Optimization of Filter Banks

Goal: Obtain a filter bank from our structure or product $\boldsymbol{P}_a(z)=\boldsymbol{F}_a\cdot\boldsymbol{D}(z)\cdot\boldsymbol{G}(z)....$, which has "good" subband filters, i.e. a good or sufficient stopband attenuation and not much pass band attenuation. An example could be a desired stopband attenuation of -60dB and a pass band attenuation of less than -3dB.

Problem to solve: The coefficients which determine the frequency responses are the coefficients in our cascade or product, the coefficients in our matrices e_j^i , h , g_j^i ... They result in the frequency response in a non-linear way, hence the usual filter design approaches don't work here, previous approaches like the remez-exchange algorithm. Those traditional filter design algorithms can also be seen as optimization algorithms for more or less linear problems (or quadratic if we take the mean squared error).

We can read out the resulting analysis baseband prototype impulse response $h_a(n)$ for instance by computing the sparse folding matrix the product

$$\boldsymbol{F}_a(z)=\boldsymbol{F}_a\cdot\boldsymbol{D}(z)\prod_{n=0}^{P-1}\boldsymbol{G}_n(z)\prod_{m=0}^{Q-1}\boldsymbol{H}_m(z)$$

(without the transform matrix $\boldsymbol{T}$), and then compare it with eq. (1) of slides 15,

$$\boldsymbol{F}_a(z)=$$

$$=\begin{bmatrix} . & 0 & -H_{2N-1}^{\downarrow 2N}(-z^2)\cdot z^{-1} & -H_{N-1}^{\downarrow 2N}(-z^2) & 0 & . \\ . & . & 0 & 0 & . & . \\ -H_{1.5N}^{\downarrow 2N}(-z^2)\cdot z^{-1} & 0 & . & . & 0 & -H_{N/2}^{\downarrow 2N}(-z^2) \\ -H_{1.5N-1}^{\downarrow 2N}(-z^2)\cdot z^{-1} & 0 & . & . & 0 & H_{N/2-1}^{\downarrow 2N}(-z^2) \\ 0 & . & 0 & . & . & 0 \\ . & 0 & -H_N^{\downarrow 2N}(-z^2)\cdot z^{-1} & H_0^{\downarrow 2N}(-z^2) & . & . \end{bmatrix}$$

$$H_n^{\downarrow 2N}(z):=\sum_{m=0}^{\infty}h_a(m2N+n)\cdot z^{-m}$$

In this way we obtain $h_a(n)$, and hence its frequency response, from our matrix coefficients $\boldsymbol{x}=[h_1, h_2, \ldots, e_j^i, \ldots, h_j^i, \ldots]$. We can do the same for the synthesis side, or we restrict the (sub-) matrices to have det(.)=-1, in which case the synthesis baseband impulse reponse is the same as for the analysis.

Another possibility to obtain the analysis baseband impulse response is to compute the polyphase matrix $\boldsymbol{P}_a(z)$. Its first column contains the **polyphase elements of the first subband filter in reverse order**. This can be

used to read out its impulse response $h_0(n)$. To obtain $h_a(n)$ we simply **divide** $h_0(n)$ by the **modulation function** for the first subband (for k=0).

Our approach:
Since we now have this non-linear dependency of our baseband impulse response from the matrix coefficients, we need to use “more powerful” optimization algorithms, which are made for just mostly convex error functions.
We first define an **error function**, and then we use an **optimization** algorithm to minimize this error function. This error function can be the sum of the magnitudes, or the squares, of the **differences between our obtained frequency response**, given our unknown variables at some point, **and our desired frequency response**. To give the stop band attenuation and the pass band ripples different weights, we can also **assign weights to the different frequency regions** for our error function.

So our starting point is a vector of all of our unknowns of our matrices,

$$\boldsymbol{x}=[h_{1,}h_2,\ldots,e^i_j,\ldots,h^i_{j\ldots}]=[x_{1,}x_{2\ldots}]$$

We now define a function which computes the

baseband impulse response out of these unknowns, by multiplying our matrices to obtain our final folding matrix or the polyphase matrix, and then read out the baseband impulse response.

This then yields the baseband prototype for our coefficient set x , which we call $h_x(n)$. This is now used to compute a **weighted frequency response, at k frequency points**, with weights w_i for each frequency point,

$$H_i = \sum_{n=0}^{L \cdot N - 1} h_x(n) e^{\omega_i \cdot n} \cdot w_i, for\, i = 0, \ldots, k-1.$$

(see also Schuller, Smith: “New Framework for Modulated Perfect Reconstruction Filter Banks”, IEEE Transactions on Signal Processing, August 1996)

We choose as many frequency points as necessary to sufficiently cover our frequency response. Usually it should be **several times the length of our filter**, to avoid sampling the frequency response accidentally near its zeros. The more important frequency points in terms of attenuation get the higher weights.

