Neural networks and their application in artificial intelligence

Jan Helm
Technical University Berlin
Email: jan.helm@alumni.tu-berlin.de

Abstract
This paper presents
in Part1 the basic theory of Neural Networks, and based on the standard (global) backpropagation algorithm, it introduces the **local backpropagation algorithm**: a layer-recurrent gradient algorithm with layer-specific target-vector.
Furthermore in Part2 , it presents calculated application examples for global backpropagation networks, local backpropagation networks and evolving cross-mutated networks.

Contents

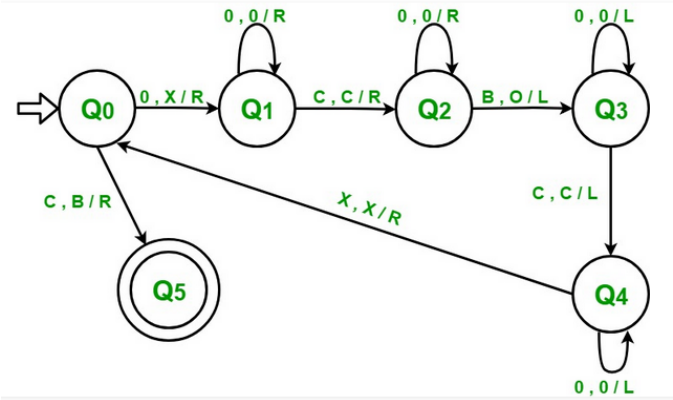
Part 1 Neural networks theory

1 Models of computation
-Functional computation
Here, operations on primitive functions are used [5].
Set of primitive functions M=

Zero-function	$\mathcal{Z}(x)=0 \quad \forall x \in \mathbb{N}$
Successor function	S(x)=x+1, e.g. S(1)=2
Projection function of a list	$U^i_n(x_1, x_2, x_3, \ldots x_i, \ldots x_n) = x_i$ e.g. $U^2_3(x_1.x_2.x_3)=x_2$
Operations	composition f12(x)=f1(f2(x)) , minimization $u(x_1.x_2.x_3,\ldots,x_n)=Min(x_1.x_2.x_3,\ldots,x_n)$

Example: addition of two numbers, m and n, by the function f(m,n).
where f(x,0) = x , (x,y+1) = S(f(x,y)
e.g. 3+2=f(3,2)=S(S(f(3,0))=5

-Turing machine
The Turing machine consists of a tape serving as the internal memory of the machine, of unlimited size, and a read/write head which moves along the tape. The Turing machine is described by the state, output and direction functions. The input on the tape is a sequence of values (0, 1, x), operation symbols (c), and blanks (B)
We can write (state, input) →(state, write, {L, R, N}) where L,R,N mean Left, Right and No movement, respectively.
The example [6], the addition 2+3 in the unary format (i.e. a number *n* is represented by *n* zeros, + is represented by c) is performed by the operation
Input 2 + 3 : 0 0 c 0 0 0
Output 5: 0 0 0 0 0
and the corresponding Turing machine is described by the diagram



step-1: Convert 0 into X and goto step-2. If symbol is “c” then convert it into blank(B), move right and goto step-6.

step-2: Keep ignoring 0’s and move towards right. Ignore “c”, move right and goto step-3.

step-3: Keep ignoring 0’s and move towards right. Convert a blank(B) into 0, move left and goto step-4.

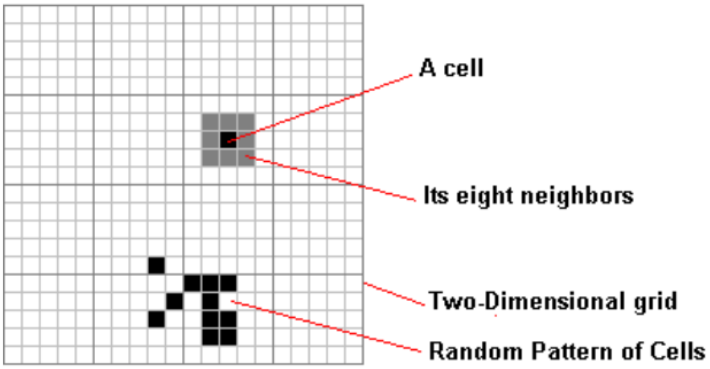
step-4: Keep ignoring 0’s and move towards left. Ignore “c”, move left and goto step-3.

step-5: Keep ignoring 0’s and move towards left. Ignore an X, move left and goto step-1.

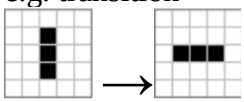
step-6: End.

-Cellular automaton

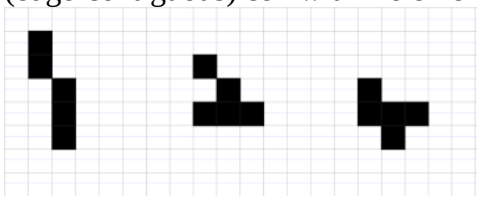
A two-dimensional cellular automaton is described by the following diagram [5]



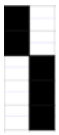
It is comprised of a two-dimensional grid of cells. Each cell can be in one of two states, on or off (dead or alive). Cells may transition from one state to the other, and become on or off, based on a set of rules.
Example: Conway's Game of Life
Rules of the Game of Life
Let N be the number of neighbors of a given cell. If N = 0 or 1, cell dies If N = 2, the cell maintains its current state (status quo)If N = 3, the cell becomes alive If N = 4,5,6,7 or 8, the cell dies.
e.g. transition



The example 2+3 → 5 is calculated by the Conway-CA in two steps, where the number *n* is represented by a (edge-contiguous) cell with *n* elements.

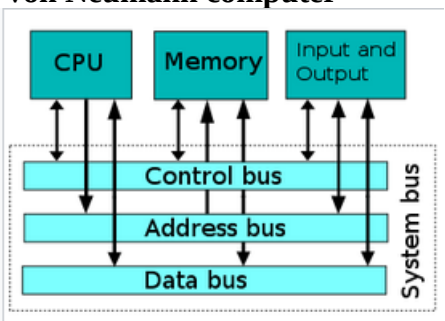


where the input (initial first state) is represented by the vertical 2-cell and a vertical 3-cell in touch at a corner



and the output (=third state) is a (contiguous) 5-cell

-von Neumann computer



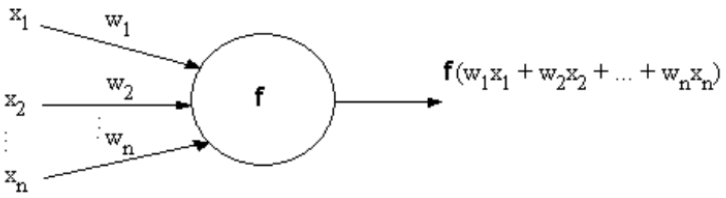
A von-Neumann-computer consists of CPU (processing unit), memory (containing opcode and data), input- and output-unit, they connected by control-bus (opcode), data-bus(data) and address-bus (memory address of current opcode).
The example addition 2+3 → 5 is schematically performed by the sequence

addr= 0 // address is initialized

input= + 2 3 // the next program instruction is read
input → memory(addr=0) // and written into memory
CPU memory(addr=0)) → 5 // CPU fetches the instruction opcode(data1, data2) performs instruction
output= 5 // and outputs result
addr = addr+1 // address is counted up

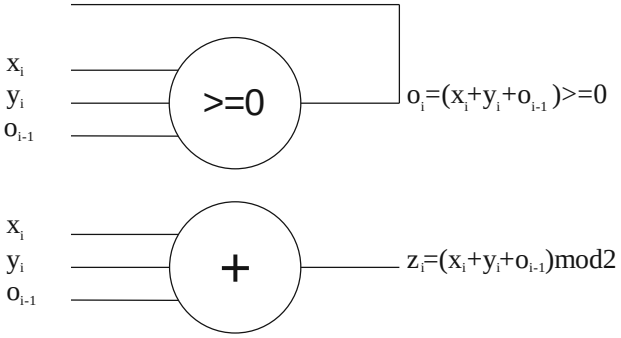
At the end memory-address *addr* points to the next memory location

-Neural network
structure of a neuron



A neural network is a network of neurons with function *f* and input vector $x=(x_1, x_2,...,x_n)$ with weights $(w_1, w_2,...,w_n)$ and the output $y=f(w_1 x_1+ w_2 x_2.+...+ w_n x_n)$. The weights are adapted in such a way that the network of m neurons yields the output vector $Y=(y_1, y_2,...,y_m)$ with desired result.

The example addition 2+3 → 5 in a binary (values=(0, 1)) network is carried out by a recursive network with weights $w_i=1$, i.e. a binary adder circuit z=add(x+y) , one stage $add_i(x_i, y_i, o_i)$ consists of a binary adder, which produces the result bit z_i, and a comparator(threshold $\theta=0$) which produces the overflow o_i for the net stage

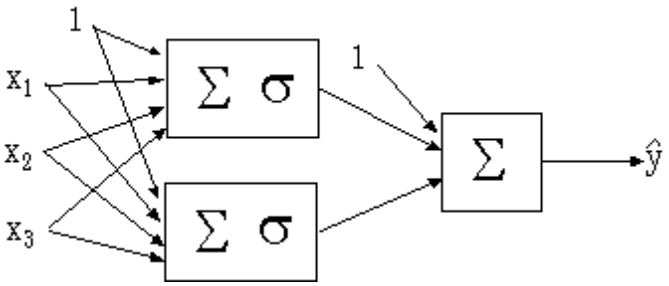


2 Basics about Neural Networks

Mathematically speaking, a Neural Network (NN) is a network of units (neurons) with several inputs with weights (real number $0<=x_i<=1$) and one output with a corresponding output function.

Let the input to a neural network be denoted by x, a real-valued (row) vector of arbitrary dimensionality or length. As such, x is typically referred to as *input*, *input vector*, *regressor* and sometimes, *pattern vector*. Typically, the length of vector x is said to be the *number of inputs* to the network. Let the network output be denoted by $\hat{y}$, an approximation of the desired output y, also a real-valued vector having one or more components, and the *number of outputs* from the network. Often data sets contain many input-output pairs. Then x and y denote matrices with one input and one output vector on each row.

Generally, a neural network is a structure involving weighted interconnections among *neurons*, or *units*, which are most often nonlinear scalar transformations, but which can also be linear. The figure below shows an example of a one-hidden-layer neural network with three inputs, $x = \{x_1, x_2, x_3\}$ that, along with a *unity bias* input, feed each of the two neurons comprising the *hidden layer*. The two outputs from this layer and a unity bias are then fed into the single output layer neuron, yielding the scalar output, $\hat{y}$. The layer of neurons is called hidden since its outputs are not directly seen in the data.



The output is given in the linear case by the following formula

$$\hat{y} = b^2 + \sum_{i=1}^{2} w_i^2 \sigma\left(b_i^1 + \sum_{j=1}^{3} w_{i,j}^1 x_j\right)$$
$$= w_1^2 \sigma\left(w_{1,1}^1 x_1 + w_{1,2}^1 x_2 + w_{1,3}^1 x_3 + b_1^1\right) + w_2^2 \sigma\left(w_{2,1}^1 x_1 + w_{2,2}^1 x_2 + w_{2,3}^1 x_3 + b_2^1\right) + b^2$$

or in general $\hat{y} = g(\theta, x)$, where θ is a real-valued vector whose components are the *weights* of the network, and $g(\theta, x)$ is the *activation function* .

Upon assigning design parameters to a chosen network, thus specifying its structure $g(\cdot,\cdot)$, the user can begin to train it. The goal of training is to find values of the *training parameters* θ so that, for any input x, the network output $\hat{y}$ is a good approximation of the desired output y. Training is carried out via suitable algorithms that tune the parameters θ so that input training data map well to corresponding desired outputs. These algorithms are iterative in nature, starting at some initial value for the parameter vector θ and incrementally updating it to improve the performance of the network.

Neural networks are used mostly in 3 applications: functional approximation, time series (reproduction of signals), classification (of patterns, e.g. letters or faces).

Apart from the training parameters θ , there are *hyperparameters* θ_h , which determine the structure (number of nodes in layers, connection topology, activation function) and the training algorithm (adjusting weights and bias from the forward or backward neighbour layer error).

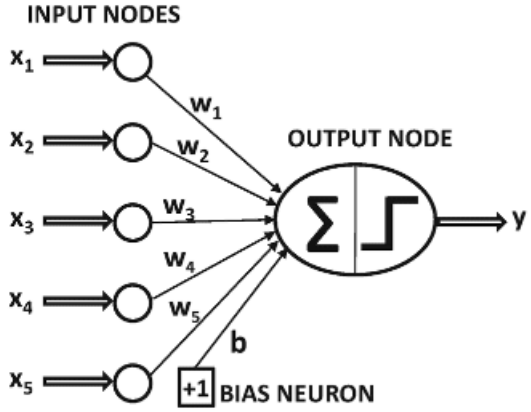
The hyperparameters can be adapted to the training goal in *evolutionary neural networks* , e.g. by *genetic algorithms* .

3 Perceptron

The simplest neural network is the perceptron [2], which contains a single input layer and an output node [1]. The training instance is of the form (X, y), where $X = (x_1, ...x_d)$ contains *d* feature variables , $W = (w_1, ...w_d)$ are the weights, and $y \in \{-1, +1\}$ is the observed value of the output. The function $W \bullet X = \sum_{j=1}^{d} w_j x_j + b$ is computed at the output node. The prediction value for *y* is : $\hat{y} = sign(\sum_{j=1}^{d} w_j x_j + b)$, i.e. the sign of the output value, with an additional bias *b* . The weight vector is updated as follows: $W' = W + \sum_{(X,y)\in S} \alpha(y - \hat{y})X$, where the summation runs over a stochastic subset of the training set *S*, and α is a scaling parameter. The perceptron performs well in prediction of the output *y* only when *S* is separable by a hyperplane.

The modified weights are originally calculated from the minimization of the loss function $L = \sum_{(X,y)\in S} (y - \hat{y})^2$

In order to achieve this, one uses the smoothed gradient of the loss function: $\nabla L = \sum_{(X,y)\in S} (y - \hat{y})X$



Perceptron with bias

Perceptron and classification

Perceptron is a realization of the *data separation problem* on the basis of neural networks [7].

Let us consider two sets of n-dimensional vectors $X_1=\{x_{1i}, i=1...n\}$ $X_2=\{x_{2i}, i=1...n\}$, which we want to separate by a hyperplane $H=\{x=(x_i) , w^t (x-b_0)=0\}$ with the direction vector $w=(w_i)$ and the distance-from-origin vector b_0 , in such a way that $w^t x+b>0$ for X_1 and $w^t x+b<0$ for X_2 , where $b= w^t b_0$.

We can reformulate the problem

$w^t x+b=d$, where $d=sign(w^t x+b)$ is the *signature* of x , d=+1 for $x \in X_1$, d=-1 for $x \in X_2$.

The problem is solvable, if there is such a hyperplane H, for which all vectors in X_1 are on one side, and all vectors in X_2 on the other side of H : X_1 and X_2 are *separable*.

