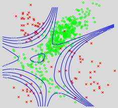
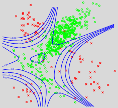


Machine Learning

Blockkurs *Neuronale Netze und Deep Learning* vom 17.5.2018

Mario Stanke
Institut für Mathematik und Informatik
Universität Greifswald



Artificial Neural Networks

For now, we consider so-called feed-forward artificial neural networks with the logistic sigmoid as so-called activator function.

Definition 1 (Artificial Neural Networks (ANN, =künstliches neuronales Netz))

A feed-forward artificial neural network with $L \geq 1$ layers of sizes $s_1, \ldots, s_L$ with $n =: s_0$ input variables $\mathbf{x} = (x_1, \ldots, x_n)^T$ and $K := s_L$ output variables $\mathbf{t} = (t_1, \ldots, t_K)^T$ is a function

$$\mathbf{t} = h_{\boldsymbol{\theta}}(\mathbf{x})$$

with parameters

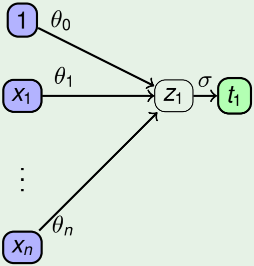
$$\boldsymbol{\theta} = (\Theta^{(0)}, \ldots, \Theta^{(L-1)}) \qquad \text{, where} \quad \Theta^{(\ell)} \in \mathbb{R}^{s_{\ell+1} \times (s_\ell + 1)},$$

defined by the following recursions

$$\begin{aligned}
\mathbf{t} &= \mathbf{g}(\mathbf{z}^{(L)}) \in \mathbb{R}^K \\
\mathbf{z}^{(\ell)} &= \Theta^{(\ell-1)} \mathbf{a}^{(\ell-1)} \in \mathbb{R}^{s_\ell} \qquad (1 \leq \ell \leq L) \\
\mathbf{a}^{(\ell)} &= \begin{pmatrix} a_0^{(\ell)} \\ a_1^{(\ell)} \\ \vdots \\ a_{s_\ell}^{(\ell)} \end{pmatrix} = \begin{pmatrix} 1 \\ \sigma(z_1^{(\ell)}) \\ \vdots \\ \sigma(z_{s_\ell}^{(\ell)}) \end{pmatrix} \qquad (1 \leq \ell < L) \\
\mathbf{a}^{(0)} &= \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}.
\end{aligned} \tag{1}$$

Here, σ is the logistic sigmoid function and we will call $\mathbf{g} : \mathbb{R}^K \to \mathbb{R}^K$ the output activation function.

Logistic regression = ANN with 1 layer and 1 output variable



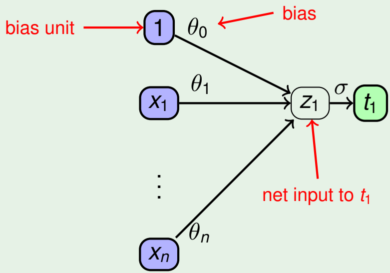
$L = 1$ layers

$$\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_n) \quad \text{parameters}$$

t_1 one output unit
$x_1, \ldots, x_n$ input units
output activation function $g = \sigma$

$$t_1 = \sigma(z_1) = \sigma\left(\theta_0 + \sum_{j=1}^{n} \theta_j x_j\right)$$

Logistic regression = ANN with 1 layer and 1 output variable



$L = 1$ layers

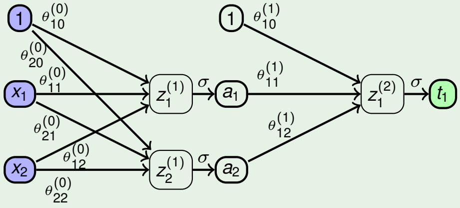
$$\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_n) \quad \text{parameters}$$

t_1 one output unit
$x_1, \ldots, x_n$ input units
output activation function $g = \sigma$

$$t_1 = \sigma(z_1) = \sigma\left(\theta_0 + \sum_{j=1}^{n} \theta_j x_j\right)$$

Artificial Neural Networks

ANN with 1 hidden layer



$L = 2$ layers, 1 hidden layer, $s_1 = 2$

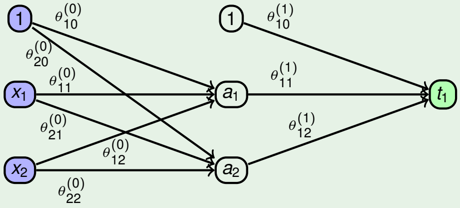
$$\boldsymbol{\theta} = (\Theta^{(0)}, \Theta^{(1)}) \quad \text{parameters}$$

x_1, x_2 input units, t_1 single output unit with logistic sigmoid output activation function

$$t_1 = \sigma\left(\theta_{10}^{(1)} + \theta_{11}^{(1)} a_1 + \theta_{12}^{(1)} a_2\right) = \sigma\left(\theta_{10}^{(1)} + \theta_{11}^{(1)} \sigma(\theta_{10}^{(0)} + \theta_{11}^{(0)} x_1 + \theta_{12}^{(0)} x_2) + \theta_{12}^{(1)} \sigma(\theta_{20}^{(0)} + \theta_{21}^{(0)} x_1 + \theta_{22}^{(0)} x_2)\right)$$

Artificial Neural Networks

ANN with 1 hidden layer



$L = 2$ layers, 1 hidden layer, $s_1 = 2$

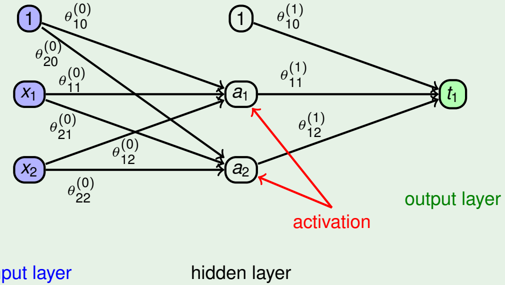
$$\boldsymbol{\theta} = (\Theta^{(0)}, \Theta^{(1)}) \quad \text{parameters}$$

x_1, x_2 input units, t_1 single output unit with logistic sigmoid output activation function

$$t_1 = \sigma\left(\theta^{(1)}_{10} + \theta^{(1)}_{11} a_1 + \theta^{(1)}_{12} a_2\right) = \sigma\left(\theta^{(1)}_{10} + \theta^{(1)}_{11}\sigma(\theta^{(0)}_{10} + \theta^{(0)}_{11} x_1 + \theta^{(0)}_{12} x_2) + \theta^{(1)}_{12}\sigma(\theta^{(0)}_{20} + \theta^{(0)}_{21} x_1 + \theta^{(0)}_{22} x_2)\right)$$

Artificial Neural Networks

ANN with 1 hidden layer



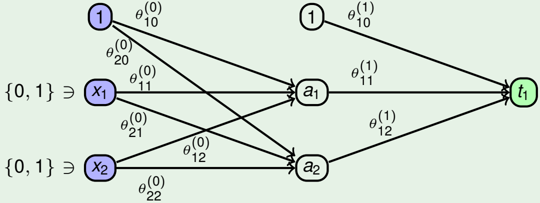
$L = 2$ layers, 1 hidden layer, $s_1 = 2$

$$\boldsymbol{\theta} = (\Theta^{(0)}, \Theta^{(1)}) \quad \text{parameters}$$

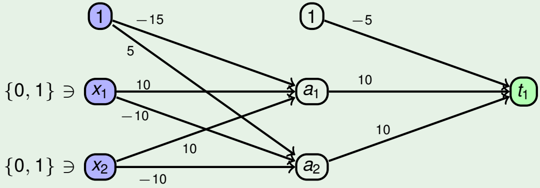
x_1, x_2 input units, t_1 single output unit with logistic sigmoid output activation function

$$t_1 = \sigma\left(\theta^{(1)}_{10} + \theta^{(1)}_{11} a_1 + \theta^{(1)}_{12} a_2\right) = \sigma\left(\theta^{(1)}_{10} + \theta^{(1)}_{11}\sigma(\theta^{(0)}_{10} + \theta^{(0)}_{11} x_1 + \theta^{(0)}_{12} x_2) + \theta^{(1)}_{12}\sigma(\theta^{(0)}_{20} + \theta^{(0)}_{21} x_1 + \theta^{(0)}_{22} x_2)\right)$$

XNOR-like neural network

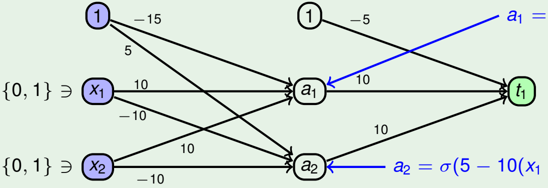


XNOR-like neural network




```
Theta <- list()
Theta[[1]] <- matrix(c(-15,10,10,5,-10,-10),
              ncol = 3, byrow=TRUE)
Theta[[2]] <- c(-5,10,10)

h <- function(x1,x2) {
  a0 <- c(1,x1,x2);
  z1 <- Theta[[1]] %*% a0;
  a1 <- c(1,sigma(z1));
  z2 <- Theta[[2]] %*% a1;
  a2 <- sigma(z2);
  return (a2);
}
hvec <- Vectorize(h)

x1 <- seq(0, 1, .01)
x2 <- seq(0, 1, .01)
t <- outer(x1, x2, hvec)
image(t, main=expression(t=h(x[1],x[2])),
      sub="white=1, black=0", xlab=~x[1],
      ylab=~x[2],col=gray(0:255 / 255),cex=3)

B <- matrix(c(0,0,1,1,0,1,0,1), ncol=2,
  dimnames = list(NULL, c("x1","x2")))
t1 <- hvec(B[,1],B[,2])

truthtable <- cbind(B,t1)
```

XNOR-like neural network



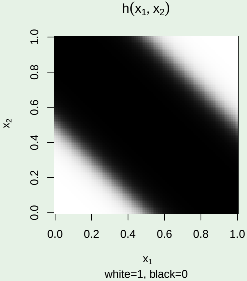
$a_1 = \sigma(10(x_1 + x_2) - 15) \approx x_1 \text{ AND } x_2$