The same can also be done for the synthesis, but remember that this is only necessary of we don't have det(..)=-1, because for det(..)=-1, we obtain identical prototype impulse responses for

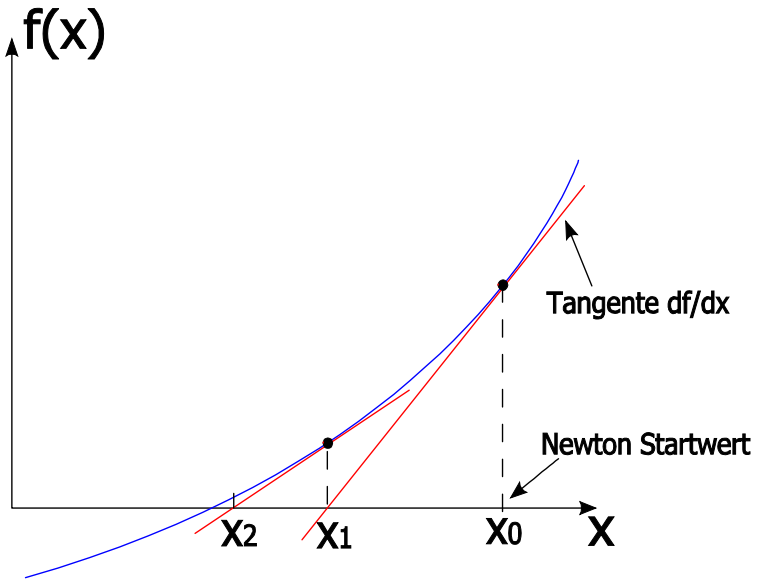
analysis and synthesis, and we only need to optimize the analysis (Lecture 13 and 14).
We also need to have a vector containing the weighted desired frequency response, which would be 1 for the pass band (multiplied by the corresponding weights), and zeros in the stop band. We call this $\boldsymbol{d}$ (for desired).
Our error function of our vector of unknowns $\boldsymbol{x}$ is now

$$f(\boldsymbol{x}) = \sum_{i=1}^{2k} |H_i(\boldsymbol{x}) - d_i|^2 = (\boldsymbol{H} - \boldsymbol{d}) \cdot \overline{(\boldsymbol{H} - \boldsymbol{d})^T}$$

Now we just need to minimize this function $f(\boldsymbol{x})$ with respect to our vector of unknows $\boldsymbol{x}$.

This leads us to optimization in general. The goal of optimization is to find the vector $\boldsymbol{x}$ which minimizes the error function $f(\boldsymbol{x})$.
We know: in a minimum, the functions derivative is zero, $f'(\boldsymbol{x}) := \frac{df(\boldsymbol{x})}{d\boldsymbol{x}} = \boldsymbol{0}$.

An approach to iteratively find the zero of a function is **Newtons method**. Take some function f(x), where x is not a vector but just a number, then we can find its minimum as depicted in the following picture,



with the iteration

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \qquad k = 0, 1, 2, \ldots$$

Now we want to find the zero not of $f(x)$, but of $f'(x)$, hence we simply replace $f(x)$ by $f'(x)$ and obtain the following iteration,

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

To obtain a minimum (not a maximum) we need the condition $f''(x_k) > 0$.

Example for finding a zero of a function:
Compute $\sqrt{2}$ numerically by finding the zero of

the function

$$f(x)=x^2-2$$

We know the solution is $\sqrt{2}$. Now apply Newtons Method to find this solution numerically. The first derivative is

$$f'(x)=2x$$

The Newton iteration is

$$x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)}=x_k-\frac{x_k^2-2}{2x_k}$$

In Matlab/Octave and with a starting value of 1 this becomes:

```
x=1
x=x-(x^2-2)/(2*x)
%x =  1.5000
x=x-(x^2-2)/(2*x)
%x =  1.4167
x=x-(x^2-2)/(2*x)
%x =  1.4142
```

We see: after only 3 iterations we obtain $\sqrt{2}$ with 5 digits accuracy!

Example: find the minimum of $\cos(x)$. We know a minimum is at $x=\pi$, so this is also a way to numerically determine π .

