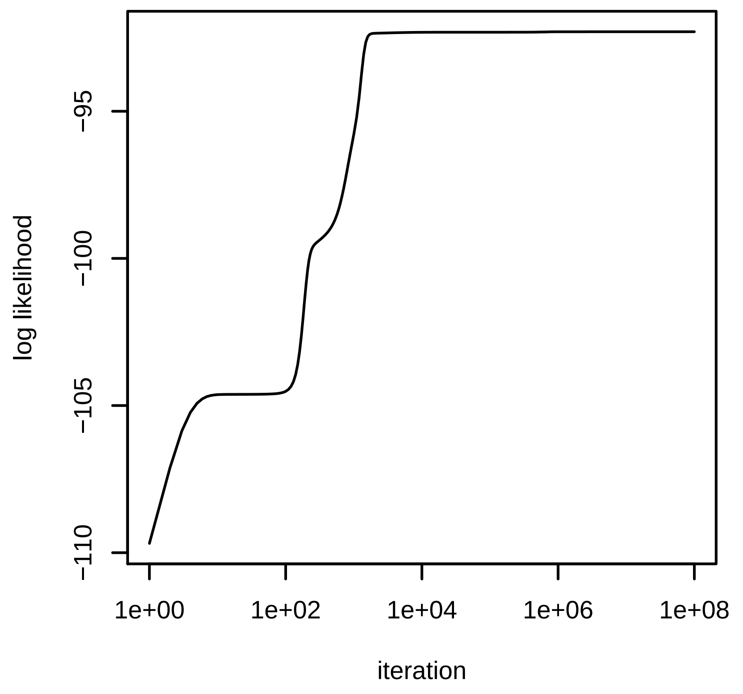
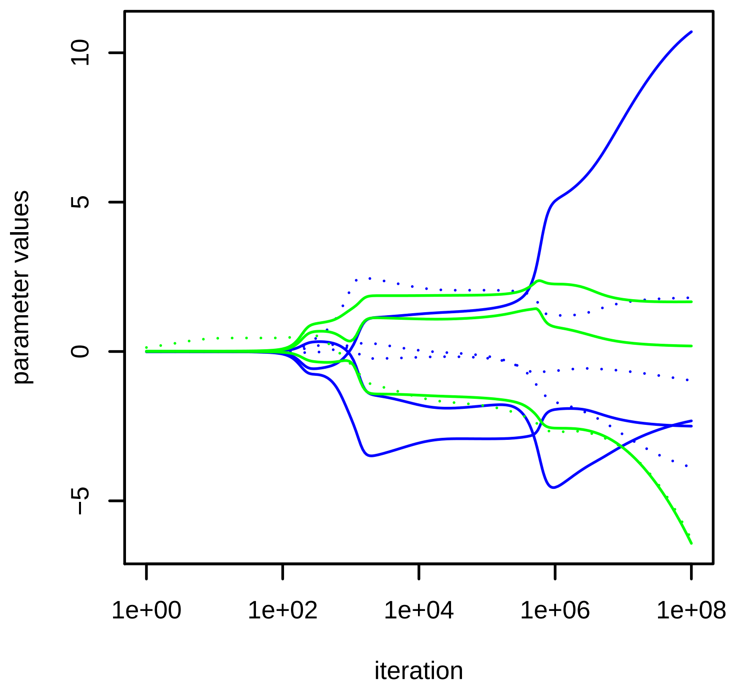
More on Neural Networks

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

Recall the MLP Training Example From Last Lecture



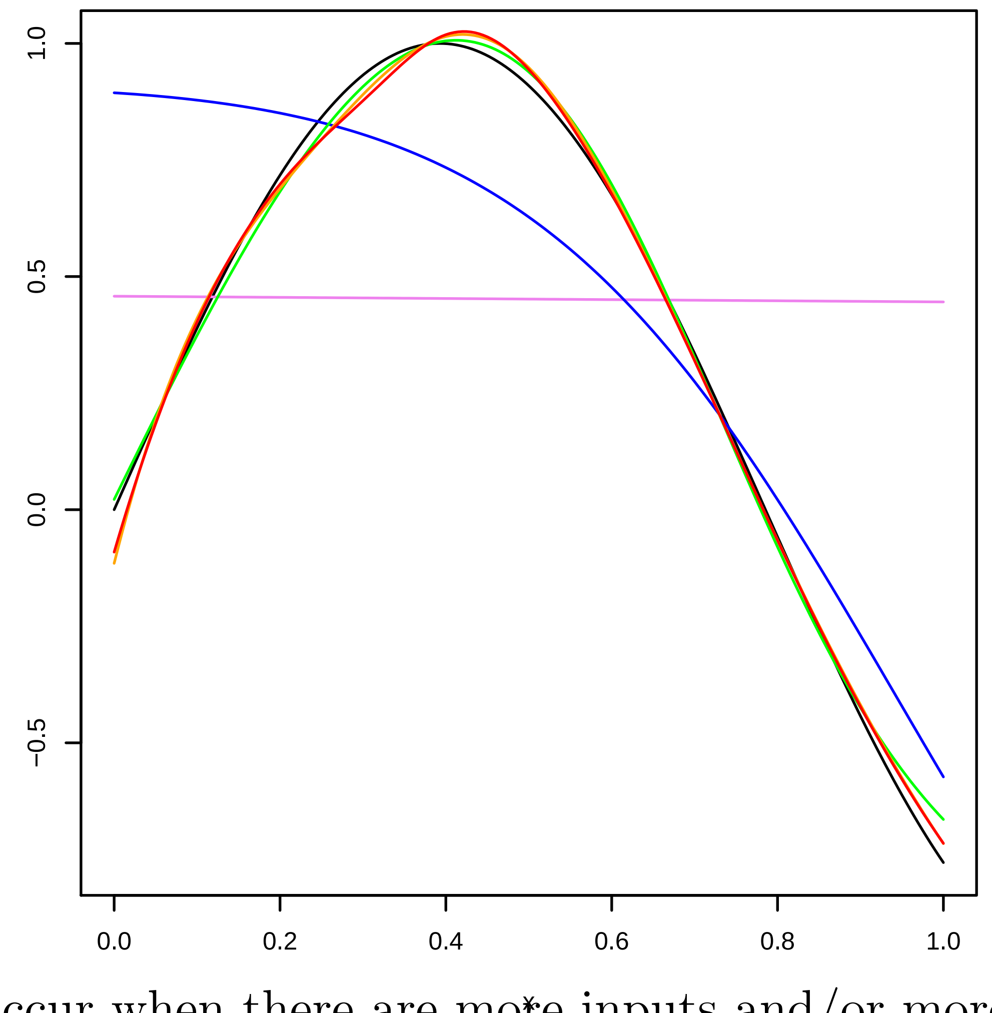


Training for around 10000 iterations seemed to have found a local maximum, but further training showed it hadn't really!

Overfitting in This Example

The network found by training for 10000000 iterations has slightly higher likelihood, but produces (slightly) overfitted results.

True regression function in black, along with 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.



Much more severe overfitting can occur when there are more inputs and/or more hidden units.

Avoiding Overfitting Using a Penalty

As for linear basis function models, we can try to avoid overfitting by maximizing the log likelihood plus a penalty function.

For an MLP with one hidden layer, a suitable penalty to add to minus the log likelihood might be

$$\lambda_1 \sum_{k=1}^{D} \sum_{j=1}^{M} \left[w_{kj}^{(1)} \right]^2 \; + \; \lambda_2 \sum_{j=1}^{M} \left[w_j^{(2)} \right]^2$$

We need to select two constants controlling the penalty, λ_1 and λ_2. Setting $\lambda_1 = \lambda_2$ isn't always reasonable, since a suitable value for λ_1 depends on the measurement units used for the inputs, whereas a suitable value for λ_2 depends on the measurement units for the response.

We might try S-fold cross-validation, but it may not work well, if each training run goes to a different local maximum. So we might use a single split into estimation and validation sets, with no re-training on the whole training set.

Avoiding Overfitting Using “Early Stopping”

We saw in the example that gradient descent spent a long time in a reasonable solution (not overfitted) before things went a bit crazy. Maybe we can somehow stop at this good point.

Some impressive results of MLP training that have been reported actually depended on having done this without meaning to — they just stopped when they lost patience with gradient descent, and that happened to be at a good point.

We’d like an “early stopping” method that doesn’t rely on being lucky.

Early Stopping Using a Validation Set

We can tell when to stop gradient descent optimization using a set of validation cases that's separate from the cases used to compute the gradient.

Here's the procedure:

1. Randomly divide the training set into an estimation set and a validation set — eg, 80% of cases in the estimation set and 20% n the validation set.

2. Randomly initialize the parameters to values near zero.

3. Repeatedly do the following:

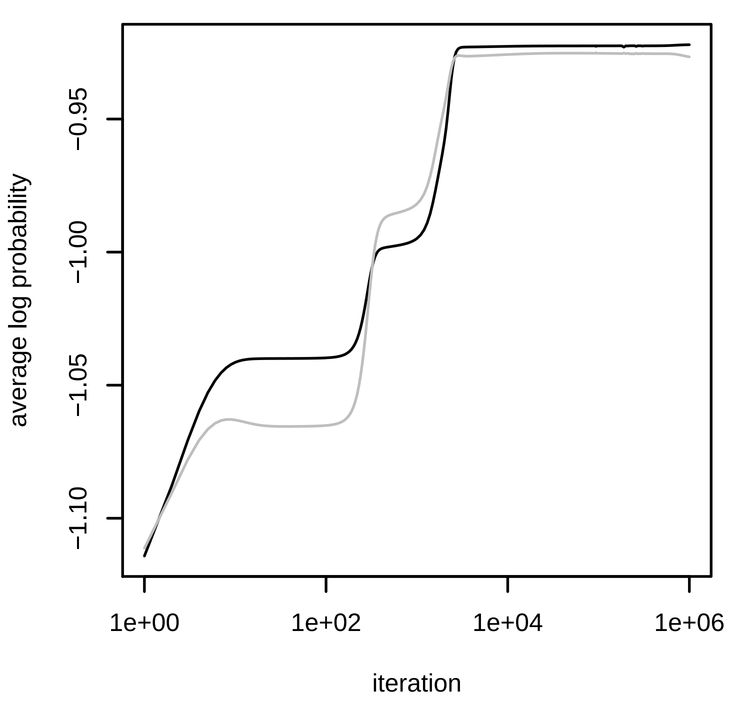
   – Compute the gradient of the log likelihood using the estimation set.
   – Update the parameters by adding η times the gradient.
   – Compute the log probability of y given x for the validation cases.

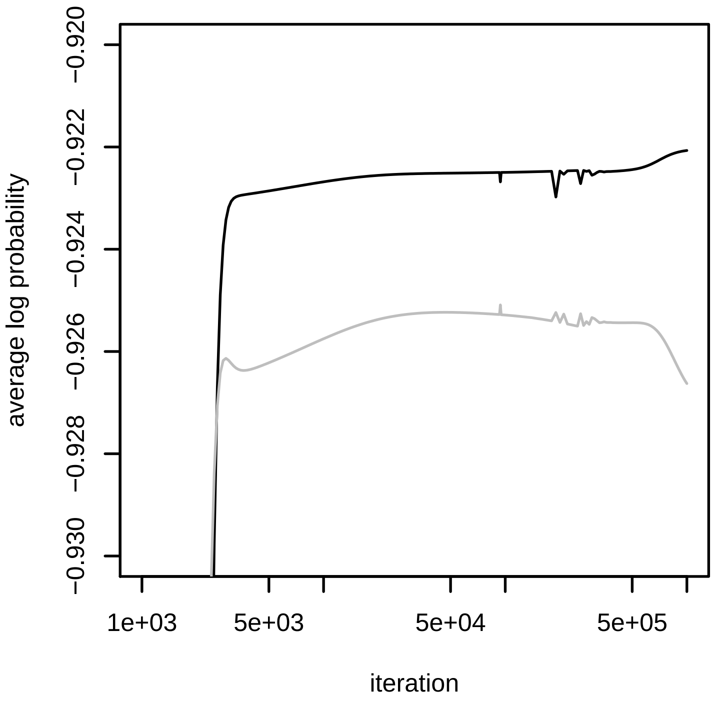
   Stop when the average log probability for validation cases is substantially less than the maximum of the values found previously (definitely getting worse).

4. Make predictions for test cases using the parameter values from the loop above that gave the highest average log probability to the validation cases.

An Example of Early Stopping Using Cross Validation

I tried same example as before, but with 75 of the 100 training cases used for estimation (computing the gradient) and 25 used for validation (deciding when to stop / which parameters to use). I used 20 hidden units this time, and set $\eta = 0.2$.



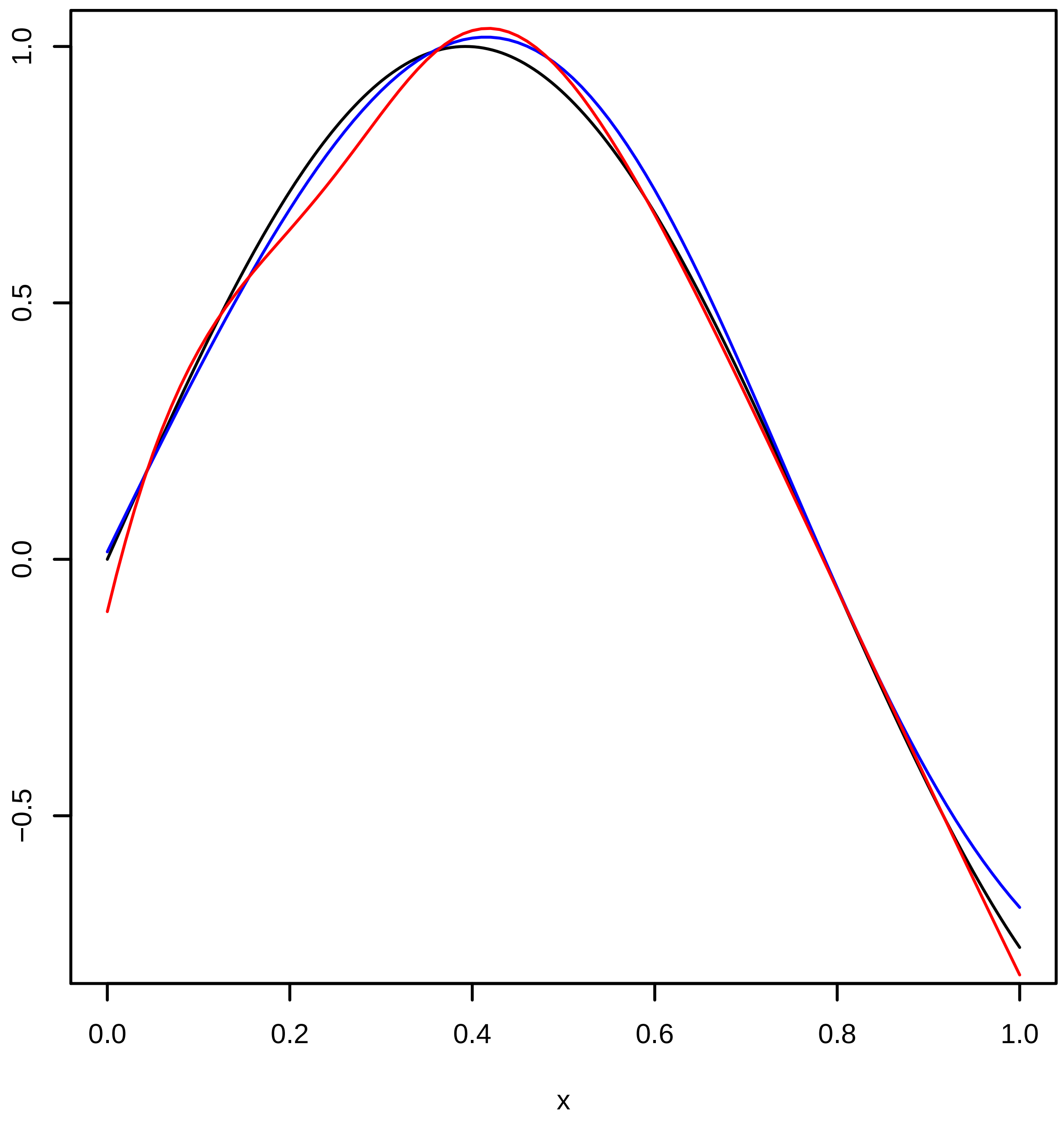


I divide total log probability by the number of cases in these plots, so that 75 vs. 25 won't matter. The average for estimation cases is shown in black, for validation cases in grey. The highest log probability for validation cases is at iteration 94000.

The Function Computed by the Network Selected

Here is the true regression function (black), the regression function defined by the network at iteration 94000 that was selected by cross validation (blue) ,and the last network at iteration 1000000 (red).

Cross validation worked this time, but 25 validation cases is really too few for reliable results.



Advantages of Early Stopping

Early stopping using a validation set has some advantages over a penalty method, in which you set λ_1 and λ_2 using S-fold cross validation, then train again on all the data with the λ_1 and λ_2 you choose.

With early stopping:

- You only have to train the network *once* — not once for each setting of λ_1 and λ_2 you consider, and then again on all the training data to get the final parameters. Important if training takes a week!
- The performance measure from cross-validation applies directly to the actual set of network parameters you will use, not to values of λ_1 and λ_2 that may not do the same thing for a different local maximum.

Disadvantages of Early Stopping

Early stopping also has some problems:

- It's very *ad hoc* — what's the justification for this procedure?
- It depends on details of the optimization method — eg, results could be different with a different η.
- In particular, we might want to use a different η for $w^{(1)}$s than for $w^{(2)}$ — sort of like using different λs for a penalty method.
- Some of the training data is used only to decide when to stop — this seems wasteful.

We could try to solve the third problem by doing training runs using different values of η for $w^{(1)}$ and for $w^{(2)}$, and then pick the best parameters (using the validation set) from all these runs.

The fourth problem can also be solved, by doing more training runs...

Using an Ensemble of Early-Stopped Networks

Rather than train just once, with a single split into estimation and validation sets, we can train several times, with several splits, to create an *ensemble* of networks.

It seems best to split the training set into S equal portions, and then train S times. Each training run uses $S-1$ portions of the training set for estimation, and one portion for validation (to choose which parameters from the run to use).

We get S sets of parameters this way. To make predictions for test cases, we *average* the predictions from all S networks.

This way we use the whole training set for estimation. Averaging several network outputs also reduces the effect of any single peculiar training run.

Modeling Non-Gaussian Response Variables

So far, we've used the function computed by the network to define a Gaussian distribution for the response variable, t, given the input variables, x:

$$t \,|\, x, \, w \quad \sim \quad N(y(x, w), \sigma^2)$$

More generally, we can use the network to define any sort of conditional distribution for t given x. Some useful examples:

- Binary classification models, where t is 0 or 1.
- Multi-class classification, where t is a value from some finite set.
- Regression with non-Gaussian residuals, where t is real, but $t|x$ is not Gaussian. Eg, we might use a heavier-tailed t-distribution if some cases have residuals that are larger than is typical of other cases.

The Logistic Model for Binary Responses

Logistic models are very common way of handling binary responses, such as when learning to classify items into two groups.

If $y(x, w)$ is the function computed by the network, we define the probability for a response of $t = 1$ to be

$$P(t = 1 \,|\, x,\, w) \;=\; \Big[1 + \exp(-y(x, w))\Big]^{-1} \;=\; \sigma(y(x, w))$$

where $\sigma(a) = 1/(1 + e^{-a})$.

In Bishop's book, he considers the σ function to be part of the network, and hence already part of $y(x, w)$, but I think it's clearer to see it as part of a model for $t|w$ that uses the network output.

One can easily show that the derivative of σ is $\sigma'(a) = \sigma(a)\,(1-\sigma(a))$. This is used when computing derivatives by backpropagation.