Iteratively Finding the MLP Parameters that Maximize the Likelihood

There is no analytical way of finding the parameters 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 β_0 to the mean of the y values, so it won't be far off if this mean is large.)

Setting all the β_k and $\gamma_{k,j}$ 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, ℓ, with respect to the network parameters. 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 just move by the amount $\eta \nabla \ell$. *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 by Backpropagation

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 derivates separately for each training case, then just sum them.

We first do "forward propagation", computing the summed input to the hidden units, s_k, then the values of the hidden units, h_k, and finally the value of the output unit, o:

$$\begin{aligned} s_k &= \gamma_{k,0} + \sum_{j=1}^{p} \gamma_{k,j} x_j \\ h_k &= \tanh(s_k) \\ o &= \beta_0 + \sum_{k=1}^{q} \beta_k h_k(x) \end{aligned}$$

We then use "backpropagation" to find the derivatives of ℓ with respect to o, h_k, and s_k. From these, we can find the derivatives with respect to the βs and γs.

Derivative of ℓ With Respect to the Network Output

To start, we find the derivative of the log likelihood with respect to the network output, $o = f(x)$. If y is real, modeled as Gaussian with mean o and variance σ^2,

$$\frac{\partial \ell}{\partial o} = \frac{\partial}{\partial o}\Big[-(1/2)\log(2\pi\sigma^2) - (y-o)^2/2\sigma^2 \Big] = (y-o)\,/\,\sigma^2$$

If y is binary, with $P(y=1\,|\,x) = 1\,/\,(1+\exp(-o))$,

$$\begin{aligned} \frac{\partial \ell}{\partial o} &= \begin{cases} \dfrac{\partial}{\partial o}\Big[-\log(1+\exp(-o))\Big] & \text{if } y=1 \\ \dfrac{\partial}{\partial o}\Big[-\log(1+\exp(+o))\Big] & \text{if } y=0 \end{cases} \\ &= y - 1\,/\,(1+\exp(-o)) = y - P(y=1\,|\,x) \end{aligned}$$

In both cases, the result is intuitive: The derivative is positive if the current output produces a prediction lower than the actual value, so that increasing the output will increase the likelihood.

Going Backwards to Derivatives of ℓ With Respect to h_k and s_k

Once we have computed $\partial\ell/\partial o$, we work backward to compute $\partial\ell/\partial h_k$, found assuming that β_k and the $h_{k'}$ for $k' \neq k$ are fixed:

$$\frac{\partial\ell}{\partial h_k} \;=\; \frac{\partial\ell}{\partial o}\,\frac{\partial o}{\partial h_k} \;=\; \beta_k\,\frac{\partial\ell}{\partial o}$$

Next, we use the fact that $\tanh'(z) \;=\; 1-\tanh(z)^2$ to work back to the derivative of ℓ with respect to s_k, the summed input to hidden unit k:

$$\frac{\partial\ell}{\partial s_k} \;=\; \frac{\partial\ell}{\partial h_k}\,\frac{\partial h_k}{\partial s_k} \;=\; (1-h_k^2)\,\frac{\partial\ell}{\partial h_k}$$

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

Computing Derivatives With Respect to Parameters From Derivatives With Respect to Unit Inputs

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

For $\beta_1,\ldots,\beta_q$:

$$\frac{\partial\ell}{\partial\beta_k} \;=\; \frac{\partial\ell}{\partial o}\,\frac{\partial o}{\partial\beta_k} \;=\; h_k\,\frac{\partial\ell}{\partial o}$$

Also, $\partial\ell\,/\,\partial\beta_0 \;=\; \partial\ell\,/\,\partial o$.

Similarly, for $\gamma_{k,1},\ldots,\gamma_{k,p}$:

$$\frac{\partial\ell}{\partial\gamma_{k,j}} \;=\; \frac{\partial\ell}{\partial s_k}\,\frac{\partial s_k}{\partial\gamma_{k,j}} \;=\; x_j\,\frac{\partial\ell}{\partial s_k}$$

Also, $\partial\ell\,/\,\partial\gamma_{k,0} \;=\; \partial\ell\,/\,\partial s_k$.

Maximizing the Likelihood by Simple Gradient Descent

Collect all the parameters into one vector, $\theta = (\beta,\,\gamma)$.

After randomly initializing the parameters to small values, we maximize the log likelihood, ℓ, by repeatedly doing the following, using some suitably chosen "learning rate" or "stepsize", η:

1. For each training case:
   - Compute the values of the units by forward propagation.
   - Compute the derivatives of ℓ w.r.t. the unit values by backpropagation.
   - Compute the derivatives of ℓ w.r.t. the parameters from these results.
2. Sum these derivatives over all training cases, to get the gradient vector, $\nabla\ell$.
3. Change the parameters by moving in the direction of the gradient, replacing θ by $\theta + \eta\nabla\ell$.

If η is too big, the changes will "overshoot", and end up decreasing the likelihood rather than increasing it. We have to set η by trial-and-error.

On-line Gradient Descent

The procedure in the previous slide is sometimes called "batch" gradient descent, since the derivatives from the training cases are handled as one batch.

It's also possible to do "on-line" gradient descent, 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 training then wastes time looking at them all before doing anything.

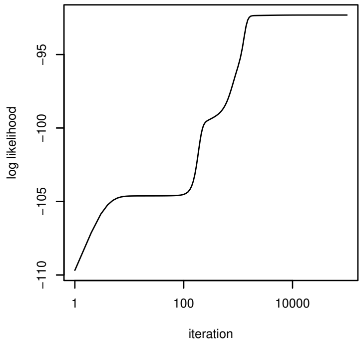
However, for on-line gradient descent to converge to a (local) maximum of ℓ, we have to decrease η with time (or switch to batch gradient descent once we're near the maximum).

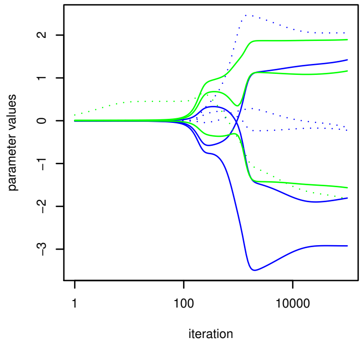
Actually, *true* on-line learning uses a new training case for each update — there is no finite training set, but rather a continuous stream of training cases. This is the situation for perceptual learning in humans.

Example of Simple Gradient Descent

Recall the example from last lecture, 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):

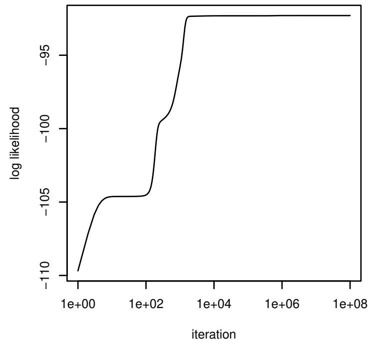


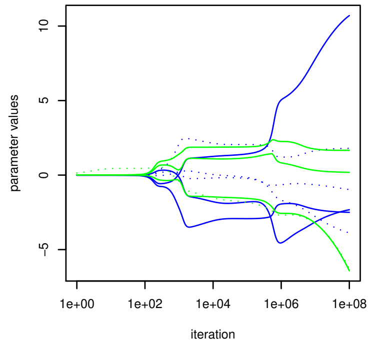


Blue lines are γs ($\gamma_{k,0}$ dotted). Green lines are βs (β_0 dotted).

Continuing the Example...

It may seem like the gradient descent maximization has converged (and this is what I assumed last lecture). 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.