neural networks
tricks of the trade

Patrick van der Smagt

December 2011

summarising the FFNN

summary: for a mapping $\mathbb{R}^d \to \mathbb{R}$ we define a feed-forward network with λ layers of weights as

$$\begin{aligned}\mathcal{M}(\mathbf{w},\mathbf{x}) \quad = \quad & \sum_{i^{(\lambda)}} w^{(\lambda)}_{i^{(\lambda)}} \phi \left(\sum_{i^{(\lambda-1)}=1}^{N_{\lambda-1}} w^{(\lambda-1)}_{i^{(\lambda)},i^{(\lambda-1)}} \phi \left(\sum_{i^{(\lambda-2)}=1}^{N_{\lambda-2}} w^{(\lambda-2)}_{i^{(\lambda-1)},i^{(\lambda-2)}} \cdots \right.\right. \\ & \left.\left.\left. \cdots \ \phi \left(\sum_{i^{(1)}=1}^{N_1=d} w^{(1)}_{i^{(2)},i^{(1)}} x_{i^{(1)}} \right) \right) \right) \right)\end{aligned}$$

Often we only use one hidden layer with N_h units and a one linear output

$$\mathcal{M}(\mathbf{w},\mathbf{x}) = \sum_{j=1}^{N_h} w^{(2)}_j \phi \left(\sum_{i=1}^{d} w^{(1)}_{ji} x_i \right) \qquad (1)$$

Usually we take $\phi(x) \equiv \tanh(x)$ for numerical stability

summarising the FFNN

The output of an FFNN is a sum of many *nonlinear parameterised* basis functions ϕ.

Conversely, note that, when $\phi(x) = x$, $\mathcal{M}$ reduces to

$$\mathcal{M}(\mathbf{w}, \mathbf{x}) = \sum_{j=1}^{N_h} w_j^{(2)} \left(\sum_{i=1}^{d} w_{ji}^{(1)} x_i \right) = \sum_{j=1}^{N_h} \sum_{i=1}^{d} w_j^{(2)} w_{ji}^{(1)} x_i = \sum_{i=1}^{d} w_i^{(3)} x_i \tag{2}$$

where $w_i^{(3)} = \sum_{j=1}^{N_h} w_j^{(2)} w_{ji}^{(1)}$

this just defines a hyperplane in the input space!

summarising the FFNN

To find the weights $\mathbf{w}$ we need parameter search methods.
The likelihood of the prior leads to minimisation of

$$E(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{n}\left(\mathbf{z}_p - \mathcal{M}(\mathbf{w}, \mathbf{x}_p)\right)^2 \tag{3}$$

Minimisation of E is normally done by following the gradient $\partial E(\mathbf{w})/\partial w_{ij}$. Back-propagation computes these partial derivatives.

Note that we often write E rather than $E(\mathbf{w})$—but mean the same.

FFNN for classification

Take a set of classification data $\mathcal{C}_0 = \{(\mathbf{x}, z = 0)\}$ and $\mathcal{C}_1 = \{(\mathbf{x}, z = 1)\}$.

If we take an FFNN with sigmoidal outputs

$$y = \phi(a) = \frac{1}{1 + \exp(-a)}$$

we can interpret the output $y(\mathbf{x}, \mathbf{w})$ as the conditional probability $p(\mathcal{C}_0 \mid \mathbf{x})$ while $p(\mathcal{C}_1 \mid \mathbf{x}) = 1 - y(\mathbf{x}, \mathbf{w})$.

But then the likelihood is a Bernoulli distribution

$$p(t \mid \mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^z \left[1 - y(\mathbf{x}, \mathbf{w})\right]^{1-z} \tag{4}$$

The negative log likelihood then leads to

$$E(\mathbf{w}) = -\sum_{p=1}^{n} \{z_p \log y(\mathbf{x}_p, \mathbf{w}) + (1 - z_p) \log[1 - y(\mathbf{x}_p, \mathbf{w})]\} \tag{5}$$

Has been shown to lead to better results!

vanishing gradient

It is usually true that

$$\partial E(\mathbf{w})/\partial w_{ij}^{(\lambda)} \gg \partial E(\mathbf{w})/\partial w_{ij}^{(\lambda-1)} \tag{8}$$

i.e., the lower you get in the network, the more the gradient vanishes.

