Artificial Neural Networks *
- Part 2 -

- Multilayer networks
- Backpropagation
- Hidden layer representations

1 Multilayer Networks of Sigmoid Units

The decision surface learned by a single perceptron is **linear**:

$$\sum_{i=1}^{l} \vec{w} \cdot \vec{x}$$

Therefore if the data are **NOT** linearly separable, the training error and, of course the testing error are very much affected (likely to be rather large) if a single perceptron (regardless whether the perceptron rule or the stochastic gradient descent rule are used) is used to learn the decision surface.

It turns out that a **network** of perceptrons, each with a nonlinear threshold, in an architecture that we call **multilayered perceptron** extends the ability of the single perceptron from the linear decision surface to a nonlinear one.

Issues that must be solved in this approach

- Introduce a new node type, hidden nodes which reside in what is usually called *hidden layer*, as different from the input layer and *output layer*.
- Replace the discontinuous (step function) perceptron threshold by a continuous nonlinear one: sigmoid threshold

  $$o = \sigma(net) \equiv \frac{1}{1 + e^{-net}}$$

  illustrated in Figure 1.
- Derive a mechanism for adjusting weights by taking advantage of the differentiability of this function.
- Derive a mechanism for training the weights for
  1. output nodes
  2. hidden nodes

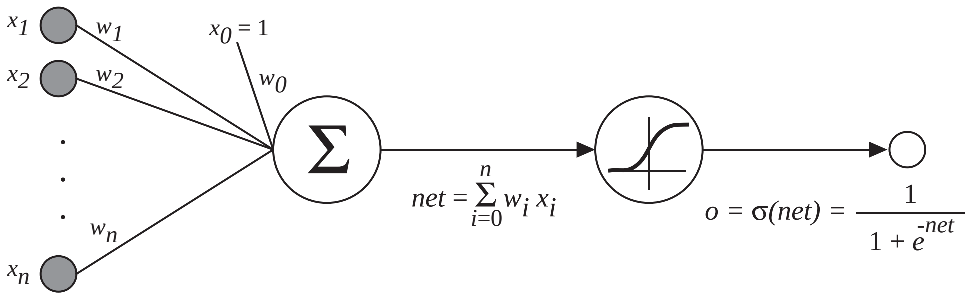


Figure 1: The sigmoid threshold unit

But first let us see the properties of the function $\sigma(t) = \frac{1}{1+e^{-t}}$:

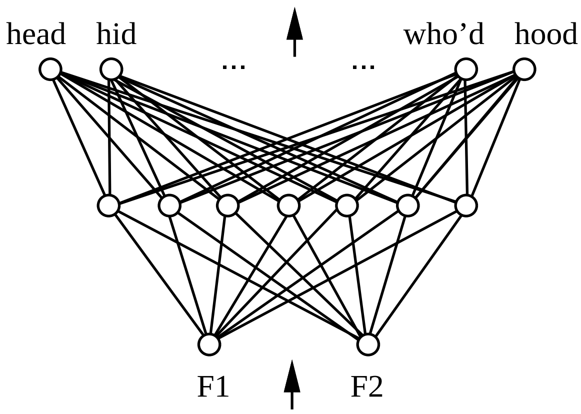
1. $\sigma : \Re \rightarrow [0, 1]$, that is, $\forall t \in \Re,\ 0 < \sigma(t) < 1$; even very large values of t are mapped into values in $(0, 1)$, hence the name of *squashing function*.

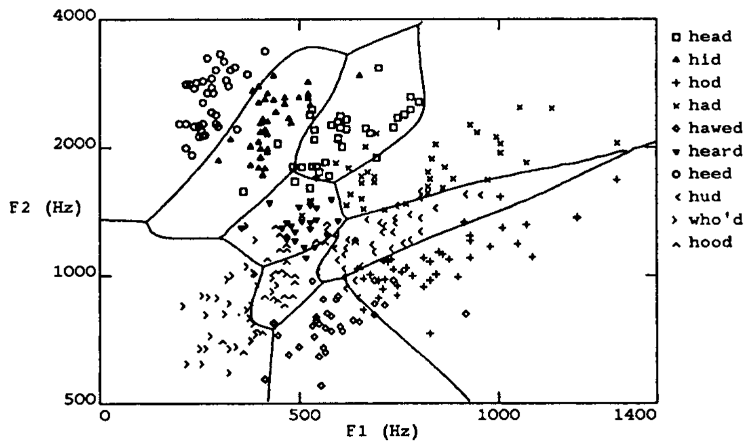
2. σ is monotonically increasing: $\sigma'(t) = \sigma(t)(1 - \sigma(t) > 0$ since $\sigma(t) \in (0, 1)$;

Remark 1 *Often the term* e^{-t} *is replaced by* e^{-kt} *for some constant* $k > 0$.