We have

$$f(x)=\cos(x)$$

$$f'(x)=-\sin(x)$$

$$f''(x) = -\cos(x)$$

Newton update:

$$x_{new} = x_{old} - f'(x_{old}) / f''(x_{old}) = x_{old} - \sin(x_{old}) / \cos(x_{old})$$

In Matlab/Octave we simply repeatedly use the expression:

```
x=x-sin(x)/cos(x)
```

We start with x=3 and proceed with the Newton update,

```
x=3;
x=x-sin(x)/cos(x)
%x =  3.1425
x=x-sin(x)/cos(x)
%x =  3.1416
```

We see that this iteration indeed converges to π ! It computes π with an accuracy of 5 digits after only 2 iterations!

For a **multi-dimensional** function, where the argument $\boldsymbol{x}$ is a vector, the first derivative is a vector called “Gradient”, with symbol Nabla ∇ , because we need the derivative with respect to each element of the argument vector $\boldsymbol{x}$,

$$\nabla f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

(where n is the number of unknowns in the argiment vector $\boldsymbol{x}$). For the second derivative, we need to take each element of the gradient vector and again take the derivative to each element of the argument vector. Hence we obtain a matrix, the **Hesse Matrix**, as matrix of second derivatives,

$$H_f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

Observe that this Hesse Matrix is symmetric. Using these definitions we can generalize our Newton algorithm to the multi-dimensional case. The one-dimensional iteration

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

turns into the multi-dimensional iteration

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - H_f^{-1}(\boldsymbol{x_k}) \nabla f(\boldsymbol{x_k})$$

For a minimum, $H_f(\boldsymbol{x})$ must be positive definite (all eigenvalues are positive).
The problem here is that for the Hesse matrix we need to compute n^2 second derivatives, which can be computationally too complex, and then we need to invert this matrix. Hence we make the simplifying assumption, that the Hesse matrix can be written as a **diagonal matrix with constant values on the diagonal**.
Observe that this is mostly is mostly a very crude approximation, but since we have an iteration with many small updates it can still work,

$$\boldsymbol{H_f}(\boldsymbol{x_k}) = \frac{1}{\alpha} \cdot \boldsymbol{I}$$

The best value of α depends on how good it approximates the Hesse matrix.
Hence our iteration $\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - \boldsymbol{H_f^{-1}}(\boldsymbol{x_k}) \cdot \nabla f(\boldsymbol{x_k})$ turns into

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - \alpha \cdot \nabla f(\boldsymbol{x_k}) \quad (1)$$

which is much simpler to compute. This is also called “Steepest Descent”, because the gradient tell us the direction of the steepest descent, or “Gradient Descent”.

We see that the update of x consists only of the gradient $\nabla f(x_k)$ scaled by the factor α . In each step, we reduce the value of $f(x)$ by moving x in the direction of the gradient. If we make α larger, we obtain larger update steps and hence quicker convergence to the minimum, but it may oscillate around the minimum. For smaller α the steps become smaller, but it will converge more precisely to the minimum.
Because of the update direction along the gradient, this method is also called "**Gradient Descent**".

Example:
Find the 2-dimensional minimum of the function
$f(x_{1,}x_2)=\cos(x_1)-\sin(x_2)$
Its gradient is
$\nabla f(x_{1,}x_2)=[-\sin(x_1),-\cos(x_2)]$

In Matlab/Octave we choose $\alpha=1$ and a starting point of $[x_{1,}x_2]=[2,2]$,

```
alpha=1;
%start:
x=[2,2]
x= x -alpha*[-sin(x(1)), -cos(x(2))]
%x =
%   2.9093   1.5839
```

```
x= x -alpha*[-sin(x(1)), -cos(x(2))]
%x =
%   3.1395   1.5708
x= x -alpha*[-sin(x(1)), -cos(x(2))]
%x =
%   3.1416   1.5708
```

We see: the minimum is obtained for $x_1 = 3.1416$ (exact value: π) and $x_2 = 1.5708$ (exact value: $\pi/2$).
Observe that we needed **3 iterations** to obtain 5 digit accuracy in this 2-dimensional case. In the 1-dimensional case with the **Newton** iteration we needed only **2 iterations** for the same accuracy.

Neural Networks

So-called “**Convolutional Neural Networks**” are like cascaded filter banks, but with a non-linear function at the output of each filter and a constant offset, see for instance http://deeplearning.net/tutorial/lenet.html.
Its coefficients are called “weights”.
The so-called “**(Artificial) Neural Networks**” simply use a weighted sum instead of the convolution, see https://en.wikipedia.org/wiki/Artificial_neural_network
and
“An Introduction to Neural Computing”, I. Aleksander, H. Morton, Chapman and Hall, 1990.

