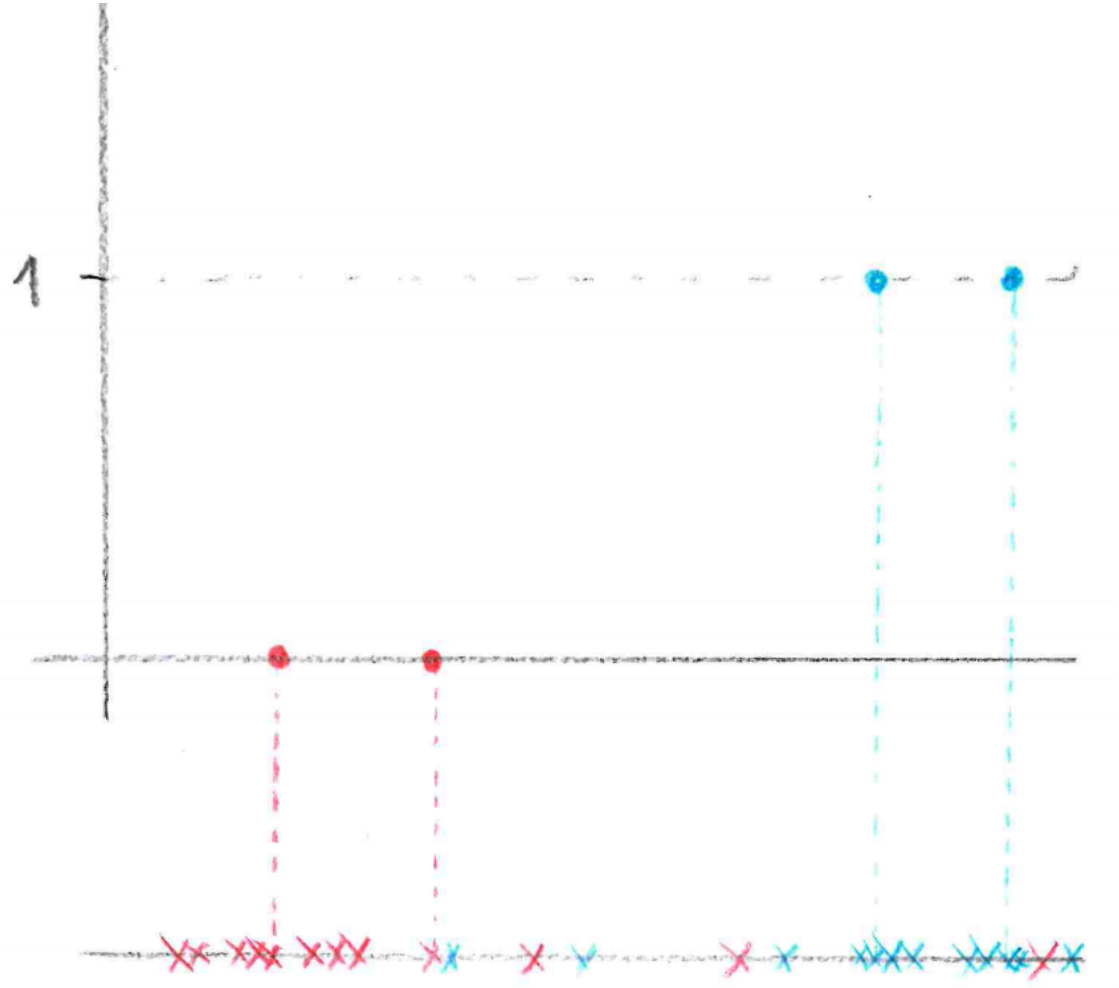
What about classification?

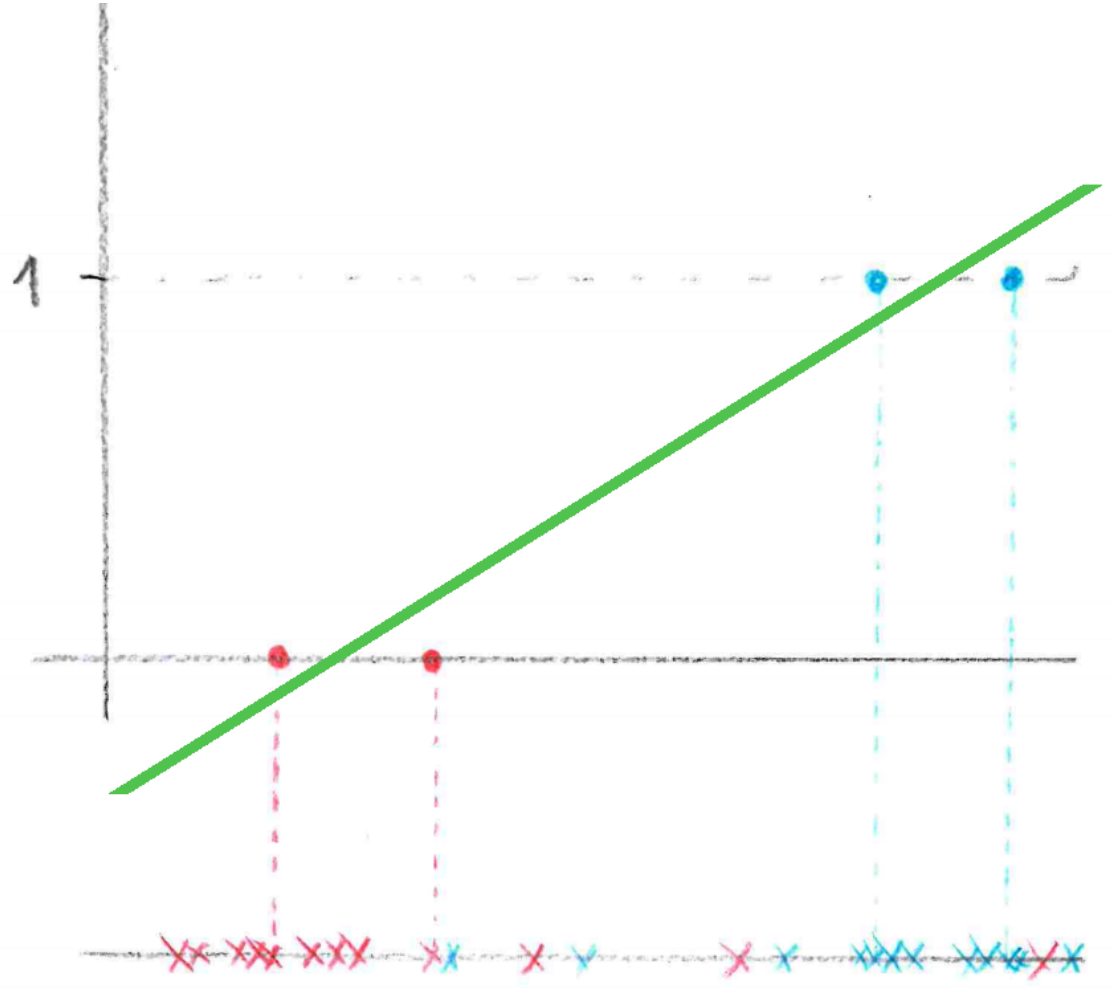
Binary classification: Desired outputs 0 and 1.