We can derive gradient descent rules to train:

- One sigmoid unit
- *Multilayer networks* of sigmoid units using a rule which is called *Backpropagation*





*based on Mitchell's notes

1.1 Error Gradient for a Sigmoid Unit

First, we set up, as before, the error function:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \tag{1}$$

where

- D denotes the set of examples, and d denotes a particular example from D;
- t_{kd} is the true (target) output value for the kth output for the example d;
- o_{kd} is the current network value for the kth output for the example d;

Note that E is a function of $\vec{w}$ because $o_{kd} = \sigma(\vec{w} \cdot \vec{x_{kd}})$.

We must alter our notation a little to convey the new network architecture, as follows:

- $i = 1, \ldots, N$, denotes a node in the network; N is the total number of nodes;
- For nodes $i \longrightarrow j$, x_{ji} denotes the input from unit i to unit j;
- likewise, w_{ji} is the weight from unit i to unit j;
- "overall input to j": $net_j = \sum_i w_{ji} x_{ji}$ the weighted sum of input to unit j;
- o_j is the network output for/by unit j;
- t_j is the target (true) output for unit j;
- σ the sigmoid threshold function;
- $Downstream(j) = \{u \mid u \textit{ is a unit in the network}, \textit{ such that } j \rightarrow u\}$
- $outputs$ denotes the set of nodes in the output layer.

Therefore $\vec{w}$ can be viewed as a 2-dimensional array.

The goal is to determine $\vec{w}$ so as to minimize $E(\vec{w})$.

We do this by applying the stochastic gradient descent algorithm to the nodes in this network. Recall that for a single perceptron, we minimized

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

in which case

$$\Delta\vec{w} = -\eta \nabla E(\vec{w})$$

or, by components,

$$\Delta w_i = -\eta \frac{\partial E_d}{\partial w_i}$$

Rewriting these for the multilayered case we obtain:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{2}$$

and

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \tag{3}$$

We need to derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ as follows:

$$
\begin{aligned}
\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \times \frac{\partial net_j}{\partial w_{ji}} (by\ the\ chain\ rule) && (4)\\
&= \frac{E_d}{net_j} \times \frac{\sum_i w_{ji} x_{ji}}{\partial w_{ji}} && (5)\\
&= \frac{E_d}{net_j} x_{ji} && (6)\\
& && (7)
\end{aligned}
$$

We need to derive $\frac{E_d}{net_j}$. To do this we consider separately two cases for the node j.

1. **Training Rule for training Output Units Weights: j is an output node**

   Note that

   - w_{ji} influences the (rest of) network only through net_j;
   - net_j influences the rest of the network only through the output corresponding to it, o_j.

   This means that by using the chain rule we can write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \times \frac{\partial o_j}{\partial net_j} \tag{8}$$

   Using equation (3) we have

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2 \tag{9}$$

$$(10)$$

$$= \frac{1}{2}(\frac{\partial(t_1 - o_1)^2)}{\partial o_j} + \cdots + \frac{\partial(t_j - o_j)^2}{\partial o_j} + \cdots)$$

$$(11)$$

$$= \frac{1}{2}(0 + \cdots + \mathbf{2}(\mathbf{t_j} - \mathbf{o_j})(-\mathbf{1}) + 0\cdots)$$

$$(12)$$

$$= -\frac{1}{2}2(t_j - o_j) = -(t_j - o_j)$$

$$(13)$$

   To derive $\frac{o_j}{net_j}$ we recall that $o_j = \sigma(net_j)$ and that $\sigma'(t) = \sigma(t)(1-\sigma(t))$. Therefore, we have:

$$\frac{\partial o_j}{\partial net_j} = \frac{\sigma(net_j)}{net_j} = o_j(1 - o_j) \tag{14}$$

   Collecting things together we obtain:

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)o_j(1 - o_j) \tag{15}$$

   We can now substitute this in the equation for δw_{ji} to obtain

$$\delta w_{ji} = \eta(t_j - o_j)o_j(1 - o_j)x_{ji} \tag{16}$$

   For simplicity denote

$$\delta_i = -\frac{\partial E_d}{\partial net_i} \tag{17}$$

2. **Training Rule for <u>Hidden</u> Output Units Weights: j is a hidden node**

   Must take into consideration the indirect ways in w_{ji} influences E_d. This is done by considering the nodes in $Downstream(j)$, and note that net_j influences E_d only through $Downstream(j)$. Then we have

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \times \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \times \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k w_{kj} o_j (1 - o_j) \qquad (18) \\
& \qquad (19)
\end{aligned}$$

   Therefore, using δ_j for the left hand-side of the above equation, we obtain

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

   and therefore,

$$\begin{aligned}
\Delta w_{ji} &= -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta \frac{\partial E_d}{\partial net_j} x_{ji} \\
&= \eta o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj} \qquad (20) \\
& \qquad (21)
\end{aligned}$$

We can now write the Backpropagation algorithm as follows:

1.2 Backpropagation Algorithm

Initialize all weights to small random numbers.