After all,

$$\frac{\partial E(\mathbf{w})}{\partial w_{ij}^{(\lambda-1)}} = \delta_j^{(\lambda-1)} x_i = \sum_l \underbrace{\delta_l^{(\lambda)}}_{\text{small}} \times \underbrace{w_{lk} x_i}_{\text{small}} = \text{smaller!} \tag{9}$$

departing from classical optimisation

Before we optimised w.r.t. E

$$E(\mathbf{w}) = \sum_{p=1}^{n} E_p(\mathbf{w}) = \sum_{p=1}^{n} \frac{1}{2} \left(\mathbf{z}_p - \mathcal{M}(\mathbf{w}, \mathbf{x}_p)\right)^2 \tag{10}$$

In "on-line" learning we optimise E_p for one particular sample $(\mathbf{x}_p, \mathbf{z}_p)$

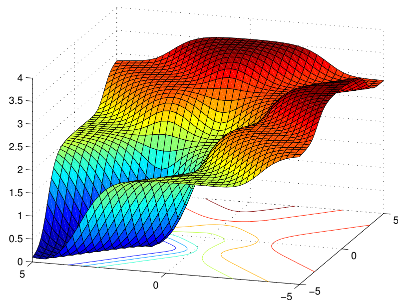
advantage:

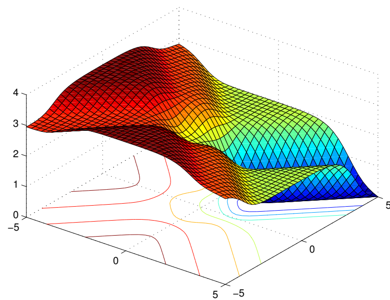
1. we add stochasticity, and
2. we can learn as our data comes in.

disadvantage:

We only have local information on our gradient, so global minimisation methods cannot be used.

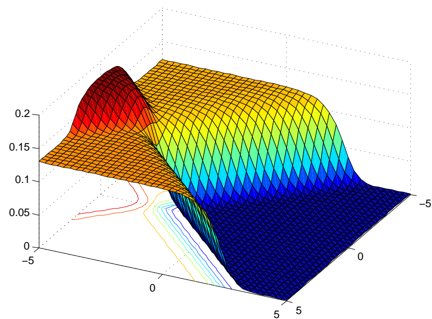
E(XOR), 1 hidden layer

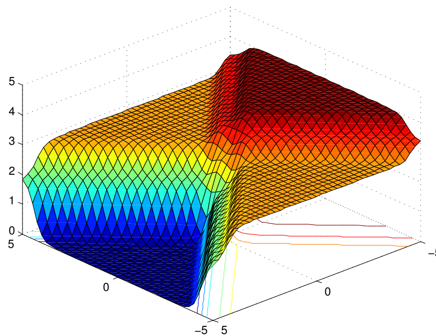




$E(w^{(1)})$

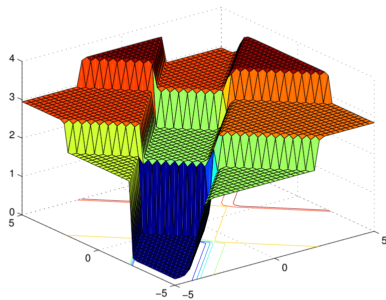
E(XOR), 3 hidden layers

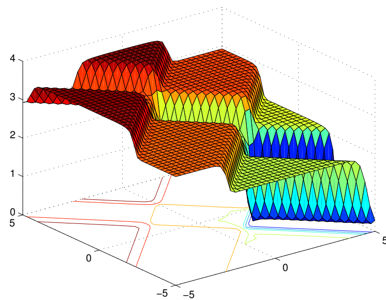




$E(w^{(1)})$

E(XOR), 10 hidden layers





$E(w^{(1)})$

trick: momentum

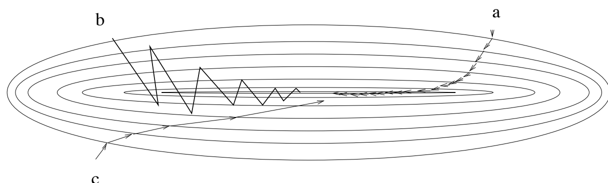
Remember: weight updates are done with

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \alpha \nabla E(\mathbf{w}(t))$$