Ideally, capture the probability distribution of classes.

What about classification?

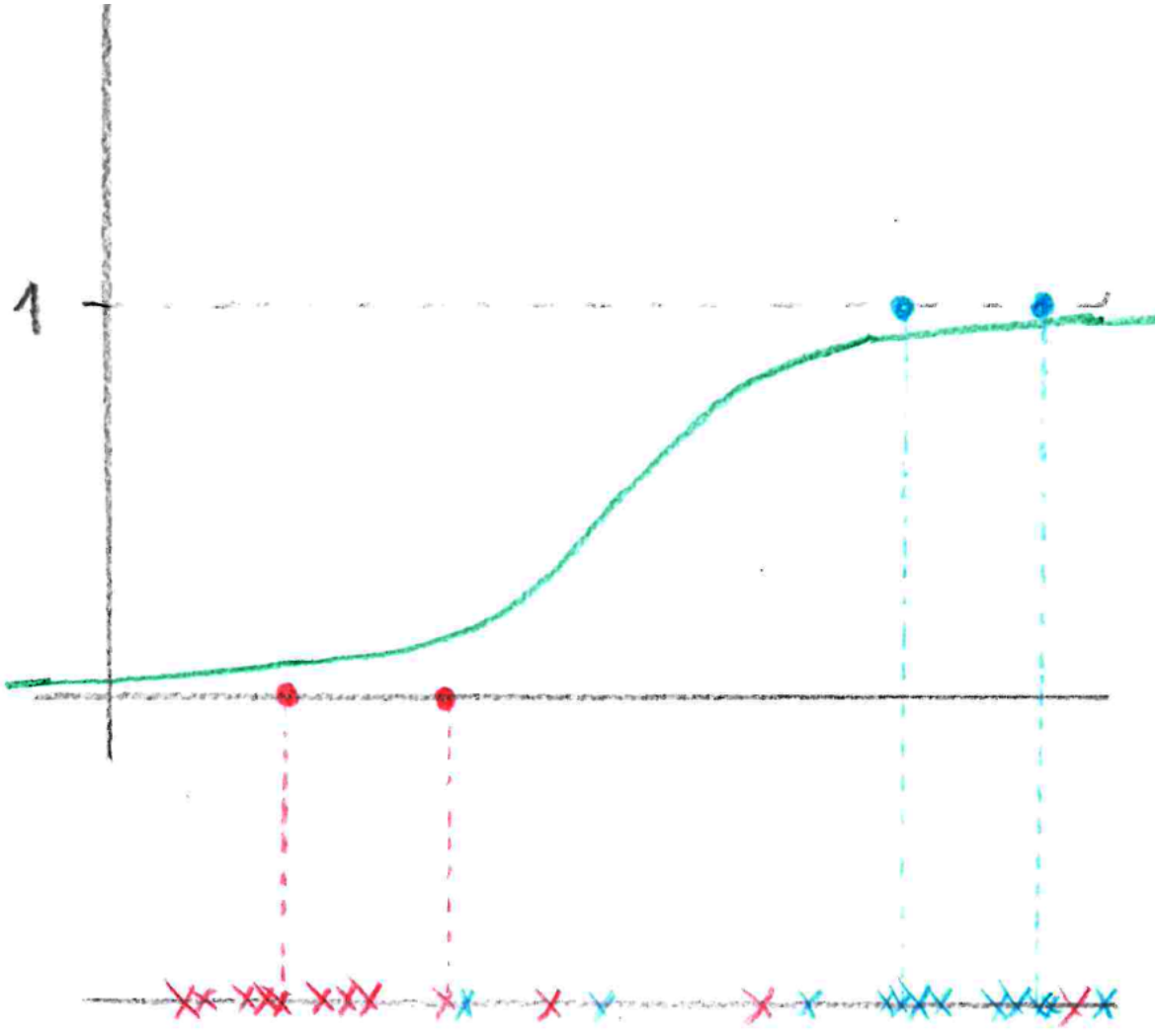
Binary classification: Desired outputs 0 and 1.



... does not capture probability well (it is not a probability at all)

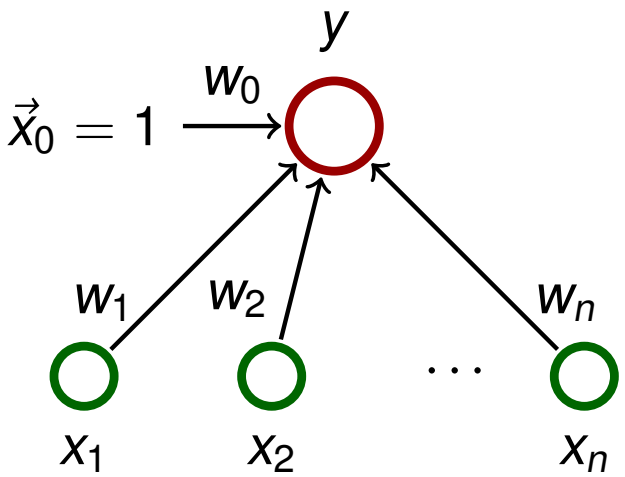
What about classification?