Until satisfied, Do

- For each training example, Do
  1. Input the training example to the network and compute the network outputs
  2. For each output unit k

     $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

  3. For each hidden unit h

     $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k} \delta_k$$

  4. Update each network weight $w_{i,j}$

     $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

     where

     $$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

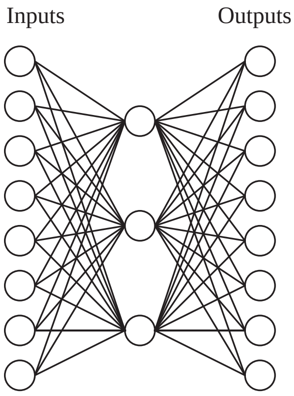
1.3 More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

2 Learning Hidden Layer Representations

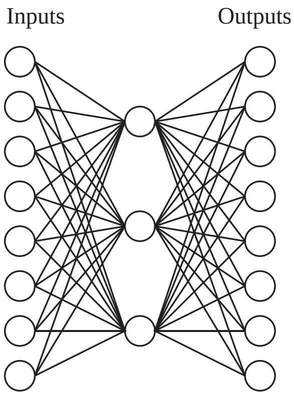


A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

A network,



Learned hidden layer representation:

Input			Hidden Values			Output
10000000	$\rightarrow$	.89	.04	.08	$\rightarrow$	10000000
01000000	$\rightarrow$	.01	.11	.88	$\rightarrow$	01000000
00100000	$\rightarrow$	.01	.97	.27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99	.97	.71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03	.05	.02	$\rightarrow$	00001000
00000100	$\rightarrow$	.22	.99	.99	$\rightarrow$	00000100
00000010	$\rightarrow$	.80	.01	.98	$\rightarrow$	00000010
00000001	$\rightarrow$	.60	.94	.01	$\rightarrow$	00000001

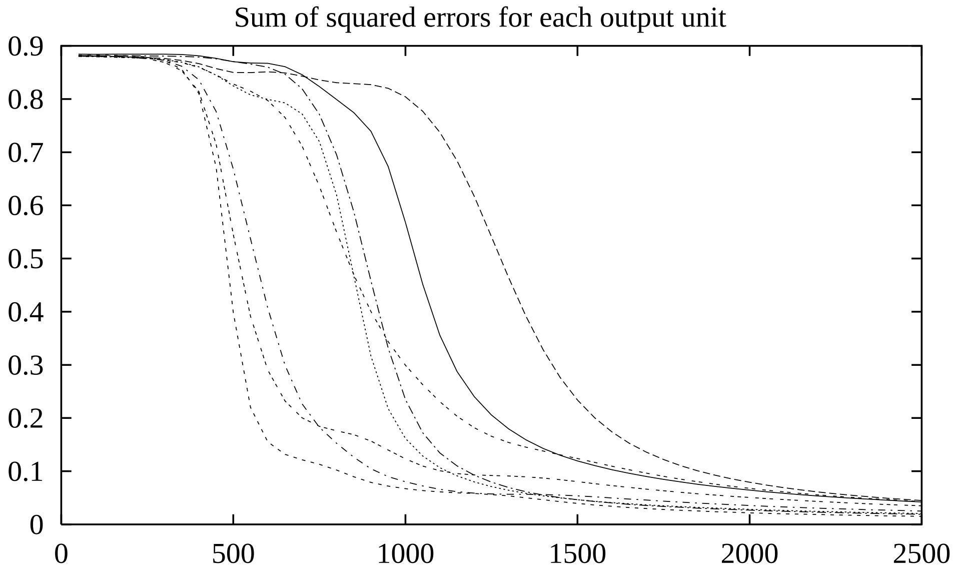


Figure 2: The $8 \times 3 \times 8$ network. Evolution of errors for each of the 8 output units, as the number of iterations increases