$t_1 = \sigma(10(x_1 + x_2) - 5) \approx a_1 \text{ OR } a_2$

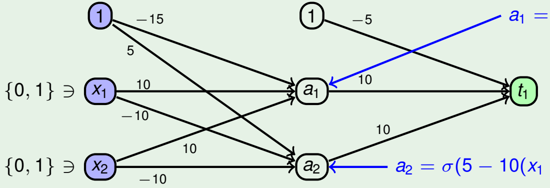
$a_2 = \sigma(5 - 10(x_1 + x_2)) \approx \neg x_1 \text{ AND } \neg x_2$

```
Theta <- list()
Theta[[1]] <- matrix(c(-15,10,10,5,-10,-10),
              ncol = 3, byrow=TRUE)
Theta[[2]] <- c(-5,10,10)

h <- function(x1,x2) {
  a0 <- c(1,x1,x2);
  z1 <- Theta[[1]] %*% a0;
  a1 <- c(1,sigma(z1));
  z2 <- Theta[[2]] %*% a1;
  a2 <- sigma(z2);
  return (a2);
}
hvec <- Vectorize(h)

x1 <- seq(0, 1, .01)
x2 <- seq(0, 1, .01)
t <- outer(x1, x2, hvec)
image(t, main=expression(t=h(x[1],x[2])),
      sub="white=1, black=0", xlab=~x[1],
      ylab=~x[2],col=gray(0:255 / 255),cex=3)

B <- matrix(c(0,0,1,1,0,1,0,1), ncol=2,
  dimnames = list(NULL, c("x1","x2")))
t1 <- hvec(B[,1],B[,2])

truthtable <- cbind(B,t1)
```

```
> Theta
[[1]]
     [,1] [,2] [,3]
[1,]  -15   10   10
[2,]    5  -10  -10

[[2]]
[1] -5 10 10

> truthtable
     x1 x2          t1
[1,]  0  0 0.992847212
[2,]  0  1 0.007644136
[3,]  1  0 0.007644136
[4,]  1  1 0.992847212
```



XNOR-like neural network (could not be achieved with logistic regression)



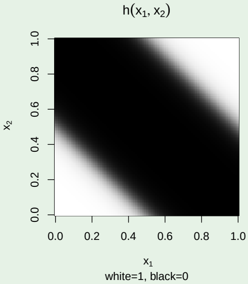
$$a_1 = \sigma(10(x_1 + x_2) - 15) \approx x_1 \text{ AND } x_2$$

$$t_1 = \sigma(10(x_1 + x_2) - 5) \approx a_1 \text{ OR } a_2$$

$$a_2 = \sigma(5 - 10(x_1 + x_2)) \approx \neg x_1 \text{ AND } \neg x_2$$

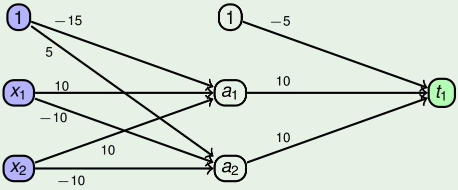
```r
Theta <- list()
Theta[[1]] <- matrix(c(-15,10,10,5,-10,-10),
                ncol = 3, byrow=TRUE)
Theta[[2]] <- c(-5,10,10)

h <- function(x1,x2) {
  a0 <- c(1,x1,x2);
  z1 <- Theta[[1]] %*% a0;
  a1 <- c(1,sigma(z1));
  z2 <- Theta[[2]] %*% a1;
  a2 <- sigma(z2);
  return (a2);
}
hvec <- Vectorize(h)

x1 <- seq(0, 1, .01)
x2 <- seq(0, 1, .01)
t <- outer(x1, x2, hvec)
image(t, main=expression(t=h(x[1],x[2])),
      sub="white=1,␣black=0", xlab=~x[1],
      ylab=~x[2],col=gray(0:255 / 255),cex=3)

B <- matrix(c(0,0,1,1,0,1,0,1), ncol=2,
  dimnames = list(NULL, c("x1","x2")))
t1 <- hvec(B[,1],B[,2])

truthtable <- cbind(B,t1)
```

```
> Theta
[[1]]
     [,1] [,2] [,3]
[1,]  -15   10   10
[2,]    5  -10  -10

[[2]]
[1] -5 10 10

> truthtable
     x1 x2          t1
[1,]  0  0 0.992847212
[2,]  0  1 0.007644136
[3,]  1  0 0.007644136
[4,]  1  1 0.992847212
```

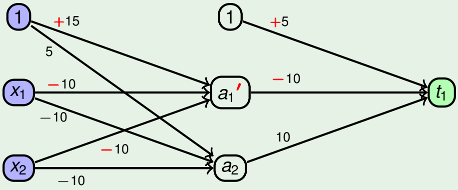


Symmetry in Parameters

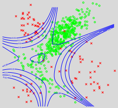
The neural network from above



Different parameters but ANN computes the same function



$$10a_1 = 10\left(1-\sigma\left(-z_1^{(1)}\right)\right) = 10 - 10\sigma\left(-\theta_{10}^{(0)} - \theta_{11}^{(0)}x_1 - \theta_{12}^{(0)}x_2\right)$$



Training

Suppose a training observation

$$\mathbf{y} = (y_1, \ldots, y_K)^T$$

for training input

$$\mathbf{x} = (x_1, \ldots, x_n)^T$$

is given (just one, for now, to keep indices simpler).

Training

Suppose a training observation

$$\mathbf{y} = (y_1, \ldots, y_K)^T$$

for training input

$$\mathbf{x} = (x_1, \ldots, x_n)^T$$

is given (just one, for now, to keep indices simpler).
Consider some error function

$$D(\boldsymbol{\theta})$$

that depends on $\boldsymbol{\theta}$ through the network output $\mathbf{t}$ only, e.g. a suitably defined mean squared error function or a cross-entropy error function.

Training

Suppose a training observation

$$\mathbf{y} = (y_1, \ldots, y_K)^T$$

for training input

$$\mathbf{x} = (x_1, \ldots, x_n)^T$$

is given (just one, for now, to keep indices simpler).
Consider some error function

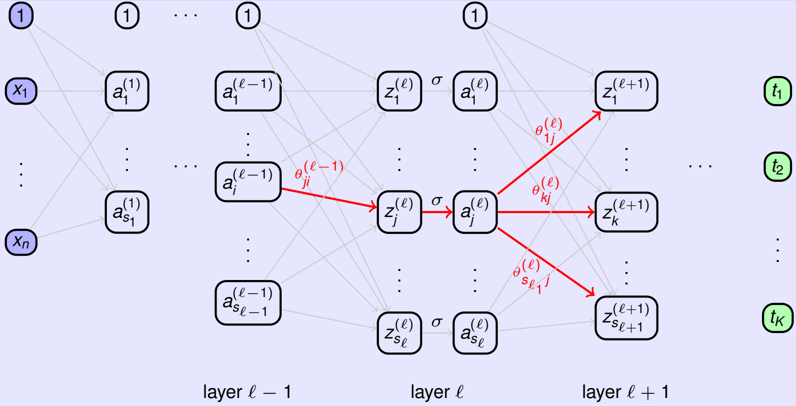
$$D(\boldsymbol{\theta})$$

that depends on $\boldsymbol{\theta}$ through the network output $\mathbf{t}$ only, e.g. a suitably defined mean squared error function or a cross-entropy error function.
To minimize the $D(\boldsymbol{\theta})$ (chosen below) we will require to compute the partial derivatives

$$\frac{\partial D}{\partial \theta_{ji}^{(\ell-1)}} \qquad (1 \leq \ell \leq L, 1 \leq j \leq s_\ell, 0 \leq i \leq s_{\ell-1}).$$

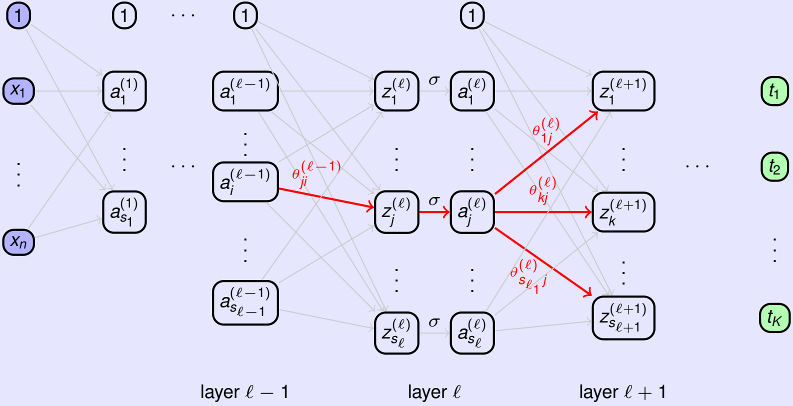
$D(\boldsymbol{\theta})$ may be non-convex and have local, non-global minima.

ANN Training



Let the activations $\mathbf{a}^{(\ell)}$ and the net inputs $\mathbf{z}^{(\ell)}$ be defined as in the definition of ANNs (1).

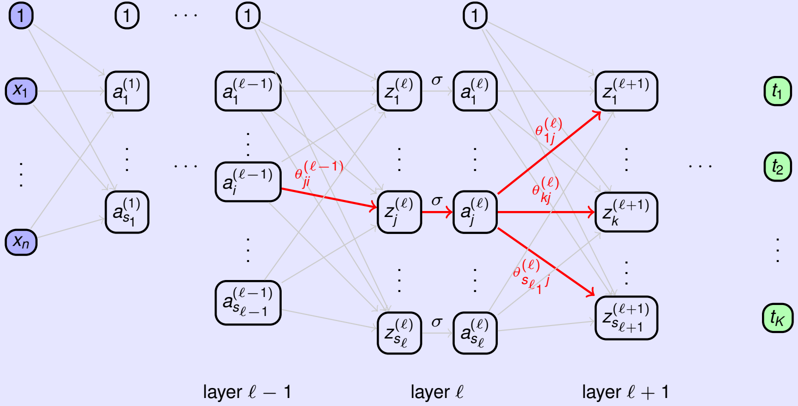
ANN Training



Let the activations $\mathbf{a}^{(\ell)}$ and the net inputs $\mathbf{z}^{(\ell)}$ be defined as in the definition of ANNs (1). In particular,

$$z_j^{(\ell)} = \sum_r \theta_{jr}^{(\ell-1)} a_r^{(\ell-1)}.$$

ANN Training



Let the activations $\mathbf{a}^{(\ell)}$ and the net inputs $\mathbf{z}^{(\ell)}$ be defined as in the definition of ANNs (1). In particular,

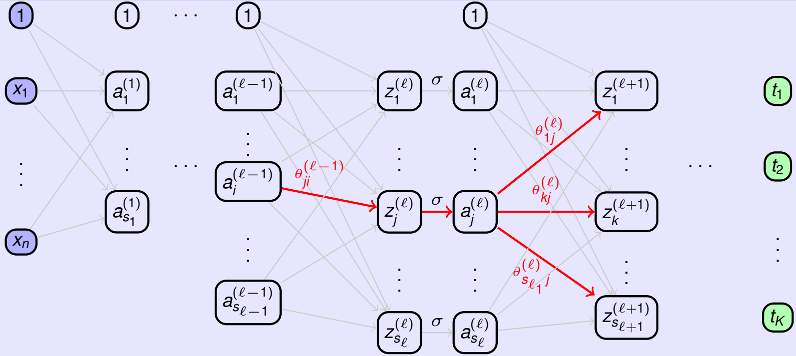
$$z_j^{(\ell)} = \sum_r \theta_{jr}^{(\ell-1)} a_r^{(\ell-1)}.$$

Then

$$\frac{\partial D}{\partial \theta_{ji}^{(\ell-1)}} = \frac{\partial D}{\partial z_j^{(\ell)}} \frac{\partial z_j^{(\ell)}}{\partial \theta_{ji}^{(\ell-1)}} \quad \text{(chain rule)}$$

$$= \delta_j^{(\ell)} a_i^{(\ell-1)} \qquad \text{with } \delta_j^{(\ell)} := \frac{\partial D}{\partial z_j^{(\ell)}}.$$