Binary classification: Desired outputs 0 and 1.



... logistic sigmoid $\frac{1}{1+e^{-(\vec{w}\cdot\vec{x})}}$ is much better!

Logistic regression



$\vec{w} = (w_0, w_1, \ldots, w_n)$ and $\vec{x} = (x_0, x_1, \ldots, x_n)$ where $x_0 = 1$.

Activity:

- inner potential: $\xi = w_0 + \sum_{i=1}^{n} w_i x_i = \sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
- activation function: $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$
- network function: $y[\vec{w}](\vec{x}) = \sigma(\xi) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x})}}$

Intuition: The output y is now interpreted as the probability of the class 1 given the input $\vec{x}$.

But what is the meaning of the sigmoid?

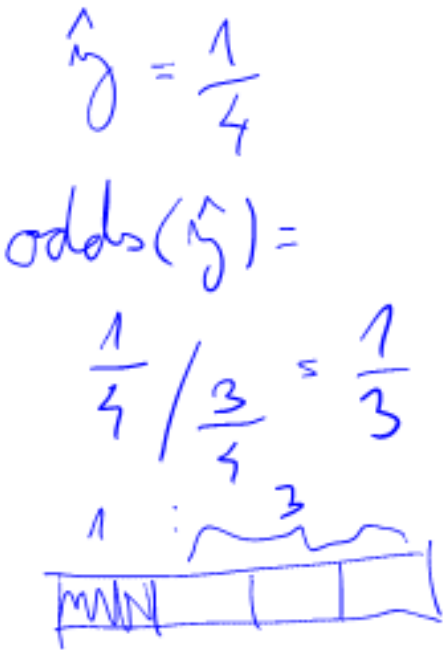
The model gives a probability y of the class 1 given an input $\vec{x}$.
But why we model such a probability using $1/(1 + e^{-\vec{w}\cdot\vec{x}})$??

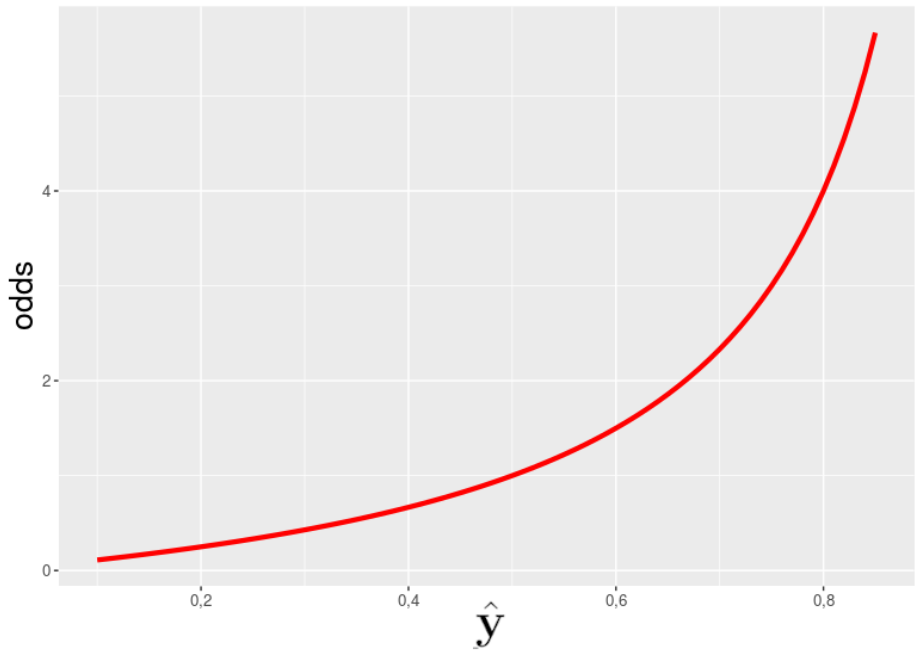
But what is the meaning of the sigmoid?

The model gives a probability y of the class 1 given an input $\vec{x}$.
But why we model such a probability using $1/(1 + e^{-\vec{w}\cdot\vec{x}})$??

Let $\hat{y}$ be the "true" probability of the class 1 to be modeled.
What about odds of the class 1?

$$odds(\hat{y}) = \hat{y}/1 - \hat{y}$$





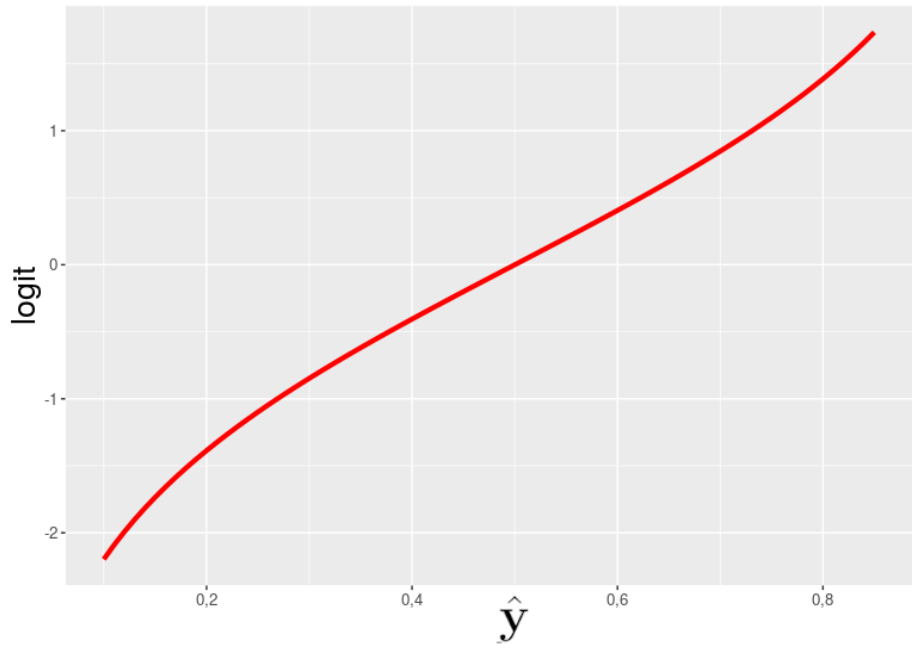
Resembles an exponential function ...

