Lecture 7: Artificial Neural Networks (Part II)

- Gradient descent
- Sigmoid units
- Backpropagation

Linear units

Idea: consider just a **linear unit**:

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

The goal is to learn w_is that minimize the squared error

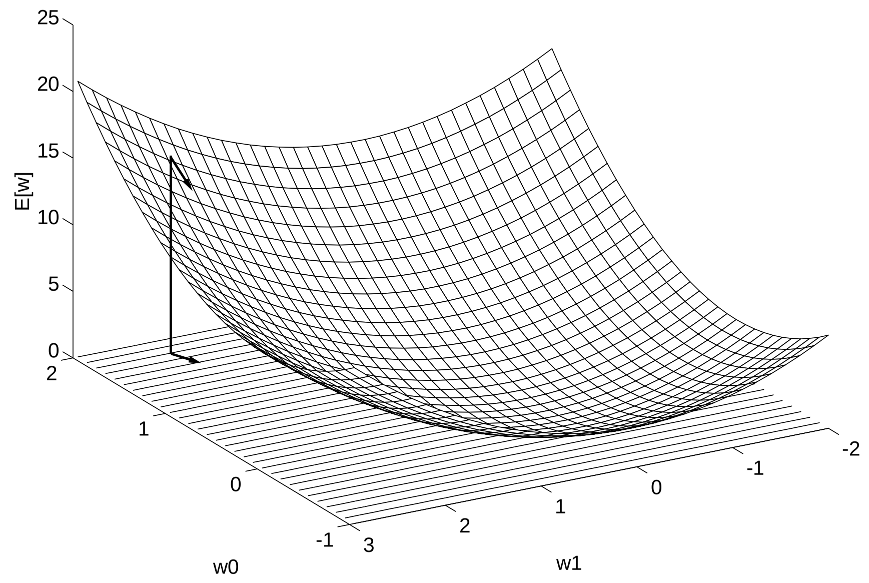
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D is set of training examples.

The function $E(\vec{w})$ defines an error surface in weight space.

Hill-climbing search for a good set of weights!

Gradient descent



The direction of the steepest descent is given by the **gradient** function: $\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n} \right]$

Training rule:

$$\Delta \vec{w} = -\alpha \nabla E[\vec{w}] \text{ i.e. } \Delta w_i = -\alpha \frac{\partial E}{\partial w_i}$$

Gradient descent for a linear unit

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

Gradient descent algorithm

```
1. Initialize each $w_i$ to some small random value
2. Until the termination condition is met, Do:
   (a) Initialize each $\Delta w_i$ to zero.
   (b) For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do:
       i. Input the instance $\vec{x}$ to the unit and compute the output $o$
       ii. For each linear unit weight $w_i$, Do
           $\Delta w_i \leftarrow \Delta w_i + \alpha(t - o)x_i$
   (c) For each linear unit weight $w_i$, Do:
       $w_i \leftarrow w_i + \Delta w_i$
```

Incremental (stochastic) gradient descent

Batch mode gradient descent: repeat until satisfied:

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \alpha \nabla E_D[\vec{w}]$

Incremental mode gradient descent: repeat until satisfied:

For each training example d in D

1. Compute the gradient $\nabla E_d[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \alpha \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental gradient descent can approximate *batch gradient descent* arbitrarily closely if α made small enough

Summary

Perceptron training rule guaranteed to succeed if:

- Training examples are linearly separable
- Sufficiently small learning rate α

Linear unit training rule uses gradient descent:

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate α
- Even when training data contains noise
- Even when training data not separable by H

The next step: increasing the expressivity of the representation!