ANN Training



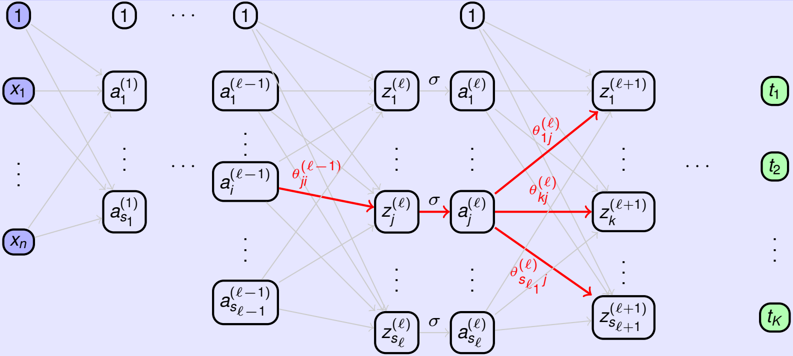
Let $\ell < L$. Use the chain rule for a function chain

$$\begin{aligned} \mathbb{R} &\to \mathbb{R}^{s_{\ell+1}} \to \mathbb{R} \\ z_j^{(\ell)} &\mapsto \mathbf{z}^{(\ell+1)} \mapsto D \end{aligned}$$

to compute for $j = 1, \dots, s_\ell$

$$\delta_j^{(\ell)} = \frac{\partial D}{\partial z_j^{(\ell)}}$$

ANN Training



Let $\ell < L$. Use the chain rule for a function chain

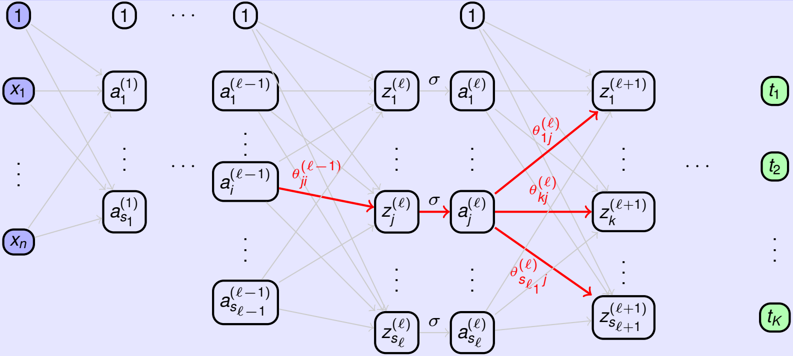
$$\begin{aligned} \mathbb{R} &\to \mathbb{R}^{s_{\ell+1}} \to \mathbb{R} \\ z_j^{(\ell)} &\mapsto \mathbf{z}^{(\ell+1)} \mapsto D \end{aligned}$$

to compute for $j = 1, \ldots, s_\ell$

$$\delta_j^{(\ell)} = \frac{\partial D}{\partial z_j^{(\ell)}}$$

$$\delta_j^{(\ell)} = \left(\frac{\partial D}{\partial z_1^{(\ell+1)}}, \cdots, \frac{\partial D}{\partial z_{s_{\ell+1}}^{(\ell+1)}} \right) \begin{pmatrix} \frac{\partial z_1^{(\ell+1)}}{\partial z_j^{(\ell)}} \\ \vdots \\ \frac{\partial z_{s_{\ell+1}}^{(\ell+1)}}{\partial z_j^{(\ell)}} \end{pmatrix}$$

ANN Training



Let $\ell < L$. Use the chain rule for a function chain

$$\begin{aligned} \mathbb{R} &\to \mathbb{R}^{s_{\ell+1}} \to \mathbb{R} \\ z_j^{(\ell)} &\mapsto \mathbf{z}^{(\ell+1)} \mapsto D \end{aligned}$$

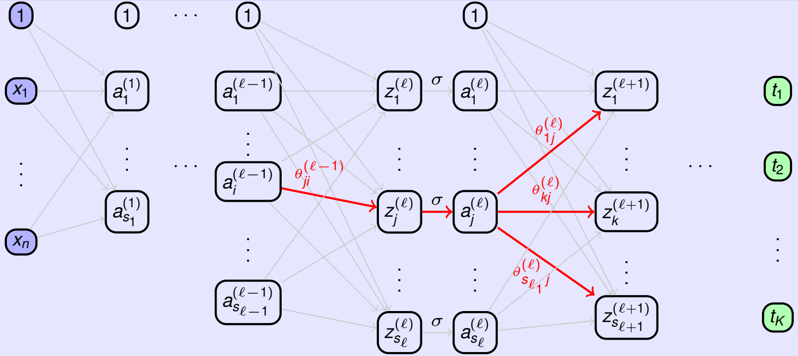
to compute for $j = 1, \ldots, s_\ell$

$$\delta_j^{(\ell)} = \frac{\partial D}{\partial z_j^{(\ell)}}$$

$$\delta_j^{(\ell)} = \begin{pmatrix} \delta_1^{(\ell+1)} \\ \vdots \\ \delta_{s_{\ell+1}}^{(\ell+1)} \end{pmatrix}^T \begin{pmatrix} \frac{\partial}{\partial z_j^{(\ell)}} \theta_{1j}^{(\ell)} \sigma(z_j^{(\ell)}) \\ \vdots \\ \frac{\partial}{\partial z_j^{(\ell)}} \theta_{s_{\ell+1}j}^{(\ell)} \sigma(z_j^{(\ell)}) \end{pmatrix}$$

$=: \boldsymbol{\delta}^{(\ell+1)}$

ANN Training



Let $\ell < L$. Use the chain rule for a function chain

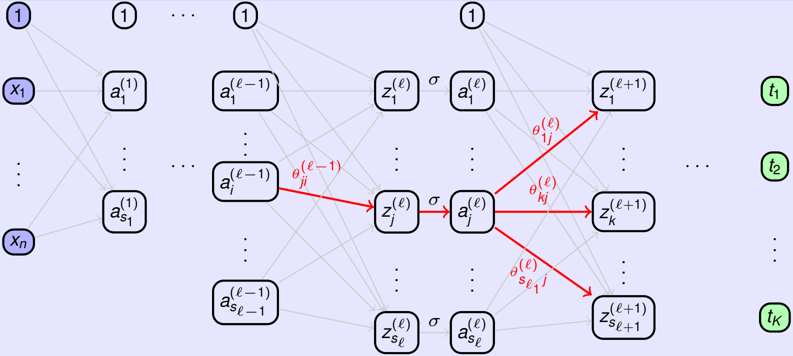
$$\begin{aligned} \mathbb{R} &\to \mathbb{R}^{s_{\ell+1}} \to \mathbb{R} \\ z_j^{(\ell)} &\mapsto \mathbf{z}^{(\ell+1)} \mapsto D \end{aligned}$$

to compute for $j = 1, \ldots, s_\ell$

$$\delta_j^{(\ell)} = \frac{\partial D}{\partial z_j^{(\ell)}}$$

$$\delta_j^{(\ell)} = (\boldsymbol{\delta}^{(\ell+1)})^T \begin{pmatrix} \theta_{1j}^{(\ell)} \\ \vdots \\ \theta_{s_{\ell+1}j}^{(\ell)} \end{pmatrix} a_j^{(\ell)}(1 - a_j^{(\ell)})$$

ANN Training



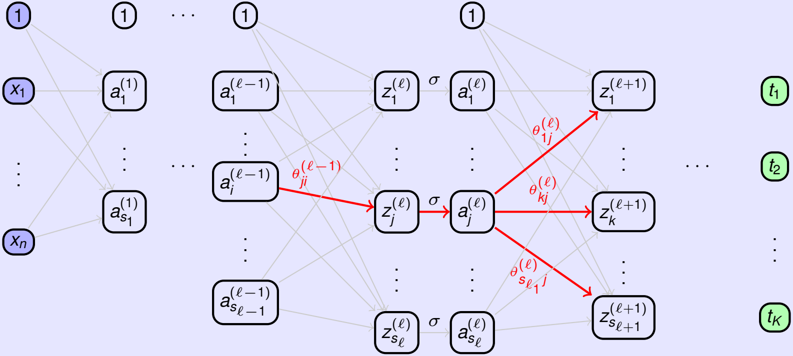
In matrix notation

$$(\boldsymbol{\delta}^{(\ell)})^T = (\boldsymbol{\delta}^{(\ell+1)})^T \cdot \Theta_{-0}^{(\ell)} \circ \left(\mathbf{a}_{-0}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}_{-0}^{(\ell)})\right)^T$$

Here,

- the subscript -0 at $\Theta_{-0}^{(\ell)}$ denotes that the column with index 0 is omitted from $\Theta^{(\ell)}$, similarly $\mathbf{a}_{-0}^{(\ell)}$ is $\mathbf{a}^{(\ell)}$ with the 0-th component removed (the terms corresponding to the bias)
- $\circ$ denotes componentwise multiplication (Hadamard product, operator * in R, .* in MATLAB).

ANN Training



In matrix notation

$$(\boldsymbol{\delta}^{(\ell)})^T = (\boldsymbol{\delta}^{(\ell+1)})^T \cdot \Theta_{-0}^{(\ell)} \circ \left(\mathbf{a}_{-0}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}_{-0}^{(\ell)})\right)^T$$

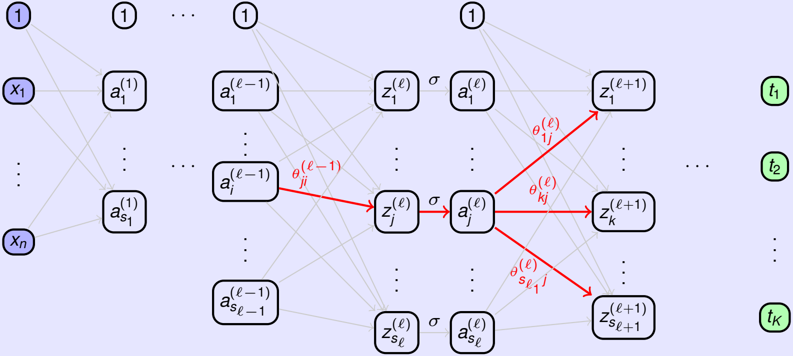
Here,

- the subscript -0 at $\Theta_{-0}^{(\ell)}$ denotes that the column with index 0 is omitted from $\Theta^{(\ell)}$, similarly $\mathbf{a}_{-0}^{(\ell)}$ is $\mathbf{a}^{(\ell)}$ with the 0-th component removed (the terms corresponding to the bias)
- $\circ$ denotes componentwise multiplication (Hadamard product, operator * in R, .* in MATLAB).