A nice application example for inage understanding can be seen in:
http://cs.stanford.edu/people/karpathy/deepimagesent/

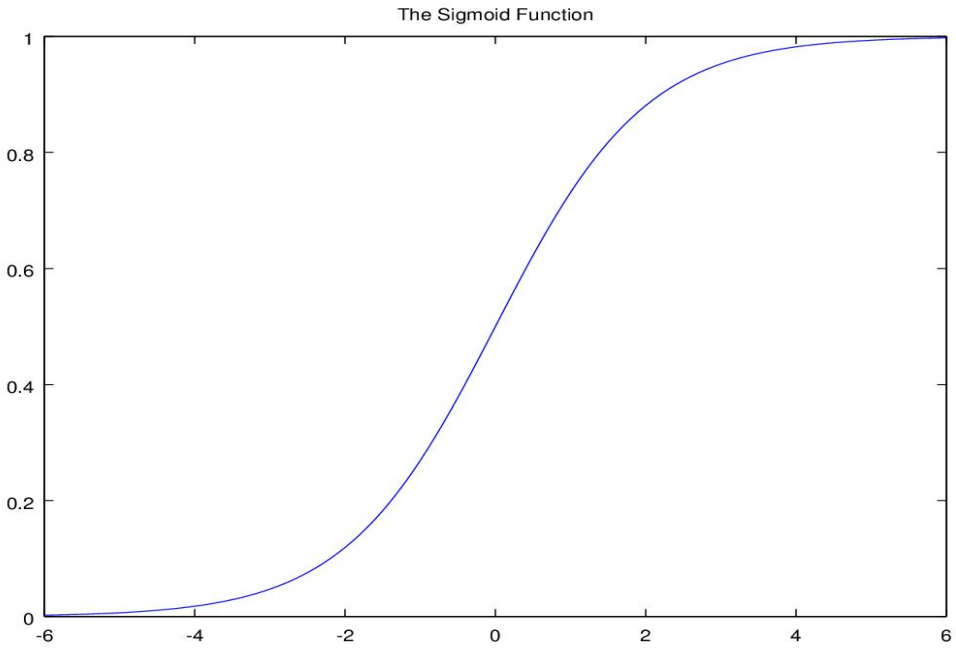
Both types use several “layers” of cascading. If there are more than 3 layers, they are called “**Deep Neural Networks**”, with “**Deep Learning**”. These are current active research areas, for instance for speech recognition and image recognition.

The non-linear function $f(x)$ is often the so-called **sigmoid function**, see also https://en.wikipedia.org/wiki/Sigmoid_function which is defined as

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

We can plot it with Matlab/Octave:

```
x=[-6:0.1:6];
y=1./(1+exp(-x));
plot(x,y)
title('The Sigmoid Function')
```