Building networks of individual units

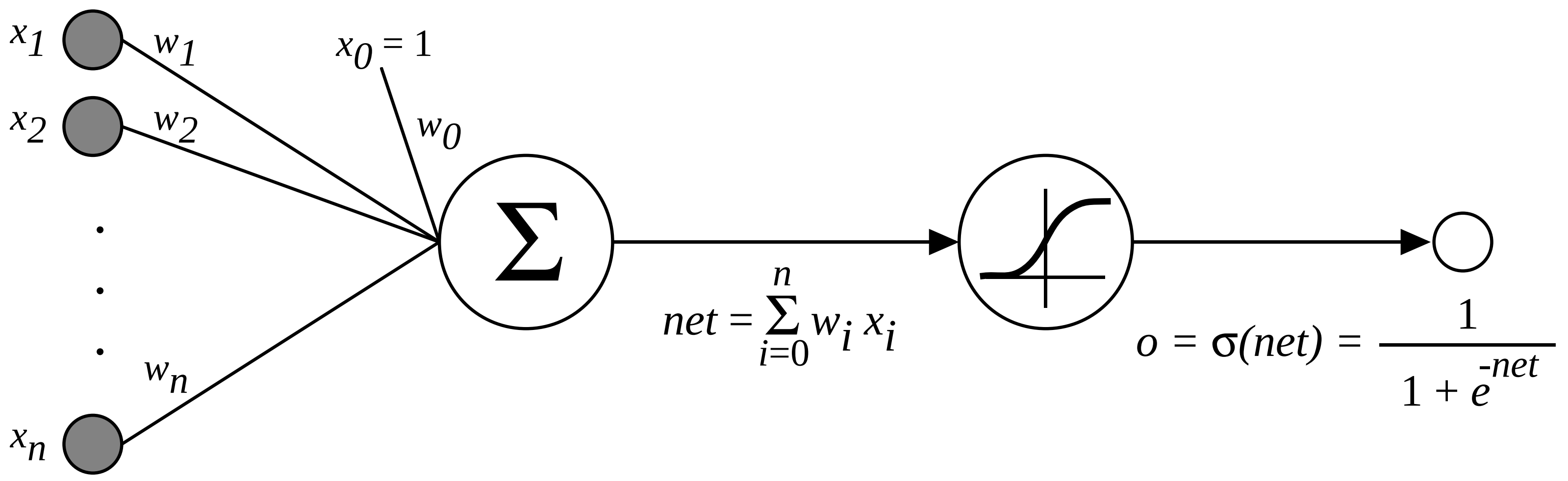
- Perceptrons have very simple decision surfaces

  If we connect them into networks, the error surface for the network is not differentiable (because of the hard threshold)

  So we cannot apply gradient descent to find a good set of weights...
- Networks of linear units are not satisfactory either (why?)
- *We would like a “soft” threshold!*

  Nicer math, and closer to biological neurons...

Sigmoid unit



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

$\sigma(x)$ is the sigmoid function: $\frac{1}{1+e^{-x}}$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient decent rules to train

- One sigmoid unit
- *Multi-layer networks* of sigmoid units $\rightarrow$ Backpropagation

Error gradient for a sigmoid unit

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\
&= -\sum_d (t_d - o_d)\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}, \text{ where} \\
net_d &= \sum_{i=0}^{n} w_i x_i
\end{aligned}
$$

Error gradient for a sigmoid unit (2)

But we know:

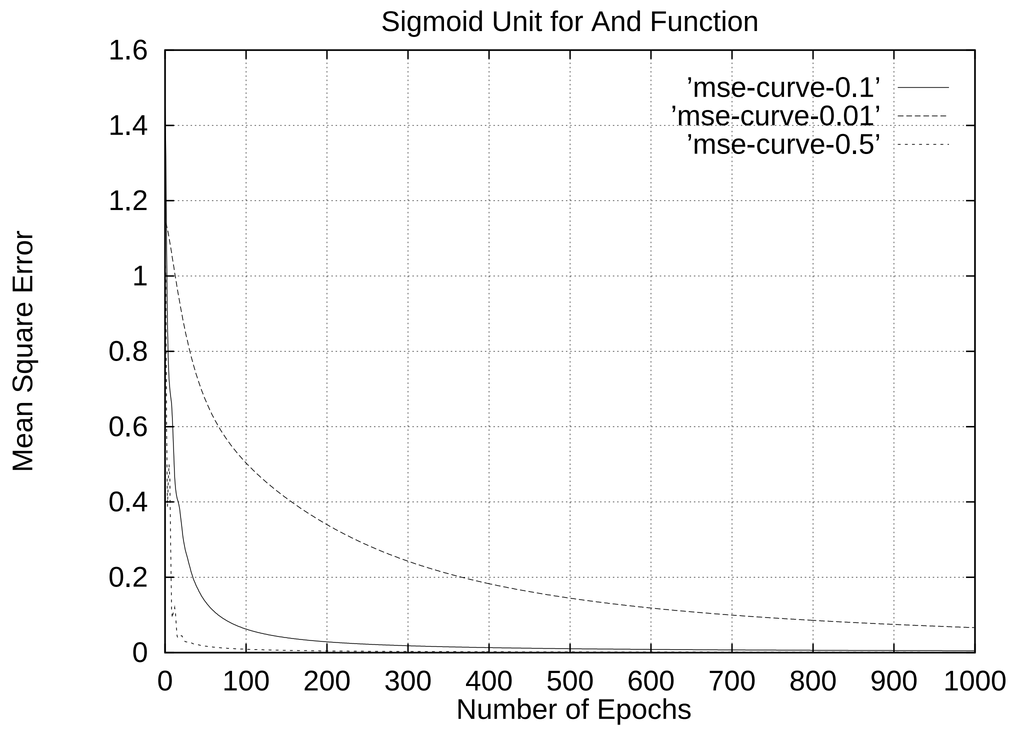
$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

