

# Neural Networks

Read Chapter 5 in the text by Bishop, except omit Sections 5.3.3, 5.3.4, 5.4, 5.5.4, 5.5.5, 5.5.6, 5.5.7, and 5.6

## “Curse of Dimensionality” for Linear Basis Function Models

Modeling a general non-linear relationship of  $t$  to  $x$  with a linear basis function model seems attractive when  $x$  is of low dimension, but when there are many inputs, we would seem to need a huge number of local basis functions to “cover” the high dimensional input space.

This is at least a computational problem — part of the “curse of dimensionality”.

One possibility is to use a relatively small number of basis functions, that cover only the actual area where  $x$  values are found, which may be the vicinity of a manifold of much lower dimension. We might:

- pick a subset of data points as centres for basis functions
- make the basis functions depend on parameters that adapt to the data.

We’ll look now at one class of models of the second type.

## “Neural Networks” with One “Hidden Layer”

The term “neural network” can refer to many models that were originally inspired by thoughts on how the brain might work. We’ll look here at the most common one — a “multilayer perceptron” network (MLP) with one “hidden layer”.

This can be seen as a linear basis function model extended to make the basis functions depend on some additional parameters. (So the model is no longer linear with respect to these additional parameters.)

As before, we model the response  $t$  for a case with inputs  $x$  as

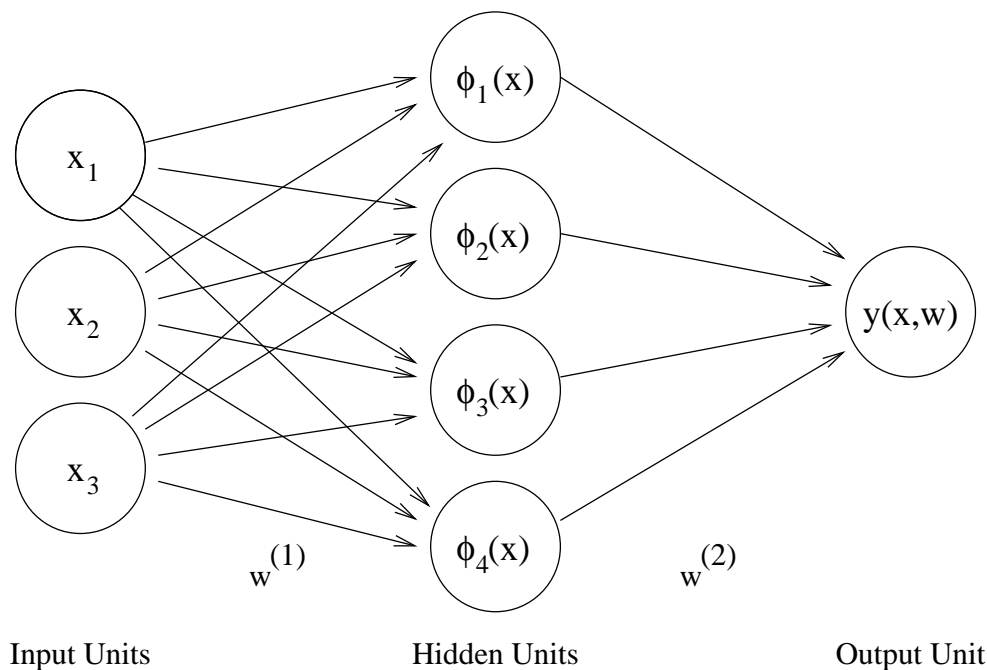
$$t = y(x, w) + \text{noise}$$

but now the parameters  $w$  — called “weights” — come in two groups:  $w^{(1)}$  defines the basis functions, and  $w^{(2)}$  defines a linear combination of these basis functions.

Note: I’ll follow notation similar to Chapter 5 in the text, except I’ll assume that we’re solving a regression problem (the text considers classification problems too), and that there is only one target value for each case (the text allows for  $K$  targets). So I’ll drop the function  $f$  and subscript  $k$  that appear in the text.

# The Architecture of a Multilayer Perceptron Network

A multilayer perceptron with one layer of four “hidden” units looks like this:



Each hidden unit computes a value based on a linear combination of the values of units that point into it.

More layers of hidden units can be added — the hidden unit values in one layer are computed from linear combinations of hidden unit values in the previous layer.