What is the best value of the learning rate? If (a) α is too small, convergence will be too slow. If it is (b) too large, we will overshoot the minimum. We can combine the two:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \alpha \nabla E(\mathbf{w}(t)) + \beta \Delta \mathbf{w}(t)$$

β is called the *momentum*



trick: momentum

How do we choose α and β? With

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \alpha \nabla E(\mathbf{w}(t)) + \beta \Delta \mathbf{w}(t)$$

if, for the sake of the argument, assume that ∇E does not change:

$$\begin{aligned} \Delta \mathbf{w} &= -\alpha \nabla E(1 + \beta + \beta^2 + \ldots) \\ &= -\frac{\alpha}{1-\beta} \nabla E \end{aligned}$$

Assuming a perfect ∇E, the best values for α and β are when

$$\frac{\alpha}{1-\beta} = 1 \qquad \Rightarrow \qquad \alpha + \beta = 1$$

Typically we choose α small and β large (of course, $\alpha, \beta > 0$).

We will soon learn better methods of setting α and β.

trick: flexible step width

It is to be expected that once the optimisation closes in on the minimum, the remaining adjustments have to become smaller to avoid overshooting and oscillations. The simplest way to do this is to set

$$\Delta \mathbf{w}(t) = -\alpha(t)\nabla E(t) \qquad \text{with} \qquad \alpha(t) = \alpha(0)\, r^{\tau} \tag{11}$$

the shrinkage factor r is typically very close to unity (e.g., 0.999). More sophisticated schemes, e.g. the bold driver technique, try to adjust α automatically based on convergence speed.

Bold driver: If the error has decreased, increase α by a small proportion (typically 1%–5%). If the error has increased by more than a tiny proportion, however, undo the last weight change, and decrease α sharply—typically by 50%.

Most of these schemes can be combined with the momentum term, though this does not automatically guarantee an improvement.

trick: separate step width for each weight

We have seen the vanishing gradient: the influence of some weight changes on E is minimal. This is a very typical for FFNN.

A trick is to maintain different learning rates for each parameter w_i:

$$\Delta w_i = -\alpha_i \nabla_i E, \qquad (12)$$

This must be combined with, e.g., the bold driver method to vary α_i.

avoidance of local minima

Langevin updating basically adds a Gaussian noise term ϵ to the weight updates, the amplitude of which exponentially decreases as training progresses. It is hoped that this will help the optimisation step out of local minima.

$$\Delta w_i = -\alpha \nabla_i E + \epsilon \tag{13}$$

this is a simplified form of. . . simulated annealing

Manhattan backpropagation

This “hack” only uses the *sign* of the gradient.

It turns out to be a very stable learning scheme:

$$\Delta w_i = -\alpha_i \operatorname{sgn}(\nabla_i E) \tag{14}$$

resilient propagation (RProp)

RProp [Riedmiller et al., 1993] uses only the sign of the gradient, but assigns each weight its own step width based on convergence behaviour.

It turns out to be a very stable (hence the name) learning scheme. Let

$$\Delta w_i(t) = -\alpha_i(t)\,\mathrm{sgn}\left[\nabla_i E(t)\right] \tag{15}$$

with $E(t) := E[\mathbf{w}(t)]$ and

$$\alpha_i(t+1) = \begin{cases} \min(\alpha_i(t)\eta^+, \alpha_{\max}) & \text{if } \nabla_i E(t)\nabla_i E(t-1) > 0 \\ \max(\alpha_i(t)\eta^-, \alpha_{\min}) & \text{if } \nabla_i E(t)\nabla_i E(t-1) < 0 \\ \alpha_i(t) & \text{otherwise} \end{cases} \tag{16}$$

In effect, α increases if the gradient's sign for that weight remains the same; α decreases otherwise.