We re-formulate the problem as an *optimization problem* for the variable vector $w=(w_i)$ and a training set $T = \{(x_k, d_k), k = 1...N\}$ of vectors x_k and signatures d_k :

constraints $d_k(w^t x_k + b) \geq 1 \quad k=1...N$

goal function $\Phi(w) = \frac{1}{2} w^t w$

optimization in w : $\Phi(w_0) = min(\Phi(w), w)$ for $w=w_0$

After introduction of Lagrange-multipliers $\alpha=(\alpha_i)$ for the conditions and the Lagrangian function as goal function

$$J(w,b,\alpha) = \frac{1}{2} w^t w - \sum_k \alpha_k (d_k (w^t x_k + b) - 1)$$

and imposing of extremum equations

$$\frac{\partial J(w,b,\alpha)}{\partial w} = 0$$

$$\frac{\partial J(w,b,\alpha)}{\partial b} = 0$$

we get the *dual optimization problem* for the variable vector α and the goal function

$$Q(\alpha) = \sum_k \alpha_k - \frac{1}{2}\sum_{k,l} \alpha_k \alpha_l d_k d_l x_k{}^t x_l$$

constraints $\sum_k \alpha_k d_k = 0$, $\alpha_k \geq 0$ $k=1...N$

optimization in α : $Q(\alpha_0) = max(Q(\alpha), \alpha)$ for $\alpha = \alpha_0$

In general, there will be vectors in X_1 and X_2 , which violate the respective condition (*misclassification*), so we have to reformulate the problem in order to minimize misclassification $d_k(w^t x_k + b) \geq 1 - \xi_k$
with *slack variable* vector $\xi=(\xi_k)$, where $\xi_k \geq 0$ and misclassification occurs when $\xi_k > 1$.
The goal function becomes:

$\Phi(w,\xi) = \frac{1}{2} w^t w + C\sum_k \xi_k$, where C is a user-specified parameter

and the primary optimization problem for w and ξ becomes

constraints $d_k(w^t x_k + b) \geq 1 - \xi_k$ $k=1...N$

optimization in w, ξ : $\Phi(w_0, \xi_0) = \min(\Phi(w,\xi); w, \xi)$ for $w=w_0$ $\xi = \xi_0$

The dual problems for α becomes with the goal function Q(α)

$$Q(\alpha) = \sum_k \alpha_k - \frac{1}{2}\sum_{k,l} \alpha_k \alpha_l d_k d_l x_k{}^t x_l$$

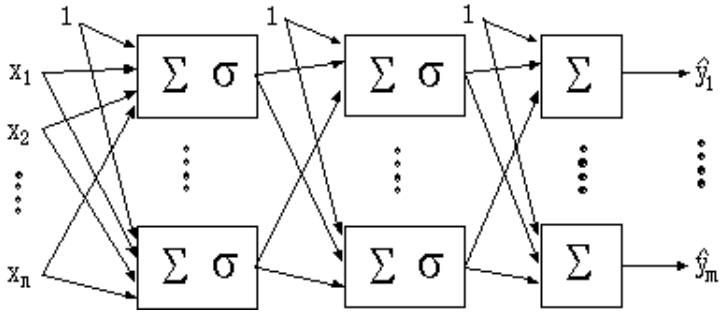
constraints $\sum_k \alpha_k d_k = 0$, $0 \leq \alpha_k \leq C$ $k=1...N$

optimization in α : $Q(\alpha_0) = max(Q(\alpha), \alpha)$ for $\alpha = \alpha_0$

4 Feedforward and RBF neural networks

Feedforward neural networks (FF networks) are the classical *classification networks* in many practical applications [2]. They have a *learning phase*, in which their weights are optimized for a training set $S = \{(X, y)\}$ of inputs $X = (x_1, ...x_d)$ and outputs *y*. In the second *recognition phase*, they sort input vectors *X* which are not in S into corresponding pattern classes *y*, according to the training.

The figure below illustrates a multi-layer FF network with inputs $x_1, ..., x_n$ and *n* outputs $y_1, ..., y_n$. Each arrow in the figure symbolizes a parameter in the network. The network is divided into *layers*. The input layer consists of just the inputs to the network. Then follows *hidden layers*, which consists of any number of *neurons*, or *hidden units* placed in parallel. Each neuron performs a weighted summation of the inputs, which then passes a nonlinear *activation function* σ, also called the *neuron* function.



A multi-layer feedforward network with several hidden layers and one output.

Mathematically the functionality of a hidden neuron is described by

$$\sigma\left(\sum_{j=1}^{n} w_j x_j + b_j\right)$$

where the weights $\{w_1, ..., w_n\}$ are symbolized with the arrows feeding into the neuron.

The neurons in the hidden layer of the network in Figure are similar in structure to those of the perceptron, with the exception that their activation functions can be any differential function. The output of this network is given by

$$\hat{y}(\theta) = g(\theta, x) = \sum_{i=1}^{nh} w_i^2 \sigma\left(\sum_{j=1}^{n} w_{i,j}^1 x_j + b_{j,i}^1\right) + b^2$$

The nonlinrear output function σ is usually the sigmoid $\text{Sigmoid}[x] = \frac{1}{1 + e^{-x}}$

Training the network means normally adjusting the weights taking into account the deviations from the output

$w_{i+1} = w_i + \eta\, x^T \epsilon_i$ where ϵ_i is the error vector, w_i the weights vector and η the learning rate

$$b_{i+1} = b_i + \eta \sum_{j=1}^{N} \epsilon_i[[j]]$$

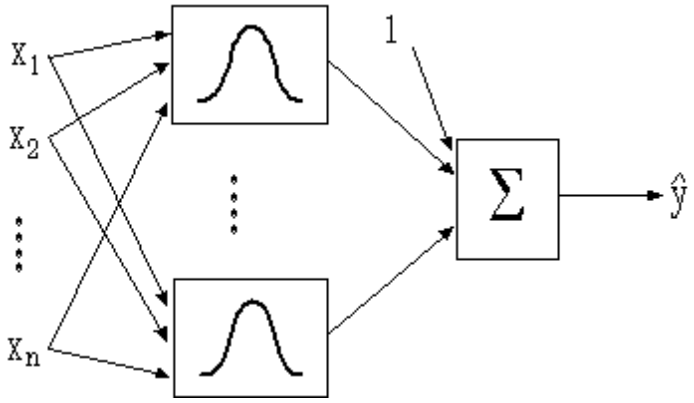
The learning rate η is normally chosen to be

$$\eta = \frac{(\text{Max}[x] - \text{Min}[x])}{N}$$

The initial values for w_i are normally chosen at random from an appropriate interval. Therefore the resulting weights can vary slightly after consecutive training sessions.

RBF networks differ from FF-networks in their use of bell-(Gauss)-functions and the distance neuron-input (radius).

The figure below illustrates an RBF network with inputs $x_1, ..., x_n$ and output $\hat{y}$. The arrows in the figure symbolize parameters in the network. The RBF network consists of one hidden layer of basis functions, or neurons. At the input of each neuron, the distance between the neuron center and the input vector is calculated. The output of the neuron is then formed by applying the basis function to this distance. The RBF network output is formed by a weighted sum of the neuron outputs and the unity bias shown.



An RBF network with one output.

The RBF network is often complemented with a linear part. This corresponds to additional direct connections from the inputs to the output neuron. Mathematically, the RBF network, including a linear part, produces an output given by

$$\hat{y}(\Theta) = g(\Theta, x) = \sum_{i=1}^{nb} w_i^2 e^{-\lambda_i^2 (x_i - w_i^1)^2} + w_{nb+1}^2 + \chi_1 x_1 + \ldots + \chi_n x_n$$

where n_b is the number of neurons, each containing a basis function. The parameters of the RBF network consist of the positions of the basis functions w_i^1, the inverse of the width of the basis functions λ_i, the weights in output sum w_i^2, and the parameters of the linear part $\chi_1,\ldots,\chi_n$. In most cases of function approximation, it is advantageous to have the additional linear part but it can be excluded by using the options.

5 Algorithms in neural networks

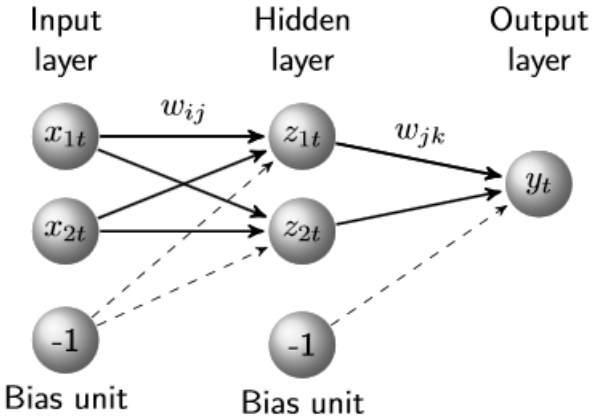
Backpropagation in multilayer networks [JH] [9]

Backpropagation is an algorithm of weight adaption, which starts with the gradient of loss (error) function at the output and proceeds layer by layer down to the input layer.
In each layer *j* the gradient of the output error function is used in respect to the weights of *the step-actual weight* $w_{j,ki}$: backpropagation is a *global-minimization algorithm.*
-forward propagation of inputs

$u_k = \sum_{i=1}^{N} w_{ki} x_i - b_k$ is the net output with weíghts w_{ki} and bias b_k and the input vector $\vec{x}$

or including bias with x_{N+1}=*-1* $w_{N+1,j}$=b_j

$$u_k = \sum_{i=1}^{N+1} w_{ki} x_i$$



[9]

where the activation function for all layers is *f(u)* (usually sigmoid)

$f_{sig}(u) = \left(1 + e^{-u}\right)^{-1}$

so in summary the output vector $\vec{y}$ of each output unit is

$$y_k = f\left(\sum_i w_{ki} x_i\right)$$

and we use the error function= sum of squared deviations from the *k* target-output

$E = \frac{1}{2}\sum_k \left(\hat{y}_k - y_k\right)^2$ where $\hat{y}_k$ is the target-output , y_k is the actual output

Every layer *j* , except the output layer (*j*=*n*) feeds its output as the input to the next layer

$y_{j-1,k} = x_{j,k}$

-backpropagation of weights in output layer [JH]

We calculate the gradient in respect to the weights of the output layer *j*=*n*

$\frac{\partial E}{\partial w_{ki}} = -(\hat{y}_k - y_k) f'(u_k) x_i$ (index *j*=*n* dropped)

W approximate the change in the weight for output layer j between input *i* and the output *k* is using the error gradient, where ε is the learning rate:

$\Delta w_{ki} = -\varepsilon \frac{\partial E}{\partial w_{ki}} = \varepsilon \delta_k x_i$

with the delta-vector $\vec{\delta}$ of the output layer

$\delta_k = f'(u_k)\left(\hat{y}_k - y_k\right)$ for the output layer

-backpropagation of weights in general layer *j* [JH]

We calculate the loss gradient in respect to the weights of the output layer *j*

$\frac{\partial E}{\partial w_{j,ki}} = -(\hat{y}_k - y_{n,k}) \frac{\partial y_{n,k}}{\partial w_{j,ki}}$

Now we have to find an iterative formula for $\frac{\partial y_{j,k}}{\partial w_{j_1,k_1 i}}$, we get

$\frac{\partial y_{j,k}}{\partial w_{j,k_1 i}} = f'(u_{j,k})\, x_{j,i} D_{kk_1}$, where D_{kk_1} is the Kronecker-delta (=1 if k=k_1 , =0 else)

$\frac{\partial y_{j,k}}{\partial x_{j,i}} = f'(u_{j,k})\, w_{j,ki}$ with the *iterative formula valid for any layer j*

$$\frac{\partial y_{j,k}}{\partial w_{j-1,k_1 i_1}} = \sum_i \frac{\partial y_{j,k}}{\partial x_{j,i}} \frac{\partial y_{j-1,i}}{\partial w_{j-1,k_1 i_1}} D_{ik_1} = \frac{\partial y_{j,k}}{\partial x_{j,k_1}} \frac{\partial y_{j-1,k_1}}{\partial w_{j-1,k_1 i_1}} = f'(u_{j,k})\, w_{j,kk_1} f'(u_{j-1,k_1})\, x_{j-1,i_1}$$

-the algorithm can be summarized [9]

initialize network weights w to random values
learning= true
while *learning* do
vector of gradients ∇E=0
foreach *association* from *t=1* to *t=T* do
set input unit states x_{it}=(*t*-th training vector)
get state of output units y_{kt}
get delta term δ_{kt} for each output unit
use output delta terms to get hidden unit delta terms
use delta terms to get vector of weight gradients ∇E_t
accumulate gradient ∇E←∇E+∇E_t
end
get weight change Δw=− ε ∇E
update weights w←w+ Δw
if |∇E|≈0 then set learning=false
end
end

ADAM backpropagation [10, 11]

Adam is a variant of the backpropagation algorithm, that is designed specifically for training deep neural networks. Adam uses a moving weighted average of the actual and the preceding value of the gradient g_t and its square $g_t.g_t$ and calculates estimator values for both in the formula for the weight update.
In total, Adam achieves much better results than the simple gradient-backpropagation and also in general outperforms its predecessor algorithm Stochastic Gradient Descent (SGD).
Adam uses in iteration *t* for the corrected gradient m_t the moving weighted average of the gradient g_t and predecessor value m_{t-1}

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

with fixed parameters β_1 , β_2 with default values of $\beta_1=0.9$ and $\beta_2=0.999$
The estimator values result from the summation of a geometric sum in powers of β_1 , β_2

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

With that, the weights update becomes:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$, where the scaling parameter η is chosen $\eta = 0.01...0.1$ and $\varepsilon << 1$ prevents the denominator from becoming zero.
Adam has the following properties [10, 11]:
The actual step size taken by the Adam in each iteration is approximately bounded the scaling parameter η.
The step size of Adam update rule is invariant to the magnitude of the gradient, which is very helpful in areas with small gradients.
The authors in [11] proved that Adam converges to the global minimum for convex goal functions.

Local backpropagation [JH]

Normal (global) backpropagation minimizes the weights of a layer in respect to the global error function of the last (output) layer. In biological networks however, a layer reacts only to its neighbor next layer.
Therefore, in this case it is advisable to use a *local* variant of the backpropagation algorithm.
In order to apply the formula for Δw_{ki} for every layer, one has to calculate the target values $\hat{y}_i$ of a layer from the target values of the next layer.
-backpropagation of target values
The input values of the layer *j* are the output values of layer *j-1*

$$\hat{y}_{j-1,k} = \hat{x}_{j,k}$$

so we minimize the error of the layer *j* : $E_j = \frac{1}{2}\sum_k \left(\hat{y}_{j,k} - y_{j,k}\right)^2$ in respect to the input $x_{j,i}$ (and not the weights $w_{j,ki}$ as before) in order to get the target values

with $\frac{\partial E_j}{\partial x_{j,i}} = -\sum_k (\hat{y}_{j,k} - y_{j,k}) f'(u_k) w_{j,ki}$ we have the gradient correction for $\hat{y}_{j-1,k} = \hat{x}_{j,k}$

$$\Delta x_{j,i} = -\varepsilon \frac{\partial E_j}{\partial x_{j,i}} = \varepsilon \sum_k \delta_{j,k} w_{j,ki}$$

where $\delta_{j,k} = (\hat{y}_{j,k} - y_{j,k}) f'(u_k)$

so we get for the target value $\hat{y}_{j-1,k}$ in first approximation

$$\hat{y}_{j-1,k} = x_{j,k} + \Delta x_{j,k}$$

-backpropagation of weights
We calculate the gradient in respect to the weights of the *j* layer

$\frac{\partial E_j}{\partial w_{j,ki}} = -(\hat{y}_{j,k} - y_{j,k}) f'(u_{j,k}) x_{j,i}$, where

$\delta_{j,k} = (\hat{y}_{j,k} - y_{j,k}) f'(u_k)$

The change in the weight for layer j between input *i* and the output *k* is using the error gradient, and where ε is the learning rate is then

$\Delta w_{j,ki} = \varepsilon \delta_{j,k} x_{j,i}$, where $x_{j,k} = y_{j-1,k}$ is the input of the layer *j* (= output layer *j-1*)

Numerical local backpropagation [JH]

The formulas given above are valid for neural networks, which consists of consecutive vector-vector linear layer with an output function. But realistic nn's contain usually linear networks with multidimensional matrices as input-output (e.g. 32x32x3 for color 32-pixel images) and pooling or convolutional layers, which transform an image layer with a kernel integration. In this case, it is advisable to use numerical differentiation for the gradient:

$\Delta x_{j,i} = -\varepsilon_x \frac{\partial E_j}{\partial x_{j,i}}$ where $\frac{\partial E_j}{\partial x_{j,i}} \approx \frac{E_j(x_{j,i} + \Delta x_{j,i}) - E_j(x_{j,i})}{\Delta x_{j,i}}$

$\Delta w_{j,ki} = -\varepsilon_w \frac{\partial E_j}{\partial w_{j,ki}}$ where $\frac{\partial E_j}{\partial w_{j,ki}} \approx \frac{E_j(w_{j,ki} + \delta_w) - E_j(w_{j,ki})}{\delta_w}$

with x-correction $\Delta x_{j,i}$ for input x , and w-correction $\Delta w_{j,ki}$ for weights *w*

so in vector notation for gradient $\nabla_x E_j := \left(\frac{\partial E_j}{\partial x_{j,i}} \right)_i$ we get for the original x-vector $\vec{x}_j$ and the corrected

x-vector $\vec{x'}_j$, where $x'_{j,i} = x_{j,i} + \Delta x_{j,i}$, the approximate relation

$\varepsilon_x \left(\nabla_x E_j \right)^2 \approx E_j(\vec{x}_j) - E_j(\vec{x'}_j)$

The algorithm parameters can be adapted for each step so as to reach the approximate corrected energy

$E_j(\vec{x'}_j) = 0$

$\varepsilon_x = \frac{E_j(\vec{x}_j)}{\left(\nabla_x E_j \right)^2}$, or uniformly for the total energy $E(x) = \sum_j E_j(\vec{x}_j)$