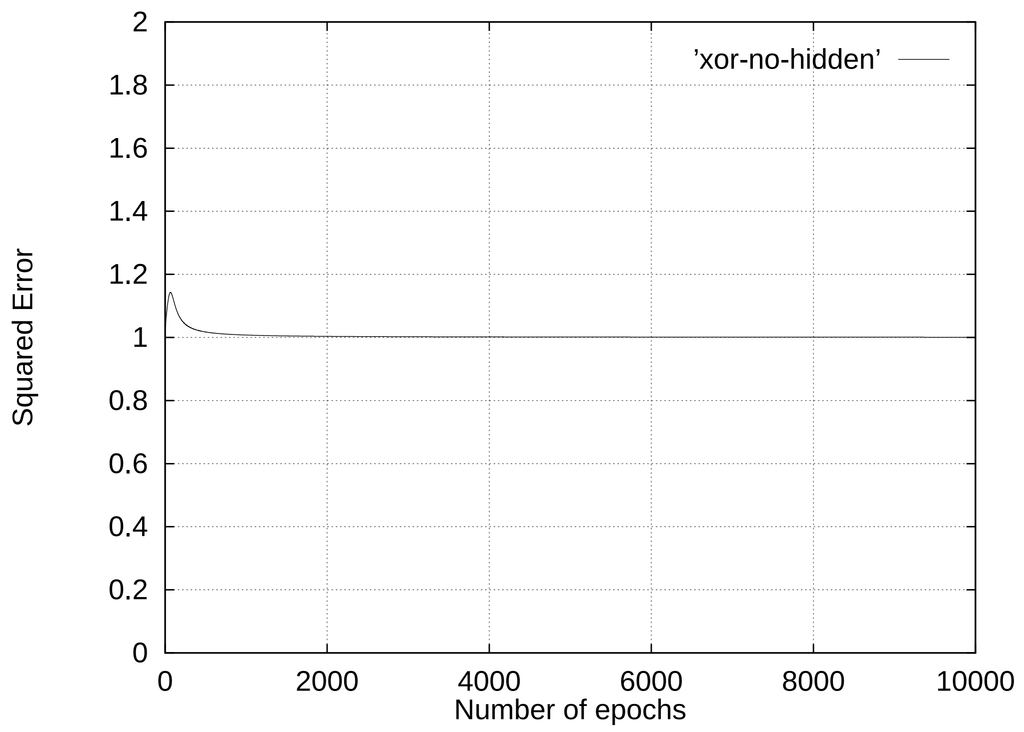
So:

$$\frac{\partial E}{\partial w_i} \quad = \quad -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