Transposing, we obtain the backwards propagation recursion

$$\boxed{\boldsymbol{\delta}^{(\ell)} = (\boldsymbol{\Theta}_{-0}^{(\ell)})^T \boldsymbol{\delta}^{(\ell+1)} \circ \mathbf{a}_{-0}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}_{-0}^{(\ell)})}$$

ANN Training



In matrix notation

$$(\boldsymbol{\delta}^{(\ell)})^T = (\boldsymbol{\delta}^{(\ell+1)})^T \cdot \Theta_{-0}^{(\ell)} \circ \left(\mathbf{a}_{-0}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}_{-0}^{(\ell)})\right)^T$$

Here,

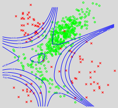
- the subscript -0 at $\Theta_{-0}^{(\ell)}$ denotes that the column with index 0 is omitted from $\Theta^{(\ell)}$, similarly $\mathbf{a}_{-0}^{(\ell)}$ is $\mathbf{a}^{(\ell)}$ with the 0-th component removed (the terms corresponding to the bias)
- $\circ$ denotes componentwise multiplication (Hadamard product, operator * in R, .* in MATLAB).

Transposing, we obtain the backwards propagation recursion

$$\boxed{\boldsymbol{\delta}^{(\ell)} = (\boldsymbol{\Theta}_{-0}^{(\ell)})^T \boldsymbol{\delta}^{(\ell+1)} \circ \mathbf{a}_{-0}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}_{-0}^{(\ell)})}$$

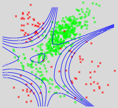
The deltas are called errors.

We need to know D (only) to compute the initial case, $\boldsymbol{\delta}^{(L)}$.



Artificial Neural Networks for Regression

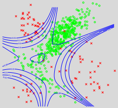
Consider first a single training instance $\mathbf{x} \in R^n$ with a single output $\mathbf{y} \in R^K$.



Artificial Neural Networks for Regression

Consider first a single training instance $\mathbf{x} \in R^n$ with a single output $\mathbf{y} \in R^K$.

- Choose the identity function as output activation function $\mathbf{g}$, i.e. $\mathbf{t} = \mathbf{z}^{(L)}$ (unbounded).

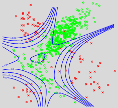


Artificial Neural Networks for Regression

Consider first a single training instance $\mathbf{x} \in R^n$ with a single output $\mathbf{y} \in R^K$.

- Choose the identity function as output activation function $\mathbf{g}$, i.e. $\mathbf{t} = \mathbf{z}^{(L)}$ (unbounded).
- Choose a squared error function

$$D(\boldsymbol{\theta}) := \frac{1}{2} \|\mathbf{t} - \mathbf{y}\|_2^2 = \frac{1}{2} \sum_{k=1}^{K} (t_k - y_k)^2$$



Artificial Neural Networks for Regression

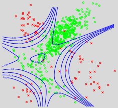
Consider first a single training instance $\mathbf{x} \in R^n$ with a single output $\mathbf{y} \in R^K$.

- Choose the identity function as output activation function $\mathbf{g}$, i.e. $\mathbf{t} = \mathbf{z}^{(L)}$ (unbounded).
- Choose a squared error function

$$D(\boldsymbol{\theta}) := \frac{1}{2} \|\mathbf{t} - \mathbf{y}\|_2^2 = \frac{1}{2} \sum_{k=1}^{K} (t_k - y_k)^2$$

- Then

$$\delta_j^{(L)} = \frac{\partial D}{\partial z_j^{(L)}} = z_j^{(L)} - y_j$$



Artificial Neural Networks for Regression

Consider first a single training instance $\mathbf{x} \in R^n$ with a single output $\mathbf{y} \in R^K$.

- Choose the identity function as output activation function $\mathbf{g}$, i.e. $\mathbf{t} = \mathbf{z}^{(L)}$ (unbounded).
- Choose a squared error function

$$D(\boldsymbol{\theta}) := \frac{1}{2} \|\mathbf{t} - \mathbf{y}\|_2^2 = \frac{1}{2} \sum_{k=1}^{K} (t_k - y_k)^2$$

- Then

$$\delta_j^{(L)} = \frac{\partial D}{\partial z_j^{(L)}} = z_j^{(L)} - y_j$$

or in vector notation $\boxed{\boldsymbol{\delta}^{(L)} = \mathbf{z}^{(L)} - \mathbf{y} = \mathbf{t} - \mathbf{y}.}$

Artificial Neural Networks for Classification

Again, consider first a single training instance. Here, let output $\mathbf{y} \in R^K$ be encoded such that

$$\mathbf{y} = \mathbf{e}_c = c\text{-th unit vector}$$

if $c \in \{1, 2, \ldots, K\}$ is the true class of the learning instance.

Artificial Neural Networks for Classification

Again, consider first a single training instance. Here, let output $\mathbf{y} \in R^K$ be encoded such that

$$\mathbf{y} = \mathbf{e}_c = c\text{-th unit vector}$$

if $c \in \{1, 2, \ldots, K\}$ is the true class of the learning instance.

- Choose the softmax function as output activation function $\mathbf{g}$.

$$\mathbf{t} = \mathbf{g}(z_1, \ldots, z_K) = \frac{1}{\sum_{k=1}^{K} e^{z_k}} \begin{pmatrix} e^{z_1} \\ \vdots \\ e^{z_K} \end{pmatrix}$$

Artificial Neural Networks for Classification

Again, consider first a single training instance. Here, let output $\mathbf{y} \in R^K$ be encoded such that

$$\mathbf{y} = \mathbf{e}_c = c\text{-th unit vector}$$

if $c \in \{1, 2, \ldots, K\}$ is the true class of the learning instance.

- Choose the softmax function as output activation function $\mathbf{g}$.

$$\mathbf{t} = \mathbf{g}(z_1, \ldots, z_K) = \frac{1}{\sum_{k=1}^{K} e^{z_k}} \begin{pmatrix} e^{z_1} \\ \vdots \\ e^{z_K} \end{pmatrix}$$

  Properties:

  - $0 < t_k < 1$, $t_1 + \cdots + t_K = 1$
  - if $z_j \gg z_i$ for $i \neq j$, then $t_j \approx 1$ and $t_i \approx 0$ for $i \neq j$
  - softmax is a generalization of the logistic sigmoid function to more than 2 classes: $K = 2, z_1 = 0 \Rightarrow t_2 = \sigma(z_2)$

Cross-entropy error function for multiclass problem

Consider the true class of the training instance to be a random variable $C \in \{1, \ldots, K\}$ with observation c and the output vector $\mathbf{t}$ of the net to be the parameters of a multinomial distribution for C. Then the likelihood is

$$L(\boldsymbol{\theta}) =$$

Artificial Neural Networks for Classification

Cross-entropy error function for multiclass problem

Consider the true class of the training instance to be a random variable $C \in \{1, \ldots, K\}$ with observation c and the output vector $\mathbf{t}$ of the net to be the parameters of a multinomial distribution for C. Then the likelihood is

$$L(\boldsymbol{\theta}) = t_c$$

Cross-entropy error function for multiclass problem

Consider the true class of the training instance to be a random variable $C \in \{1, \ldots, K\}$ with observation c and the output vector $\mathbf{t}$ of the net to be the parameters of a multinomial distribution for C. Then the likelihood is

$$L(\boldsymbol{\theta}) = t_c$$

and we seek to minimize the negative log-likelihood

$$-\ln L(\boldsymbol{\theta}) = -\ln t_c = -\sum_{k=1}^{K} y_k \ln t_k$$

Cross-entropy error function for multiclass problem

Consider the true class of the training instance to be a random variable $C \in \{1, \ldots, K\}$ with observation c and the output vector $\mathbf{t}$ of the net to be the parameters of a multinomial distribution for C. Then the likelihood is

$$L(\boldsymbol{\theta}) = t_c$$

and we seek to minimize the negative log-likelihood

$$-\ln L(\boldsymbol{\theta}) = -\ln t_c = -\sum_{k=1}^{K} y_k \ln t_k$$

Define the cross-entropy error function

$$D(\boldsymbol{\theta}) = -\sum_{k=1}^{K} y_k \ln t_k.$$

Artificial Neural Networks for Classification

Cross-entropy error function for multiclass problem

Consider the true class of the training instance to be a random variable $C \in \{1, \dots, K\}$ with observation c and the output vector $\mathbf{t}$ of the net to be the parameters of a multinomial distribution for C. Then the likelihood is

$$L(\boldsymbol{\theta}) = t_c$$

and we seek to minimize the negative log-likelihood

$$-\ln L(\boldsymbol{\theta}) = -\ln t_c = -\sum_{k=1}^{K} y_k \ln t_k$$

Define the cross-entropy error function

$$D(\boldsymbol{\theta}) = -\sum_{k=1}^{K} y_k \ln t_k .$$

For $K = 2$ this yields

$$D(\boldsymbol{\theta}) := -\big(y_2 \ln t_2 + (1 - y_2) \ln(1 - t_2)\big),$$

the cross-entropy function for logistic regression with 2 classes.

Artificial Neural Networks for Classification

Cross-entropy error function for multiclass problem

In the case of

$$D(\boldsymbol{\theta}) = -\sum_{k=1}^{K} y_k \ln t_k$$

the errors for the last layer (start of backwards propagation) are computed as follows. Let $c \in \{1, \dots, K\}$ be such that $y_c = 1$. Then

$$\delta_j^{(L)} = \frac{\partial D}{\partial z_j^{(L)}} = -\frac{\partial}{\partial z_j^{(L)}} \ln t_c = -\frac{\partial}{\partial z_j^{(L)}} \ln \frac{e^{z_c^{(L)}}}{\sum_{k=1}^{K} e^{z_k^{(L)}}} = \frac{\partial}{\partial z_j^{(L)}} \left\{ \ln \left(\sum_{k=1}^{K} e^{z_k^{(L)}} \right) - z_c^{(L)} \right\}$$

For $j \neq c$ we obtain

Artificial Neural Networks for Classification

Cross-entropy error function for multiclass problem

In the case of

$$D(\boldsymbol{\theta}) = -\sum_{k=1}^{K} y_k \ln t_k$$

the errors for the last layer (start of backwards propagation) are computed as follows. Let $c \in \{1, \dots, K\}$ be such that $y_c = 1$. Then

$$\delta_j^{(L)} = \frac{\partial D}{\partial z_j^{(L)}} = -\frac{\partial}{\partial z_j^{(L)}} \ln t_c = -\frac{\partial}{\partial z_j^{(L)}} \ln \frac{e^{z_c^{(L)}}}{\sum_{k=1}^{K} e^{z_k^{(L)}}} = \frac{\partial}{\partial z_j^{(L)}} \left\{ \ln \left(\sum_{k=1}^{K} e^{z_k^{(L)}} \right) - z_c^{(L)} \right\}$$

For $j \neq c$ we obtain $\delta_j^{(L)} = t_j$ and for $j = c$ we obtain

Cross-entropy error function for multiclass problem

In the case of

$$D(\boldsymbol{\theta}) = -\sum_{k=1}^{K} y_k \ln t_k$$

the errors for the last layer (start of backwards propagation) are computed as follows. Let $c \in \{1, \ldots, K\}$ be such that $y_c = 1$. Then

$$\delta_j^{(L)} = \frac{\partial D}{\partial z_j^{(L)}} = -\frac{\partial}{\partial z_j^{(L)}} \ln t_c = -\frac{\partial}{\partial z_j^{(L)}} \ln \frac{e^{z_c^{(L)}}}{\sum_{k=1}^{K} e^{z_k^{(L)}}} = \frac{\partial}{\partial z_j^{(L)}} \left\{ \ln \left(\sum_{k=1}^{K} e^{z_k^{(L)}} \right) - z_c^{(L)} \right\}$$

For $j \neq c$ we obtain $\delta_j^{(L)} = t_j$ and for $j = c$ we obtain $\delta_j^{(L)} = t_j - 1$.