$\varepsilon_x = \frac{E(x)}{\left\| \nabla_x E \right\|^2}$

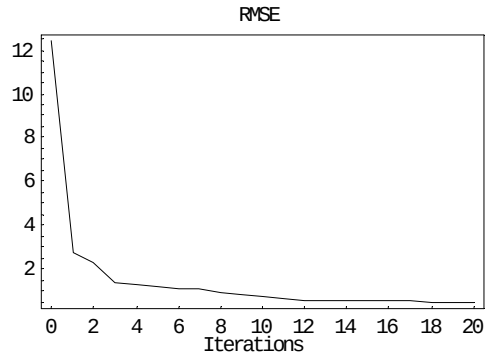
and accordingly for *w*

$\varepsilon_w = \frac{E(w)}{\left\| \nabla_w E \right\|^2}$

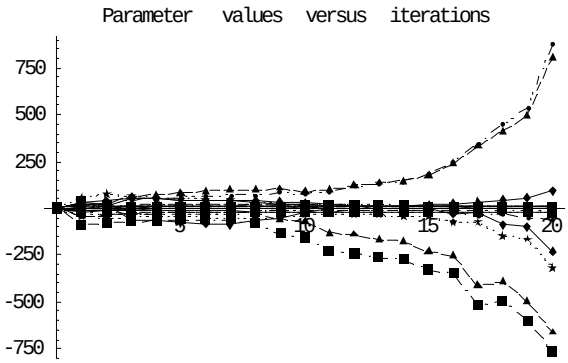
6 Calculations with FF neural networks

Fitting measurement data with feed-forward (FF) neural networks

In the case described here [3], there are 4 components dissolved in water: fv (fructose sugar), sv (vinegar acid), av (ethylic alcohol), gv (glucose sugar) and 3 measured values: KW (cold elasticity), WW (hot elasticity), LW (conductivity). The fitting problems amounts to fitting 3 functions of 4 variables simultaneously.
A feedforward network sigmoid type with 10 neurons, 20 training iterations was used.
The result was as follows (RMSE= root of mean squares):



mean relative error= 3.6%, the build-up of weights is depicted below:

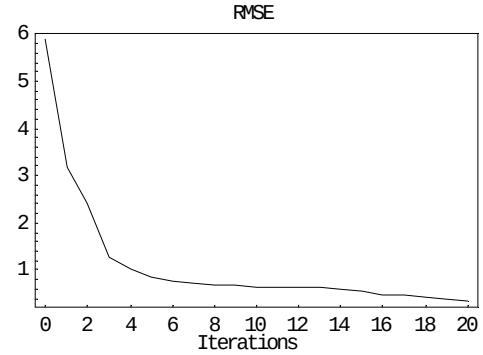


The fit delivers an analytic nonlinear expression, which can be used like a normal fit function (a better-behaved approximation than polynomials, because the functions involved are bounded in the positive region):

$$286.434 - \frac{11.9247}{1+e^{-0.0730337+0.727227\,av+0.750932\,fv+0.850967\,gv-0.917168\,sv}} - \frac{22.711}{1+e^{-0.416058-0.74665\,av-0.319244\,fv-0.590845\,gv-0.131603\,sv}} - \,,,,$$

Calculating component concentrations from measurement values

In practice this means the inversion of the measurement: calculate the 4 components from the 3 measured values. For the problem to be generally solvable, one has to reduce the number of components to 3: then one has 3 (nonlinear) equations for 3 variables. In the above example, one can e.g. introduce the variables sugar zv=fv+gv, acid sgv=sv, alcohol av and reformulate the measurement data accordingly (3 values with 3 components). Then one simply exchanges the input and output: one considers the components as functions of the measurement values and fits the resulting data with a neural network.
First, an approximation with a small feedforward network was tried: 5 neurons, 20 training iterations, 5 trial sessions:



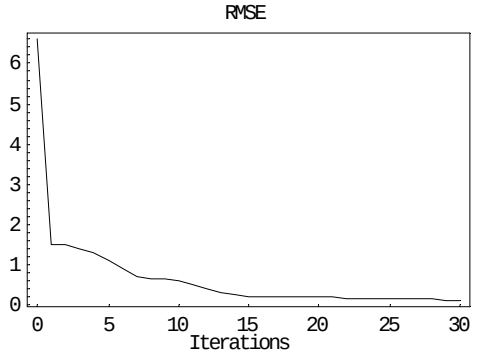
best trial mean relative error= 5.7% (=0.057)

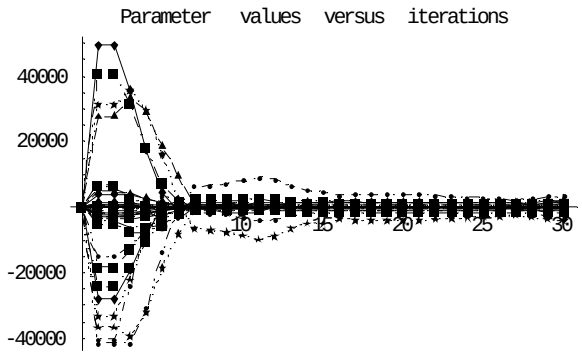
all trial mean relative errors: {0.0957865,0.0746036,0.0686699,0.0570039,0.15861}
complete output expression:

$$\left\{314.847+\frac{11.0818}{1+e^{0.380053+0.836313\,Kw-0.31142\,LW-1.77111\,WW}}-\frac{15.8137}{1+e^{-0.549193-1.16958\,Kw-0.640356\,LW-0.0676694\,WW}}-\right.$$
$$\frac{561.065}{1+e^{-0.195474+0.00313543\,Kw-0.000489175\,LW-0.00205271\,WW}}+\frac{2.25662}{1+e^{0.841638-1.93511\,Kw-0.522092\,LW+0.000177889\,WW}}-\frac{3.50584}{1+e^{-0.0954056+0.713352\,Kw-0.384836\,LW+0.914091\,WW}},$$
$$-1.46466-\frac{0.0903424}{1+e^{0.380053+0.836313\,Kw-0.31142\,LW-1.77111\,WW}}+\frac{3.15354}{1+e^{-0.549193-1.16958\,Kw-0.640356\,LW-0.0676694\,WW}}-$$
$$\frac{1.5761}{1+e^{-0.195474+0.00313543\,Kw-0.000489175\,LW-0.00205271\,WW}}-\frac{0.71649}{1+e^{0.841638-1.93511\,Kw-0.522092\,LW+0.000177889\,WW}}+\frac{1.0765}{1+e^{-0.0954056+0.713352\,Kw-0.384836\,LW+0.914091\,WW}},$$
$$390.774-\frac{5.59672}{1+e^{0.380053+0.836313\,Kw-0.31142\,LW-1.77111\,WW}}+\frac{2.42352}{1+e^{-0.549193-1.16958\,Kw-0.640356\,LW-0.0676694\,WW}}-$$
$$\left.\frac{715.846}{1+e^{-0.195474+0.00313543\,Kw-0.000489175\,LW-0.00205271\,WW}}+\frac{2.73643}{1+e^{0.841638-1.93511\,Kw-0.522092\,LW+0.000177889\,WW}}+\frac{3.38759}{1+e^{-0.0954056+0.713352\,Kw-0.384836\,LW+0.914091\,WW}}\right\}$$

The input start values are chosen at random, therefore the output, too, varies slightly: the better the approximation (more neurons), the less error and output variation results. In the presented case of 5 neurons only, the average mean error is 9.0%, its std (standard deviation) = 4.0%, so the trial relative deviation is 44% (!).

A much better accuracy can be reached with 20 neurons, 30 training iterations, 5 trial sessions:





best trial mean relative error= 1.4% (=0.014)
all trial mean relative errors: {0.0195788,0.0142116,0.0197352,0.0201921,0.0162758}
average mean error= 1.80% , std = 0.262%, so rel. deviation=14.6% .
The output formula consists of 20 exponentials (for 20 neurons) for each of the 3 functions, as compared to 5 in the 5 neurons case (see above).

Training Feedforward NN's with backpropagation
In a multi-layer NN the gradient of a composition function is computed using the backpropagation algorithm. It contains two main phases, referred to as the forward and backward phases, respectively. The forward phase is required to compute the output values and the local derivatives at various nodes, and the backward phase is required to accumulate the products of these local values over all paths from the node to the output:
1.Forward phase: In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights. The final predicted output can be compared to that of the training instance and the derivative of the loss function with respect to the output is computed.
2.Backward phase: The main goal of the backward phase is to learn the gradient of the loss function with respect to the different weights by using the chain rule of differential calculus. Consider a sequence of hidden units $h_1, h_2, \ldots, h_k$ followed by output o, with respect to which the loss function L is computed. Furthermore, assume that the weight of the connection from hidden unit h_r to h_{r+1} is $w_{(h_r,h_{r+1})}$.
Then, in the case that a single path exists from h_1 to o, one can derive the gradient of the loss function with respect to any of these edge weights using the chain rule:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k}\prod_{i=r}^{k-1}\frac{\partial h_{i+1}}{\partial h_i}\right]\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \quad \forall r \in 1 \ldots k$$

In general, there are several paths, so we sum over them:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\sum_{[h_r,h_{r+1},\dots h_k,o] \in \mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$$

7 Hopfield networks

Hopfield networks [1, 9] are the *classical memory networks*, which are able to find a similar memorized pattern when an incomplete or distorted pattern is presented as input.

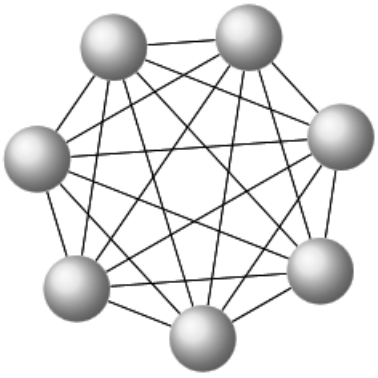
A Hopfield network is an undirected network, with *K* units and connections of the form *(i, j)* . Each connection *(i, j)* is undirected, and is associated with a symmetric weight $w_{ij}=w_{ji}$. Each neuron *i* is associated with a binary state $y_i \in \{-1,+1\}$ (in biological terms: firing or non-firing). The associated energy *E* of a particular combination of states $y = (y_1,...y_K)$ of the Hopfield network can be defined as follows:

$E = -\sum_i b_i y_i - \sum_{i,j} w_{ij} y_i y_j$ resp. $E = -\sum_{i,j} w_{ij} y_i y_j$ with “included” bias

with weights w_{ij} and biases b_i . The Hopfield network is a state machine, where at time points t=1,2,... the state values are updated dependending on the weights and the state vector at thre previous time point with the update rule

$$y_i = \begin{cases} +1 \ \textit{if} \ \sum_{j,j\neq i} w_{ij} y_j + b_i \geq 0 \\ -1 \ \textit{otherwise} \end{cases} \quad \text{resp.} \quad y_i = \begin{cases} +1 \ \textit{if} \ \sum_{j,j\neq i} w_{ij} y_j \geq 0 \\ -1 \ \textit{otherwise} \end{cases} \quad \text{with “included” bias}$$

Hopfield networks are trained on a training set $X = \{(x_{i1},..,x_{iK}), i = 1...T\}$ of *T* test vectors, where the weights w_{ij} are set in such a way, that they minimize the energy near the test vectors values for the neurons, the network is said to ”memorize” the test data. Afterwards, starting with some initial state vector, the network converges to one of the memorized vectors.



A Hopfield network with seven units and $(7 \times 7 - 7)/2 = 21$ weights (bias unit and thresholds not shown).

The states of *K* units can be represented as a column vector

$\mathbf{y} = (y_1, \ldots, y_K)^\mathsf{T}$, where a state is +1 or -1

we wish the network to memorize a particular set of values

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

for T training vectors, we sum the weights for each training vector (learning rule)

$$w_{ij} = \epsilon \sum_{t=1}^{T} x_{it}\, x_{jt},$$

the energy of the Hopfield network is

$$E(\mathbf{y}) = -\sum_{i=1}^{K} \sum_{j=i+1}^{K} w_{ij}\, y_i\, y_j.$$

where the energy contributed by unit U_i is

$$E_i = -y_i \sum_{j=1}^{K} w_{ij} y_j$$

The defining property of the Hopfield net energy function *E(y)* is that its value always decreases (or stays the same) as the network evolves, i.e. the network converges by learning to a stable state: the final state is a minimum of *E* .

The algorithm of the Hopfield network can be summarized

Hopfield Net: Learning

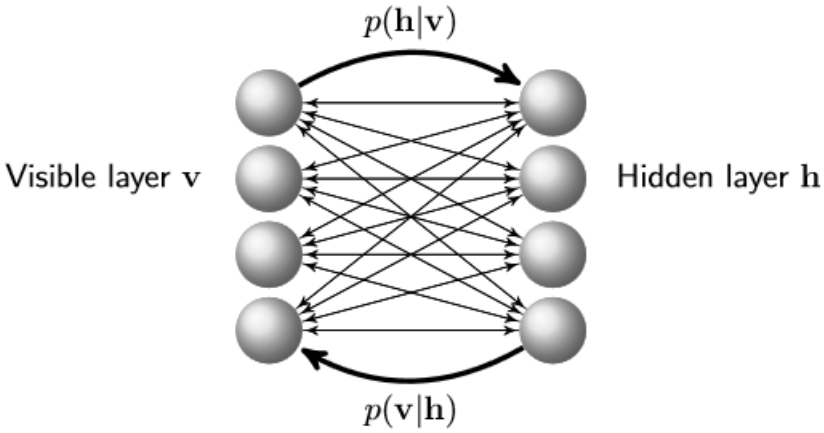
initialise network weights W to zero
foreach *training vector from* $t = 1$ *to* T **do**
  find weights for the vector $\mathbf{x}_t$: $W_t = \mathbf{x}_t \mathbf{x}_t^\intercal$
  update weights $W \leftarrow W + W_t$
end

Hopfield Net: Recall

Recall vector x from corrupted vector x′
set network state to $\mathbf{y} = \mathbf{x}'$
while *stable* = *false* **do**
  set stable to true (can be reset in loop below)
  reset the set of unit indices to $J = \{1, \ldots, K\}$
  foreach k *from* 1 *to number of units* K **do**
    choose unit index j from J without replacement
    find input to unit j: $u_j = \sum_i w_{ij} y_i$
    note current state as $y_{\text{last}} = y_j$
    get state of unit j: $y_j = f(u_j)$
    if $y_j \neq y_{\text{last}}$ **then**
      set stable to false
    end
  end
end

8 Restricted Boltzmann machines (RBM)

A RBM is a Boltzmann machine, which has no connections within a layer, so the layers can be stacked [9].
The RBM's are *generative stack neural networks* , after a learning phase with defined input and output, they generate "similar" output from an untrained input (e.g. Van-Gogh-style image from a photograph of New York skyline).
A RBM is essentially a Hopfield net, but with the addition of hidden layers.
The output of a unit (state) is binary: $y_i=\{0, 1\}$, generated by a threshold from probability *P*.



A restricted Boltzmann machine (RBM) with four input units and four hidden units.

The output of a unit is $y_j = P\left(u_j\right)$ where $u_j = \sum_{i=1}^{K+1} w_{ij} y_i$ with the sigmoid probability function P=P(y)

(probabílity of y) with $P = \left(1 + e^{-u_j/T}\right)^{-1}$, where T is the temperature.

The RBM has the same energy function as the Hopfield network

$$E(\mathbf{y}) \quad = \quad -\sum_{i=1}^{K} \sum_{j=i+1}^{K} w_{ij} y_i y_j .$$

Learning in a Boltzmann machine comprises two nested loops.
On each iteration of the inner loop, the correlation between unit states y_i and y_j is measured under two conditions:
1 when visible units are set to training vectors (wake phase), which yields the expectation value $E[y_i y_j]_{Wake}$;
2 when visible units are random (sleep phase), which yields the expectation value $E[y_i y_j]_{Sleep}$.
On each iteration of the outer loop, the weights are adjusted so that after learning $E[y_i y_j]_{Wake} \approx E[y_i y_j]_{Sleep}$.