But what is the meaning of the sigmoid?

The model gives a probability y of the class 1 given an input $\vec{x}$.
But why we model such a probability using $1/(1+e^{-\vec{w}\cdot\vec{x}})$??

Let $\hat{y}$ be the "true" probability of the class 1 to be modeled.
What about log odds (aka logit) of the class 1?

$$logit(\hat{y}) = \log(\hat{y}/(1-\hat{y}))$$



Looks almost linear ...

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

$$\log(\hat{y}/(1-\hat{y})) = \vec{w} \cdot \vec{x}$$

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

$$\log(\hat{y}/(1-\hat{y})) = \vec{w} \cdot \vec{x}$$

Then

$$\log((1-\hat{y})/\hat{y}) = -\vec{w} \cdot \vec{x}$$

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

$$\log(\hat{y}/(1-\hat{y})) = \vec{w} \cdot \vec{x}$$

Then

$$\log((1-\hat{y})/\hat{y}) = -\vec{w} \cdot \vec{x}$$

and

$$(1-\hat{y})/\hat{y} = e^{-\vec{w} \cdot \vec{x}}$$

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

$$\log(\hat{y}/(1-\hat{y})) = \vec{w} \cdot \vec{x}$$

Then

$$\log((1-\hat{y})/\hat{y}) = -\vec{w} \cdot \vec{x}$$

and

$$(1-\hat{y})/\hat{y} = e^{-\vec{w} \cdot \vec{x}}$$

and

$$\hat{y} = \frac{1}{1+e^{-\vec{w} \cdot \vec{x}}}$$

That is, if we model log odds using a linear function, the probability is obtained by applying the logistic sigmoid on the result of the linear function.

$T = \{(x, d)\}$, $y = \sigma(\xi)$, $\xi = wx$ | $\sigma' = \sigma(1-\sigma) = y(1-y)$

$E = -d \log(y) - (1-d) \log(1-y)$

$d = 1$: $\frac{\partial E}{\partial w} = \frac{\partial -d \log y}{\partial w} = -\frac{1}{y} \cdot \sigma' \cdot x =$

$= \frac{-1 \cancel{y}(1-y) \cdot x}{\cancel{y}} = -(1-y)x$

if $y \sim 0 \Rightarrow \frac{\partial E}{\partial w} = -x$

Logistic regression

Learning:

- Given a **training dataset**

$$\mathcal{T} = \left\{\left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right)\right\}$$

Here $\vec{x}_k = (x_{k0}, x_{k1} \ldots, x_{kn}) \in \mathbb{R}^{n+1}$, $x_{k0} = 1$, is the k-th input, and $d_k \in \{0, 1\}$ is the expected output.

$T = \{(x, d)\}$, $y = \sigma(\xi)$, $\xi = wx$ | $\sigma'(\xi) = \sigma(\xi)\cdot(1-\sigma(\xi))$

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

$$\frac{\partial E}{\partial w} = (y-d)\cdot\sigma'\cdot x = (y-d)\cdot\sigma\cdot(1-\sigma)\cdot x$$

($y = \sigma$, $y = \sigma$)

if $d=1$, $y\sim 0 \Rightarrow \frac{\partial E}{\partial w} \sim 0$

$d=0$, $y\sim 1 \Rightarrow \frac{\partial E}{\partial w} \sim 0$

Logistic regression

Learning:

- Given a **training dataset**

$$\mathcal{T} = \left\{\left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right)\right\}$$

Here $\vec{x}_k = (x_{k0}, x_{k1} \ldots, x_{kn}) \in \mathbb{R}^{n+1}$, $x_{k0} = 1$, is the k-th input, and $d_k \in \{0, 1\}$ is the expected output.

What error function?

(Binary) cross-entropy:

$$E(\vec{w}) = \sum_{k=1}^{p} -(d_k \log(y_k) + (1 - d_k)\log(1 - y_k))$$

What?!?

Log likelihood is your friend!

- Let’s have a "coin" (sides 0 and 1).

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is $\hat{y}$ and is unknown!

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is $\hat{y}$ and is unknown!
- You have tossed the coin 5 times and got a training dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Consider this to be a very special case where the input dimension is 0

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is $\hat{y}$ and is unknown!
- You have tossed the coin 5 times and got a training dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Consider this to be a very special case where the input dimension is 0

- What is the best model y of $\hat{y}$ based on the data?

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is $\hat{y}$ and is unknown!
- You have tossed the coin 5 times and got a training dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Consider this to be a very special case where the input dimension is 0

- What is the best model y of $\hat{y}$ based on the data?

Answer: The one that generates the data with maximum probability!

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Assume that the data was generated by independent trials, then the probability of getting exactly $\mathcal{T}$ from our model is

$$L = y \cdot y \cdot (1 - y) \cdot (1 - y) \cdot y$$

How to maximize this w.r.t. y?

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Assume that the data was generated by independent trials, then the probability of getting exactly $\mathcal{T}$ from our model is

$$L = y \cdot y \cdot (1 - y) \cdot (1 - y) \cdot y$$

How to maximize this w.r.t. y?

Maximize

$$LL = log(L) = \log(y) + \log(y) + \log(1-y) + \log(1-y) + \log(y)$$

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Assume that the data was generated by independent trials, then the probability of getting exactly $\mathcal{T}$ from our model is

$$L = y \cdot y \cdot (1 - y) \cdot (1 - y) \cdot y$$

How to maximize this w.r.t. y?

Maximize

$$LL = log(L) = \log(y) + \log(y) + \log(1-y) + \log(1-y) + \log(y)$$

But then

$$-LL = -1 \cdot log(y) - 1 \cdot log(y) - (1-0) \cdot log(1-y) - (1-0) \cdot \log(1-y) - 1 \cdot \log(y)$$

i.e. $-LL$ is the cross-entropy.