Some playing around finds good values of
$\eta^+ = 1.2$, $\eta^- = 0.5$, $\alpha_{\max} = 50$, $\alpha_{\min} = 10^{-6}$

Problem: the system tends to be unstable near flat points (including minima!)

quick-propagation [Fahlman, 1988]

Yet another one:

$$\Delta w_{ij}(t+1) = \Delta w_{ij}(t) \left(\frac{\nabla_{ij} E(t)}{\nabla_{ij} E(t-1) - \nabla_{ij} E(t)} \right) \tag{17}$$

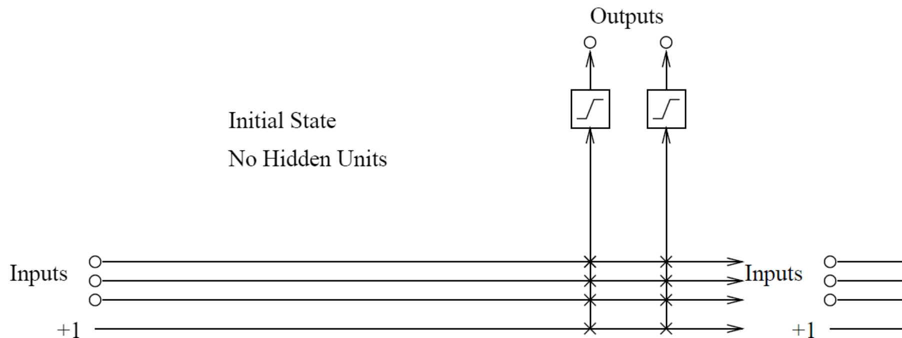
Here the assumption is that $E(\mathbf{w})$ can be well-approximated by a second-order polynomial.

If so, the above brings E immediately to the minimum.

cascade correlation [Fahlman, 1991]

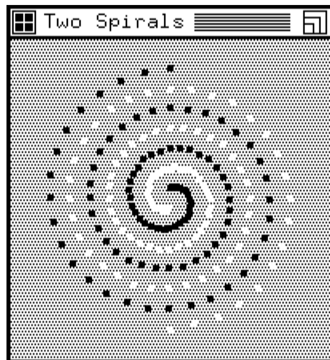
combine two basic ideas:

1. cascade architecture: hidden units are added to the network one at a time and *do not change* after they have been added
2. each new hidden unit maximizes correlation between its output and the residual error

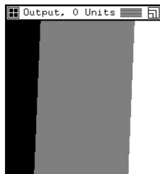
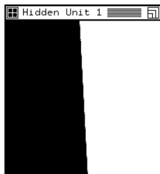
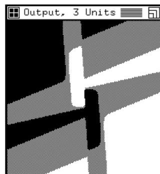
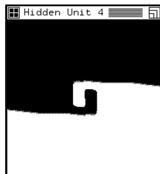
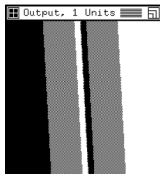
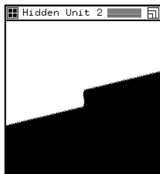
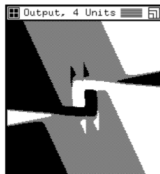
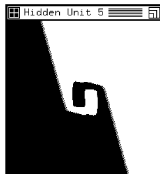
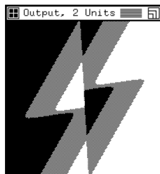
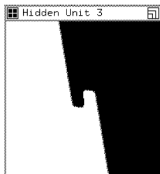
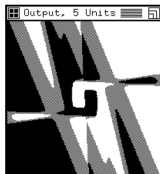


CC example: two spirals problem

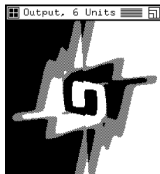
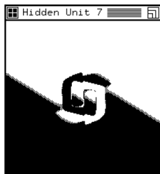
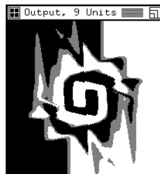
two-class problem (notoriously difficult), 194 training points, logistic output neurons.