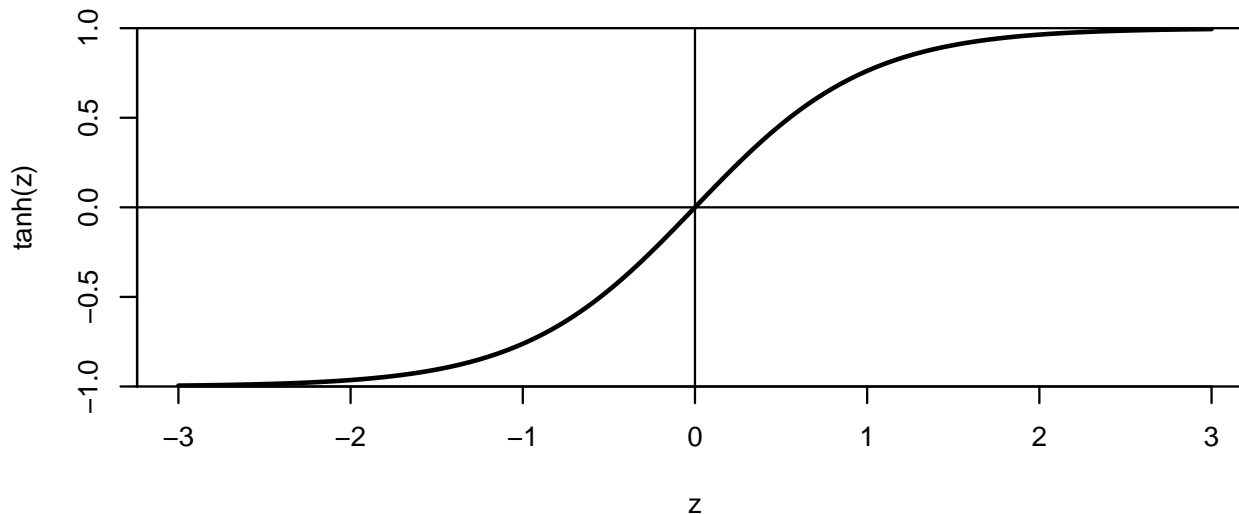
## The Function Computed by the Network

The function,  $y(x, w)$ , computed by a multilayer perceptron network with one hidden layer of  $M$  units and one output unit can be written as follows:

$$y(x, w) = w_0^{(2)} + \sum_{j=1}^M w_j^{(2)} \phi_j(x, w), \quad \phi_j(x, w) = h\left(w_{0j}^{(1)} + \sum_{k=1}^D w_{kj}^{(1)} x_k\right)$$

The network can approximate any function (better as  $M$  increases) if the *activation function*,  $h(a)$ , is any non-polynomial function.

A traditional choice of activation function (which I'll assume in later slides) is  $\tanh(a) = (e^a - e^{-a}) / (e^a + e^{-a})$ . It looks like this:



# Training the Network by Maximum Likelihood

The simplest training procedure for an MLP network is to simply adjust the parameters ( $w$ ) to maximize some measure of fit to the training data. The most obvious measure of fit is likelihood.

For a regression problem, we may model the target values in training cases,  $t_1, \dots, t_N$ , as being independent (given  $x_1, \dots, x_N$  and  $w$ ), with each  $t_i \sim N(y(x_i, w), \sigma^2)$ , for some noise variance  $\sigma^2$ .

The log of the likelihood (ignoring terms not involving  $w$ ) is then

$$-\frac{1}{2\sigma^2} \sum_{i=1}^N (t_i - y(x_i, w))^2$$

So maximum likelihood estimation for  $w$  just minimizes total squared error. But this is a complicated function of  $w$ , with no analytical way to find the maximum.

In real problems, the dimension of  $w$ , which is  $M(D+2) + 1$ , may be in the hundreds, thousands, tens of thousands, ...

There usually are many local maxima, and finding a global maximum is hopeless. Fortunately, good results are often obtained from fairly good local maxima.

# Multiple Ways of Fitting the Data with an MLP

We may not need to find the global maximum of the likelihood because an MLP can fit the data in multiple ways — some producing identical results, others producing nearly identical results.

First, there are exact symmetries:

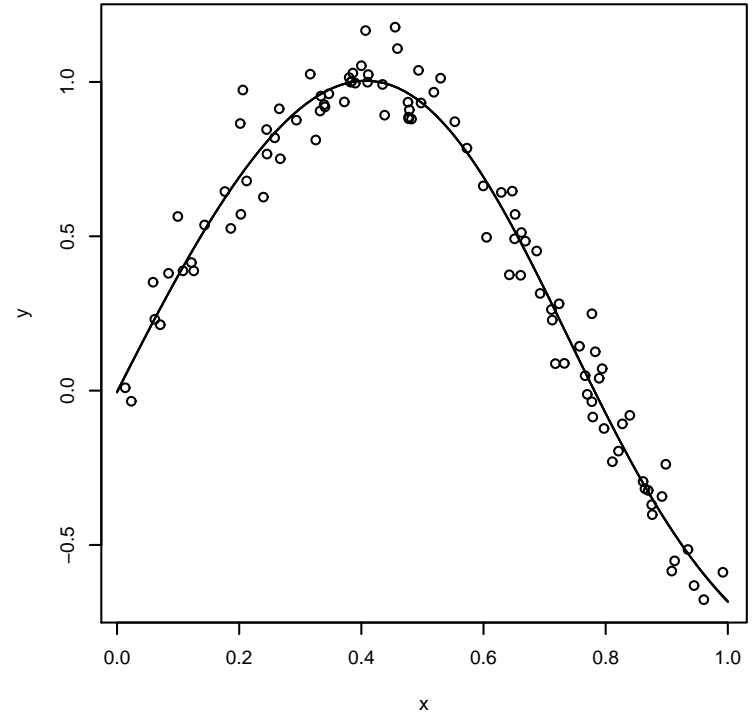
- Permuting the order of hidden units.
- For any hidden unit,  $j$ , negating  $w_j^{(2)}$  and  $w_{kj}^{(1)}$  for all  $k$  (including 0).

There are also similar, but different, ways to fit the data.

Suppose we have 10 hidden units, but 3 hidden units are enough to roughly fit the data. The other 7 hidden units could fit various different slight wiggles, with any of these overall fits being fairly good.

# Example of Fitting a Network

Here are 100 training points (circles) that I artificially generated as  $t = \sin(4x) + \text{noise}$ , and the functions computed by three MLP networks with three hidden units found by fitting this data, with different starting points of the optimization procedure. The three fits are indistinguishable. The biggest difference in predicted value over the range  $(0, 1)$  is about 0.003.



Here are the parameter values for the three fits:

$w_{11}^{(1)}$	$w_{12}^{(1)}$	$w_{13}^{(1)}$	$w_{01}^{(1)}$	$w_{02}^{(1)}$	$w_{03}^{(1)}$	$w_1^{(2)}$	$w_2^{(2)}$	$w_3^{(2)}$	$w_0^{(2)}$
-1.807	1.422	-2.923	-0.159	-0.224	2.052	-1.565	1.162	1.894	-1.829
-1.906	3.018	-0.218	0.005	-2.132	0.436	-2.140	-1.690	-0.530	-1.418
2.924	0.621	1.906	-2.078	-0.435	0.004	-1.867	0.885	1.987	-1.459



# Iteratively Finding the MLP Parameters that Maximize the Likelihood

There is no analytical way of finding the parameters,  $w$ , of a multilayer perceptron network that maximize the likelihood, so iterative methods are used.

To start, we set all the network parameters to small random values — eg, values drawn uniformly from  $[-0.1, +0.1]$ . (Except we might set  $w_0^{(2)}$  to the mean of  $t$  in the training cases, so it won't be far off if this mean is large.)

Setting all the  $w_j^{(2)}$  and  $w_{kj}^{(1)}$  to zero would not work, since the hidden units are then all identical, and an iterative method that treats them symmetrically could never make them non-identical.

We stop trying to increase the likelihood when either (a) the likelihood seems to be at a local maximum, or (b) it seems that trying to increase the likelihood further would actually make performance on test cases worse, or (c) our patience runs out. For (b), we need to use a held-out set of “validation” cases.

The optimization may be quick for simple problems, but for the most difficult problems, network training can take days (or even weeks).

# Three Approaches to Optimization

We'll see that it's easy to compute the partial derivatives of the log likelihood. Second derivatives can also be computed.

Using these derivatives, several iterative approaches to maximizing the log likelihood are possible:

- Newton and quasi-Newton methods — use the matrix of second derivatives (or estimates) to approximate the log likelihood as a quadratic function of the parameters. Move to the maximum of this approximation at each iteration. *Infeasible if there are many parameters (matrix too big); may not work well anyway, if the likelihood is very non-quadratic.*
- Conjugate gradient methods — cleverly choose good directions to look for the maximum, moving to the maximum along that direction at each iteration.
- Gradient descent — always move in the direction of the gradient (vector of partial derivatives). In the simplest scheme, we change  $w$  by some constant times the gradient of the log likelihood. *The original, and crudest, method, but sometimes the best — it doesn't need a big matrix of second derivatives, and can be used “on-line”, without looking at all training cases at once.*

# Computing Derivatives of the Log Likelihood

Partial derivatives of the log likelihood with respect to the network parameters are found by applying the chain rule backwards from the network output. Since training cases are assumed to be independent, we calculate derivatives separately for each training case, then just sum them. I'll denote **minus** the log likelihood for just one training case by  $E$  (somewhat matching Bishop's notation).

We first do “forward propagation”, computing the summed input to each hidden unit,  $a_j$ , then the values of the hidden units,  $z_j$ , and finally the value of the output unit,  $y$ :

$$a_j = w_{0j}^{(1)} + \sum_{k=1}^D w_{kj}^{(1)} x_k$$

$$z_j = \tanh(a_j)$$

$$y = w_0^{(2)} + \sum_{j=1}^M w_j^{(2)} z_j$$

We then use “backpropagation” to find the derivatives of  $E$  with respect to  $y$ , to  $z_j$ , and to  $a_j$ . From these, we can find the derivatives with respect all the components of  $w$ .

## Derivatives of $E$ With Respect to $y$ , $z_j$ , and $a_j$

To start, we find the derivative of  $E$  with respect to the network output,  $y$ .

If the response,  $t$ , is modeled as Gaussian with mean  $y$  and variance  $\sigma^2$ , minus the log likelihood for just one case is  $E = (t - y(x, w))^2 / 2\sigma^2$ , and its derivative with respect to  $y$  is

$$\frac{\partial E}{\partial y} = (y - t) / \sigma^2$$

Once we have computed  $\partial E / \partial y$ , we work backward to (for all  $j$ ) compute  $\partial E / \partial z_j$ , found assuming that  $w_j^{(2)}$  and the  $z_{j'}$  for  $j' \neq j$  are fixed:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial z_j} = w_j^{(2)} \frac{\partial E}{\partial y}$$

Next, we use the fact that  $\tanh'(a) = 1 - \tanh(a)^2$  to work back to the derivative of  $E$  with respect to  $a_j$ , the summed input to hidden unit  $j$ :

$$\frac{\partial E}{\partial a_j} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial a_j} = (1 - z_j^2) \frac{\partial E}{\partial z_j}$$

If we had more than one hidden layer, we would continue working backwards, finding the derivatives of  $E$  with respect to the values of all hidden units, and with respect to the summed inputs for these hidden units.

## Derivatives of $E$ With Respect to $w$

We can find the derivative of  $E$  w.r.t. a parameter on a connection from unit A to unit B by multiplying the value of A by the derivative of  $E$  w.r.t. the summed input to B.

For  $w_1^{(2)}, \dots, w_M^{(2)}$ :

$$\frac{\partial E}{\partial w_j^{(2)}} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial w_j^{(2)}} = z_j \frac{\partial E}{\partial y}$$

Also,  $\partial E / \partial w_0^{(2)} = \partial E / \partial y$ .

Similarly, for  $w_{1j}^{(1)}, \dots, w_{Dj}^{(1)}$ :

$$\frac{\partial E}{\partial w_{kj}^{(1)}} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial w_{kj}^{(1)}} = x_k \frac{\partial E}{\partial a_j}$$

Also,  $\partial E / \partial w_{0j}^{(1)} = \partial E / \partial a_j$ .

# Maximizing the Likelihood by Simple Gradient Ascent

After randomly initializing the parameters,  $w$ , to small values, we maximize the log likelihood repeatedly doing the following, using some suitably chosen “learning rate” or “stepsize”,  $\eta$ :

1. For each training case:
  - Compute the values of the units by forward propagation.
  - Compute the derivatives of  $E$  w.r.t. the unit values by backpropagation.
  - Compute the derivatives of  $E$  w.r.t. the parameters from these results.
2. Sum these derivatives over all training cases, to get the gradient vector,  $g$ , of minus the total log likelihood.
3. Change  $w$  by moving in the direction of (minus) this gradient, replacing  $w$  by  $w - \eta g$ .

If  $\eta$  is too big, the changes will “overshoot”, and end up decreasing the likelihood rather than increasing it. We have to set  $\eta$  by trial-and-error.

# On-line Learning

The procedure in the previous slide is sometimes called “batch” gradient ascent learning, since the derivatives from the training cases are handled as one batch.

It’s also possible to do “on-line” learning, in which we update the parameters based on each training case in turn. This may work better if the training cases are redundant — many cases provide the same information. Batch learning then wastes time looking at them all before doing anything.

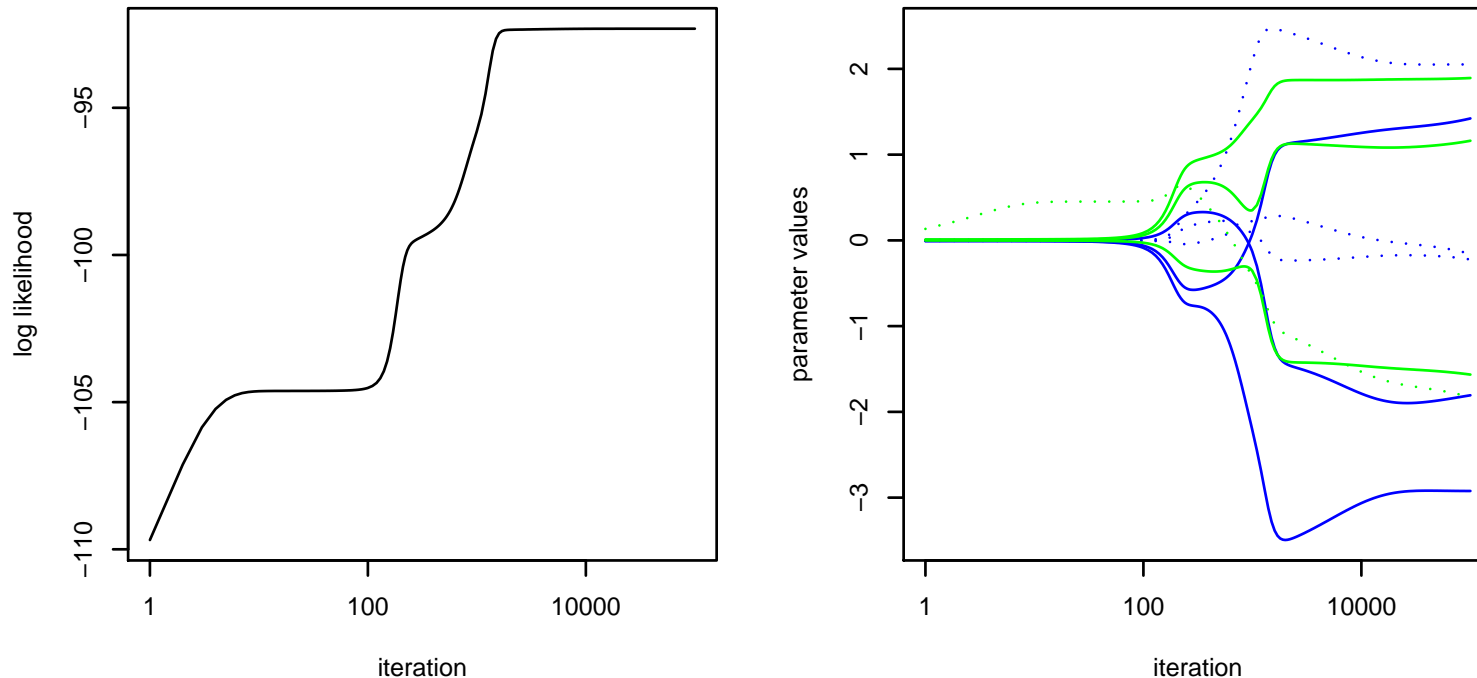
However, for on-line learning to converge to a (local) maximum, we will have to decrease  $\eta$  with time (or switch to batch learning once we’re near the maximum).

Actually, *true* on-line learning uses a new training case for each update — there is no finite training set, but rather a continuous stream of training cases. This is the situation for perceptual learning in humans.

# Example of Simple Gradient Ascent Learning

Recall the previous example, with one input (ranging over  $(0, 1)$ ), one real target (equal to  $\sin(4x) + \text{noise}$ ), and 100 training cases.

I trained a MLP network with three hidden units for 100000 iterations of simple gradient descent ( $\eta = 0.3$ ). Here are plots of the log likelihood and of the 10 parameters as a function of iteration (log scale):

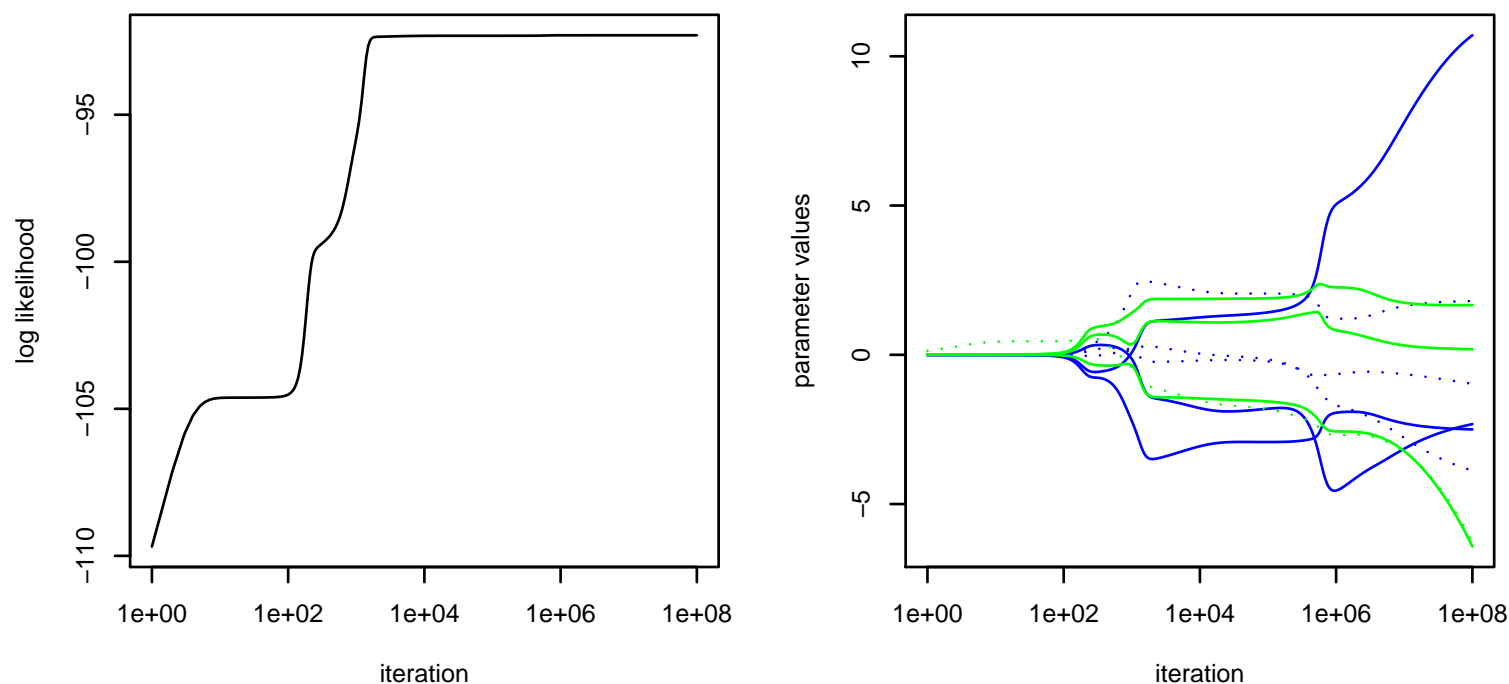


Blue lines are  $w_{kj}^{(1)}$  ( $k = 0$  dotted). Green lines are  $w_j^{(2)}$  ( $j = 0$  dotted).



## Continuing the Example...

It may seem like the optimization has converged after 100000 iterations. But what happens if we continue training for many more iterations?



This shows how tricky the MLP likelihood function can be! In practice, we don't try to find the absolute maximum, since with complex networks it is sure to overfit the training data anyway...

## Overfitting in This Example

Here is the true regression function,  $\sin(4x)$ , in black, and the regression functions defined by the networks after training for  $10^2$  (violet),  $10^3$  (blue),  $10^4$  (green),  $10^6$  (orange), and  $10^8$  (red) iterations.

The last two curves may have slightly overfit the data.