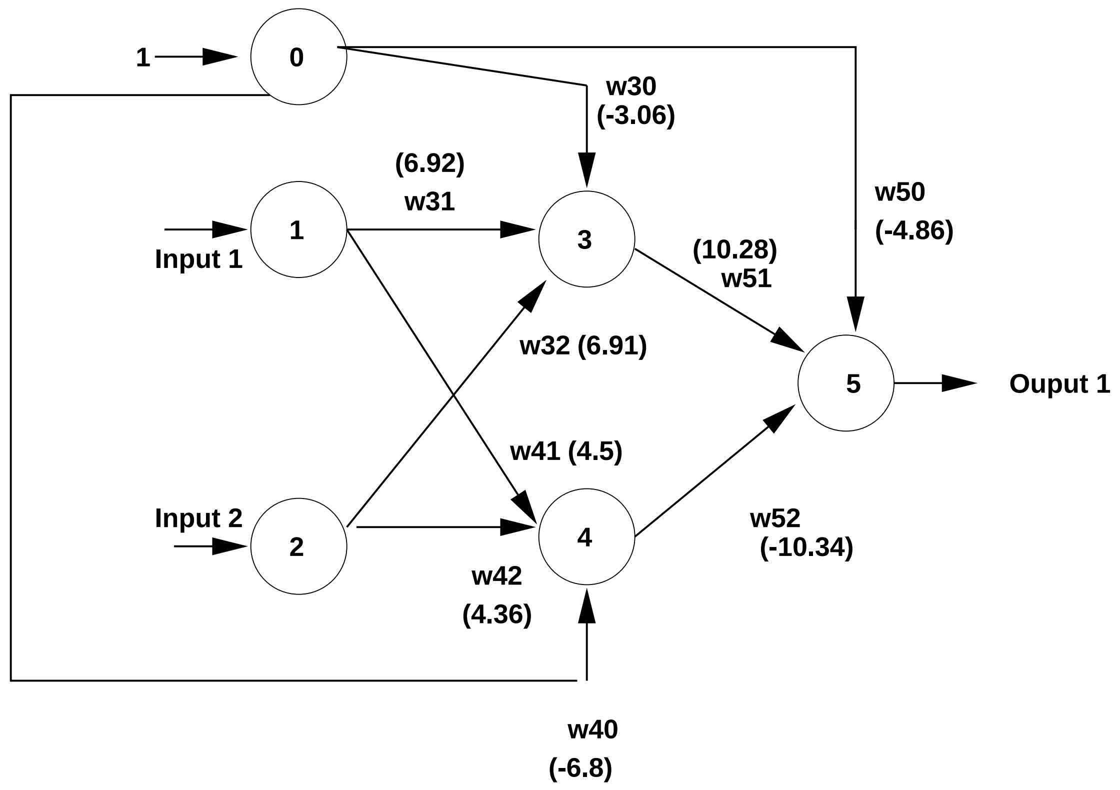
Learning curves for AND function



A single sigmoid unit cannot learn XOR!