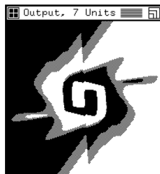
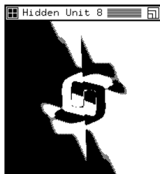
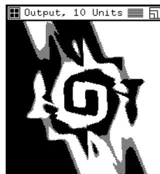
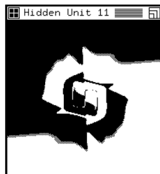
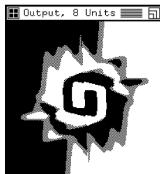
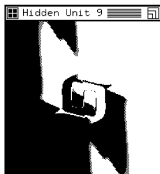
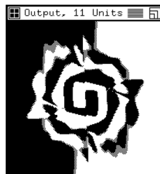
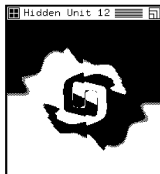


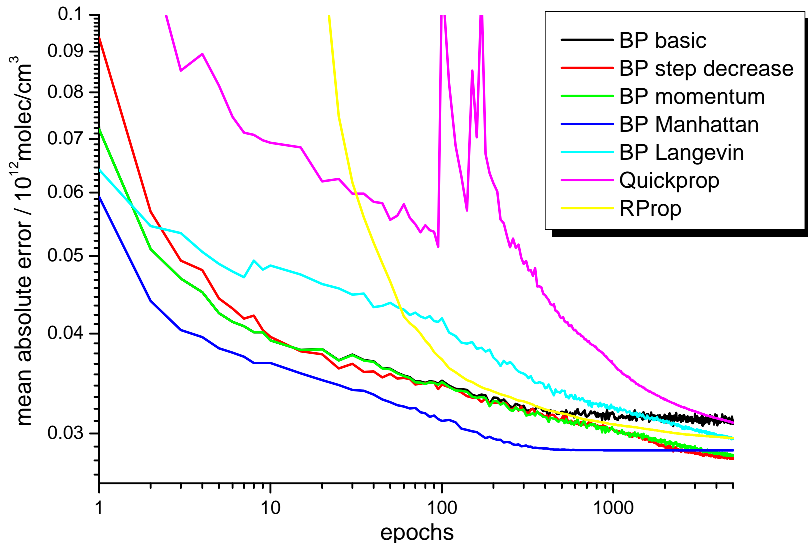
(note highest density of points in the centre)


Output, 0 Units

Hidden Unit 1

Output, 3 Units

Hidden Unit 4

Output, 1 Units

Hidden Unit 2

Output, 4 Units

Hidden Unit 5

Output, 2 Units

Hidden Unit 3

Output, 5 Units

Hidden Unit 6


Output, 6 Units

Hidden Unit 7

Output, 9 Units

Hidden Unit 10

Output, 7 Units

Hidden Unit 8

Output, 10 Units

Hidden Unit 11

Output, 8 Units

Hidden Unit 9

Output, 11 Units

Hidden Unit 12



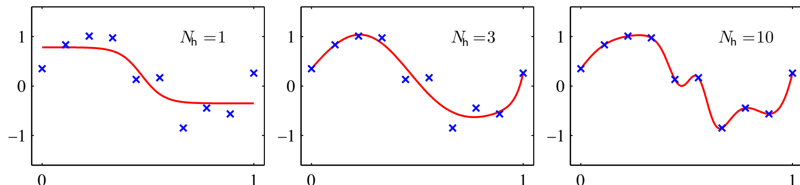
Training error for the example problem
mean absolute error / 10^{12} molec/cm^3
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
1
10
100
1000
epochs
BP basic
BP step decrease
BP momentum
BP Manhattan
BP Langevin
Quickprop
RProp

options for regularising neural networks

Modification	Issues
use small enough network	⊖ how choose # hidden units?
early stopping	⊖ need extra data for testing ⊖ which data to choose for cross-validation?
introduce weight decay term	⊖ how choose λ?
add Gaussian noise to inputs	⊖ is it realistic? ⊖ increases training time
full Bayesian treatment	⊖ nonlinear transfer functions prohibit that