In wake phase, the correlation between the *i* visible unit's state and the *j* hidden unit's state is

$$E\left[v_i P_j\right]_{wake} = \frac{1}{T} \sum_{t=1}^{T} v_{it} P_{jt}$$

In sleep phase, the correlation between the *i* visible unit's state and the *j* hidden unit's state is estimated as

$$E\left[P_i P_j\right]_{sleep} = \frac{1}{T} \sum_{t=1}^{T} P_{it} P_{jt}$$

The RBM learning rule with the learning rate ε is:

$$\Delta w_{ij} = \varepsilon \left(E\left[v_i P_j\right]_{wake} - E\left[P_i P_j\right]_{sleep} \right)$$

The training algorithm of RBM is

Training RBM
foreach *iteration* do
foreach *training vector* x_t t= 1 to T do
sleep phase: get data v_i , P_j for input= x_t
set each visible unit state $v_{it} = x_{it}$
use v_t to calculate input u_{jt} to each hidden unit

calculate probability P_{jt}
record wake products $v_{it}P_{jt}$ between connected units
wake phase:get data v_i , P_j for random input
initially use hidden state probabilities P_{jt} from above
foreach k= 1 to N samples do
sample binary hidden states h_{kt}
use h_{kt} to calculate input u_{kit} to each visible unit
probability that i visible unit is on is P_{kit}
sample binary visible states v_{kt} , use to calculate input u_{kit} to each hidden unit
probability that j hidden unit is on is P_{kjt}
record sleep products P_{kit} P_{kjt} between connected units
end
end
calculate $E\left[v_i P_j\right]_{wake}$
calculate $E\left[P_i P_j\right]_{sleep}$
update weights with Δw_{ij}
end

In the recall phase RBM “recalls” the learned contents from distorted input

Recall RBM
set vector input=x
generate output vector y with $y_j = P\left(u_j\right)$
calculate binary output $\hat{y}_j = step(y_j, \theta)$ with the threshold θ

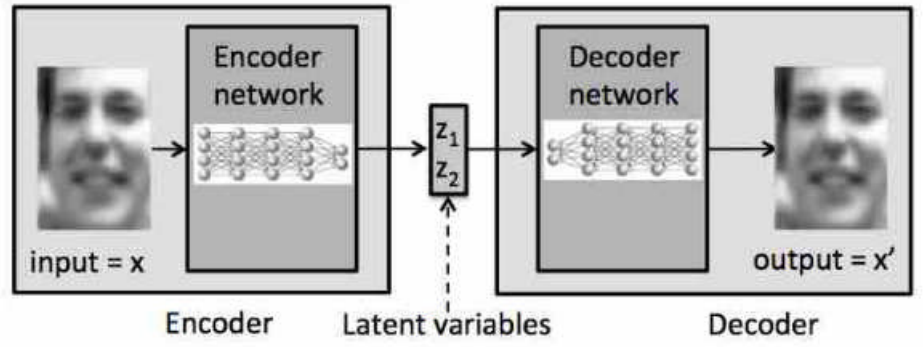
9 Variational Autoencoders

Variational autoencoders are basically *data-compressors* [9].

A variational autoencoder consists of two modules, an encoder and a decoder. The data are a set of T images$\{x\}=\{x_1,...,x_T\}$. The encoder maps each input x_t to a layer of *2J* encoder output units, which represent the values of a small number *J* of *latent variables* (e.g. orientation of a face image, expression of a face). Each latent variable is represented by the mean μ and standard deviation σ of a Gaussian distribution $P_G(x, \mu, \sigma)$ of the corresponding random variable *x* .

Thus, each encoder output state consists of two J-element vectors $\mu= (\mu_1,...,\mu_J)$ and $\sigma= (\sigma_1,...,\sigma_J)$.

A variational autoencoder consists of two modules, an encoder and a decoder, with a "bottleneck" of latent variables in between.



Simplified schematic of a variational autoencoder, which comprises an encoder and a decoder. The encoder maps each input image x to several output unit states (z_1 and z_2), which can be assumed to be equalto the latent variables for now. The outputs of the encoder act as inputs to the decoder, which produces an approximation x' of the input image.

The assessed deviation between x and x' is for the encoder

$$L_t^{\text{enc}} = \frac{1}{2}\sum_{j=1}^{J}(1+\log\sigma_{jt}^2-\sigma_{jt}^2-\mu_{jt}^2),$$

and for the decoder

$$L_t^{\text{dec}} = c-\frac{1}{2N_z}\sum_{i=1}^{N_z}|\mathbf{x}_t-\mathbf{x}'_{it}|^2.$$

$$c = -\log\left((2\pi)^n\det(\Sigma_{\mathbf{x}})\right)^{1/2}, \text{ n=2}$$

where N_z is the number of samples, Σ_x is a n×n-covariance matrix of x, and $\det(\Sigma_x)$ is its determinant.

Variational autoencoders training algorithm is as follows

Training a Variational Autoencoder

set dimensionality J of latent space $\mathbf{z}$
initialise weight vector $\mathbf{w}$ to random values
set number N_z of $\mathbf{z}$ samples per input vector
set learning rate ϵ
foreach *learning iteration* **do**
  foreach *batch of* T_{batch} *training vectors* $\{\mathbf{x}\}$ **do**
    set $L = 0$ and gradient vector $\nabla L = 0$
    foreach *training vector* $\mathbf{x}_t$ *from* $t = 1$ *to* T_{batch} **do**
      Encoder
      given an input $\mathbf{x}_t$, the encoder outputs are $\boldsymbol{\mu}_t$ and $\boldsymbol{\sigma}_t$
      the encoder regularisation term is L_t^{enc}
      Decoder
      set $L_t^{\text{dec}} = 0$
      foreach *sample* i *from 1 to* N_z **do**
        obtain sample $\mathbf{z}_{it}$ from $q(\mathbf{z}|\mathbf{x}_t) = \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t)$
        use $\mathbf{z}_{it}$ to obtain decoder output $\mathbf{x}'_{it}$
        reconstruction term: L_{it}^{dec}
        accumulate mean: $L_t^{\text{dec}} = L_t^{\text{dec}} + L_{it}^{\text{dec}}/(2N_z)$
      end
      Estimate L_t **and its gradient**
      $L_t = L_t^{\text{dec}} + L_t^{\text{enc}}$
      $\nabla L_t = \nabla L_t^{\text{dec}} + \nabla L_t^{\text{enc}}$
      accumulate L and gradient for this batch
      $L = L + L_t$
      $\nabla L = \nabla L + \nabla L_t$
    end
    Update weights
    $\mathbf{w} = \mathbf{w} + \epsilon \nabla L$
  end
end

10 Unsupervised Networks

Unsupervised networks are *structure-finding* networks [1]. They can, for instance, be used to find clusters of data points, or to find a one-dimensional relation in the data.

An unsupervised network consists of a number of *codebook vectors,* which constitute cluster centers. The codebook vectors are of the same dimension as the input space, and their components are the parameters of the unsupervised network. The codebook vectors are called the *neurons* of the unsupervised network.

An unsupervised network is trained by adapting the locations of the codebook vectors so that the mean Euclidian distance between each data point and its closest codebook vector is minimized.

In the training algorithm, the codebook vectors are adapted in a recursive manner, considering one data sample in each update. There are two basic algorithms:

Standard competitive learning rule

Given *N* data vectors $\{x_k\}$, *k=1,...,N,* in each update, the following steps are performed.

1. *k* is chosen randomly from a uniform integer distribution between 1 and *N,* where the whole range is considered each time this step is executed.
2. The codebook vector closest to x_k, called the winning neuron, or the winning codebook vector, is identified. Its index is indicated by *i*.
3. The winning codebook vector is changed according to

$w_i=w_i+SL[n] * (x_k-w_i)$

where *n* is the iteration number.

4. The described steps are repeated *N* times in each iteration.

where *SL[n]* is the StepLength function

SOM or Kohonen's algorithm

Given *N* data vectors $\{x_k\}$, *k=1,...N,* in each update, the following steps are performed.

1. *k* is chosen randomly from a uniform integer distribution between *1* and *N,* where the whole range is considered each time this step is executed.
2. The codebook vector closest to x_k, called the winning neuron, or the winning codebook vector, is identified. Its index is indicated by $\{i_{win},j_{win}\}$.
3. All the codebook vectors are changed according to

$$w_{i,j} = w_{i,j} + SL[n] * Exp[-NS[n] * NM[[c_1 - i_{win} + i, c_2 - j_{win} + j]]] * (x_k - w_{i,j})$$

where *n* is the iteration number and $\{c_1,c_2\}$ is the center position of the neighbor matrix,

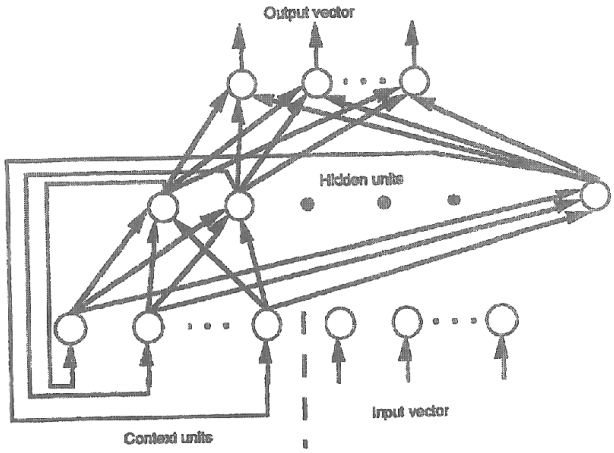
4. The described steps are repeated *N* times in each iteration.

where *SL[n]* is the StepLength function and *NS[n]* is the NeighborStrength function, *NM* is the neighbor matrix dictating which codebook vectors are neighbors. The neighbor matrix *NM* should have its minimum at its center element $\{c_1,c_2\}$, so that the winning neuron update is most pronounced. One *iteration* of the stochastic algorithm (that is, *n* incremented by *1*), consists of *N* updates via equation in step 3.

11 Recurrent networks

Recurrent networks are used for *learning of time series*, e.g. for next word prediction in text analysis [2]. They are used mainly for text analysis and word prediction, and can even "compose", i.e. generate texts according to the learning samples like Shakespearean speech.

Elman network



An Elman network consists of n_i input units $X = (x_1,...x_{n_i})$, and n_o output units $Y = (y_1,...y_{n_o})$. There is a *moving frame* of *p* units, which are mapped onto *p* hidden layer units $H = (h_1,...h_p)$ in every time step.

The algorithm for Elman network is

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

where σ is the activation function, usually *sigmoid=1/(1+Exp(-x))* , W_h and U_h is a (p,n_i) matrix, b_h and b_y are bias vectors, W_y is a (p,n_o) matrix .

Jordan network

is an alternative model, where in addition the output vector instead of the hidden layer is mapped onto the hidden layer

$$h_t = \sigma_h(W_h x_t + U_h y_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

12 Convolutional networks

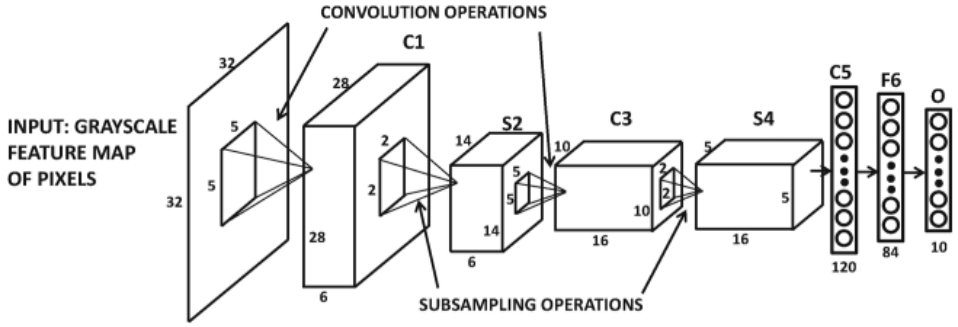
Convolutional neural networks (CNN [1]) work on grid-structured inputs with strong spatial dependencies in local regions of the grid, i.e. they are *image-classification* networks. The name is derived from their function of filtering, which is in most cases a mathematical discrete convolution (i.e. discrete integration with a local kernel function like a gaussian kernel). Taking 3 basic colors into account, the 2-dimensional filter kernel becomes 3-dimensional with *depth=3* .

The CNN contains several hidden layers, each performing a certain function e.g. edge detection, where the convolutional operations for a kernel with $F_q \times F_q \times d_q$ parameters from the layer q to the layer $(q+1)$ are defined as follows:

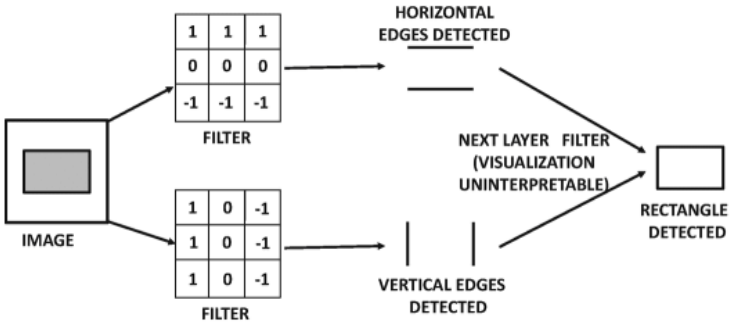
$$h_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)} \qquad \begin{matrix} \forall i \in \{1 \ldots, L_q - F_q + 1\} \\ \forall j \in \{1 \ldots B_q - F_q + 1\} \\ \forall p \in \{1 \ldots d_{q+1}\} \end{matrix}$$

where we have L_q positions along the height and B_q positions along the width of the image.

The general structure of a CNN is:



The detection operations are divided into basic filters e.g. a rectangle detection runs as follows:



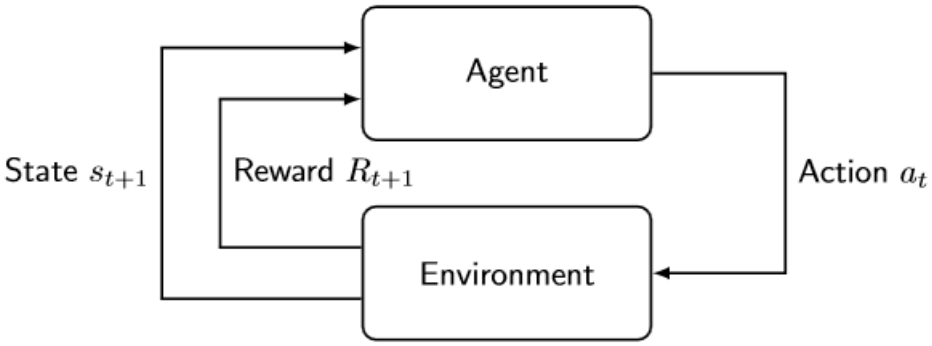
The weights in the filter are adapted by backpropagation via convolution of loss (=deviation) function gradient from layer $(q+1)$ to layer q .

13 Reinforcement learning in neural networks (REL)

In reinforcement learning, we have an *agent* (neural network being in a particular *state* s_i at a discrete time point t_i) that interacts with the environment with the use of *actions* [1, 9]. For example, the player is the agent in a video game, and moving the joystick in a certain direction in a video game is an action. These actions change the environment and lead to a new state s_{i+1} . The environment gives the agent *rewards*, depending on how well the goals of the learning application are being met. A particular episode of this process is a finite sequence of actions, states, and rewards:

$s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_t a_t r_t \ldots s_n a_n r_n$

Examples are video games, robot locomotion, self-driving cars.



An agent uses the current state of the environment to generate an action a_t, which changes the state of the environment tos_{t+1} and usually elicits an immediate reward R_{t+1}.

The state transitions $s_t \rightarrow s_{t+1}$ happen with probabilities $p(s_{t+1}|s_t)$, which depend only on s_t and s_{t+1} , not on the previous states: such a random process is called Markov process.

A Markov decision process is defined in terms of the state *s*, the action *a*, the reward *r*, and the transition probabilities $p(s_{t+1}|s_t)$, the underlying process in reinforcement learning is assumed to be a Markov process.

Important quantities

•An episode is a complete sequence of states (like a game of chess).

•The policy π specifies the probability of choosing each of a set of possible actions$\{a_t\}$, given the current state s_t at time t.

•The reward signal R_{t+1} is an immediate reward for the current state s_t, which usually results from the action just taken.

•The return G_t is the cumulative total reward acquired from time t+ 1 to the end of an episode.

•The state-value function $v(s_t)$ is the expected return based on the current state s_t, that is, the return G_t averaged over all instances of the state s_t. The estimate of $v(s_t)$ is $V(s_t)$.

•The action-value function $q_\pi(s_t,a_t)$ defines the expected return based on the current state s_t and the action a_t specified by the policy π. The estimate of $q_\pi(s_t,a_t)$ is $Q(s_t,a_t)$.

•The eligibility trace is a scalar parameter that indicates how often each state has been visited. Each state has its own eligibility trace, which is given a boost every time that state is visited, before decaying exponentially thereafter at a rate determined by the trace-decay parameter λ.

•Temporal discounting means that immediate rewards are valued more than future rewards; it is determined by the parameter γ, which affects the state-value and action-value functions.