In vector notation:

Artificial Neural Networks for Classification

Cross-entropy error function for multiclass problem

In the case of

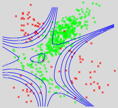
$$D(\boldsymbol{\theta}) = -\sum_{k=1}^{K} y_k \ln t_k$$

the errors for the last layer (start of backwards propagation) are computed as follows. Let $c \in \{1, \ldots, K\}$ be such that $y_c = 1$. Then

$$\delta_j^{(L)} = \frac{\partial D}{\partial z_j^{(L)}} = -\frac{\partial}{\partial z_j^{(L)}} \ln t_c = -\frac{\partial}{\partial z_j^{(L)}} \ln \frac{e^{z_c^{(L)}}}{\sum_{k=1}^{K} e^{z_k^{(L)}}} = \frac{\partial}{\partial z_j^{(L)}} \left\{ \ln \left(\sum_{k=1}^{K} e^{z_k^{(L)}} \right) - z_c^{(L)} \right\}$$

For $j \neq c$ we obtain $\delta_j^{(L)} = t_j$ and for $j = c$ we obtain $\delta_j^{(L)} = t_j - 1$.

In vector notation: $\boxed{\boldsymbol{\delta}^{(L)} = \mathbf{t} - \mathbf{y}}$



Artificial Neural Networks

Training set

Let

$$\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$$

be training inputs and

$$\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}$$

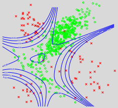
be the corresponding training outputs. Let

$$\mathbf{t}^{(i)} = h_\Theta(\mathbf{x}^{(i)}) \qquad (i = 1..m)$$

be the outputs of the ANN. Put a upper left index i on the errors and net inputs that denotes the sample number:

$${}^{i}\delta_j^{(\ell)}$$

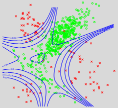
$${}^{i}z_j^{(\ell)}$$



Artificial Neural Networks

Error function with regularization

$$E(\boldsymbol{\theta}) = D(\boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta})$$



Artificial Neural Networks

Error function with regularization

$$E(\boldsymbol{\theta}) = D(\boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta})$$

An easy-to-differentiate choice for the regularization term is

$$R(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{\ell=0}^{L-1} \sum_{i=1}^{s_\ell} \sum_{j=1}^{s_{\ell+1}} \left(\theta_{ji}^{(\ell)}\right)^2 .$$

Note that the bias terms $\theta_{j0}^{(\ell)}$ are left out.

Artificial Neural Networks

Error function with regularization for regression

$$E(\boldsymbol{\theta}) = \frac{1}{2m}\left\{\sum_{i=1}^{m}\sum_{k=1}^{K}\left(t_k^{(i)} - y_k^{(i)}\right)^2 + \lambda\sum_{\ell=0}^{L-1}\sum_{i=1}^{s_\ell}\sum_{j=1}^{s_{\ell+1}}\left(\theta_{ji}^{(\ell)}\right)^2\right\} \tag{2}$$

Error function with regularization for regression

$$E(\boldsymbol{\theta}) = \frac{1}{2m}\left\{\sum_{i=1}^{m}\sum_{k=1}^{K}\left(t_k^{(i)} - y_k^{(i)}\right)^2 + \lambda \sum_{\ell=0}^{L-1}\sum_{i=1}^{s_\ell}\sum_{j=1}^{s_{\ell+1}}\left(\theta_{ji}^{(\ell)}\right)^2\right\} \tag{2}$$

Error function with regularization for classification

$$E(\boldsymbol{\theta}) = \frac{1}{m}\left\{-\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \ln t_k^{(i)} + \frac{\lambda}{2}\sum_{\ell=0}^{L-1}\sum_{i=1}^{s_\ell}\sum_{j=1}^{s_{\ell+1}}\left(\theta_{ji}^{(\ell)}\right)^2\right\} \tag{3}$$

Derivatives with respect to the parameters θ (both regression and classification)

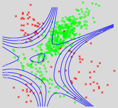
Putting above results together we obtain the following backwards propagation (backprop) recursions to compute the “gradient” of the parameters

$$\frac{\partial E}{\partial \theta_{jk}^{(\ell)}} = \frac{1}{m}\left\{\sum_{i=1}^{m} {}^{i}\delta_j^{(\ell+1)} \cdot {}^{i}a_k^{(\ell)} + \lambda\theta_{jk}^{(\ell)}\right\} \qquad \begin{array}{l}(0 \le \ell < L,\\ 1 \le k \le s_\ell,\\ 1 \le j \le s_{\ell+1})\end{array} \tag{4}$$

$$\frac{\partial E}{\partial \theta_{j0}^{(\ell)}} = \frac{1}{m}\sum_{i=1}^{m} {}^{i}\delta_j^{(\ell+1)} \qquad (0 \le \ell < L, 1 \le j \le s_{\ell+1}) \tag{5}$$

$${}^{i}\boldsymbol{\delta}^{(\ell)} = (\Theta_{-0}^{(\ell)})^T\, {}^{i}\boldsymbol{\delta}^{(\ell+1)} \circ {}^{i}\mathbf{a}_{-0}^{(\ell)} \circ (\mathbf{1} - {}^{i}\mathbf{a}_{-0}^{(\ell)}) \qquad (0 \le \ell < L) \tag{6}$$

$${}^{i}\boldsymbol{\delta}^{(L)} = \mathbf{t}^{(i)} - \mathbf{y}^{(i)} \tag{7}$$



ANN Forward Propagation Algorithm

Let an ANN with L layers, with parameters as in the definition and output activation function $\mathbf{g}$ be given.
Let $\mathbf{x}$ be an input vector.
The recursions defining the ANN directly suggest the following algorithm to compute the activations $\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(L-1)}$ and the output vector $\mathbf{t}$.

FORWARDPROP($\mathbf{x}$)

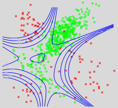
```
r ← ()     // will hold result: (a^(1), ..., a^(L-1), t)
a ← x
prepend 1 to a
for ℓ = 1..L − 1 do
    a ← z ← Θ^(ℓ−1) a
    apply σ elementwise to a and then prepend 1 to a
    append a to r
z ← Θ^(L−1) a // net input of last layer L
t ← g(z)
append t to r
return r
```

NN.r ...

```r
# artificial neural network (ANN)
# Mario Stanke, 11.11.2014

# logistic sigmoid function
sigma <- function(t){
   return (1/(1 + exp(-t)))
}
# softmax function
softmax <- function(z){
   z <- exp(z)
   return (z/sum(z))
}

setClass(Class = "ANN",
   representation = representation(L = "numeric",
                              sizes = "numeric",
                              theta = "list",
                              regression = "logical"),
   validity = function(object){
       if (object@L+1 == length(object@sizes)){
          return(TRUE)
       } else {
          stop ("[ANN: validation] The array ",
               "\'sizes\' must contain L+1 ",
               "elements.")
       }
   }
)
setGeneric (
   name = "ForwardProp",
   def  = function(nn, x, Theta){
            standardGeneric("ForwardProp")
         }
)
```

... NN.r

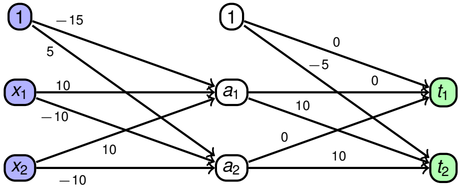
```r
setMethod (
    f = "ForwardProp",
    signature = "ANN",
    definition = function(nn, x, Theta){
       if (length(x) != nn@sizes[1]){
          stop("[ANN: ForwardProp] Input ",
               "vector has a different size",
               " than the input layer.")
       }
       if (missing(Theta)) {
          Theta <- nn@Theta
       }
       r = list()
       a <- c(1,x)
       for (l in 1:(nn@L-1)) {
          z <- Theta[[l]] %*% a
          a <- c(1,sigma(z))
          r[[l]] <- a
       }
       z <- Theta[[nn@L]] %*% a
       if (nn@regression) {
          t <- z
       } else {# using ANN for classification
          t <- softmax(z)
       }
       r[[nn@L]] <- t
       return (r)
    }
)
```



... NN.r ...

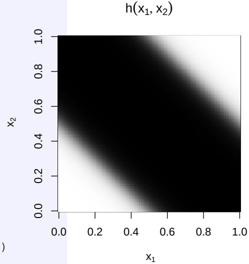
```
# a function that returns just the prediction
setGeneric (
   name = "predict",
   def = function(nn, x){standardGeneric("predict")}
)

setMethod (
   f = "predict",
   signature = "ANN",
   definition = function(nn, x){
      A <- ForwardProp(nn, x)
      return (A[[nn@L]])
   }
)
```

XNOR-like neural network (revisited)




```r
source("NN.r")
Theta <- list()
Theta[[1]] <- matrix(c(-15,10,10,5,-10,-10),
                     ncol = 3, byrow=TRUE)
Theta[[2]] <- matrix(c(0,-5,0,10,0,10),
                     ncol=3)

ann = new ("ANN", L=2,
          sizes = c(2,2,2),
          Theta = Theta,
          regression = FALSE)

ForwardProp(ann, c(0,0))

h <- function(x1,x2) {
  return (predict(ann, c(x1,x2))[2])
}
hvec <- Vectorize(h)

x1 <- seq(0, 1, .01)
x2 <- seq(0, 1, .01)
y <- outer(x1, x2, hvec)
image(y, main=expression(y=h(x[1],x[2])),
      sub="white=1,␣black=0", xlab=~x[1],
      ylab=~x[2],col=gray(0:255 / 255),cex=3)
```

```
> ann
An object of class "ANN"
Slot "L":
[1] 2
Slot "sizes":
[1] 2 2 2
Slot "Theta":
[[1]]
     [,1] [,2] [,3]
[1,]  -15   10   10
[2,]    5  -10  -10

[[2]]
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]   -5   10   10

Slot "regression":
[1] FALSE


> ForwardProp(ann, c(0,0))
[[1]]
[1] 1.000e+00 3.059e-07 9.933e-01

[[2]]
            [,1]
[1,] 0.007152788
[2,] 0.992847212
```



white=1, black=0

ANN Backward Propagation Algorithm (both Regression and Classification)

In a given training set, let $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$ be the inputs with corresponding outputs $\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}$. Let $\lambda \geq 0$ be the L^2-regularization parameter.
The following recursion computes the partial derivatives ("gradient") or the error function wrt. all parameters

$$D_{jk}^{(\ell)} := \frac{\partial E(\theta)}{\partial \theta_{jk}^{(\ell)}} \qquad (0 \leq \ell < L,\ 0 \leq k \leq s_\ell,\ 1 \leq j \leq s_{\ell+1}).$$