Let the coin depend on the input

Consider our model:

$$y = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x})}}$$

Let the coin depend on the input

Consider our model:

$$y = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x})}}$$

The training dataset is now standard:

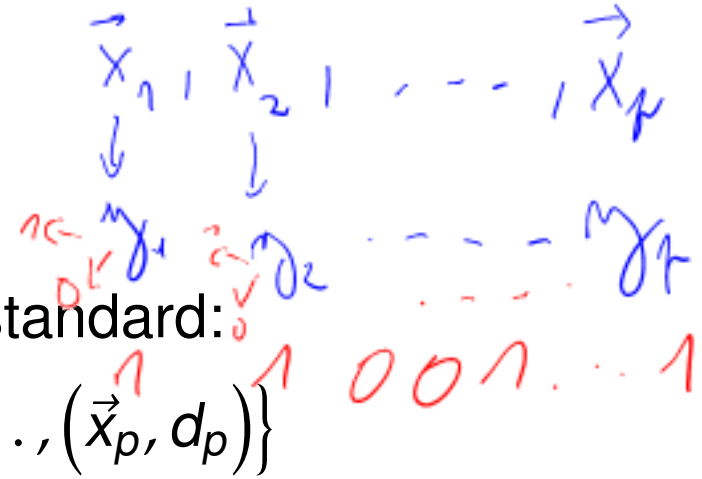
$$\mathcal{T} = \left\{ \left(\vec{x}_1, d_1 \right), \left(\vec{x}_2, d_2 \right), \ldots, \left(\vec{x}_p, d_p \right) \right\}$$

Here $\vec{x}_k = (x_{k0}, x_{k1} \ldots, x_{kn}) \in \mathbb{R}^{n+1}$, $x_{k0} = 1$, is the k-th input, and $d_k \in \{0, 1\}$ is the expected output.

Let the coin depend on the input

Consider our model:

$$y = \frac{1}{1 + e^{-(\vec{w}\cdot\vec{x})}}$$



The training dataset is now standard:

$$\mathcal{T} = \left\{\left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right)\right\}$$

Here $\vec{x}_k = (x_{k0}, x_{k1} \ldots, x_{kn}) \in \mathbb{R}^{n+1}$, $x_{k0} = 1$, is the k-th input, and $d_k \in \{0, 1\}$ is the expected output.

The likelihood:

$$L = \prod_{k=1}^{p} y_k^{d_k} \cdot (1 - y_k)^{(1-d_k)}$$

and $LL = \log(L) = \sum_{k=1}^{p} (d_k \log(y_k) + (1 - d_k) \log(1 - y_k))$
and thus $-LL$ = the cross-entropy.

Minimizing the cross-netropy maximizes the log-likelihood (and vice versa).

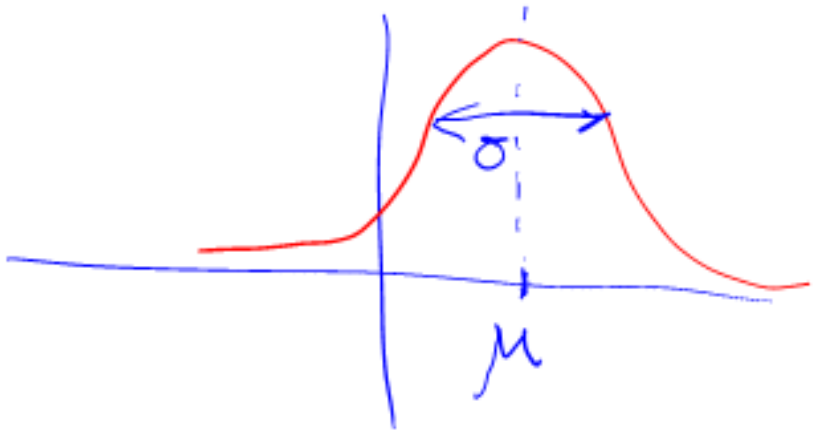
Normal Distribution

Distribution of continuous random variables.

Density (one dimensional, that is over $\mathbb{R}$):

$$p(x) \quad = \quad \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \quad =: \quad N[\mu, \sigma^2](x)$$

μ is the expected value (the mean), σ^2 is the variance.



Maximum Likelihood vs Least Squares (Dim 1)

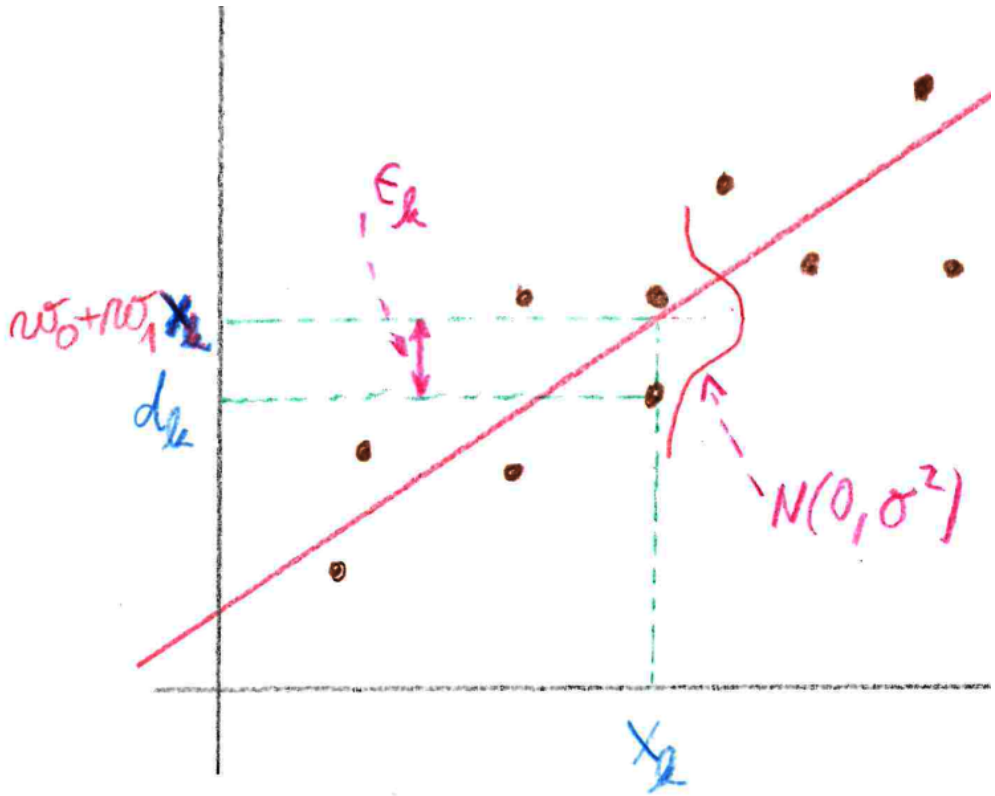
Fix a training set $D = \left\{ (x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right) \right\}$