•The temporal difference error or TD error is the difference between the value $v(s_t)$ and the estimated value $V(s_t)$

The most important equation in REL is the **Bellmann equation**

$$v^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v^\pi(s_{t+1})|s_t = s].$$

The Bellman equation says that the value $v(s_t)$ of the state s_t equals the expected sum of the immediate reward R_{t+1} and the discounted return $\gamma v_\pi(s_{t+1})$ of the next state s_{t+1}.

A good learning algorithm for REL is **Q-learning**

Q-Learning
set $Q(s_T, a) = 0$, where s_T is the terminal state in an episode
foreach *episode* **do**
  initialise state s_t
  while *state s_t is not a terminal state* **do**
    choose action a_t from Q using ε-greedy policy
    take action a_t
    get new state s_{t+1} and observe reward R_{t+1}
    get optimal action for state s_{t+1}: $a^*_{t+1} = \text{argmax}_a\, Q(s_{t+1}, a)$
    use $Q(s_{t+1}, a^*_{t+1})$ to update the action-value function:
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \epsilon[R_{t+1} + \gamma Q(s_{t+1}, a^*_{t+1}) - Q(s_t, a_t)]$
    $s_t \leftarrow s_{t+1}$
  end
end

Here *greedy policy* means choosing an action so that it maximizes the estimated return

$$a^*_{t+1} = \underset{a}{\text{argmax}}\ Q(s_{t+1}, a).$$

ε-greedy policy adds a random-action with probability ε .

Part 2 Applications of neural networks

1 Implementation on GPU hardware: Nvidia CUDA

([3] test CUDA)

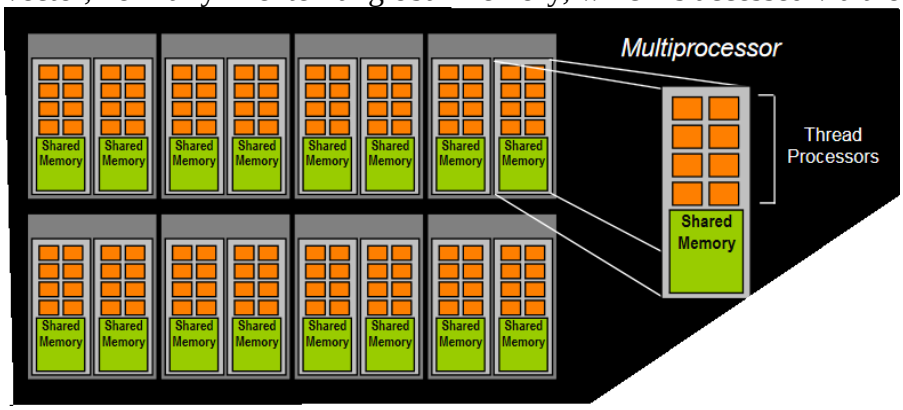
Currently, the most advanced parallel hardware implementation of neural networks runs on parallel graphic processors (GPU) of graphic interface hardware. In this area, CUDA-C-interface of Nvidia is a de-facto standard.

In the following, the theory of Nvidia GPU hardware and its actual implementation in C and in Mathematica is described.

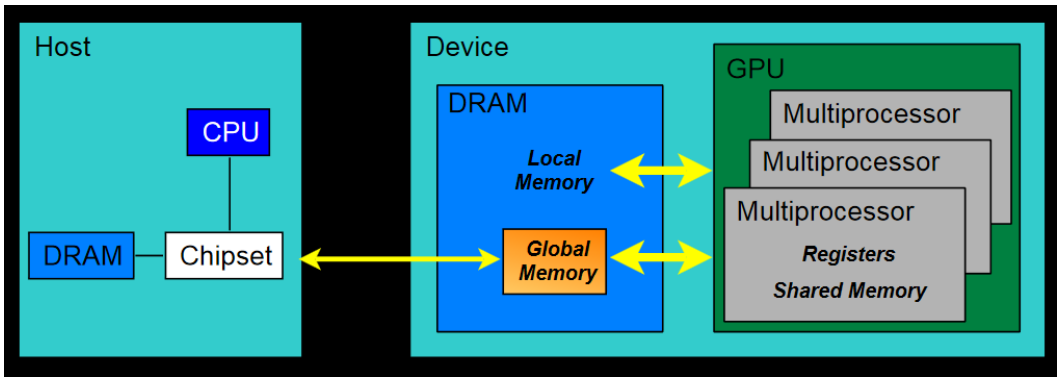
In the context of neural networks, the GPU's can be considered as an array of (32 bit) floating-point arithmetic units with dedicated shared(fast) memory, constant memory and normal (slow) memory.. For example, the graphic card Nvidia Quadro RTX4000 contains 2034 GPU-kernels located in 36 multiprocessors (64 kernels per multiprocessor), shared (fast) memory 49kB/block, constant memory 65kB/block, total normal GPU memory 8.5GB.

The GPU-subroutines are programmed in C++ via Nvidia C-compiler and CUDA-library and the code is executed via drivers in parallel on GPU-processes (threads). Subroutine parameters are vectors or matrices (2-dimensional vectors) or matrix-vectors (3-dimensional vectors) passes as pointers (addresses), and the vector length, and the result is stored in another vector pointer. The length is subdivided into one- or two- or three-dimensional blocks (of threads), with maximum number of 1024 threads in a block. For instance, a subroutine for vector(4096) with matrix(4096x4096) multiplication could have block length L_b=1024, i.e. 1024 threads work in parallel on

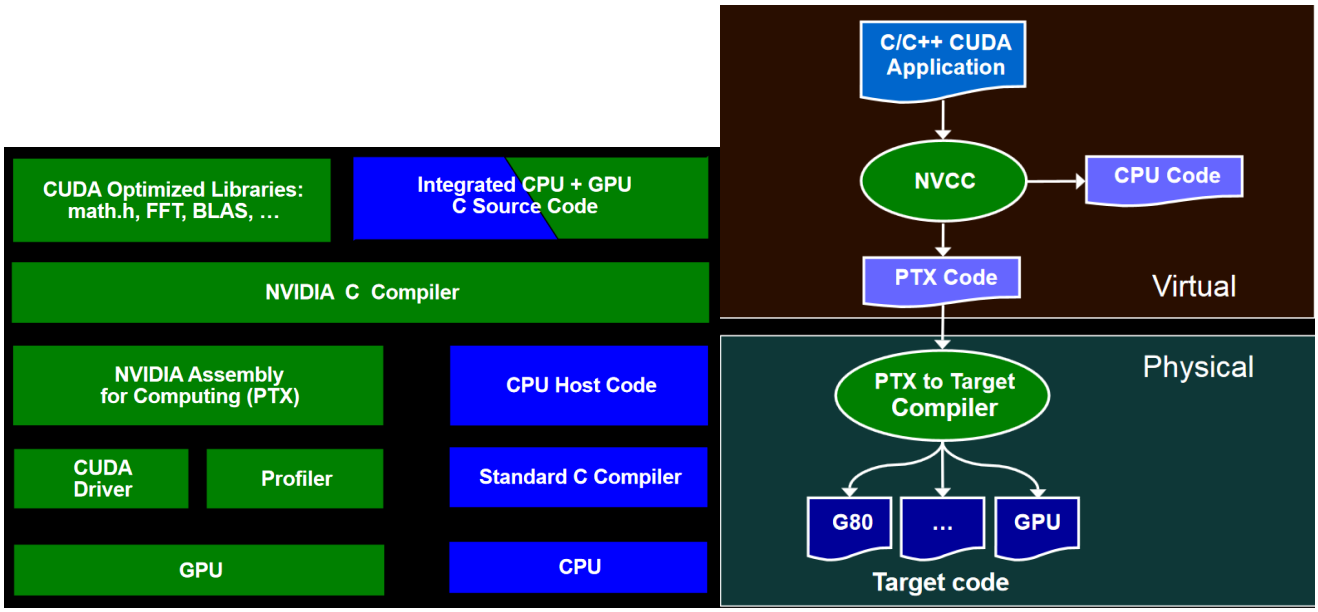
32-chunks of the vector and 32x32-chunks of the matrix. The threads execute the code asynchronously, and in the code only the current block index (*blkIdx.x, blkIdx.y*) and the current thread index (*threadIdx.x, threadIdx.y*) within the block are known at execution time, so intermediate storage is a dedicated vector, normally in external global memory, which is accessed via those indices.



Kernel architecture [13]: every multiprocessor manages 64 kernels with dedicated shared memory

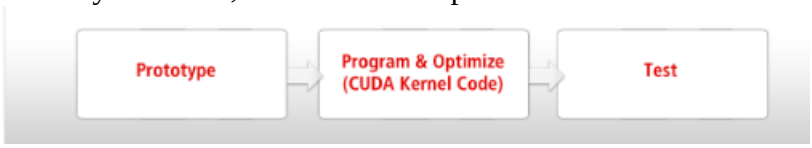


Memory architecture [13]

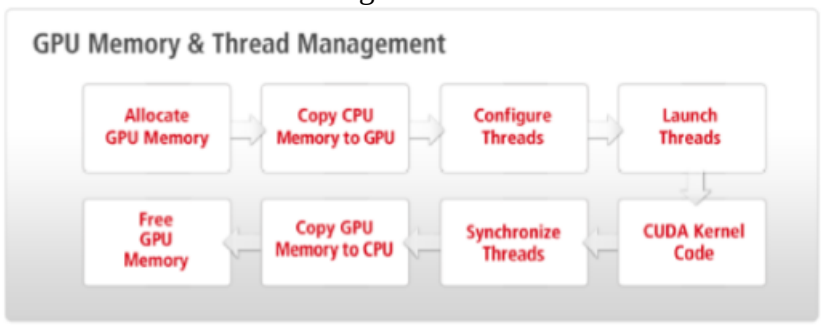


Software architecture [13]: the NVCC-compiler generates CPU-code for the host CPU, and PTX-code for the GPU-hardware

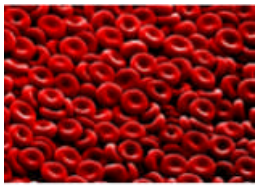
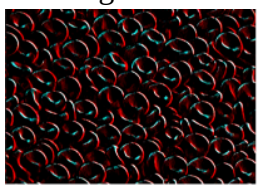
Neural network applications in this paper are programmed and executed under Mathematica, so the high-level code generation and execution runs via Mathematica C-compiler and the Mathematica CUDA-Link [14], where the Mathematica allocates and frees the GPU-memory, copies the CPU-host parameter memory to the GPU memory and back, and in-between passes control to the NVCC-compiler and CUDA-drivers



whereas the low-level code generation and execution runs via NVCC-compiler and CUDA-drivers



Mathematica uses built-in CUDA image-processing subroutines like CUDAImageConvolve (image filtering)

 ---------->  [14]

and supports parallel GPU-processing for ADAM-backpropagation FF-networks via built-in functions
NetChain, NetGraph for network generation
NetTrain, NetMeasurement for network training and testing
which gives an acceleration factor of 30 compared to pure CPU-processing [3].

An example of Mathematica-based user-programmed CUDA-code is vector-matrix product [3]
matDotV(invector.matrix -> outvector)

```
code =
  "
 __global__ void matDotV(Real_t * y, Real_t * x, Real_t * z, mint N){
    extern __device__ Real_t cache[64*1024*1024];
    int tidx = threadIdx.x+ blockIdx.x*blockDim.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int cacheIndex = threadIdx.x ;

    Real_t temp=0;
    while (tidx<N){
        temp+=y[row * N + tidx]*x[tidx];
        tidx +=blockDim.x*gridDim.x;
    }
    cache[cacheIndex + row * N]=temp;
    __syncthreads();
    int i=blockDim.x/2;
    while(i!=0){
        if (cacheIndex<i)
            cache[cacheIndex + row * N]+=cache[cacheIndex+ row * N +i];
        __syncthreads();
        i/=2;
    }
    if(cacheIndex==0)
        z[blockIdx.x  + row * N]=cache[cacheIndex + row * N];
}
";

dotMatV = CUDAFunctionLoad[code, "matDotV", {{_Real, _, "Input"},
{_Real, _, "Input"}, {_Real, _, "Output"}, _Integer}, {32, 32}, "ShellCommandFunction" → Print]
```

with the actual call *dotMatV(invector A, matrix B, outvector C1, length listSize=4096)* :

```
res = dotMatV[A, B, C1, listSize];
```

In the above example, two-dimensional parallel threads of 32x32=1024 threads are used, where *threadIdx.x* and *threadIdx.y* are the thread indices within a 2-dimensionsl block, where *blkIdx.x* and *blkIdx.y* are the block indices, and the external array *cache* is used for intermediate memory storage. The results for block-wise scalar product are stored in consecutive locations in the result vector *C1* .

However, since at every subroutine call the (huge) CPU-host parameter memory is copied to GPU memory and back, there is no significant acceleration for neural network applications when using this subroutine. In order to use the whole power of parallel GPU-hardware, one has to program the entire network algorithm in CUDA-C code, which is a time-consuming task. Therefore, in this paper the user-programmed local backpropagation was executed in parallel CPU-code with N=12 CPU kernels, whereas the built-in global propagation was executed using the CUDA-link with N=1024 GPU kernels, which is much faster.

2 Implementation of backpropagation FF networks with CIFAR data

([3] Example network classification 2-out CIFAR-100 different NN's)

Training database CIFAR-100

CIFAR-100 is a standard training database for neural networks. It is an rgb 32x32 pixels image database with 50000 records. It is used in the following as training and as test database. A dataset consists of an image, “Label” i.e. index of category (e.g. “food containers”) (ic=1…20), and “SubLabel” i.e. index of sub-category (e.g. “bottle”) (is=1…5) .
The networks are trained on CIFAR for Label and SubLabel simultaneously.

Example: a random sample of 5 elements

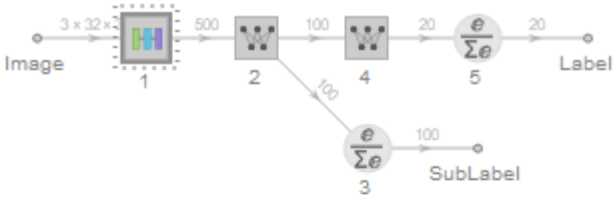
Image	Label	SubLabel
	large man-made outdoor things	castle
	food containers	bottle
	non-insect invertebrates	lobster
	large natural outdoor scenes	cloud
	household furniture	chair

Types of networks

In a performance test with global ADAM-backpropagation , we use 4 types of networks with varying number of layers: convolution, recurrent, linear-pooling and simple linear.

Layers are positioned in series in groups consisting of one or two number-reducing layers and one element-wise function (in general a Ramp or a Sigmoid function). Number-reducing means that the number of output nodes should be in general smaller than the number of the input nodes, so that the final output matches the vector of 20 Label indices, resp. 100 SubLabel indices.

Convolution-pool network consists of convolution layers (moving average with a specifiable kernel), grouped with pooling layers (moving mean or max over a specified sub-block), and element-wise output function (here Ramp, i.e. positive(x)), in this example with 2 convolution layers and one output linear layer.



Input Port

Image: image

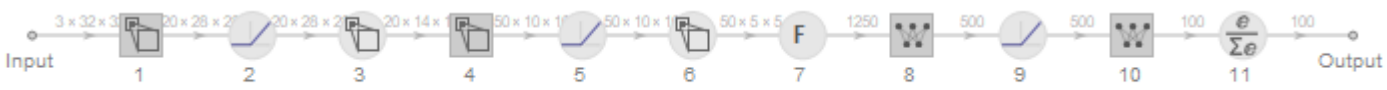
Output Ports

SubLabel: class

Label: class

1: NetChain

	Input	array (size: 3×32×32)
1	ConvolutionLayer	array (size: 20×28×28)
2	Ramp	array (size: 20×28×28)
3	PoolingLayer	array (size: 20×14×14)
4	ConvolutionLayer	array (size: 50×10×10)
5	Ramp	array (size: 50×10×10)
6	PoolingLayer	array (size: 50×5×5)
7	FlattenLayer	vector (size: 1250)
8	LinearLayer	vector (size: 500)
9	Ramp	vector (size: 500)
	Output	vector (size: 500)



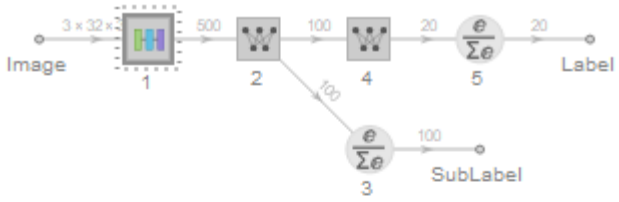
Input Port

Input: image

Output Port

Output: vector (size: 100)

Recurrent-pool network consists of (recurrent) GatedRecurrentLayer's, grouped with pooling and linear layers, and element-wise output function (here Ramp),), in this example with 2 recurrent layers and one output linear layer.