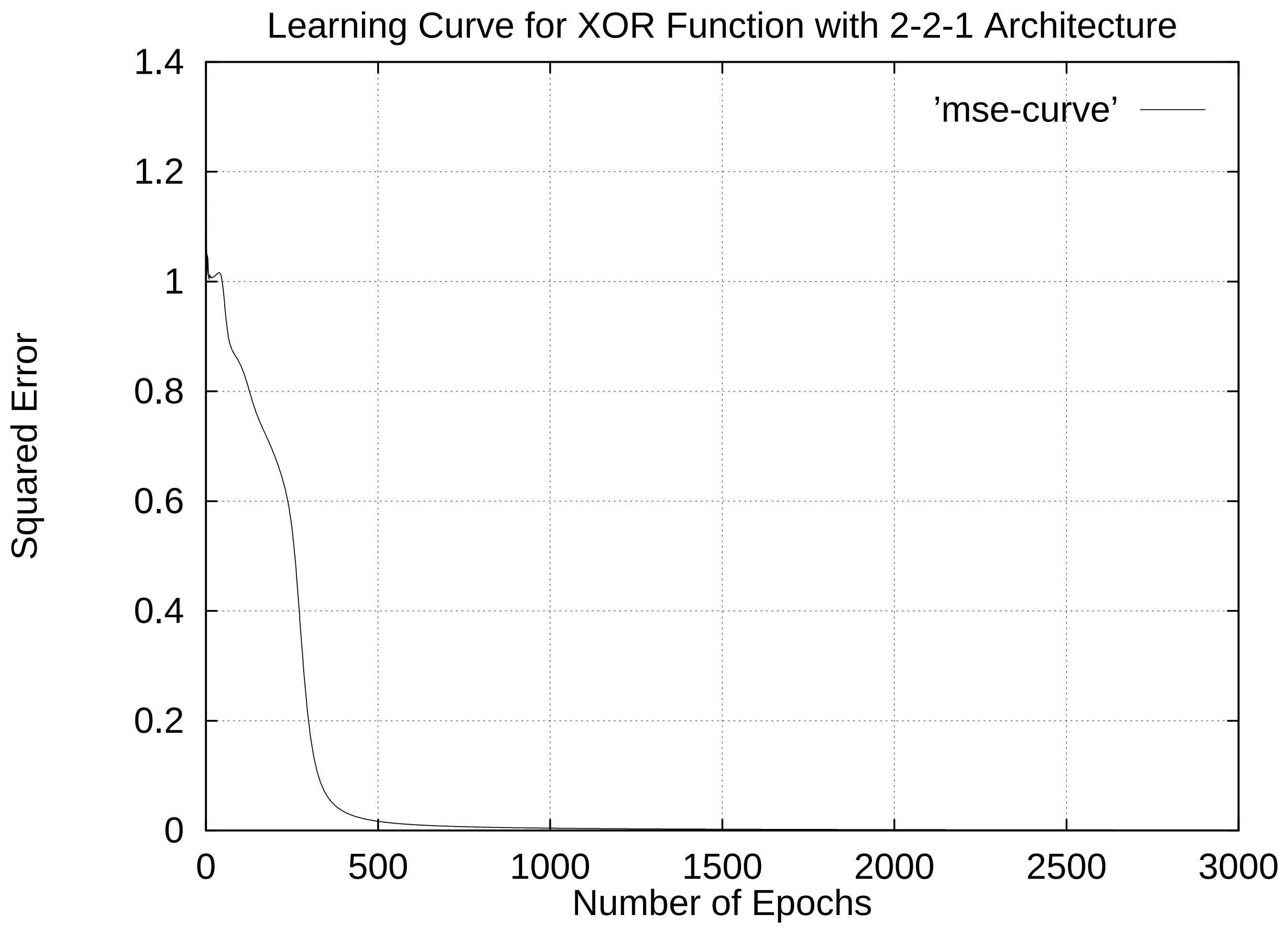


A network of sigmoid units can represent XOR...



Input1	Input2	o3	o4	Ouput 1
0	0	0.04	0.001	0.011
0	1	0.98	0.08	0.99
1	0			
1	1			

...And it can learn XOR too!



Backpropagation algorithm

1. Initialize all weights to small random numbers.
2. Repeat until satisfied:
   (a) Pick a training example
   (b) Input the training example to the network and compute the network outputs
   (c) For each output unit k

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

   (d) For each hidden unit h

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

(e) Update each network weight w_{ij}

$$w_{ij} \leftarrow w_{ij} + \eta \delta_j x_{ij}$$

x_{ij} is the input from unit i into unit j (so for the output neurons, the x's are the signals received from the hidden layer neurons)

This algorithm is the *incremental* version.

Alternatively, we can do a *batch version*: cycle through the training data, accumulate the weight changes for all instances, then change the weights.

Terminology: *epoch* = one pass through all the training instances

Why this algorithm?

For the output units, this is just like the update for a single neuron.

The only difference is that now the error function for the whole network is defined over all the outputs:

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

where t_{kd} and o_{kd} are the target and output values associated with the kth output unit and dth training example.

For the hidden units, we have to compute how much they influence the overall error.

But they only influence the error of the units immediately downstream from them!

The rest is a matter of applying the chain rule.

Convergence of backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Can be much worse than global minimum
- There can be MANY local minima (Auer et al, 1997)

Partial solution: train multiple nets with different initial weights

Restarting is a standard trick in hill-climbing algorithms

More tricks:

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses
- *Make sure the units start with different weights, to break symmetry!*

Expressiveness of feed-forward neural networks

- Every Boolean function can be represented by a network with single hidden layer, but might require exponential (in number of inputs) hidden units
- Every bounded continuous function can be approximated with arbitrarily small error, by a network with one, sufficiently large hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Inductive bias is roughly *smooth interpolation between points*

More on backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs (not only two layers)
- In theory it will find a local, not necessarily global error minimum, but in practice, it often works well (can run multiple times)
- Minimizes error over *training* examples

  Will it generalize well to subsequent examples?