$\textsc{BackProp}(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}, \mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}, \lambda)$

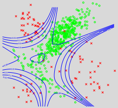
$D_{ij}^{(\ell)} \leftarrow 0 \quad (\forall \ell, i, j)$ // initialize derivates
for $i = 1..m$ **do**
  $(\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(L-1)}, \mathbf{t}) \leftarrow \textsc{ForwardProp}(\mathbf{x}^{(i)})$
  $\boldsymbol{\delta} \leftarrow \mathbf{t} - \mathbf{y}^{(i)}$
  for $\ell = L - 1..1$ **do**
    $D^{(\ell)} \leftarrow D^{(\ell)} + \boldsymbol{\delta} \cdot (\mathbf{a}^{(\ell)})^T$
    $\boldsymbol{\delta} \leftarrow (\Theta^{(\ell)})^T \cdot \boldsymbol{\delta} \circ \mathbf{a}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}^{(\ell)})$
    delete first element of $\boldsymbol{\delta}$ // bias term not used/defined
  $D^{(0)} \leftarrow D^{(0)} + \boldsymbol{\delta} \cdot (\mathbf{a}^{(0)})^T$
$D_{jk}^{(\ell)} \leftarrow D_{jk}^{(\ell)} + \lambda \cdot \theta_{jk}^{(\ell)} \quad (\forall \ell, j, k \neq 0)$ // regularization, exluding bias term
$D^{(\ell)} \leftarrow D^{(\ell)} / m \quad (0 \leq \ell < L)$ // averaging over samples
return $(D_{jk}^{(\ell)})_{\ell,j,k}$

... NN.r ... (BackProp)

```
setMethod (
   f = "BackProp",
   signature = "ANN",
   definition = function(nn, X, Y, lambda, Theta){
      if (missing(lambda)) {
          lambda <- 0 # default value for regularization parameter
      }
      if (missing(Theta)) {
          Theta <- nn@Theta
      }
      m <- dim(Y)[1] # number of samples
      if (m != dim(X)[1]){
          stop ("[ANN: BackProp] Number of training inputs and outputs not equal.")
      }
      s = nn@sizes
      L = nn@L
      # initialize derivatives as all zeros, same dimensions as Theta
      D <- list()
      for (l in 1:L) {
          D[[l]] <- matrix (data=0, nrow = s[l+1], ncol = s[l]+1)
      }
      for (i in 1:m) { # loop over samples
          x <- t(X[i,])
          A <- ForwardProp(nn, x, Theta) # compute activations and output
          t <- A[[L]] # output of neural network on this sample
          delta <- t - t(Y[i,])  # error of last layer
          for (l in L:2) {
              D[[l]] <- D[[l]] + delta %*% t(A[[l-1]])
              delta <- t(nn@Theta[[l]]) %*% delta * A[[l-1]] * (1 - A[[l-1]])
              delta <- delta[-1] # throw away 0-th component of delta
          }
          D[[1]] <- D[[1]] + delta %*% t(c(1,x))
      }
      # L2-regularization, excluding bias terms
      for (l in 1:L) {
          D[[l]] <- D[[l]] + lambda * (nn@Theta[[l]] %*% diag(c(0,rep(1,s[l]))))
          D[[l]] <- D[[l]] / m
      }
      return (D)
  }
)
```

... NN.r ... ("gradient" -> gradient)

```
# conversion for parameters between a single vector and a list of matrices
setMethod (
   f = "roll",
   signature= "ANN",
   definition = function(nn, thetavec){
      s <- nn@sizes
      numpars = 0;
      for (l in 1:nn@L) {
          numpars <- numpars + (s[l]+1) * s[l+1]
      }
      if (length(thetavec) != numpars) {
          stop ("[ANN roll] number of parameters does not match total size of weight matrices")
      }
      numpars = 0;
      Theta <- list()
      for (l in 1:nn@L) {
          Theta[[l]] <- matrix(thetavec[(1+numpars) : (numpars + s[l+1] * (s[l]+1))], nrow=s[l+1])
          numpars <- numpars + (s[l]+1) * s[l+1]
      }
      return(Theta)
   }
)
setMethod (
   f = "unroll",
   signature = "ANN",
   definition = function(nn, Theta){
      s <- nn@sizes
      if (missing(Theta)) { Theta <- nn@Theta  }
      numpars <- 0
      for (l in 1:nn@L) {
          numpars <- numpars + (s[l]+1) * s[l+1]
      }
      thetavec <- numeric(numpars)
      numpars <- 0
      for (l in 1:nn@L) {
          thetavec[(1+numpars) : (numpars + (s[l]+1) * s[l+1])] <- as.vector(Theta[[l]])
          numpars <- numpars + (s[l]+1) * s[l+1]
      }
      return (thetavec)
   }
)
```



Check for Errors in BackProp

Second way to approximate gradient (brute force)

As $E(\boldsymbol{\theta})$ is differentiable, we have for small $\epsilon > 0$

$$\frac{\partial E(\boldsymbol{\theta})}{\partial \theta_i} \approx \frac{E(\boldsymbol{\theta} + \epsilon \cdot \mathbf{e}_{\mathbf{i}}) - E(\boldsymbol{\theta} - \epsilon \cdot \mathbf{e}_i)}{2\epsilon},$$

where $\mathbf{e}_i$ is the i-th unit vector.

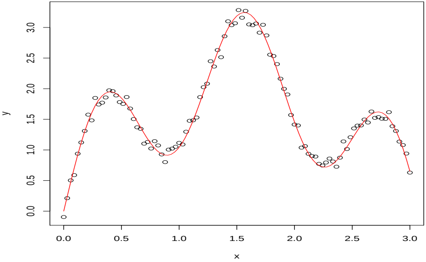
... NN.r ... (compute error $E(\theta)$)

```
setMethod (
   f = "E",
   signature= "ANN",
   definition = function(nn, thetavec, X, Y, lambda){
      if (missing(thetavec)) Theta <- nn@Theta
      else                   Theta <- roll(nn, thetavec)
      # compute data-dependent term D(theta)
      cumD = 0;
      m <- dim(Y)[1] # number of samples
      for (i in 1:m) { # loop over samples
         # compute output of neural network on this sample
         x <- X[i,] # x is a (column) vector
         A <- ForwardProp(nn, x, Theta)
         t <- A[[nn@L]]
         y <- Y[i,] # (column) vector
         if (nn@regression){
            cumD <- cumD + (t-y)^2 / 2
         } else {
            cumD <- cumD - t(y) %*% log(t)
         }
      }
      cumD <- cumD / m
      # compute regularization term R(theta)
      cumR <- 0
      for (l in 1:nn@L) { # square all parameters, except bias terms
         cumR <- cumR + sum((Theta[[l]][,-1])^2)
      }
      cumR <- cumR / 2 / m
      return (cumD + lambda * cumR)
   }
)
```

... `NN.r` ... (compute gradient analytically and numerically)

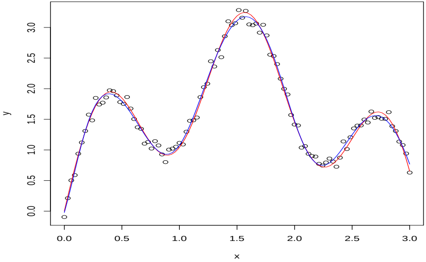
```
# gradient of E(theta)

setMethod (
   f = "gradient",
   signature= "ANN",
   definition = function(nn, X, Y, lambda, thetavec, numeric){
       if (missing(numeric)){
           numeric = FALSE
       }
       if (!numeric) {
           Theta <- roll(nn, thetavec)
           D <- BackProp(nn, X, Y, lambda, Theta=Theta)
           return (unroll(nn, D))
       } else {
           # for testing correctness only,
           # determine gradient numerically by calling E only
           numpars <-  length(thetavec)
           ng <- numeric(numpars)
           epsilon = 1e-3
           for (j in 1:numpars) {
               d1 <- d2 <- thetavec
               d1[j] = d1[j] - epsilon
               d2[j] = d2[j] + epsilon
               E1 <- E(nn, d1, X, Y, lambda)
               E2 <- E(nn, d2, X, Y, lambda)
               ng[j] = (E2-E1) / 2 / epsilon
           }
           return (ng)
       }
   }
)
```

Regression with ANN on simulated 1-dim data




```
# true unknown function underlying simulated data
f <- function(x){
  return (sin(5*x) + 3 * x - x^2)
}
f <- Vectorize(f)

m = 100
x <- seq(0,3,length.out=m)
y <- f(x) + rnorm(m, mean=0, sd = 0.1)

plot(x, y)
curve(f, 0, 3, add=TRUE, col="red")

X <- matrix(x, ncol=1)
Y <- matrix(y, ncol=1)
```

Regression with ANN on simulated 1-dim data



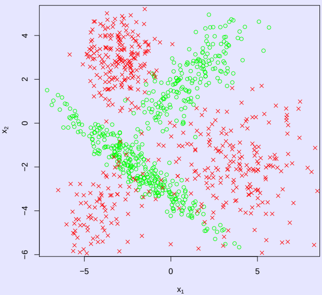

```r
source("NN.r")
ann = new("ANN", L=3,
          sizes = c(1,2,2,1),
          regression = TRUE)

# compare analytic and numeric gradient
ann <- rinit(ann)
thetavec <- unroll(ann)
ag <- gradient(ann, X=X, Y=Y, lambda=1, thetavec)
ng <- gradient(ann, X=X, Y=Y, lambda=1, thetavec,
               numeric = TRUE)
ng - ag

ann = new("ANN", L=3,
          sizes = c(1,8,8,1),
          regression = TRUE)
thetavec <- train(ann, X,Y,lambda=0)
ann@Theta <- roll(ann, thetavec)

a = seq(0,3,length.out=100)
b = lapply(a, predict, nn=ann)
lines(a,b, col="blue")
```

```
> ann
An object of class "ANN"
Slot "L":
[1] 3

Slot "sizes":
[1] 1 2 2 1

Slot "Theta":
[[1]]
          [,1]        [,2]
[1,] 0.4629961 -0.31457947
[2,] 0.7209802 -0.01415564

[[2]]
           [,1]      [,2]       [,3]
[1,]  0.7077066 0.4530151  0.6910659
[2,] -0.2254940 0.3682434 -0.9343973

[[3]]
          [,1]       [,2]     [,3]
[1,] 0.4569235 -0.0467557 0.163784


Slot "regression":
[1] TRUE

> ng - ag
 [1]  2.286626e-10 -4.648023e-10  1.192649e-09 -2.713589e-09
 [6]  2.602437e-09  6.606340e-12  3.365808e-10  1.421071e-11
[11] -5.282441e-13 -5.129230e-14  1.346145e-13

function gradient
   34057     9894
optimization successful
Target error E after optimization:  0.0032765
```