Maximum Likelihood vs Least Squares (Dim 1)

Fix a training set $D = \left\{(x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right)\right\}$
Assume that each d_k has been generated randomly by

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

- w_0, w_1 are **unknown numbers**
- ϵ_k are normally distributed with mean 0 and an unknown variance σ^2



Maximum Likelihood vs Least Squares (Dim 1)

Keep in mind:

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

Assume that $\epsilon_1, \ldots, \epsilon_p$ were generated **independently**.

Maximum Likelihood vs Least Squares (Dim 1)

Keep in mind:

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

Assume that $\epsilon_1, \ldots, \epsilon_p$ were generated **independently**.

Denote by $p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$ the probability density according to which the values $d_1, \ldots, d_n$ were generated assuming fixed $w_0, w_1, \sigma^2, x_1, \ldots, x_p$.

Maximum Likelihood vs Least Squares (Dim 1)

Keep in mind:

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

Assume that $\epsilon_1, \ldots, \epsilon_p$ were generated **independently**.

Denote by $p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$ the probability density according to which the values $d_1, \ldots, d_n$ were generated assuming fixed $w_0, w_1, \sigma^2, x_1, \ldots, x_p$.

The independence and normality imply

$$\begin{aligned} p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2) &= \prod_{k=1}^{p} N[w_0 + w_1 x_k, \sigma^2](d_k) \\ &= \prod_{k=1}^{p} \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(d_k - w_0 - w_1 x_k)^2}{2\sigma^2}\right\} \end{aligned}$$

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Theorem

(w_0, w_1) maximizes $L(w_0, w_1, \sigma^2)$ for arbitrary σ^2 **iff** *(w_0, w_1) minimizes squared error $E(w_0, w_1) = \sum_{k=1}^{p} (d_k - w_0 - w_1 x_k)^2$.*

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Theorem

(w_0, w_1) maximizes $L(w_0, w_1, \sigma^2)$ for arbitrary σ^2 **iff** *(w_0, w_1) minimizes squared error $E(w_0, w_1) = \sum_{k=1}^{p} (d_k - w_0 - w_1 x_k)^2$.*

Note that the maximizing/minimizing (w_0, w_1) does not depend on σ^2.

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Theorem

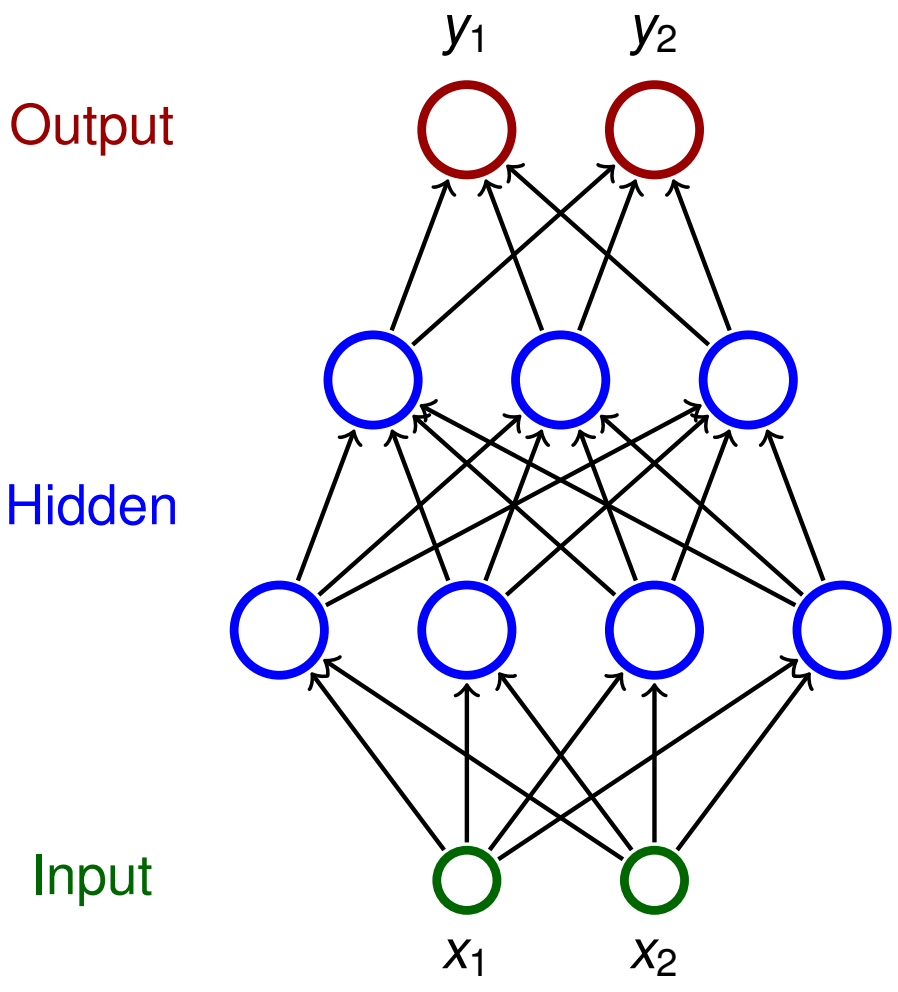
(w_0, w_1) maximizes $L(w_0, w_1, \sigma^2)$ for arbitrary σ^2 **iff** *(w_0, w_1) minimizes squared error $E(w_0, w_1) = \sum_{k=1}^{p} (d_k - w_0 - w_1 x_k)^2$.*

Note that the maximizing/minimizing (w_0, w_1) does not depend on σ^2.