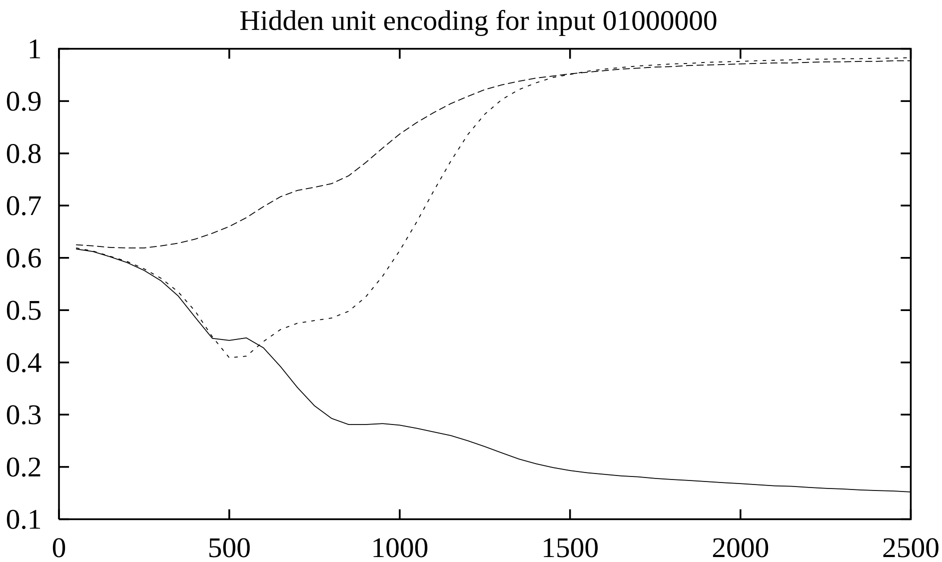


Figure 3: The $8 \times 3 \times 8$ network: the evolving hidden layer for the string 010000000, as the number of iterations increases

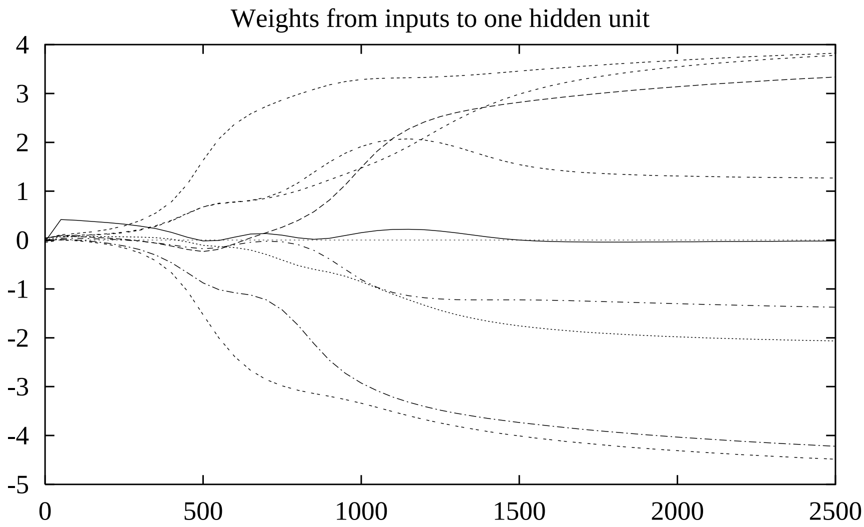


Figure 4: The $8 \times 3 \times 8$ network: the evolving weights for one the three hidden nodes, as the number of iterations increases

2.1 Convergence of Backpropagation

Gradient descent to some local minimum, iteratively reducing the error E between the training examples target value and the network output.

- Convergence is guarranteed to local minima only;
- Add momentum to the weight updating rule:

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{22}$$

  where α is called *momentum*; First term in the equation (22) is the weight-update rule in the BACKPROPAGATION; the second term of (22) guides the search on the error surface.

- Stochastic gradient descent can also be used to avoid local minima;
- Train multiple nets with different initial weights; use validation data sets to select among the networks obtained.

Nature of convergence

- Initialize weights near zero;
- Therefore, initial networks near-linear;
- Increasingly non-linear functions possible as training progresses;

2.2 Expressive Capabilities of ANNs

Boolean functions (but might require exponential (in number of inputs) hidden units):

- Every boolean function can be represented by network with single hidden layer; Sceme for representing arbitrary boolean functions:
  1. for each possibleinput vector create a hidden node that activates if and only if the input to the network is that vector;
  2. It follows the hidden layer will always have exactly one active node;
  3. Implement the output unit as an OR gate: will activate only for the desired input patterns.