Input Port

Image: image

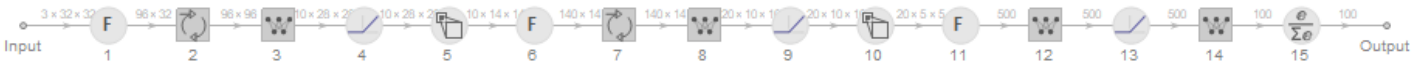
Output Ports

SubLabel: class

Label: class

1: NetChain

	Input	array (size: 3×32×32)
1	LinearLayer	array (size: 20×28×28)
2	Ramp	array (size: 20×28×28)
3	PoolingLayer	array (size: 20×14×14)
4	LinearLayer	array (size: 50×10×10)
5	Ramp	array (size: 50×10×10)
6	PoolingLayer	array (size: 50×5×5)
7	FlattenLayer	vector (size: 1250)
8	LinearLayer	vector (size: 500)
9	Ramp	vector (size: 500)
	Output	vector (size: 500)



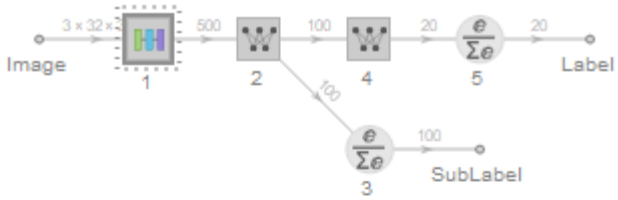
Input Port

Input: image

Output Port

Output: vector (size: 100)

Linear-pool network consists of linear layers, grouped with pooling layers, and element-wise output function (here Ramp), in this example with 2 linear layers and one output linear layer.



Input Port

Image:	image

Output Ports

SubLabel:	class
Label:	class

1: NetChain

	Input	array (size: 3×32×32)
1	LinearLayer	array (size: 20×28×28)
2	Ramp	array (size: 20×28×28)
3	PoolingLayer	array (size: 20×14×14)
4	LinearLayer	array (size: 50×10×10)
5	Ramp	array (size: 50×10×10)
6	PoolingLayer	array (size: 50×5×5)
7	FlattenLayer	vector (size: 1250)
8	LinearLayer	vector (size: 500)
9	Ramp	vector (size: 500)
	Output	vector (size: 500)

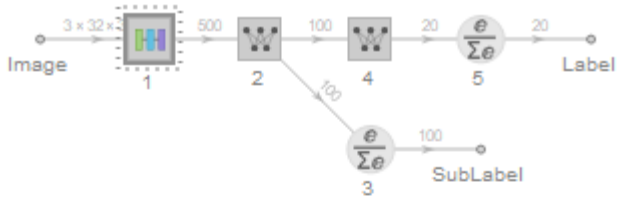
Input Port

Input:	image

Output Port

Output:	vector (size: 100)

Linear (simple) network consists of linear layers, and element-wise output function (here Ramp), in this example with 6 linear layers and one output linear layer.



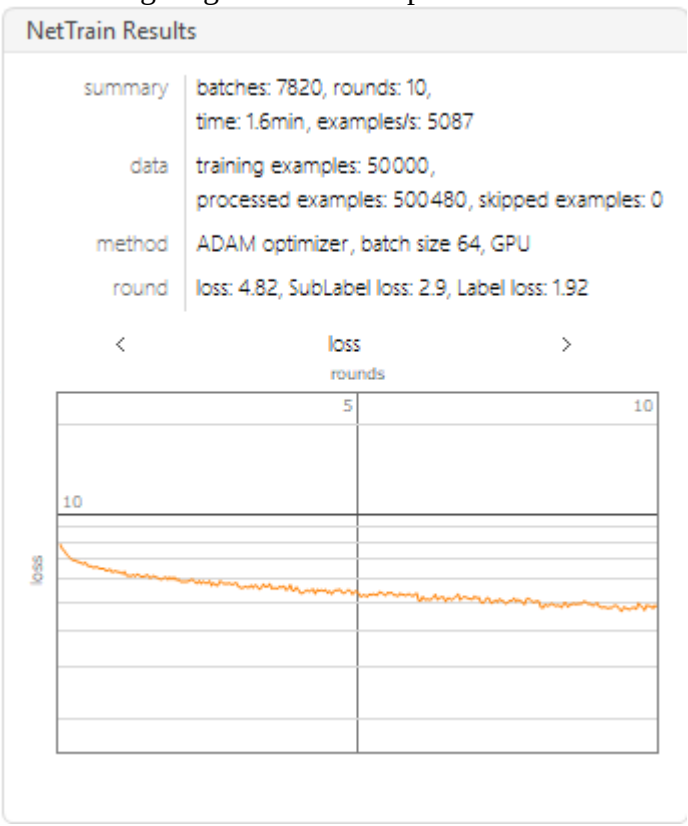
Input Port	
Image:	image
Output Ports	
SubLabel:	class
Label:	class

1: NetChain		
	Input	array (size: 3×32×32)
1	LinearLayer	array (size: 20×28×28)
2	Ramp	array (size: 20×28×28)
3	PoolingLayer	array (size: 20×14×14)
4	LinearLayer	array (size: 50×12×12)
5	Ramp	array (size: 50×12×12)
6	PoolingLayer	array (size: 50×6×6)
7	LinearLayer	array (size: 100×8×8)
8	Ramp	array (size: 100×8×8)
9	PoolingLayer	array (size: 100×4×4)
10	LinearLayer	array (size: 500×2×2)
11	Ramp	array (size: 500×2×2)
12	PoolingLayer	array (size: 500×1×1)
13	FlattenLayer	vector (size: 500)
14	LinearLayer	vector (size: 500)
15	Ramp	vector (size: 500)
	Output	vector (size: 500)

Input Port	
Input:	image
Output Port	
Output:	vector (size: 100)

Training and recognition rate

Now, 9 networks are trained on CIFAR-100 and their recognition rate on the same database is tested:
2-convolpool, 3-convolpool, 2-recurpool, 3-recurpool, 2-linpool, 3-linpool, 4-linpool, 6-linear, 7-linear .
The training diagramm for 2-linpool looks like that:



The training lasted 1.6 min on GPU's (and approximately 30 min on CPU), it ran in 10 iterations (rounds), and the final loss=error-vector-norm (sqrt(square sum)) was reduced from total 8. to total 4.82 , 2.9 for Label, 1.91 for SubLabel .
The recognition rate on the database was as follows:

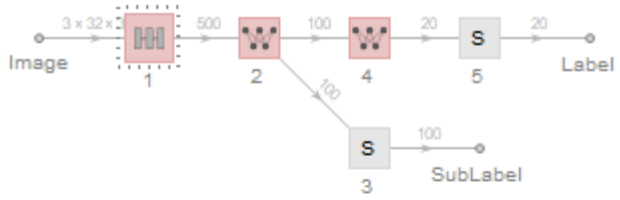
```
2-convol-pool {0.78594`,0.72230},
3-convol-pool {0.5465`,0.46768`},
2-recur-pool {0.198040,0.097740},
3-recur-pool {0.95524,0.8904},
2-lin-pool {0.4252,0.30342},
3-lin-pool {0.41282,0.28076},
4-lin {0.39066,0.25894},
6-lin {0.33308,0.19332},
7-lin {0.05,0.01}
```

One could assume that the recognition rate increases with the number of layers, but this is not the case in this application. From the results, we can draw the following conclusions:
-convolution nets have the best performance, and the optimal number of layers is 2
-recurrent nets have a sharp peak at 3 layers (afterward, the rate decreases), where the performance is excellent
-linear and lin-pool networks have the best performance at 2 layers, and the performance decreases slowly until it reaches the untrained-value at 7 layers: (1/20, 1/100) is the recognition expectation value without training.

Structure-gradient algorithm

[3] gradient-optimized network linnet3p
An interesting question is, whether one can change the structure, i.e. the dimensional node numbers of a layer, so that the total loss decreases, and eventually find the optimal structure in this sense.
We applied the gradient-method in respect to the dimensional node numbers, reducing the loss in each step. Basically, the method works, but progress is slow, and so is the change of the dimensional node numbers.
Here are the results of this structure-gradient algorithm for the net 3-lin-pool:

35



1: NetChain

	Input	array (size: 3×32×32)
1	LinearLayer	array (size: 20×28×28)
2	Ramp	array (size: 20×28×28)
3	PoolingLayer	array (size: 20×14×14)
4	LinearLayer	array (size: 50×12×12)
5	Ramp	array (size: 50×12×12)
6	PoolingLayer	array (size: 50×6×6)
7	LinearLayer	array (size: 100×10×10)
8	Ramp	array (size: 100×10×10)
9	PoolingLayer	array (size: 100×5×5)
10	FlattenLayer	vector (size: 2500)
11	LinearLayer	vector (size: 500)
12	Ramp	vector (size: 500)
	Output	vector (size: 500)

progress of the structure-gradient algorithm

```
linnet3p {0.41668,0.29884`}, {0.262`,0.1348}, loss 6.15,
step1 gr linnet3p {0.2858,0.1686}, {0.206,0.0884}, loss 6.46,
step1 mstep linnet3p {0.31636`,0.1901}, {0.2744`,0.1236}, loss 6.19,
step2 mstep linnet3p {0.4296,0.30628`}, {0.25`,0.1156}, loss 6.327,
step3 mstep linnet3p {0.43904`,0.31048`}, {0.2388`,0.1176}, loss 6.38,
step4 mstep linnet3p {0.41306`,0.2880}, {0.2924`,0.1552`}, loss 6.07,
step5 mstep gr=0 linnet3p {0.4430,0.32202`}, {0.3052,0.1560}, loss 6.13,
step6 mstep linnet3p {0.45218,0.33414`}, {0.2976`,0.1580}, loss 5.92,
step7 mstep  linnet3p {0.43326`,0.31248`}, {0.3084`,0.1589}, loss 6.09,
step8 mstep  linnet3p {0.43326`,0.31248`}, {0.2920,0.1416}, loss 6.05,
step9 mstep  linnet3p {0.43326`,0.31248`}, {0.2756,0.12840}, loss 6.26,
step10 mstep  linnet3p {0.42256,0.296520}, {0.242`,0.1416}, loss 6.18,
step11 mstep  linnet3p {0.41946,0.2925`}, {0.2772`,0.1216}, loss 6.26,
step12 mstep  linnet3p {0.43798,0.31884}, {0.2776,0.1548}, loss 6.19,
step13 mstep  linnet3p {0.39776`,0.2800}, {0.2748,0.1356}, loss 6.07,
```

and corresponding structures in the format
lin1, lin2, lin3, pool1, pool2, pool3 :

```
{{{{20,28,28},{50,12,12},{100,10,10}},{{2,2,2,2},{2,2,2,2},{2,2,2,2}}},{{{20,28,28},{50,12,12},{99,10,10}},{{2,2,2,2},{2,2,2,2},{2,2,2,2}}},
 {{{21,28,29},{50,12,12},{98,10,10}},{{2,2,2,2},{2,2,2,2},{2,2,2,2}}},{{{22,27,30},{51,13,13},{97,10,10}},{{2,2,2,2},{2,2,2,2},{2,2,2,2}}},
 {{{22,26,30},{51,13,13},{97,10,10}},{{2,2,2,2},{2,2,2,2},{2,2,2,2}}},{{{22,26,30},{51,13,13},{98,10,10}},{{2,2,2,2},{2,2,2,3},{2,2,2,2}}},
 {{{22,26,30},{51,13,13},{98,10,10}},{{2,2,2,2},{2,2,2,3},{2,2,2,2}}},{{{23,26,30},{51,13,13},{98,10,10}},{{2,3,2,2},{2,2,2,3},{2,2,2,2}}},
 {{{23,26,30},{51,12,13},{99,10,10}},{{2,3,2,2},{2,2,2,3},{3,2,2,2}}},{{{23,26,30},{51,12,13},{99,10,10}},{{2,3,2,2},{2,2,2,3},{3,2,2,2}}},
 {{{23,26,30},{51,12,13},{100,10,10}},{{2,3,2,3},{2,2,2,3},{3,2,2,2}}},{{{23,27,30},{51,12,13},{99,10,10}},{{3,3,2,3},{2,2,2,3},{3,2,2,2}}},
 {{{23,27,30},{51,13,13},{99,10,10}},{{3,3,2,3},{2,2,2,3},{3,2,2,2}}},{{{23,27,30},{51,13,13},{99,10,10}},{{4,2,2,3},{2,2,2,3},{2,2,2,2}}}}
```

There is a slow increase in dimensional node numbers in the first linear layer, elsewhere there is only fluctuation, the total loss was reduced from 6.46 to 6.07 in 13 steps, which is very inefficient.

3 Local backpropagation in serial FF networks

[3] manual backprop goal-output minimization linlayer-output parallel subclass linnet2 Ident, ytk4={0,1},adapted indiv. eta, symbolic derivative backprop parallel in datasets , 10 training-drecords

In this chapter, the subject is the implementation of the local backpropagation algorithm described in chapter 1.5.

The local backpropagation minimizes the output deviation $E_j = \frac{1}{2}\sum_k \left(\hat{y}_{j,k} - y_{j,k}\right)^2$ of the layer-j output $y_{j,i}$ from the target vector $\hat{y}_{j,i}$ by adaptation of the weights $w_{j,ki}$, but it does this for *every layer* , not only for the last, as the global propagation does. In order to do this, it back-propagates the target values $\hat{y}_{j,i}$, not only the weights, to the layer below. Therefore the local backpropagation is a truly layer-*iterative* algorithm, where each step uses only *local* data from the next higher layer, and not from the last layer, like the global propagation does. In addition, the ADAM variant is used for the weight correction Δw_{ki} , like in the case of global backpropagation in the preceding chapter.

The network and the variables

The initial network *netls2N0* is a *2-lin-pool* network with 2 linear-pool layers and the preliminary output linear layer (1) plus the sub-label linear layer (2) and the sub-label output function (3), the input is the pixel data array of the image *data*, and the final output is the sub-label index *is* , the label index of CIFAR-100 is omitted in this implementation. The network weights are initialized at start with gaussian random values in interval [-1, +1] .

Input Port

data:	array (size: 32×32×3)

Output Port

is:	vector (size: 100)

1: NetChain

	Input	array (size: 32×32×3)
1	LinearLayer	array (size: 20×28×28)
2	*x*	array (size: 20×28×28)
3	PoolingLayer	array (size: 20×14×14)
4	LinearLayer	array (size: 50×10×10)
5	*x*	array (size: 50×10×10)
6	PoolingLayer	array (size: 50×5×5)
7	FlattenLayer	vector (size: 1250)
8	LinearLayer	vector (size: 500)
9	*x*	vector (size: 500)
	Output	vector (size: 500)

The overall structure of the network *netls2N0* is

Input Port

Input:	array (size: 32×32×3)

Output Port

Output:	vector (size: 100)

with the linear layers (1), (4), (8), (10), pooling layers (3), (6), output functions identity (2), (5), (9), output function (11) is *prob(is)* , where is=1,…100 . The function gives ideally *prob=1* for the correct sub-label index and *prob=0* for the all others, in a trained network the "answer" of the network to the input *data* is the highest *prob* in the *Output*-vector.

We have 4 lin-layers here, so we have also 4 weight arrays with corresponding 4 bias-vectors and dimensions: *(wwlinlayer1, bwlinlayer1), dim(wwlinlayer1)={15680,3072}*

(wwlinlayer2, bwlinlayer2), dim(wwlinlayer2)= {5000,3920}
(wwlinlayer3, bwlinlayer3), dim(wwlinlayer3)= {500,1250}
(wwlinlayer4, bwlinlayer4), dim(wwlinlayer4)= {100,500}
where (wwlinlayer4, bwlinlayer4) corresponds to the last linear layer (10), and the target vector of the final vector *Output* is the final vector of the current training data-record *n*: *ytk4fA[n]* , where *ytk4fA[n, i]*=1 for *i=is[n]*, *ytk4fA[n, i]*=0 for other *i* (here *is[n]* is the sub-label index in the training data-record *n*, which is to be trained).
So in total *ytk4fA* is an array of dimension (*Ntrain* , 100), where *Ntrain* is the number of training data records. In the first run we set *Ntrain=10*: we use at first 10 data records for training, which is very small compared to the total of *Ntrain*(CIFAR)=50000.