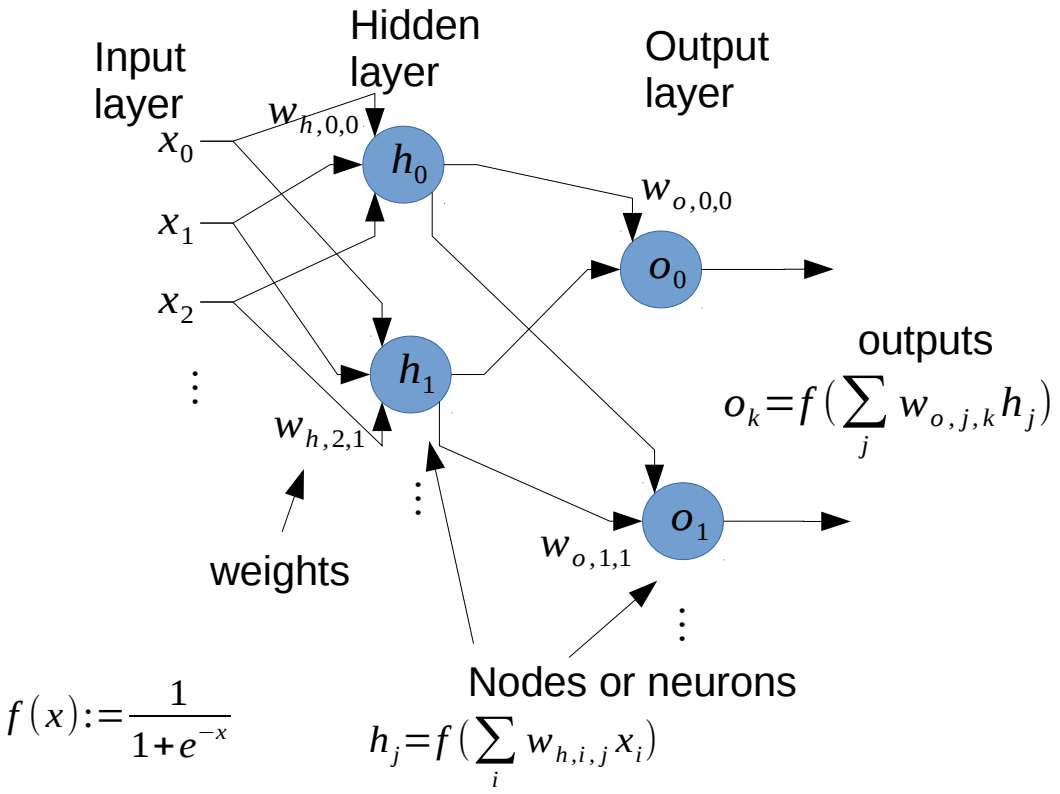


We see that it behaves like a “soft-limiter” function.

Its derivative is

$$f'(x)=\frac{d}{dx}f(x)=\frac{e^x}{(1+e^x)^2}$$

The following diagram shows a 3-layer artificial neural network,



In general we have several outputs in our neural network, **output k** is the non-linear function of a sum $s_{o,k}$,

$$o_k = f\left(s_{o,k}\right)$$

The sum is the weighted sum from the hidden layer outputs h_j . We choose $x_0 = h_0 = 1$, such that we also obtain a constant offset as part of the sum,

$$s_{o,k} = \sum_j w_{o,j,k} h_j$$

The output of the hidden layer h_j is again a non-linear function f of a sum

$$h_j = f\left(s_{h,j}\right)$$

The hidden layer sum is a weighted sum of input values x_i :

$$s_{h,j} := \sum_i w_{h,i,j} x_i$$

The output of our neural network depends on the weights $\boldsymbol{w}$ and the inputs $\boldsymbol{x}$ (we assume a fixed sigmoid function). We assemble the inputs in
the vector x which contains all the inputs,

$$\boldsymbol{x} = \left[x_{0,} x_{1,\ldots}\right]$$

and vector w which contains all the weights (from the hidden and the output layer),

$$\boldsymbol{w}=\left[w_{h,0,0},w_{h,0,1},\ldots,w_{o,0,0},w_{o,0,1}\ldots\right]$$

To express this dependency, we can rewrite the **output k** as

$$o_k\left(\boldsymbol{x},\boldsymbol{w}\right)$$

Now we would like to "train" the network, meaning we would like to determine the weights such that if we present the neural network with a training pattern in x, the output produces a desired value. For instance, if we present an image with an object in it, the output indicates that the object was there with a desired output value. Hence we have **training inputs**, and **desired outputs** d_k .

We now use **optimization** to update the weights $\boldsymbol{w}$ to obtain outputs o_k as close as possible to the desired outputs d_k with a given input $\boldsymbol{x}$.

We use **Gradient Descent** for this optimization. We start with an error function delta,

$$\delta_k\left(\boldsymbol{x},\boldsymbol{w}\right)=o_k\left(\boldsymbol{x},\boldsymbol{w}\right)-d_k$$

An always positive Error function is its square,

$$Err_k\left(\boldsymbol{x},\boldsymbol{w}\right)=\frac{1}{2}\left(\delta_k\left(\boldsymbol{x},\boldsymbol{w}\right)\right)^2$$

To this error function we can now **apply Gradient Descent** (see above eq. (1)),

$$\boldsymbol{x}_{new}=\boldsymbol{x}_{old}-\alpha\cdot\nabla f\left(\boldsymbol{x}_{old}\right)$$

in our case this becomes

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} - \alpha \cdot \nabla Err_k(\boldsymbol{x}, \boldsymbol{w}_{old}) \quad (2)$$

For the gradient ∇ we need to compute the derivative to each weight w_i of the weight vector $\boldsymbol{w}$. In our case we apply the chain rule (outer derivative times inner derivative),

$$\frac{d}{dw_i} Err_k(\boldsymbol{x}, \boldsymbol{w}) = \frac{d}{dw_i} \frac{1}{2} \cdot \delta_k^2(\boldsymbol{x}, \boldsymbol{w}) =$$

$$= \delta_k(\boldsymbol{x}, \boldsymbol{w}) \cdot \underbrace{\frac{d}{dw_i} o_k(\boldsymbol{x}, \boldsymbol{w})}_{inner\ derivative} \quad (3)$$