Maximizing σ^2 satisfies $\sigma^2 = \frac{1}{p} \sum_{k=1}^{p} (d_k - w_0 - w_1 \cdot x_k)^2$.

MLP training – theory

Architecture – Multilayer Perceptron (MLP)



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the i-th layer are connected with all neurons in the $i + 1$-st layer
- Architecture of a MLP is typically described by numbers of neurons in individual layers (e.g. 2-4-3-2)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

(define $y_0 = 1$ is the value of the formal unit input)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)

MLP – activity

Activity:

- inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

MLP – activity

Activity:

- inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- activation function σ_j for neuron j (arbitrary differentiable)
  [e.g. logistic sigmoid $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$]

MLP – activity

Activity:

- inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- activation function σ_j for neuron j (arbitrary differentiable)
  [e.g. logistic sigmoid $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$]
- State of non-input neuron $j \in Z \setminus X$ after the computation stops:

$$y_j = \sigma_j(\xi_j)$$

(y_j depends on the configuration $\vec{w}$ and the input $\vec{x}$, so we sometimes write $y_j(\vec{w}, \vec{x})$)

MLP – activity

Activity:

- inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- activation function σ_j for neuron j (arbitrary differentiable)
  [e.g. logistic sigmoid $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$]
- State of non-input neuron $j \in Z \setminus X$ after the computation stops:

$$y_j = \sigma_j(\xi_j)$$

  (y_j depends on the configuration $\vec{w}$ and the input $\vec{x}$, so we sometimes write $y_j(\vec{w}, \vec{x})$)
- The network computes a function $\mathbb{R}^{|X|}$ do $\mathbb{R}^{|Y|}$. Layer-wise computation: First, all input neurons are assigned values of the input. In the ℓ-th step, all neurons of the ℓ-th layer are evaluated.

MLP – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{ \left(\vec{x}_k, \vec{d}_k \right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

MLP – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{ \left(\vec{x}_k, \vec{d}_k \right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function:**

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj} \right)^2$$

MLP – learning algorithm

Batch algorithm (gradient descent):

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

MLP – learning algorithm

Batch algorithm (gradient descent):

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

where

$$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$$

is a *weight update* of w_{ji} in step $t+1$ and $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$.

MLP – learning algorithm

Batch algorithm (gradient descent):

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

where

$$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$$

is a *weight update* of w_{ji} in step $t+1$ and $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$.

Note that $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$ is a component of the gradient ∇E, i.e. the weight update can be written as $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \varepsilon(t) \cdot \nabla E(\vec{w}^{(t)})$.

MLP – error function gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

MLP – error function gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

MLP – error function gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

and for every $j \in Z \setminus X$ we get

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

MLP – error function gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

and for every $j \in Z \setminus X$ we get

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{for } j \in Z \setminus (Y \cup X)$$

(Here all y_j are in fact $y_j(\vec{w}, \vec{x}_k)$).

MLP – error function gradient

- If $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$ for all $j \in Z$, then

$$\sigma_j'(\xi_j) = \lambda_j y_j (1 - y_j)$$

MLP – error function gradient

- If $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$ for all $j \in Z$, then

$$\sigma_j'(\xi_j) = \lambda_j y_j (1 - y_j)$$

and thus for all $j \in Z \setminus X$:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \lambda_r y_r (1 - y_r) \cdot w_{rj} \qquad \text{for } j \in Z \setminus (Y \cup X)$$

MLP – error function gradient

- If $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$ for all $j \in Z$, then

$$\sigma_j'(\xi_j) = \lambda_j y_j (1 - y_j)$$

and thus for all $j \in Z \setminus X$:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \lambda_r y_r (1 - y_r) \cdot w_{rj} \qquad \text{for } j \in Z \setminus (Y \cup X)$$

- If $\sigma_j(\xi) = a \cdot \tanh(b \cdot \xi_j)$ for all $j \in Z$, then

$$\sigma_j'(\xi_j) = \frac{b}{a}(a - y_j)(a + y_j)$$

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

Initialize $\mathcal{E}_{ji} := 0$

(By the end of the computation: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

Initialize $\mathcal{E}_{ji} := 0$

(By the end of the computation: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

For every $k = 1, \ldots, p$ do:

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

Initialize $\mathcal{E}_{ji} := 0$

(By the end of the computation: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

For every $k = 1, \ldots, p$ do:

1. **forward pass:** compute $y_j = y_j(\vec{w}, \vec{x}_k)$ for all $j \in Z$

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

Initialize $\mathcal{E}_{ji} := 0$

(By the end of the computation: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

For every $k = 1, \ldots, p$ do:

1. **forward pass:** compute $y_j = y_j(\vec{w}, \vec{x}_k)$ for all $j \in Z$
2. **backward pass:** compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ using *backpropagation* (see the next slide!)

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

Initialize $\mathcal{E}_{ji} := 0$

(By the end of the computation: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

For every $k = 1, \ldots, p$ do:

1. **forward pass:** compute $y_j = y_j(\vec{w}, \vec{x}_k)$ for all $j \in Z$
2. **backward pass:** compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ using *backpropagation* (see the next slide!)
3. compute $\frac{\partial E_k}{\partial w_{ji}}$ for all w_{ji} using

$$\frac{\partial E_k}{\partial w_{ji}} := \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

Initialize $\mathcal{E}_{ji} := 0$

(By the end of the computation: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

For every $k = 1, \ldots, p$ do:

1. **forward pass:** compute $y_j = y_j(\vec{w}, \vec{x}_k)$ for all $j \in Z$
2. **backward pass:** compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ using *backpropagation* (see the next slide!)
3. compute $\frac{\partial E_k}{\partial w_{ji}}$ for all w_{ji} using

$$\frac{\partial E_k}{\partial w_{ji}} := \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

4. $\mathcal{E}_{ji} := \mathcal{E}_{ji} + \frac{\partial E_k}{\partial w_{ji}}$

The resulting $\mathcal{E}_{ji}$ equals $\frac{\partial E}{\partial w_{ji}}$.