The algorithm
The algorithm runs in 4 steps
-step1
(wwlinlayer4, bwlinlayer4) is modified by ADAM-gradient to minimize the deviation of *Output* from the target vector *ytk4fA[n]* for all data records *n=1…Ntrain*
The target vector *ytk3fA* for the lin-layer3 (8) is calculated ..
-step2
(wwlinlayer3, bwlinlayer3) is modified by ADAM-gradient to minimize the deviation of *output-function(9)* from the target vector *ytk3fA[n]* for all data records *n=1…Ntrain*
The target vector *ytk2fA* for the lin-layer2 (4) is calculated .
-step3
(wwlinlayer2, bwlinlayer2) is modified by ADAM-gradient to minimize the deviation of *output-function(5)* from the target vector *ytk2fA[n]* for all data records *n=1…Ntrain*
The target vector *ytk1fA* for the lin-layer1 (1) is calculated .
-step4
(wwlinlayer1, bwlinlayer1) is modified by ADAM-gradient to minimize the deviation of *output-function(2)* from the target vector *ytk1fA[n]* for all data records *n=1…Ntrain*
There is no target vector calculation, as lin-layer1 is the innermost layer.

Timing and memory
The algorithm is run with 12 kernels in parallel, the total time for 4 steps is about 8000s, and the memory requirement 23GB/kernel, because of the huge size of the weight arrays: *wwlinlayer2* in step3 has 19.6 10^6 elements, *wwlinlayer1* in step4 has 48.16 10^6 elements. The execution time increases linearly in *Ntrain*, but the increase in memory is a little below linear*: mem=190GB*/kernel for *Ntrain=100*.

The timing with 12 parallel kernels and memory is given in the following table

step	time (s)	memory (GB/kernel)
1	5, 2.4/loop	
2	47, 18/loop	
3	1700, 600/loop	
4	6300, 1450/loop	23

Results
The most important result is of course the *recognition rate* of the resulting network *netls2D0* on the trained dataset *trainingDatarD* and on whole database CIFAR
mtrained(*netls2D0, trainingDatarD*)= 0.9
mtrained(*netls2D0,CIFAR*)= 0.02
For comparison, we train the initial (random weights) network *netls2N0* with the normal (global) ADAM-backpropagation on the trained dataset to the network *trained*(*netls2N0, trainingDatarD*) and on CIFAR to the network *trained*(*netls2N0, CIFAR*) and for the original network *netls2N0* and measure the recognition rate
mtrained(*trained*(*netls2N0, trainingDatarD*), *trainingDatarD*)=0.4
mtrained(*trained*(*netls2N0, trainingDatarD*), *CIFAR*)=0.0111
mtrained(*trained*(*netls2N0, CIFAR*), *trainingDatarD*)=0.1
mtrained(*trained*(*netls2N0, CIFAR*), *CIFAR*)=0.11902

mtrained(*netls2N0, trainingDatarD*)=0.1
mtrained(*netls2N0, CIFAR*)=0.013

We have the following conclusions
-the recognition rate *mtrained* on the trained dataset is 0.9 for the local backprop, and only 0.4 for the global backprop, so the local backprop is **better by the factor 2.25** .
-on the whole database CIFAR the local backprop achieves *mtrained=0.02* and the global backprop achieves *mtrained=0.0111* , where the random rate is of course 1/100=0.01, so also here the local backprop is better by the factor almost 2.
-the CIFAR-trained global-prop network *trained*(*netls2N0, CIFAR*) achieves on CIFAR a moderate rate of mtrained=0.11902 , where the random rate is 1/10=0.1

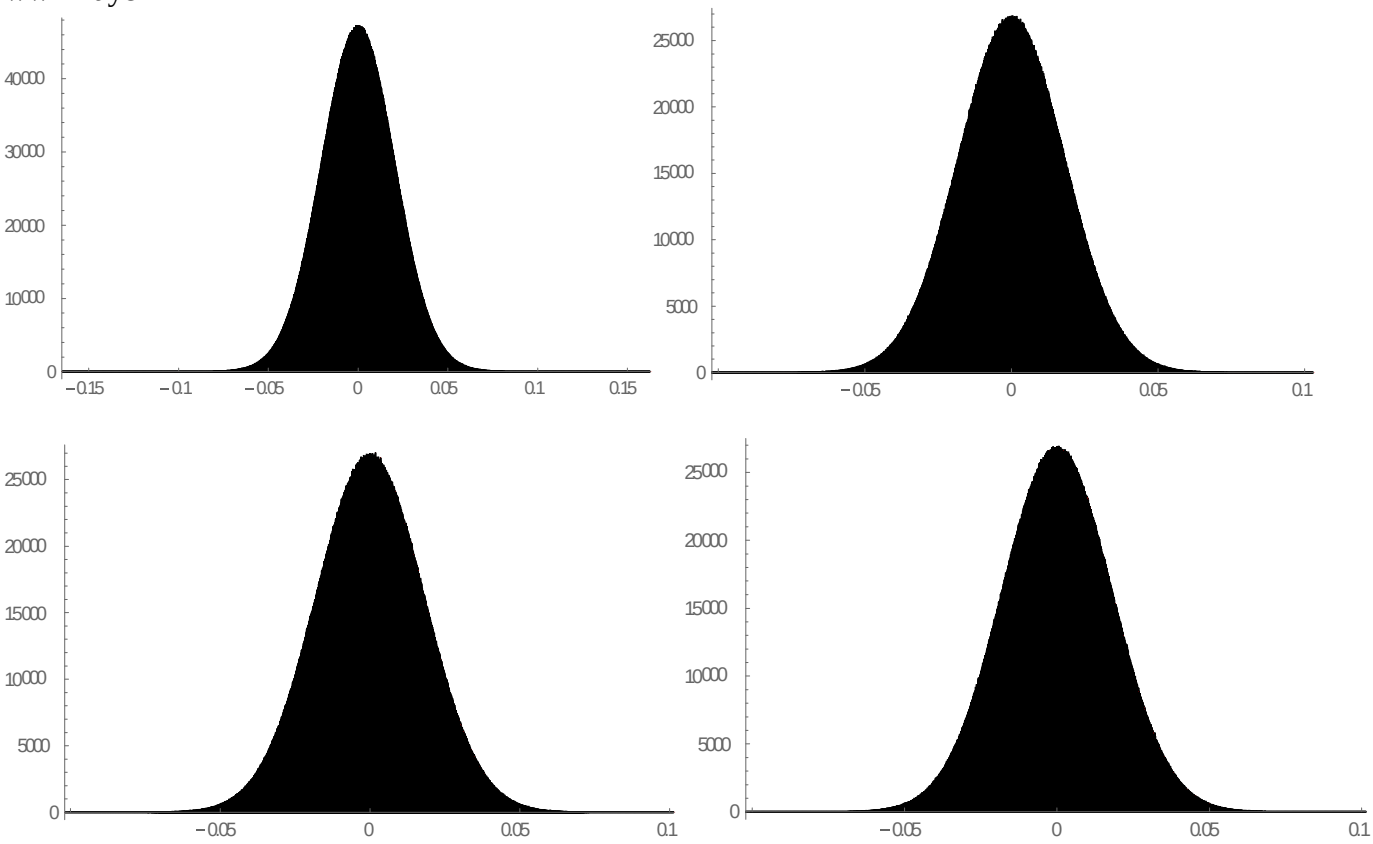
Values and histograms of the weights
For the local-backprop result network *netls2D0* we get the following values for the weights and the target vectors

name	max	min	mean	mean(abs)
wwlinlayer4	0.334528	-0.40843	-0.000285342	0.0404414
wwlinlayer3	0.148006	-0.135242	0.0000531141	0.0225776
wwlinlayer2	0.0866414	-0.0841181	$-3.65985*10^{-6}$	0.0127423
wwlinlayer1	0.101447	-0.101818	$1.97439*10^{-6}$	0.0143918
bwlinlayer4	2.00836	-0.17522	0.0862463	0.280228
bwlinlayer3	0.0703841	-0.085721	-0.00117297	0.0204053
bwlinlayer2	0.0645067	-0.0715514	-0.0000113144	0.0127455
bwlinlayer1	0.077166	-0.0706738	$-3.06135*10^{-6}$	0.0153956
ytk4fA	1.	0	0.01	0.01
ytk3fA	1.01385	-1.2154	-0.0136743	0.254063
ytk2fA	0.614329	-0.643174	-0.00230229	0.103551
ytk1fA	1.25602	-1.28643	0.00256237	0.193817

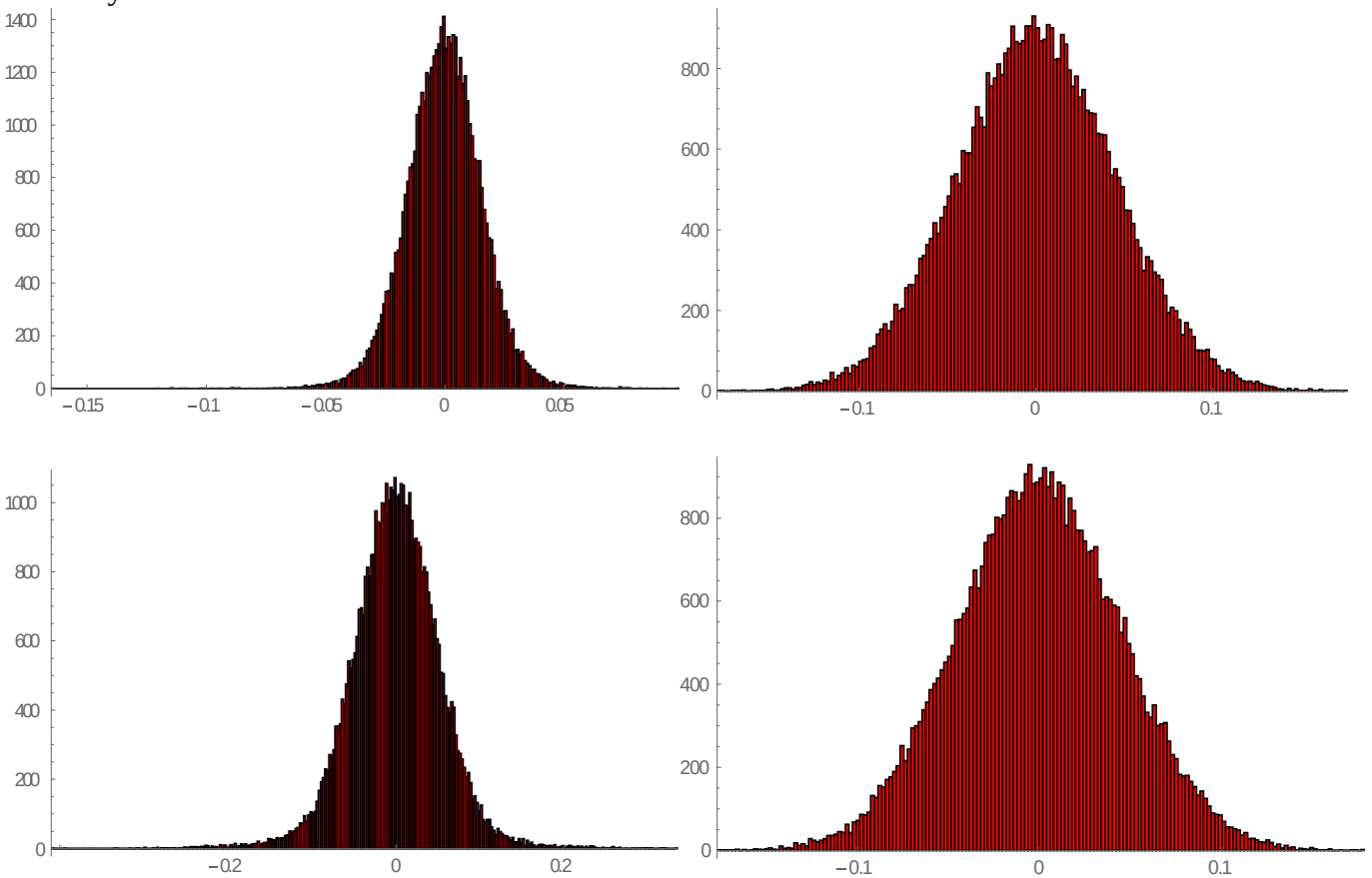
From this table result some interesting properties
-weights and biases 1…3 have values typically around +-0.01, for wwlinlayer4 it is +-0.04, for bwlinlayer4 it is +-0.3
-target vectors 1…3 value range is 0-symmetric, it is approximately [-1,1] for target vector 1 and 3, it is approximately [-0.6,0.6] for target vector 2

Finally, we can compare value histograms of the weights for the networks
trained(*netls2N0, CIFAR*) , *trained*(*netls2N0, trainingDatarD*) , *netls2D0* , *netls2N0*

wwlinlayer1



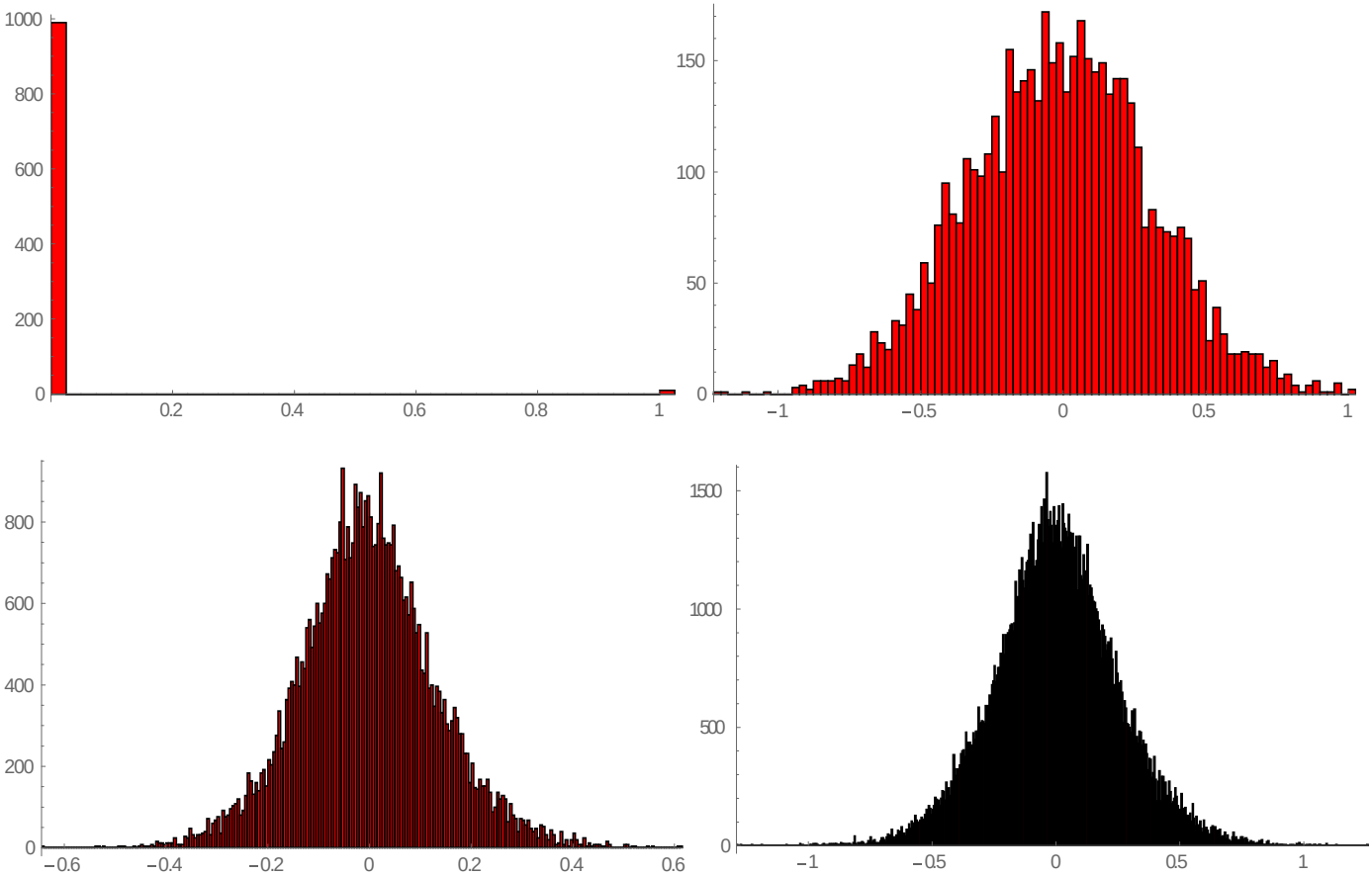
wwlinlayer4



We see that
-the value distribution for the CIFAR-trained network is much more “slim” than the others: the training concentrates the weights more around zero

-for the wwlinlayer4 we observe the same for the second diagram glob-backprop *trained*(*netls2N0, trainingDatarD*) and loc-backprop trained *netls2D0* : the latter is “slimmer” , apparently better trained.