Classification with ANN on simulated 2-dim data

Sample from a mixture of normal distributions

```r
# same distr. as in the logistic regression example
set.seed(3) # to get reproducible resuls
library(MASS)
# simulate two 2-dim data sets from
# mixtures of normal distributions
pos <- rbind(mvrnorm(n = 200, c(1,2),
                matrix(c(3,2,2,2),2)),
             mvrnorm(n = 300, c(-2,-2),
                matrix(c(5,-3,-3,2),2)))
neg <- rbind(mvrnorm(n = 200, c(4,-2),
                matrix(c(4,0,0,3),2)),
             mvrnorm(n = 200, c(-3,3),
                matrix(c(1,0,0,1),2)),
             mvrnorm(n = 100, c(-4,-4),
                matrix(c(2,1,1,2),2)))
plot(pos,col="green",xlim=c(-7,8),xlab="x_1",ylab="x_2")
points(neg,col="red",pch=4)
```



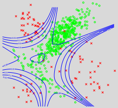
Unknown true distribution

$C \in \{1, 2\}$, $K = 2$

$\mathbf{X} = (X_1, X_2)$ has conditional density

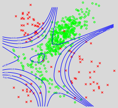
$$f_{\mathbf{X}|C=2}(\mathbf{x}) = \frac{2}{5}\varphi(\mathbf{x}; \begin{pmatrix}1\\2\end{pmatrix}, \begin{pmatrix}3 & 2\\2 & 2\end{pmatrix}) + \frac{3}{5}\varphi(\mathbf{x}; \begin{pmatrix}-2\\-2\end{pmatrix}, \begin{pmatrix}5 & -3\\-3 & 2\end{pmatrix})$$

$$f_{\mathbf{X}|C=1}(\mathbf{x}) = \frac{2}{5}\varphi(\mathbf{x}; \begin{pmatrix}4\\-2\end{pmatrix}, \begin{pmatrix}4 & 0\\0 & 3\end{pmatrix}) + \frac{2}{5}\varphi(\mathbf{x}; \begin{pmatrix}-3\\-3\end{pmatrix}, \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}) + \frac{1}{5}\varphi(\mathbf{x}; \begin{pmatrix}-4\\-4\end{pmatrix}, \begin{pmatrix}2 & 1\\1 & 2\end{pmatrix})$$

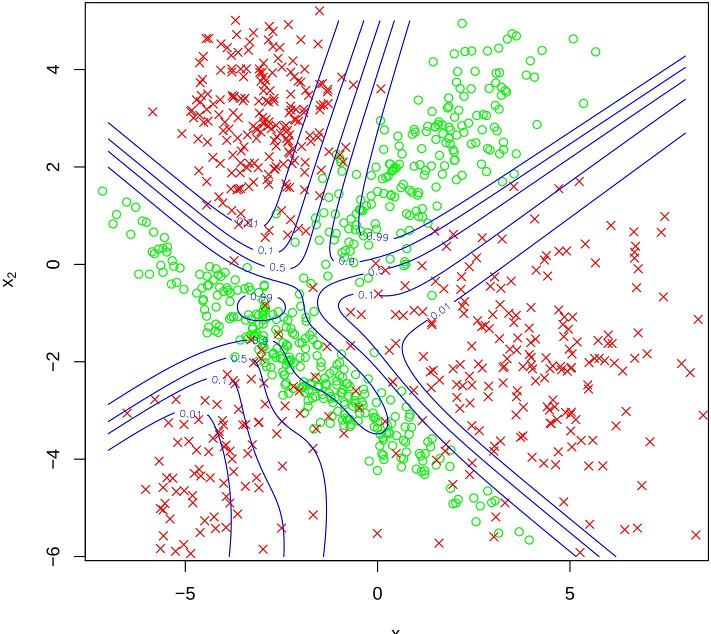


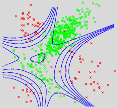
Classification with ANN on simulated 2-dim data

```r
X <- rbind(pos,neg)
mpos <- dim(pos)[1] # number of positive examples
mneg <- dim(neg)[1] # number of negative examples
m <- mpos + mneg # sample size
Y <- matrix(c(rep(c(0,1),mpos),
              rep(c(1,0),mneg)),
            ncol=2, byrow=TRUE)

source("NN.r")

ann = new("ANN", L=3,
          sizes = c(2,20,20,2),
          regression = FALSE)

thetavec <- train(ann, X,Y,lambda=.1)
ann@Theta <- roll(ann, thetavec)

# make a contour plot of the function specified by the ANN
h <- function (v)
  return (predict(ann, v)[2])

a <- seq(-7,8,by=0.05);
b <- seq(-6,5,by=0.05);
G <- matrix(nrow=length(a), ncol=length(b))
for (i in 1:nrow(G))
  for (j in 1:ncol(G))
    G[i, j] <- h(c(a[i],b[j]));

contour(a,b,G,add=1,col="blue", levels=c(0.01,0.1,0.5,0.9,0.99))
```

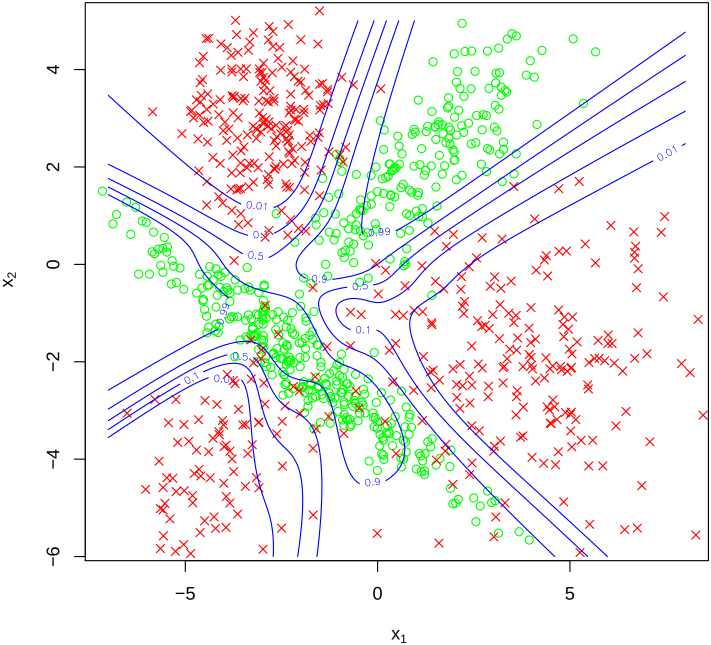


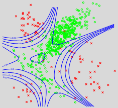
1 hidden layer of size 5, $\lambda = 0$ (no regularization)



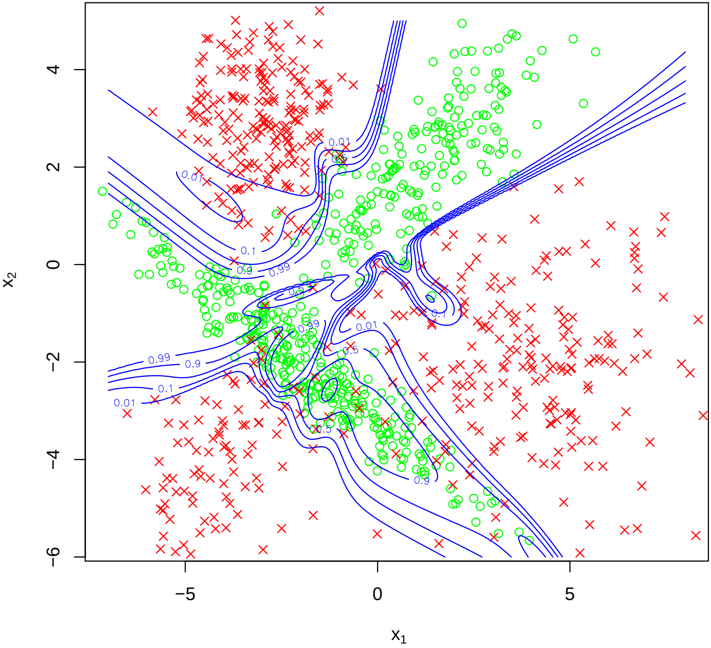


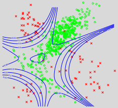
2 hidden layers of size 8 each, $\lambda = 0$ (no regularization)



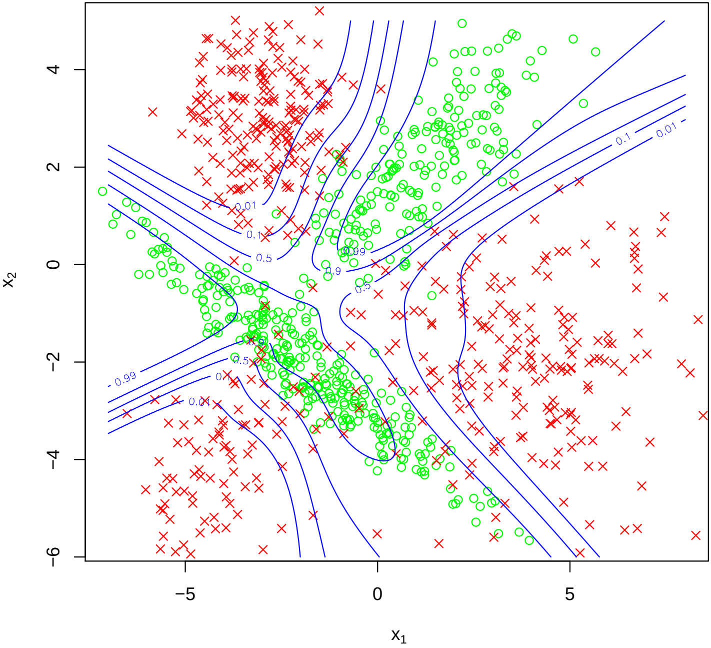


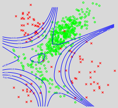
2 hidden layers of size 20 each, $\lambda = 0$ (no regularization)



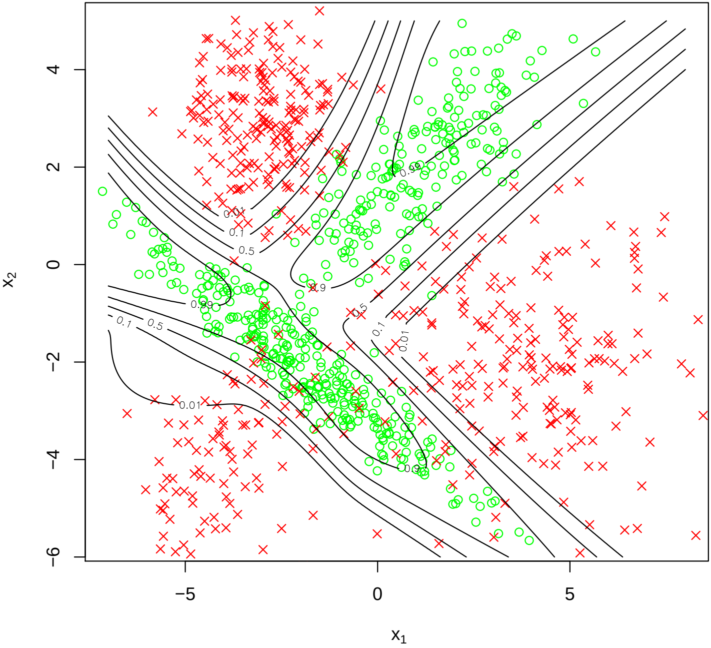


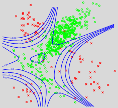
2 hidden layers of size 20 each, $\lambda = 0.1$ (with regularization)