We compute this inner derivative first for the **output weights**, again with the chain rule,

$$\frac{d}{dw_{o,j,k}} o_k(\boldsymbol{x}, \boldsymbol{w}) =$$

$$= \frac{d}{dw_{o,j,k}} f(s_{o,k}) = f'(s_{o,k}) \cdot \frac{d}{dw_{o,j,k}} s_{o,k}$$

$$= f'(s_{o,k}) \frac{d}{dw_{o,j,k}} \sum_{j'} w_{o,j',k} h_j$$

finally we get

$$\frac{d}{dw_{o,j,k}} o_k(\boldsymbol{x}, \boldsymbol{w}) = f'(s_{o,k}) \cdot h_j \quad (4)$$

We now plug this result (4) into eq. (3) and eq. (2), and obtain the update rule for the Gradient Descent for the output weights,

$$w_{o,j,new} = w_{o,j,old} - \alpha \cdot \delta_k(\boldsymbol{x}, \boldsymbol{w}) f'(s_{o,k}) \cdot h_j$$

which says: **update** = **alpha** times **output delta** times **output derivative** times its **input** h_j from the hidden nodes.

For the hidden weights the inner derivative of eq. (3) becomes a little more complicated,

$$\frac{d}{dw_{h,i,j}} o_k(\boldsymbol{x}, \boldsymbol{w}) =$$

$$= \frac{d}{dw_{h,i,j}} f(s_{o,k}) = f'(s_{o,k}) \cdot \frac{d}{dw_{h,i,j}} s_{o,k}$$

$$= f'(s_{o,k}) \frac{d}{dw_{h,i,j}} \sum_j w_{o,j,k} h_j$$

$$= f'(s_{o,k}) \cdot w_{o,j,k} \frac{d}{dw_{h,i,j}} h_j$$

$$= f'(s_{o,k}) \cdot w_{o,j,k} \frac{d}{dw_{h,i,j}} f(s_{h,j})$$

apply the chain rule:

$$= f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \frac{d}{dw_{h,i,j}} s_{h,j}$$

$$= f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \frac{d}{dw_{h,i,j}} \sum_i w_{h,i,j} x_i$$

finally we get

$$\frac{d}{dw_{h,i,j}} o_k(\boldsymbol{x},\boldsymbol{w}) = f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \cdot x_i \quad (5)$$

We now plug this result for the hidden weights (5) again into eq. (3) and eq. (2), and obtain the **update rule** for the Gradient Descent for the **hidden weights**

$$w_{h,i,j,new} = w_{h,i,j,old} - \alpha \cdot \delta_k(\boldsymbol{x},\boldsymbol{w}) f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \cdot x_i$$

We can simplify this update equation if we define a new "**back propagated delta**" term as

$$\delta_{h,k}(\boldsymbol{x},\boldsymbol{w}) := \delta_k(\boldsymbol{x},\boldsymbol{w}) \cdot f'(s_{o,k}) \cdot w_{o,j,k}$$

Hence we get

$$w_{h,i,j,new} = w_{h,i,j,old} - \alpha \cdot \delta_{h,k}(\boldsymbol{x},\boldsymbol{w}) \cdot f'(s_{h,j}) \cdot x_i \quad (6)$$

which says: **update** = **alpha** times **back propagated delta** times **derivative of hidden function** times its **input** x_i .

This algorithm is also called **Back-Propagation**.

We need to do this update for each output node k .

This is in principle the same rule as for the output nodes, just with its corresponding input and output.

This means: **Back-Propagation is** just a consequence of applying the **Gradient Descent** algorithm to Neural Networks.

Hence for all nodes we obtain a **“local” processing**. We just look at one node, call its input x_i , its output o , its weights w_i , and its desired output d . Then we obtain its output with

$$s=\sum_i w_i x_i$$

$$o=f\left(s\right)$$

if it is an **output node** the difference delta to the desired d is:

$$\delta=o-d$$

if it is a **hidden node** we use the “**back propagated**” δ from an output difference δ_o , weight to the output node w_o , and sum for the output node s_o :

$$\delta=\delta_o\cdot f'\left(s_o\right)\cdot w_o$$

Then we get the **local weight update** as

$$w_{i,new}=w_{i,old}-\alpha\cdot\delta\cdot f'\left(s\right)\cdot x_i$$

This is depicted in the following diagram,

x_0

w_0 $f(x) := \frac{1}{1+e^{-x}}$

x_1 w_1

o

$\vdots$

δ

$$w_{i,new} = w_{i,old} - \alpha \cdot \delta \cdot f'(s) \cdot x_i$$

Matlab/Octave Example

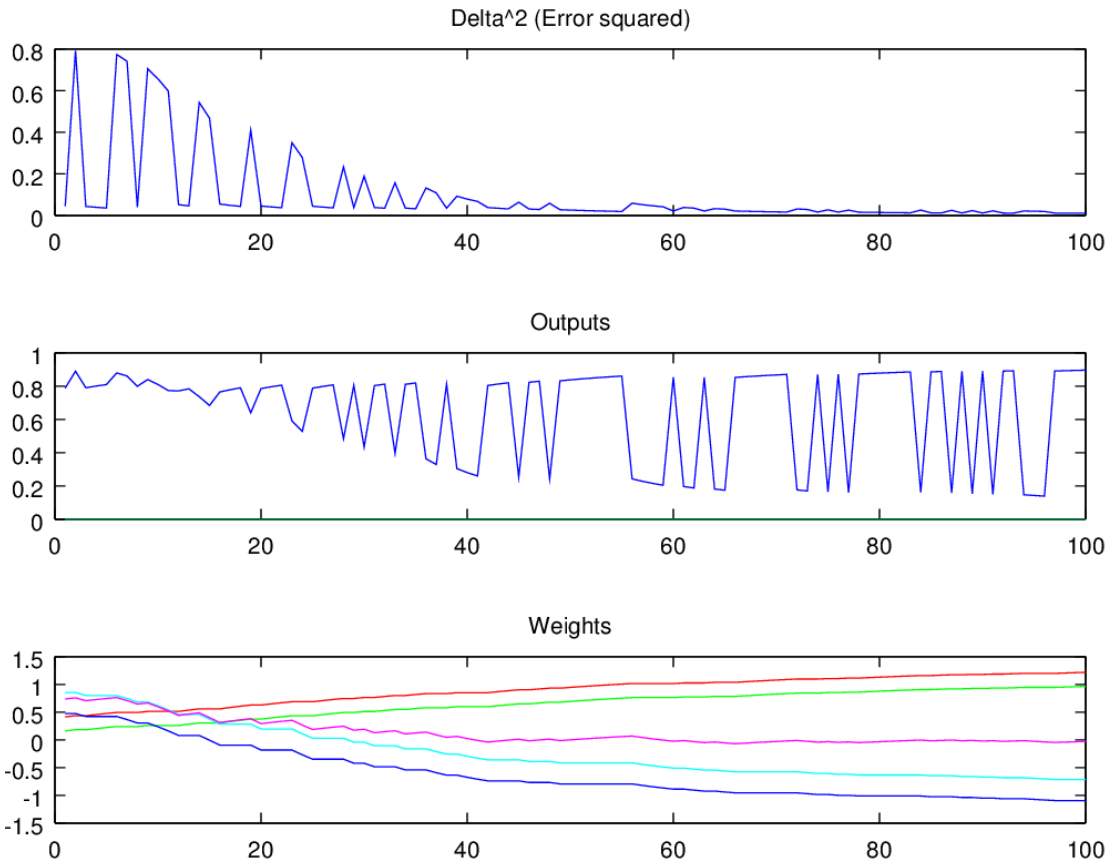
We simulate:
*1 Neuron with 4 signal inputs and 1 constant (1) input.
*We want to learn it 2 pattern:
$x=[1,1,0,0]$, where the desired output should be $d=1$, and
$x=[0,0,1,1]$, where the desired output should be $d=0$.

In the program we **append** the constant 1 to the end of the vector x , hence we get a total of **5 weights**.
The program **trains** the neuron with a **random sequence** of the to patterns to be recognized. In the end we have the weights vector w for recognizing the 2 patterns.

We start the program in Matlab/Octave with
neuron

We then see in the plots how the weights converge.
Observe that the program also implements the **Gradient descent** directly, using **computed derivatives**. This leads indeed to the same results for the weights, but to higher

computational complexity.



In the upper plot we see how the error decreases over the training sequence, the middle plot shows the output during training (depending on the random sequence), and the lower plot shows the 5 converging weights, for a length of the trainings sequence of 100.