The comparison of the histograms for the target vectors *ytk4fA, ytk3fA, ytk2fA, ytk1fA* of the local-backprop result network *netls2D0* is:



Here we can see, as mentioned above, that the target vector values are 0-symmetric. Furthermore, the distribution for target2 and target1 is much “slimmer” than for target3. The target4 (outermost) values are 0 and 1, so the histogram is trivial.

Results for different sizes of training data sets

For Ntrain=10 we have the recognition rates stated above
mtrained(*netls2D0, trainingDatarD*)= 0.9
mtrained(*netls2D0,CIFAR*)= 0.02
mtrained(*trained*(*netls2N0, trainingDatarD*), *trainingDatarD*)=0.4
mtrained(*trained*(*netls2N0, trainingDatarD*), *CIFAR*)=0.0111
mtrained(*trained*(*netls2N0, CIFAR*), *trainingDatarD*)=0.1
mtrained(*trained*(*netls2N0, CIFAR*), *CIFAR*)=0.11902
mtrained(*netls2N0, trainingDatarD*)=0.1
mtrained(*netls2N0, CIFAR*)=0.013

For Ntrain=50 we get
mtrained(*netls2D0, trainingDatarD*)= 0.18
mtrained(*netls2D0,CIFAR*)= 0.01526
mtrained(*trained*(*netls2N0, trainingDatarD*), *trainingDatarD*)=0.06
mtrained(*trained*(*netls2N0, trainingDatarD*), *CIFAR*)=0.01
mtrained(*trained*(*netls2N0, CIFAR*), *trainingDatarD*)=0.14
mtrained(*trained*(*netls2N0, CIFAR*), *CIFAR*)=0.194
mtrained(*netls2N0, trainingDatarD*)=0.
mtrained(*netls2N0, CIFAR*)=0.0079

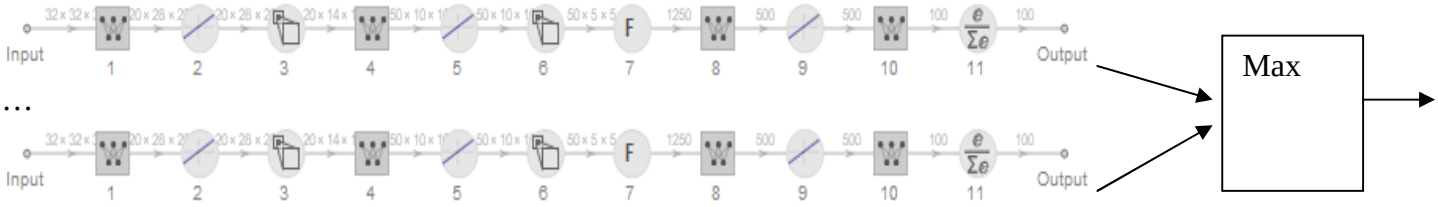
For Ntrain=100 we get
mtrained(*netls2D0, trainingDatarD*)= 0.07
mtrained(*netls2D0,CIFAR*)= 0.01206
mtrained(*trained*(*netls2N0, trainingDatarD*), *trainingDatarD*)=0.11
mtrained(*trained*(*netls2N0, trainingDatarD*), *CIFAR*)=0.0157
mtrained(*trained*(*netls2N0, CIFAR*), *trainingDatarD*)=0.13
mtrained(*trained*(*netls2N0, CIFAR*), *CIFAR*)=0.1308
mtrained(*netls2N0, trainingDatarD*)=0.1
mtrained(*netls2N0, CIFAR*)=0.00974

We conclude from these data that
-the results for local backprop deteriorate with larger training dataset *trainingDatarD*, it may be the consequence of decreasing convergence speed, as the target vector arrays become ever larger
-the rate for the CIFAR-trained glob-backprop network is approximately equal on CIFAR and training data set, as expected
-the rate for the CIFAR-trained glob-backprop network on CIFAR is around 0.13 , largely independent of the test dataset
-the rate for the glob-backprop network trained on *trainingDatarD* also decreases at first with the size of the training dataset

4 Local backpropagation in parallel FF networks

[3] manual backprop goal-output minimization linlayer-output parallel subclass linnet2 Ident, ytk4={0,1},adapted indiv. eta, symbolic derivative backprop parallel in datasets , parallel NN Ndata==5x(Ntrain==10 training-drecords)

Calculation of local backprop networks in Mathematica without GPU support is, even with parallel CPU kernels, time consuming, and, more importantly, has increasing memory requirements.
Therefore it is important to find a viable alternative, and such an alternative is to train an *array of serial networks* on small datasets, and to connect their *outputs in parallel*. This happens also in the brain during higher-level perception processing [15].
Based on this, we present here parallel network with 5 trained networks
parallel(*trained*(*netls2N0 , trainingDatarDA[i]*), *i=1…Ndata*) ,
where *Ndata*=5 , so we have in total a network trained on 5x10=50 datasets , combined from 5 networks connected with Maximum-function:



Timing and memory

The algorithm is run with 12 kernels in parallel, the total time for 1 dataset is about 800s, for 5 datasets time=4100s, and the memory requirement 12GB/kernel.

Results

The recognition rate of the resulting **parallel network** *netls2D0* on the trained dataset *trainingDatarD* and on whole database CIFAR is
mtrained(*netls2D0, trainingDatarD*)= 0.2
mtrained(*netls2D0,CIFAR*)= 0.03
The recognition rate of the global-backprop-trained network *trained*(*netls2N0, trainingDatarD*) and the network *trained*(*netls2N0, CIFAR*) , and for the original network *netls2N0* is
mtrained(*trained*(*netls2N0, trainingDatarD*), *trainingDatarD*)=0.06
mtrained(*trained*(*netls2N0, trainingDatarD*), *CIFAR*)=0.012
mtrained(*trained*(*netls2N0, CIFAR*), *trainingDatarD*)=0.08
mtrained(*trained*(*netls2N0, CIFAR*), *CIFAR*)=0.104
mtrained(*netls2N0, trainingDatarD*)=0.02
mtrained(*netls2N0, CIFAR*)=0.018
whereas the random recognition rate on 50 data records is 1/50=0.02

in comparison, with **serial network** on *Ntrain=50* we get (see above)
mtrained(*netls2D0, trainingDatarD*)= 0.18
mtrained(*netls2D0,CIFAR*)= 0.01526
mtrained(*trained*(*netls2N0, trainingDatarD*), *trainingDatarD*)=0.06
mtrained(*trained*(*netls2N0, trainingDatarD*), *CIFAR*)=0.01
mtrained(*trained*(*netls2N0, CIFAR*), *trainingDatarD*)=0.14
mtrained(*trained*(*netls2N0, CIFAR*), *CIFAR*)=0.194
mtrained(*netls2N0, trainingDatarD*)=0.
mtrained(*netls2N0, CIFAR*)=0.0079

We have the following conclusions
-the recognition rate *mtrained* on the trained dataset is 0.2 for the local backprop, and 0.06 for the global backprop, so the local backprop is better by the factor 3.3
-the recognition rate *mtrained* for the serial network with 50 data records are very similar:
on the trained dataset it is 0.18 for the local backprop, and 0.06 for the global backprop

-on the whole database CIFAR the local backprop achieves *mtrained=0.03* and the global backprop achieves *mtrained=0.02*
-the CIFAR-trained global-prop network *trained*(*netls2N0, CIFAR*) achieves on CIFAR mtrained=0.104 and on *trainingDatarD* it achieves 0.08, which is better than *netls2D0* by the factor 1.3

5 Evolving mutation cross-optimized networks
[3] mutation-cross-optimized evolving network classification 2-out CIFAR-100

We present here several scenarios of evolving population of *structure-changing* networks. The changes proceed by modification of (integer) layer *structure parameters*,
LinearLayer[{20,28,28}] has three output dimensions as 3 structure parameters: the output is 20x28x28 array
PoolingLayer[{2,2},{2,2}] has the kernel size 2x2 and the kernel offset 2x2 as 4 structure parameters
ConvolutionLayer[{20,5,5,0,0}] has the output dimension3 (= number of output matrices) and the kernel size 5x5 and kernel padding size 0x0 as 5 structure parameters

In every generation there is a fixed number of agents *Nagent=15* , from which the best *Nbest=5* are selected by recognition rate after training on CIFAR.
These best agent *mutate randomly* by ±1 or ±2 in structure parameters with the probability *Nmut=2* (from *Nagent*) and then enter the new generation by *genetic "crossing" pairwise* agent A[k] with agent A[k-1], A[k-2], …
(layer(A[k],1), layer(A[k-1],2), layer(A[k],3), layer(A[k-1],4),…)
The genetic crossing of 5 agents produces 10 "crossed" agents c(A[5],A[4]), c(A[5],A[3]), …, c(A[2],A[1])
In this way the new generation with again 15 agents is produced
G[n+1]={best agents= A[1],…A[5]} & {crossed agents= c(A[5],A[4]), c(A[5],A[3]), …, c(A[2],A[1]) }

Scenario1:
[3] scenario1: 9x3-lin-pool, 6x2-lin-pool, mut=2/15
Generation G1={9xLin2, 6xLin3}
Lin2 = 2 linear-pool layers
Lin3= 3 linear-pool layers
Mean recognition rates for the 2 layers are
```
mtrainedl0=0.037  mtrainedl1=0.036;
```
weighted mean wmean=0.0366
and the results for the first 5 generations:
```
sAm1trainedl={0.0338,0.034,0.036,0.0364,0.0366,0.0366,0.0366,0.0368,0.0368,0.1172,0.1342,0.1648,0.185,0.188,0.204}
sAm1trainedl={0.0356,0.0358,0.0364,0.0368,0.0368,0.0372,0.0372,0.0372,0.085,0.1228,0.176,0.1814,0.1836,0.1924,0.1992}
sAm1trainedl={0.0338,0.0338,0.034,0.0342,0.0342,0.0348,0.0358,0.0358,0.107,0.1174,0.165,0.1982,0.1988,0.209,0.2116}
sAm1trainedl={0.0348,0.0356,0.0356,0.0358,0.0358,0.0366,0.0376,0.0376,0.1732,0.1794,0.186,0.1992,0.2038,0.2042,0.2044}
sAm1trainedl={0.033,0.034,0.0342,0.0346,0.035,0.035,0.0356,0.0356,0.084,0.13,0.1682,0.1684,0.1732,0.1924,0.1952}
```
The maximum rate achieved is about 0.2, and the mean best rate (of the 5 best) is **stagnating** after the third generation (5 best ≈0.17), the mean rate has not increased significantly (mean(G1)= 0.0877867, mean(G5)= 0.09256)

Scenario2:
[3] scenario2: 9x3-lin-pool, 6x(2-lin-pool-pool), mut=2/15
Generation G1={3xLin3, 2xLin2poolpool}
Lin3=3 linear-pool layers
Lin2poolpool= 2 linear-pool-pool layers
Mean recognition rates for the 2 layers are
```
mtrainedl0=0.1318  mtrainedl2=0.0368;
```
weighted mean wmean= 0.0938
and the results for the first 5 generations

```
sAm1trainedl={0.034,0.035,0.036,0.0362,0.037,0.0414,0.1182,0.1206,0.1696,0.1748,0.1888,0.1904,0.2018}
sAm1trainedl={0.0324,0.0332,0.035,0.035,0.035,0.0362,0.1288,0.1354,0.1372,0.1398,0.1476,0.1494,0.1668,0.1988,0.2096}
sAm1trainedl={0.0346,0.0356,0.0362,0.0374,0.0962,0.1308,0.153,0.153,0.1532,0.1798,0.1824,0.1886,0.2108}
sAm1trainedl={0.0336,0.0342,0.035,0.036,0.1124,0.1352,0.1372,0.1462,0.1512,0.1882,0.1904,0.1944,0.199,0.2008,0.2302}
sAm1trainedl={0.0344,0.0358,0.0396,0.1558,0.172,0.1838,0.185,0.1854,0.189,0.1928,0.194,0.2002,0.2012,0.209,0.2206}
```

The maximum rate achieved is about 0.2, and the mean performance is **better** after 5 generations (11 best ≈0.18, mean(G1)= 0.106446, mean(G5)= 0.159907)

Scenario3:
[3] scenario3: 7x(3-lin-pool), 7x(2-lin-pool), 1x(2-conv-pool) , mut=2/15
Generation G1={7xLin3, 7xLin2, 1xConv2}
Lin2 = 2 linear-pool layers
Lin3= 3 linear-pool layers
Conv2= 2convolution-pool layers
Mean recognition rates for the 3 layers are

```
mtrainedl0=0.1318  mtrainedl1=0.036 mtrained3=0.7664;
```

weighted mean wmean=0.18779
and the results for the first 5 generations

```
sAm1trainedl={0.034,0.034,0.034,0.0342,0.0352,0.1082,0.2056,0.2164,0.2302,0.2314,0.4694,0.6746,0.9052,0.9088,0.9216}
sAm1trainedl={0.5308,0.5376,0.5622,0.6472,0.6734,0.6884,0.7112,0.7394,0.7492,0.901,0.9118,0.9184,0.9186,0.9188,0.9214}
sAm1trainedl={0.3972,0.6042,0.6492,0.6764,0.7216,0.724,0.7744,0.8434,0.8792,0.888,0.8902,0.895,0.9034,0.9114,0.9114}
sAm1trainedl={0.521,0.547,0.6246,0.6438,0.681,0.6848,0.709,0.7392,0.8726,0.89,0.9022,0.9134,0.9136,0.9314,0.9354}
sAm1trainedl={0.5996,0.638,0.6836,0.7142,0.724,0.8726,0.897,0.9078,0.9086,0.91,0.917,0.9184,0.9248,0.9276,0.9376}
```

The maximum rate achieved is about 0.9, and the mean performance is **much better** after 5 generations (mean(G1)= 0.336187, mean(G5)= 0.832053), the best 10 achieve about 0.9

Scenario4:
[3] scenario4: 15x(3-lin-pool)
Generation G1={15xLin3}
Lin3= 3 linear-pool layers
Mean recognition rates for the 1 layer are

```
mtrainedl0=0.1318;
```

and the results for the first 5 generations

```
sAm1trainedl={0.0376,0.0382,0.0382,0.0388,0.0388,0.114,0.1588,0.1804,0.2,0.218}
sAm1trainedl={0.0338,0.0344,0.0362,0.0362,0.0362,0.0754,0.081,0.1056,0.195,0.2014}
sAm1trainedl={0.0346,0.0348,0.0354,0.036,0.0362,0.0538,0.0558,0.1098,0.1706,0.1802}
sAm1trainedl={0.0352,0.0364,0.0372,0.0792,0.1074,0.1906}
sAm1trainedl={0.0376,0.038,0.038,0.038,0.0384,0.0812,0.083,0.0988,0.1358,0.14}
```

The final maximum rate achieved is about 0.1, and the mean performance has decreased (mean(G1)= 0.10628, mean(G5)= 0.07288), performance has **deteriorated** after 5 generations

We draw the following (preliminary) conclusions
-the final mean rate depends on the weighted mean (*wmean*) of the initial rates
-there is a threshold for *wmean*, where the mean rate increases until all agents have approximately the same, larger rate
-the threshold is approximately *wmean/wmin=3*, where *wmin*=minimum initial rate

References

[1] C. Aggarwal, Neural Networks and Deep Learning, Springer 2018
[2] J.A. Freeman, Simulating neural networks with Mathematica, Addison-Wesley 1994
[3] Mathematica-notebook NeuralNetworks.nb, www.janhelm-works.de
[4] R. Zemel et al., Neural networks, lecture University of Toronto, 2016
[5] S. Lucci, Artificial Neural Networks Lecture Notes, Brooklyn University, 2008
[6] www.geeksforgeeks.org
[7] S. Haykin, Neural networks and learning machines, Pearson Education, 2009
[8] M. Maynard, Neural networks, Maynard 2020
[9] J. Stone, Artificial Intelligence Engines, Sebtel Press, 2020
[10] Vitaly Bushaev, towardsdatascience.com, 10/2018
[11] D. P. Kingma, Jimmy Lei Ba., Adam : A method for stochastic optimization, arXiv:1412.6980, 2014
[12] CUDA C++ programing guide, Nvidia, 2021
[13] Getting started with CUDA, Nvidia, 2008
[14] CUDA programming within Mathematica, Wolfram, 2011
[15] M. Sigman S. Dehaene, Brain mechanisms of serial and parallel processing, J. of Neuroscience, 07/2008