MLP – backpropagation

Compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ as follows:

MLP – backpropagation

Compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ as follows:

- if $j \in Y$, then $\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$

MLP – backpropagation

Compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ as follows:

- if $j \in Y$, then $\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$
- if $j \in Z \setminus Y \cup X$, then assuming that j is in the ℓ-th layer and assuming that $\frac{\partial E_k}{\partial y_r}$ has already been computed for all neurons in the $\ell + 1$-st layer, compute

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj}$$

(This works because all neurons of $r \in j^{\rightarrow}$ belong to the $\ell + 1$-st layer.)

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

Proof sketch: The algorithm does the following p times:

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$
2. backpropagation, i.e. computes $\frac{\partial E_k}{\partial y_j}$

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$
2. backpropagation, i.e. computes $\frac{\partial E_k}{\partial y_j}$
3. computes $\frac{\partial E_k}{\partial w_{ji}}$ and adds it to $\mathcal{E}_{ji}$ (a constant time operation in the unit cost framework)

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$
2. backpropagation, i.e. computes $\frac{\partial E_k}{\partial y_j}$
3. computes $\frac{\partial E_k}{\partial w_{ji}}$ and adds it to $\mathcal{E}_{ji}$ (a constant time operation in the unit cost framework)

The steps 1. - 3. take linear time.

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

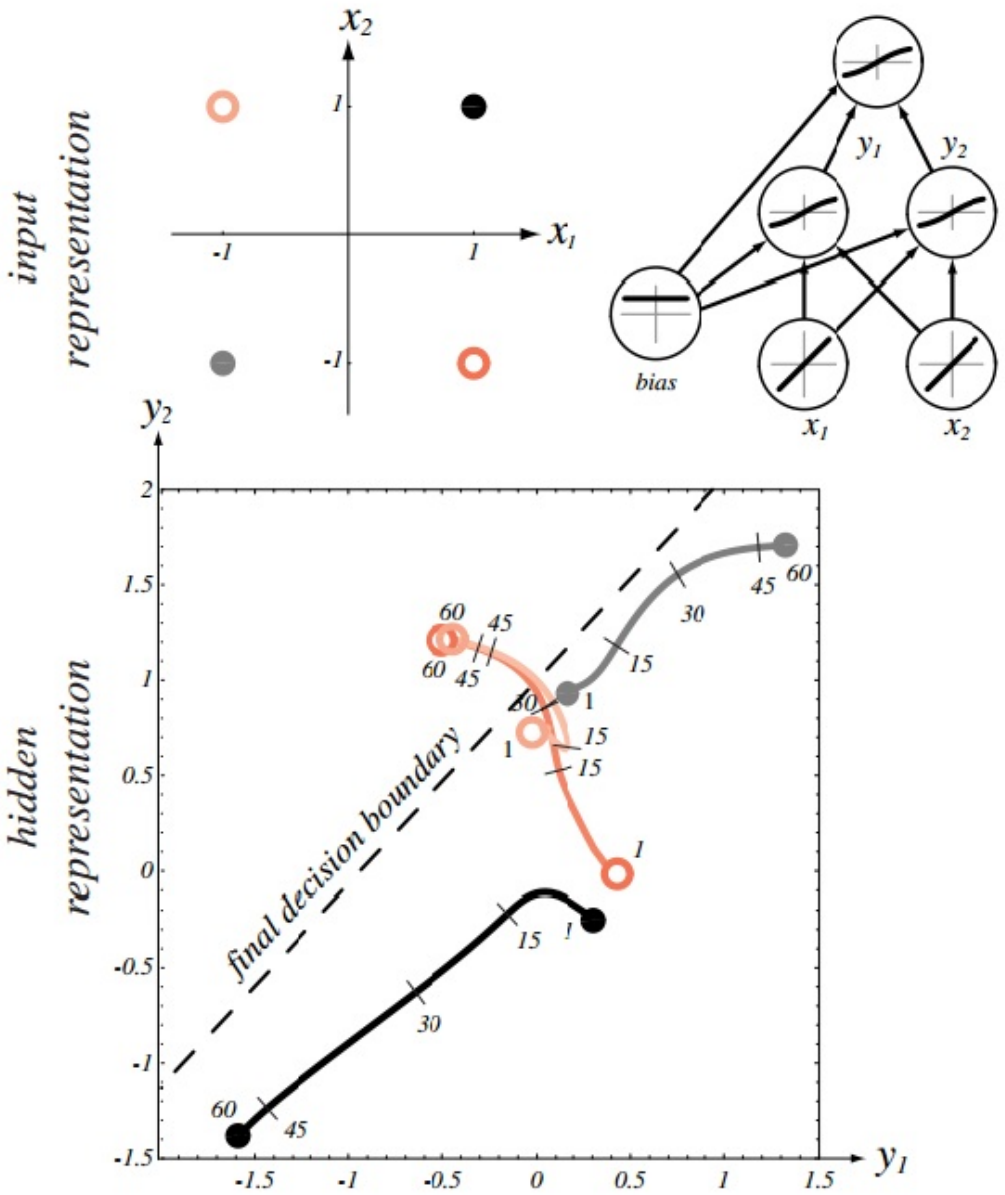
Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$
2. backpropagation, i.e. computes $\frac{\partial E_k}{\partial y_j}$
3. computes $\frac{\partial E_k}{\partial w_{ji}}$ and adds it to $\mathcal{E}_{ji}$ (a constant time operation in the unit cost framework)

The steps 1. - 3. take linear time.

Note that the speed of convergence of the gradient descent cannot be estimated ...

Illustration of the gradient descent – XOR



Source: Pattern Classification (2nd Edition); Richard O. Duda, Peter E. Hart, David G. Stork

MLP – learning algorithm

Online algorithm:

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

where

$$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E_k}{\partial w_{ji}}(w_{ji}^{(t)})$$

is the *weight update* of w_{ji} in the step $t+1$ and $0 < \varepsilon(t) \leq 1$ is the *learning rate* in the step $t+1$.

There are other variants determined by selection of the training examples used for the error computation (more on this later).

SGD

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

where

$$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially.