risk of getting stuck
  - Usually converges faster
- Common compromise: Batch learning, for smaller subsets – "mini-batches"

Backprop videos on YouTube

- There are many, of course
- One of my favourite channels: 3Blue1Brown
  - For example, watch the video "What is backpropagation really doing?" (from 3:09)
  - (https://www.youtube.com/watch?v=Ilg3gGewQ5U)





Challenges

All are possible exam questions

- The loss function could be any differentiable function, but is very often assumed to be the squared error
  - Which part of the update equations would change, if we replaced the objective function?
- If we use pattern learning, why don't we update the weight directly, in step 4? (instead of as a new step 4.5)
- What would happen if we initialized all weights to zero, instead of small random values?
- In a deep network (many layers) the chain of partial derivatives gets very long. A problem?

Super Challenge

- As shown in the previous lecture and this one:

  $ab, \overline{ab}, a+b, \overline{a+b}$, are all linearly separable,

  $(a+b)\overline{c}$ (the three input solution to XOR) is too,

  but $a \oplus b = a\overline{b} + \overline{a}b = (a+b)\overline{ab}$ is not!

- *Where is the limit? Under which condition(s) is a Boolean expression linearly separable, when viewed as a classification problem?*

(This challenge is beyond the scope of this course, so don't worry if you can't figure this one out)