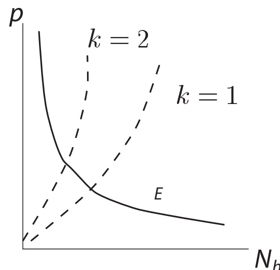
choosing the best hidden layer size N_h

Just like in generalised linear regression, for a typical toy problem we may get something like this:



Methods exist (Barron, 1991; Vysniauskas, 1993) which estimate the optimal number of hidden units N_h and number of samples p by fitting the approximation error E as

$$E(p, N_h) \approx \sum_{p} \sum_{i+j=p} \frac{\gamma_{ij}}{p^i N_h^j} \qquad (18)$$



early stopping and weight decay

recap: in Bayesian linear regression, assuming a Gaussian prior for the weights **w** lead to a quadratic regularisation term in the log likelihood.

Exactly the same happens with MLPs.

Here this procedure is called weight decay:

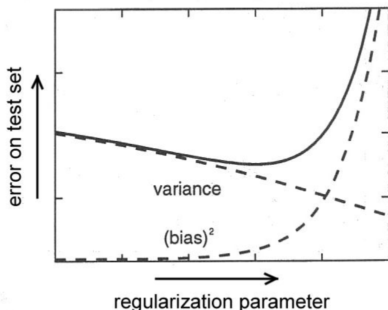
$$E = \frac{1}{2} \sum_n \|\mathcal{M}(\mathbf{x}_n, \mathbf{w}) - \mathbf{z}_n\|^2 + \frac{\lambda}{2} \mathbf{w}^\mathsf{T} \mathbf{w} \tag{19}$$

Minimisation of E leads to our MAP solution $\mathbf{w}_{\text{MAP}}$.

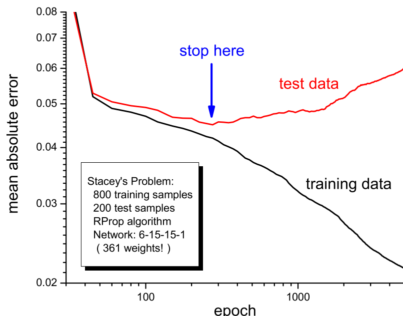
early stopping

early stopping is probably the most widely used regularisation technique for MLPs.

Bias-variance dilemma



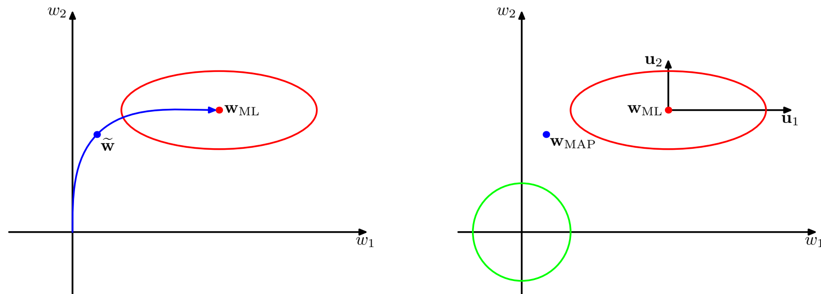
Real-world example



Note: In typical applications, use about 10% to 25% of the data for testing.

early stopping and weight decay

A typical training algorithm follows the steepest gradient first, starting with weights near zero:



We follow a trajectory in weight space and stop at $\widetilde{\mathbf{w}}$, when the validation set tells us so. In effect, this moves the weight to $\widetilde{\mathbf{w}}$ along an ev of $\nabla^2 E$ with small ev. The increase in E is small but the decrease in $|\mathbf{w}|$ large!

This is approximately the same location as $\mathbf{w}_{\mathrm{MAP}}$! With learning rate α and iteration t, it can be shown (under some conditions) that $\alpha t \propto \lambda^{-1}$.