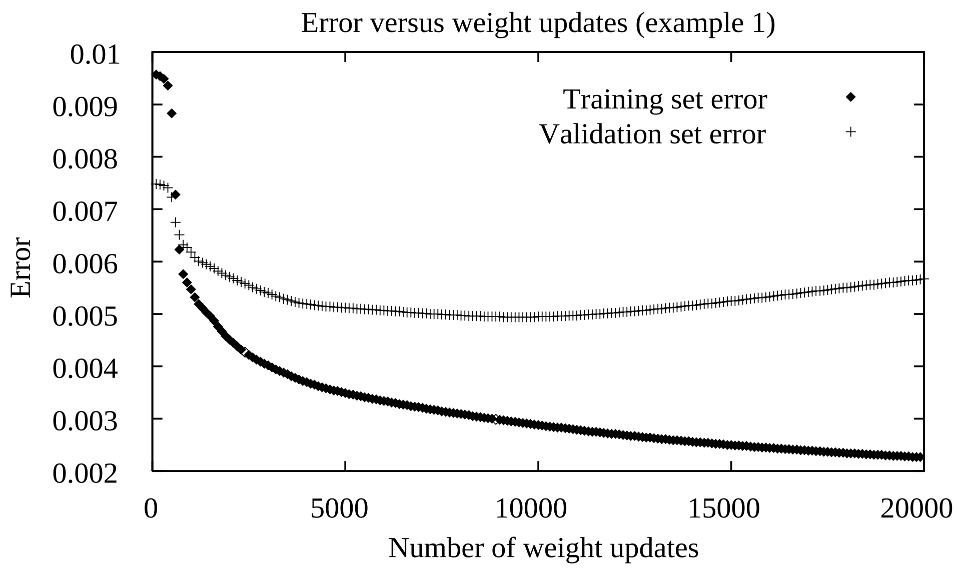


Figure 5: Error as function of number of weights updates

Continuous Functions

- Every **bounded continuous** function can be approximated with arbitrarily small error, by a network with:
  1. one hidden layer whose size depends on the function;
  2. sigmoid units at the hidden layer;
  3. un-thresholded linear units at the output layer.
- **Any function** can be approximated to arbitrary accuracy by a network with **three hidden** layers [Cybenko 1988].

2.3 The Hypothesis Search Space, Inductive Bias, Hidden Layer Represenatations

- The Hypothesis space:

$$H = \{\vec{w}; \vec{w} = (w_1, \ldots, w_n); w_i \in \Re\} \equiv \Re^n$$

  Hence H is continous (ID3 is discrete);

2.4 Overfitting in ANNs

- Search in BP is based on the differentiability of E (to calculate and search according to the gradient);
  1. Symbolic searches: general-to-specific order
  2. ID3 (C4.5): simple-to-complex
- The Inductive Bias:*smooth interpolation between <u>successive</u> data points*
- Hidden Layer Representations: these can emerge from weights setting whichminimize the error E.

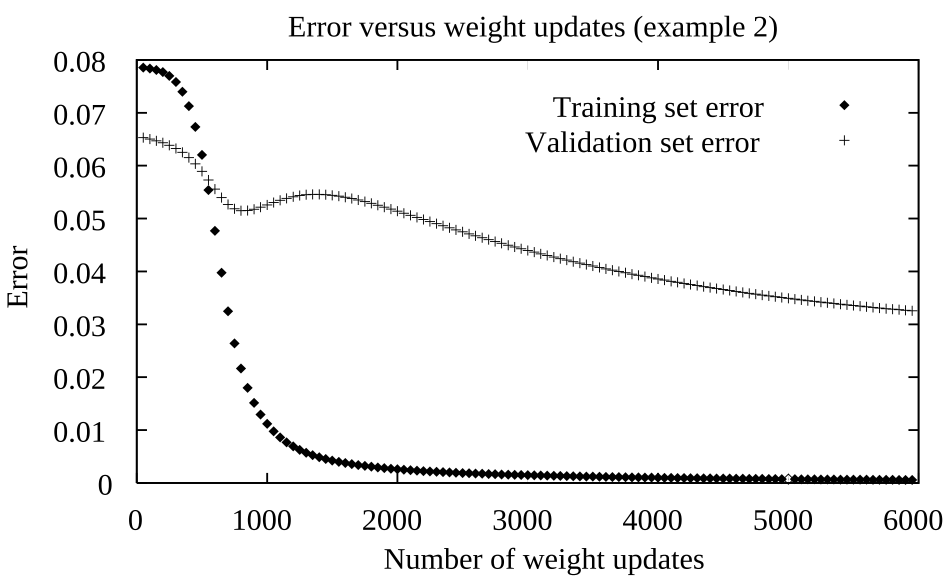


Figure 6: Error as function of number of weights updates (local min)