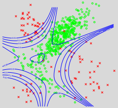




Posterior probability of class (theoretical optimum if distr. was known)







An ANN for the Classification of Images of Handwritten Digits

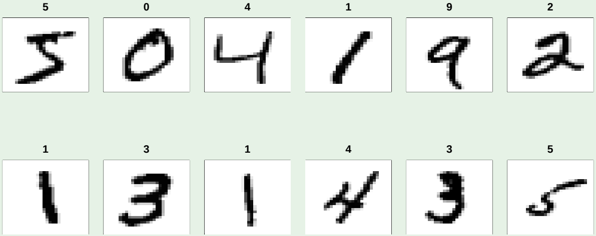
Input

Images from the MNIST benchmark database at

`http://yann.lecun.com/exdb/mnist.`

- each image is a vector $\mathbf{x} = (x_1, \ldots, x_n)^T$
- the images are centered and sized to $n = 28 \times 28 = 784$ pixels
- each pixel $x_i \in \{0, \ldots, 255\}$ is a 1-byte gray-value

Some labeled training images



Training an ANN for Handwritten Digits

```
# open training images and their labels for reading in binary mode (rb)
trainset = file("train-images-idx3-ubyte", "rb")
trainlabels = file("train-labels-idx1-ubyte", "rb")
# read away meta data bytes until the training data starts
readBin(trainset, integer(), n=4)
readBin(trainlabels, integer(), n=2)


m <- 3000 # training set size
n <- 28*28 # number of features


X <- matrix(readBin(trainset, integer(), size=1, signed=0, n=m*n),
            ncol=n, byrow=TRUE)
X <- (X - 127.5) / 127.5 # normalize input to [-1,1]


c <- readBin(trainlabels, integer(), size=1, signed=0, n=m)
Y <- matrix(0, ncol=10, nrow=m)
for (i in 1:m)
   Y[i, 1+c[i]] = 1


source("NN.r")
ann = new("ANN", L=3, sizes = c(n, 50, 20, 10), regression = FALSE)
thetavec <- train(ann, X, Y, lambda=0.5)
ann@Theta <- roll(ann, thetavec)
```

Testing the ANN

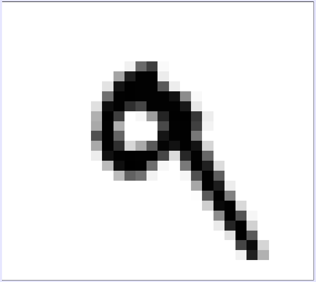
```
# open test images and their labels for reading in binary mode (rb)
testset = file("t10k-images-idx3-ubyte", "rb")
testlabels = file("t10k-labels-idx1-ubyte", "rb")
# read away meta data bytes until the training data starts
readBin(testset, integer(), n=4)
readBin(testlabels, integer(), n=2)

r <- 10000 # test set size
c <- numeric(r) # actual classes
t <- numeric(r) # predicted classes

for(i in 1:r){
  # read i-th test image and its true label
  x <- (readBin(testset, integer(), size=1, signed=0, n=n) - 127.5) / 127.5
  c[i] <- readBin(testlabels, integer(), size=1, signed=0, n=1)

  y <- predict(ann, x) # probabilities for each of the 10 classes
  t[i] <- which.max(y) - 1 # most likely class
}

print ("test␣error␣rate:")
sum(t!=c) / r

save(ann, file="ann.MS.RData")
```

Output:

```
test error rate:
0.139
```

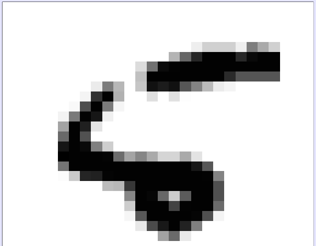
Examples: A Correct and a False Prediction

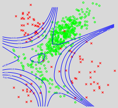
is a 9 pred 9, 5



probs= 0.00,0.00,0.00,0.00,0.00,0.01,0.00,0.00,0.00,0.98

is a 5 pred 6, 2



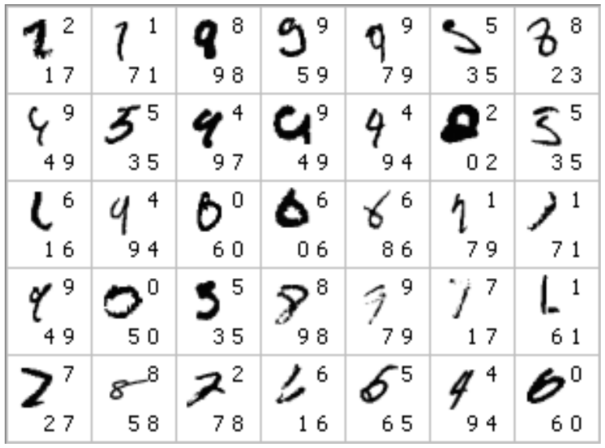


A better ANN from the literature

Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition
Ciresan, Meier, Gambardella, Schmidhuber, *arXiv*, 2010

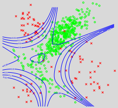
- $m = 60\,000$ training images
- additionally training images are deformed (rotation, scaling, shearing)
- layer sizes 2500, 2000, 1500, 1000, 500, 10
- 114.5 hours on 1 GeForce GTX 280 + 1 CPU
- test error rate is $35/10000 = 0.35\%$

35 images misclassified by ANN of Ciresan et al.

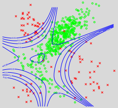


Ciresan, Meier, Gambardella, Schmidhuber, "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition", arXiv, 2010

top right: true label
bottom: two most likely labels (predictions)



1 Artificial Neural Networks

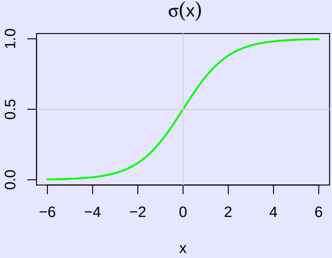


Activation Functions

Logistic sigmoid

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

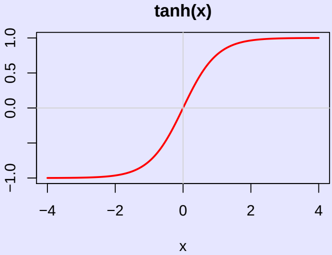
$$\sigma' = \sigma(1 - \sigma)$$

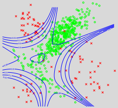


Tangens hyperbolicus

$$\begin{aligned} \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ &= 2\sigma(2x) - 1 \end{aligned}$$

$$\tanh'(x) = 1 - (\tanh(x))^2$$



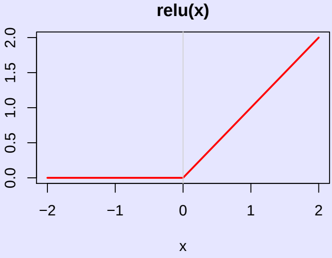


Rectified linear unit (ReLU)

$$\text{relu}(x) = x^+ = \max\{0, x\}$$

$$\text{relu}'(x) = \begin{cases} 1 & \text{, if } x > 0 \\ 0 & \text{, if } x < 0 \end{cases}$$

$\text{relu}'(0) := 1$ (arbitrary)



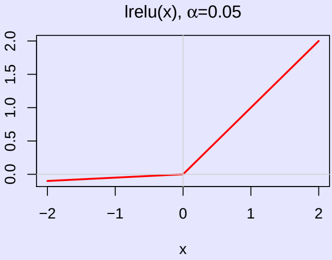
Leaky rectified linear unit (Leaky ReLU)

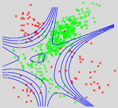
$$\text{lrelu}(x) = \max\{\alpha x, x\}$$

$$\text{lrelu}'(x) = \begin{cases} 1 & \text{, if } x > 0 \\ \alpha & \text{, if } x < 0 \end{cases}$$

$\text{lrelu}'(0) := 1$ (arbitrary)

$$\alpha = 0.01$$



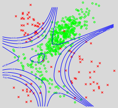


Comparison of Activation Functions

Sigmoid versus tanh

- are equivalent up to linear transformations of net input and output.
- Recall that
$$\frac{\partial D}{\partial \theta_{ji}^{(\ell-1)}} = \delta_j^{(\ell)} a_i^{(\ell-1)}.$$
If all activations $a_i^{(\ell-1)}$ are positive (log. sigmoid), then of any row j of the matrix
$$\left(\frac{\partial D}{\partial \theta_{ji}^{(\ell-1)}} \right)$$
all entries have the same sign – that of $\delta_j^{(\ell)}$. Increasing some values in a row of $\Theta^{(\ell-1)}$ while increasing others is therefore only possible though multiple iterations and can be inefficient.
- Better to have input layer and hidden layers averaging close to 0.
- tanh therefore preferrable to logistic sigmoid.

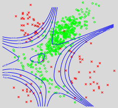
LeCun, Bottou, Orr and Müller: Efficient BackProp, 1998



Comparison of Activation Functions

Vanishing gradients

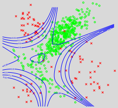
- Recal that for the logistic sigmoid activation function we had the update step
$$\boldsymbol{\delta} \leftarrow (\Theta^{(\ell)})^T \cdot \boldsymbol{\delta} \circ \mathbf{a}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}^{(\ell)})$$
during BackProp.
- $\mathbf{a}^{(\ell)} \circ (\mathbf{1} - \mathbf{a}^{(\ell)})$ is a vector with entries in $[0, 0.25]$.
- In deep networks (with may layers) entries of δ and therefore partial derivatives to weights can therefore numerically vanish. These parameters are then not changed at all and the optimization can get stuck far from an optimum.
- $\tanh'(x) \in (0, 1)$ and multiplying partial derivatives for many layers may also result in vanishing gradients (albeit less quicky).



Comparison of Activation Functions

ReLU

- introduced in 2010 (Nair and Hinton, “Rectified linear units improve restricted Boltzmann machines”, *Proc. ICML*, 2010)
- The derivative is either 0 or 1.
- If $a_j^{(\ell)} = 0$ for a training example, then no computation is necessary to obtain $\delta_j^{(\ell)}$ – it vanishes as well. Sparsity makes computation efficient.
- Averaging over all training examples, $D^{(\ell)}$ may still be non-sparse.



Comparison of Activation Functions

Leaky ReLU vs ReLU

- If an particular activation $a_j^{(\ell)}$ vanishes on all training examples, the neuron has no influence on the outcomes anymore ("dying ReLU problem"). 'Resurrection' may not happen, as the derivative to the most important weights influencing the value $a_j^{(\ell)}$ are vanishing as well.
- Leaky ReLU overcomes the dying ReLU problem
- at the cost of loosing efficiency from sparsity.