The program code is:

```
clc
% description of the script

%% Input data
% images that are to be recognized
% x1 = [1;1;0;0];
% x2 = [0;0;1;1];
X = [1 0
     1 0
     0 1
     0 1];
X = [X;ones(1,size(X,2))];
% desired reactions for the images
% d1 = 1;
% d2 = 0;
d = [1;0];
% number of inputs
M = 5; % four variables

K = 1; % number of neurons

T = 100; % duration of the training sequence

%% Neuron model
% neural function and its derivative
f = @(x) 1./(1+exp(-x));
f_prime = @(x) (exp(x)./((1+exp(x)).^2));

S = @(x,w) (x.'*w); % neural sum (including a
constant)
o = @(x,w) f(S(x,w)); % output of a neuron
```

```
%% Training set
w_0 = rand(M,1); % initial value of the weighting
vector
alpha = 0.6; % descent speed
delta_w = 0.01;% derivative step

%Training sequence
sequence = randi(2,[1 T]);

%% Training phase
Er = @(x,w,d) ((o(x,w) - d).^2)/2;
w = w_0;
delta_history = zeros(1,T);
w_history = zeros(M,T);
output_history = zeros(T);

for ii=1:T
    x_current = X(:,sequence(ii));
    d_desired = d(sequence(ii));

    delta = o(x_current,w) - d_desired;
    delta_history(ii) = delta;
    w_history(:,ii) = w;
    output_history(ii) = o(x_current,w);

%% the last equation, updating weitghs w
    w = w - alpha*...
             delta*...
             f_prime(S(x_current,w))*...
             x_current;

%% alternative equation #2 with gradient, updating
weitghs w
%    Err = Er(x_current,w,d_desired);
%    Grad_Err = (1/(2*delta_w))*...
%        (Er(x_current,w*ones(1,M)
+delta_w*eye(M),d_desired*ones(1,M)) - ...
```

```
%         Er(x_current,w*ones(1,M)-
delta_w*eye(M),d_desired*ones(1,M)));
%    w = w - alpha*Grad_Err.';

end

%% Plot results
%plot(delta_history.^2);
%plot(output_history);

figure;
subplot(3,1,1), plot(delta_history.^2);
title('Delta^2 (Error squared)')
subplot(3,1,2), plot(output_history);
title('Outputs')
    subplot(3,1,3), plot(w_history(1,:),'r');
    hold on;
    subplot(3,1,3), plot(w_history(2,:),'g');
    hold on;
    subplot(3,1,3), plot(w_history(3,:),'b');
    hold on;
    subplot(3,1,3), plot(w_history(4,:),'c');
    hold on;
    subplot(3,1,3), plot(w_history(5,:),'m');
    hold on;
    title('Weights');
```

Python Example

Most of Neural Network development is done in Python. It has powerful libraries for it, for instance “Keras” and “Theano” or “Tensorflow”. Here is an example using the Keras and Theano libraries for a 3 layer feedforward fully connected

neural network.

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
import numpy as np

def generate_dummy_data():
    #Method to generate some artificial data in an
numpy array form in order to fit the network.
    #:return: X, Y numpy arrays used for training
    X = np.array([[0.5,1.,0], [0.2,0.7,0.3],
[0.5,0,1.], [0,0,1.]])
    Y = np.array([[1], [0], [1], [1]])
    return X, Y

def generate_model():
    #    Method to construct a fully connected
neural network using keras and theano.
    #    :return: Trainable object
    # Define the model. Can be sequential or graph
    model = Sequential()
    model.add(Dense(output_dim = 4, input_dim = 3,
init="normal"))
    model.add(Activation("sigmoid"))
    model.add(Dense(output_dim = 1, input_dim = 3,
init="normal"))
    model.add(Activation("sigmoid"))
    # Compile appropriate theano functions
    model.compile(loss='mse', optimizer='sgd')
    return model

if __name__ == '__main__':
    #   Demonstration on using the code.
    X, Y = generate_dummy_data() # Acquire Training
Dataset
    model = generate_model()     # Compile an
```

```
neural net
    model.fit(X, Y, nb_epoch=100, batch_size=4)
    model.predict(X) # Make Predictions
    model.save_weights('weights.hdf5') #save
weights to file
```

Before we can run it first we need to install the libraries,

```
(sudo apt install python-pip)
sudo pip install Theano
sudo pip install keras
sudo apt install python.h5py
```

open an editor to switch the backend (after running keras at least once):

```
gedit ~/.keras/keras.json &
```

edit the backend line to:
"backend": "theano"
(see also https://keras.io/backend/)

Then run it from a terminal with

```
python kerasexamples.py
```

Observe how the loss function decreases during the training over the epochs.