  See the overfitting issue...
- Training can take thousands of iterations $\rightarrow$ VERY SLOW!

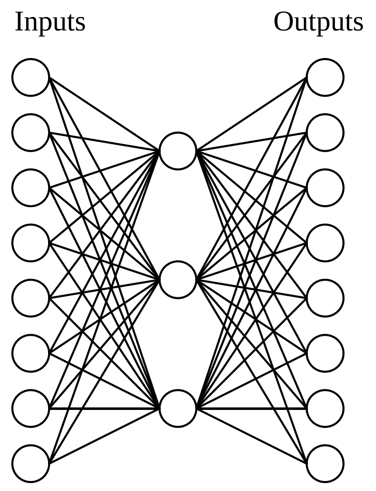
  But using network after training is very fast.

Example: Learning an encoder 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??

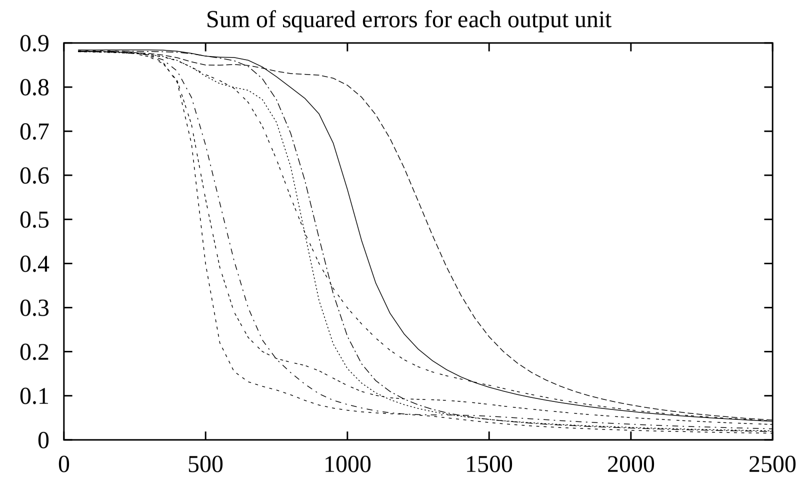
Learning hidden layer representations

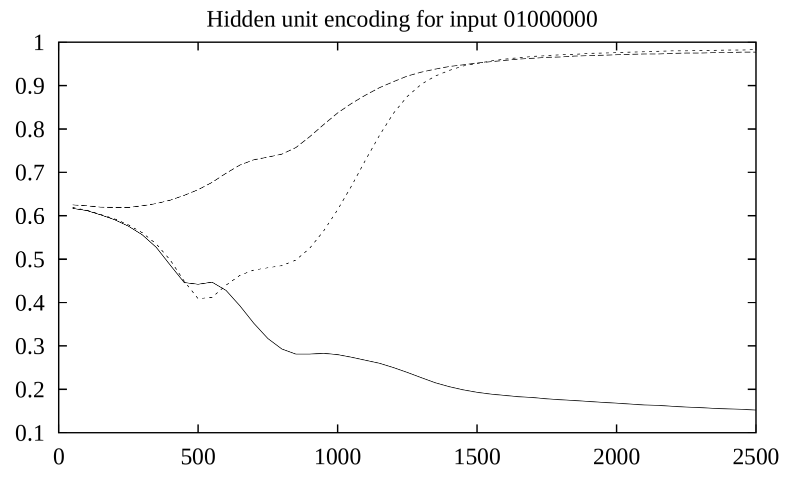


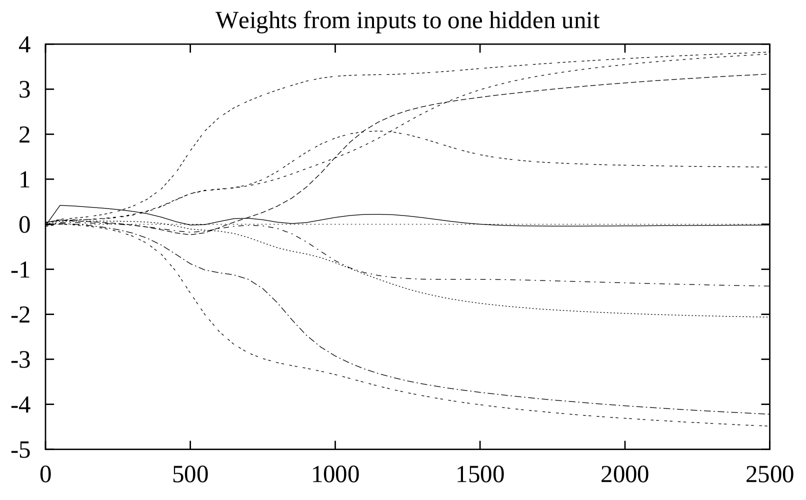
Learned hidden layer representation:

Input		Hidden Layer				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

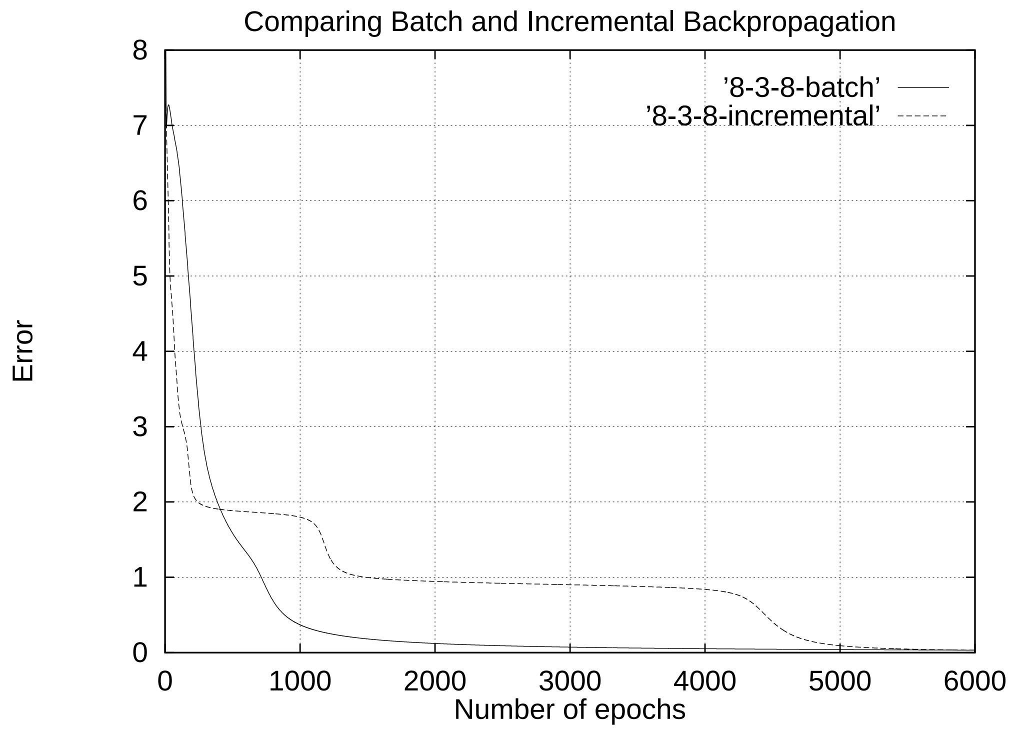
Evolution during training







Batch vs. incremental learning



Adding momentum

On the n-th training sample, instead of the update:

$$\Delta w_{ij} \leftarrow \eta \delta_j x_{ij}$$

we do:

$$\Delta w_{ij}(n) \leftarrow \eta \delta_j x_{ij} + \alpha \Delta w_{ij}(n-1)$$

The second term is called *momentum*

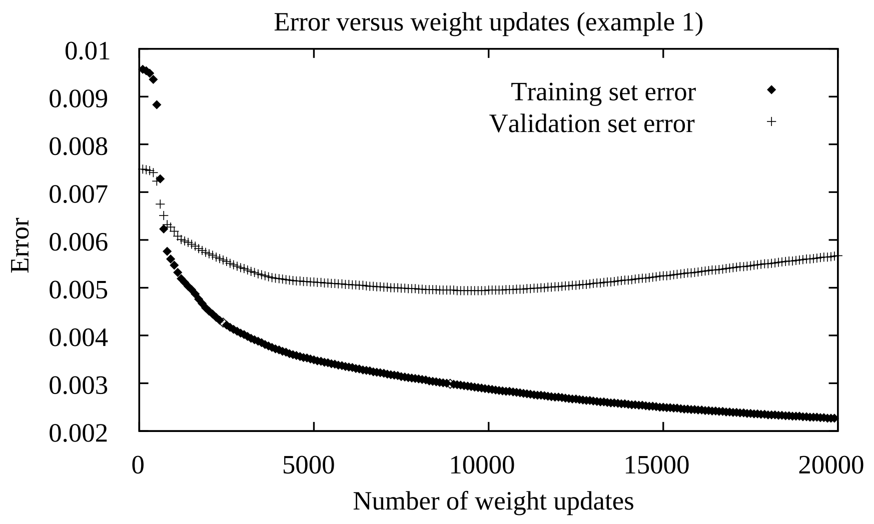
Advantages:

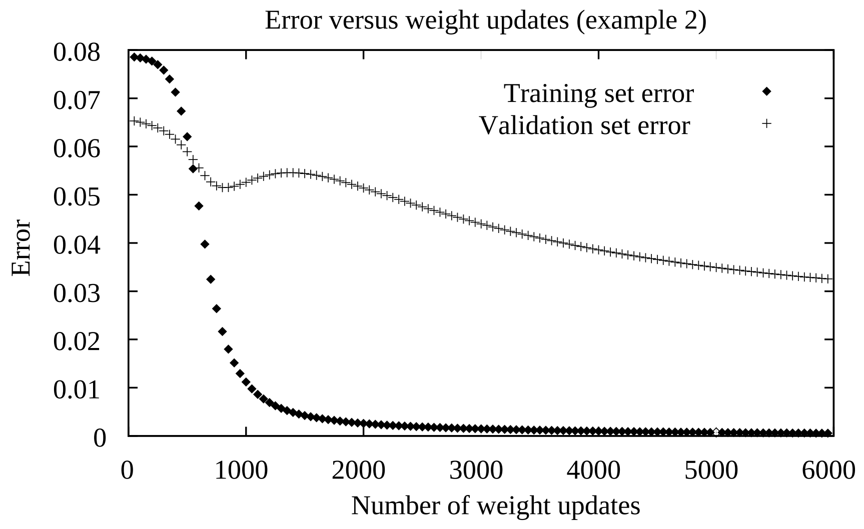
- Easy to pass small local minima
- Keeps the weights in areas where the error is flat
- Increases the speed where the gradient stays unchanged

Disadvantages:

- With too much momentum, it can get out of a global maximum!
- One more parameter to tune, and more chances of divergence

Overfitting in feed-forward networks





Use a validation set to decide when to stop training!

Practical issues

- The choice of initial weights has great impact on convergence! If the input size is N, and N is large, a good heuristic is to choose initial weights between $-1/N$ and $1/N$.
- Backpropagation is very sensitive to the size of the learning rate! If it is too large, the weights diverge.
- Sometimes it is appropriate to use different learning rates for different layers.
- The choice of input encoding and network topology can drastically affect learning!
  - It is bad to have inputs of very different magnitude
  - A thermometer encoding can be better than a 1-of-n
  - Too many hidden units hurt (why?)! Good heuristic: $\log(N)$.

Alternative error functions

Penalize large weights:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Used to avoid overfitting.

Train on target slopes as well as values:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Tie together weights: Train each weight individually, but then replace the values with the mean of the weights obtained by backprop.

Constructive methods for neural networks

Meiosis networks (Hanson):

- Start with just one hidden unit, train using backprop
- Compute the variance of each weight during training
- If a unit has one or more weights of high variance, it is split into two units, and the weights are perturbed

Cascade correlation (Fahlman & Lebiere):

- Start with outputs only and train using backprop
- Add a neuron connected to all inputs, and train it to correlate to the residual error
- Connect the neuron to the output node, then freeze its weights and train the output again
- Continue until the residual error is below a threshold

<u>When to consider using neural networks</u>

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued, or a *vector of values*
- Possibly noisy data
- Training time is unimportant
- Form of target function is unknown
- Human readability of result is unimportant
- The computation of the output based on the input has to be fast

Examples:

- Speech phoneme recognition [Waibel] and synthesis [Nettalk]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction