More Ways Beyond Linear Methods for Supervised Learning

Recall that ordinary linear regression, logistic regression, and Naive Bayes all end up looking only at a linear combination of the inputs, x. But often y is related to x in a non-linear fashion.

We can add extra input variables, like x_1^2, $\sin(x_2)$, $x_1 x_2$, etc. But how can we choose which functions of x to add?

Two approaches:

- Add an *infinite* number of extra functions, that can be used to create any function relating y to x (in some infinite class).

  With an infinite number of inputs, we'd have severe overfitting problems if we used maximum likelihood. So this method needs a way of handling that — penalty methods (splines), Bayesian methods (Gaussian process models), or margin methods (Support Vector Machines).
- Add a finite (but fairly large) number of extra functions, but make these functions depend on additional model parameters, fit along with the βs. Maximum likelihood may be OK now (but a penalty may still help).

We'll look at the second approach next.

Neural Network Models

Many machine learning methods have been inspired by biology, and in particular the operation of the brain.

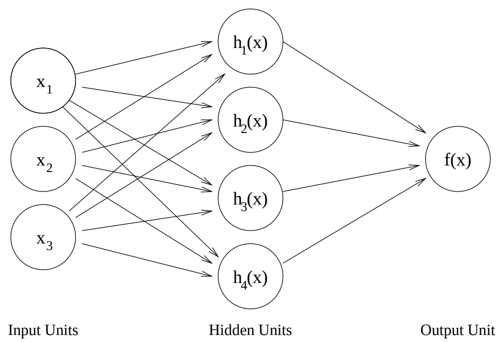
The brain has about 10^{11} *neurons*, each of which communicates with up to about 10^4 other neurons. We know that the brain learns many things very well, so it's plausible that machine learning methods with similar architectures may work well.

For supervised learning, a popular scheme of this sort is the *multilayer perceptron* (MLP), or *backprop* network, developed by Geoffrey Hinton and David Rumelhart in the early 80's.

It defines extra functions of the inputs called "hidden features", computed by "neurons", which are then linearly combined. I'll call these artificial neurons "units", since they aren't closely connected with real neurons.

The Architecture of a Multilayer Perceptron Network

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



Each 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, so that the hidden features linearly combined at the output can themselves be computed from linear combinations of earlier hidden features.

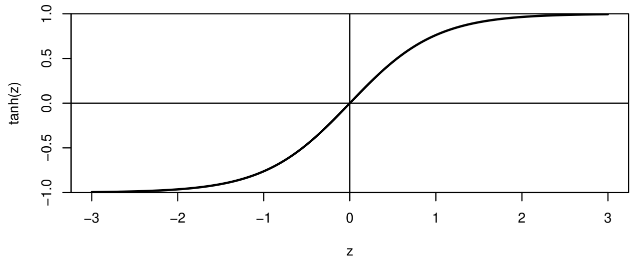
The Function Computed by the Network

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

$$f(x) \;=\; \beta_0 \;+\; \sum_{k=1}^{q} \beta_k h_k(x), \qquad h_k(x) \;=\; a(\gamma_{k,0} \;+\; \sum_{j=1}^{p} \gamma_{k,j} x_j)$$

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

A traditional choice of activation function is $\tanh(z) = (e^z - e^{-z}) \,/\, (e^z + e^{-z})$, which looks like this:



Defining a Distribution Using the Network Output

Our goal for supervised learning is to model $P(y \mid x)$. We can do this using the function, $f(x)$, defined by a multilayer perceptron.

If y is real-valued, we can model it as Gaussian with mean $f(x)$ and some variance (typically assumed to be the same for all cases).

If y is binary, we can let $P(y=1 \mid x) \;=\; 1\,/\,(1+\exp(-f(x))$.

If y takes on values in $\{0,\ldots,C-1\}$, we can extend the network to have C outputs, defining functions $f_c(x)$ for $c=0,\ldots,C-1$, and then let

$$P(y=c \mid x) \;=\; \frac{\exp(f_c(x))}{\sum\limits_{c'=0}^{C-1}\exp(f_{c'}(x))}$$

Training the Network by Maximum Likelihood

The simplest training procedure for an MLP network is to simply adjust the parameters — the βs and γs — to maximize some measure of fit to the training data. The most obvious measure of fit is likelihood.

Suppose the inputs in the training cases are $x^{(1)},\ldots,x^{(n)}$, and the corresponding targets are $y^{(1)},\ldots,y^{(n)}$, and that these cases are independent. If the targets are binary, the log of the likelihood can be written as

$$\log\prod_{i=1}^{n}\Big[1+\exp(-(2y^{(i)}-1)f(x^{(i)}))\Big]^{-1} \;=\; -\sum_{i=1}^{n}\log\Big[1+\exp(-(2y^{(i)}-1)f(x^{(i)}))\Big]$$

This is a complicated function of the β and γ parameters. Often, there are hundreds, thousands, or tens of thousands of parameters.

There usually are many local maxima, and finding the global maximum is hopeless. Fortunately, results are often good when using a relatively good local maximum.

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:

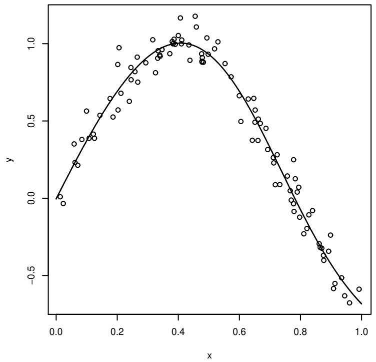
- Permuting the order of hidden units.
- For any hidden unit, k, negating β_k and $\gamma_{k,i}$ for all i.

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 $y=\sin(4x)+\text{noise}$, 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 y value over the range $(0,1)$ is about 0.003.



Here are the parameter values for the three fits:

$\gamma_{1,1}$	$\gamma_{2,1}$	$\gamma_{3,1}$	$\gamma_{1,0}$	$\gamma_{2,0}$	$\gamma_{3,0}$	β_1	β_2	β_3	β_0
-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