4 Nonlinear Regression and Multilayer Perceptron

In nonlinear regression the output variable y is no longer a linear function of the regression parameters plus additive noise. This means that estimation of the parameters is harder. It does not reduce to minimizing a convex energy functions – unlike the methods we described earlier.

The perceptron is an analogy to the neural networks in the brain (over-simplified). It receives a set of inputs $y = \sum_{j=1}^{d} \omega_j x_j + \omega_0$, see Figure (3).

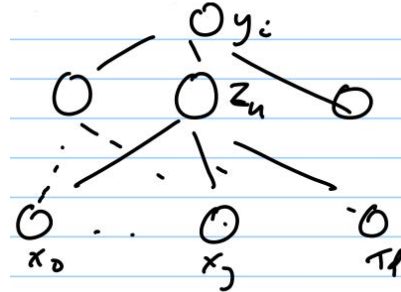

Figure 3: Idealized neuron implementing a perceptron.

It has a threshold function which can be *hard* or *soft*. The hard one is $\zeta(a) = 1$, if $a > 0$, $\zeta(a) = 0$, otherwise. The soft one is $y = \sigma(\vec{\omega}^T \vec{x}) = 1/(1 + e^{\vec{\omega}^T \vec{x}})$, where $\sigma(\cdot)$ is the *sigmoid function.*

There are a variety of different algorithms to train a perceptron from labeled examples.

Example: The quadratic error:
$E(\vec{\omega}|\vec{x}^t, y^t) = \frac{1}{2}(y^t - \vec{\omega} \cdot \vec{x}^t)^2$,
for which the update rule is $\Delta\omega_j^t = -\Delta \frac{\partial E}{\partial \omega_j} = +\Delta(y^t \vec{\omega} \cdot \vec{x}^t)\vec{x}^t$. Introducing the sigmoid function $r^t = \text{sigmoid}(\vec{\omega}^T \vec{x}^t)$, we have
$E(\vec{\omega}|\vec{x}^t, y^t) = -\sum_i \left\{ r_i^t \log y_i^t + (1 - r_i^t) \log(1 - y_i^t) \right\}$, and the update rule is
$\Delta\omega_j^t = -\eta(r^t - y^t)x_j^t$, where η is the learning factor. I.e, the update rule is the learning factor $\times$ (desired output – actual output)$\times$ input.

4.1 Multilayer Perceptrons

Multilayer perceptrons were developed to address the limitations of perceptrons (introduced in subsection 2.1) – i.e. you can only perform a limited set of classification problems, or regression problems, using a single perceptron. But you can do far more with multiple layers where the outputs of the perceptrons at the first layer are input to perceptrons at the second layer, and so on.

Two ingredients: (I) A standard perceptron has a discrete outcome, $sign(\vec{\omega} \cdot \vec{x}) \in \{\pm 1\}$. It is replaced by a graded, or *soft*, output $z_h = \sigma(\vec{\omega}_h \cdot \vec{x}) = 1/\{1 + e^{-\sum_{j=1}^{d}(\omega_{hj} \cdot x_j + \omega_{0j})}\}$, with $h = 1..H$. See figure (4). This makes the output a differentiable function of the weights $\vec{\omega}$.

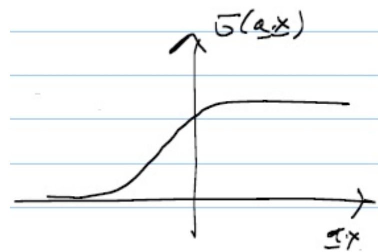

Figure 4: The sigmoid function of $(\vec{\omega}_h \cdot \vec{x})$ tends to 0 for small $(\vec{a} \cdot \vec{x})$ and tends to 1 for large $(\vec{a} \cdot \vec{x})$.

(II) Introduce hidden units, or equivalently, multiple layers, see figure (5).

The output is

$$y_i = \vec{\nu}_i^T z = \sum_h^H = \nu_{hi} z_h + \nu_{0i}.$$

Other output function can be used, e.g. $y_i = \sigma(\vec{\nu}_i^T \vec{z})$.

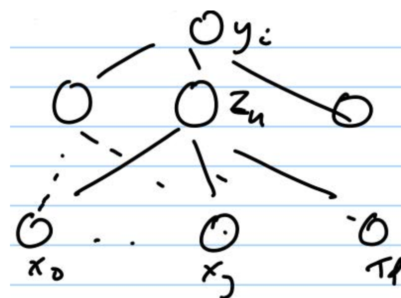

Figure 5: A multi-layer perceptron with input x's, hidden units z's, and outputs y's.

Many levels can be specified. What do the hidden units represent? Many people have tried to explain them but it is unclear. The number of hidden units is related to the capacity of the perceptron. Any input-output function can be represented as a multilayer perceptron with enough hidden units.

4.2 Training Multilayer Perceptrons

For training a multilayer perceptron we have to estimate the weights ω_{hj}, ν_{ij} of the perceptron. First we need an error function. It can be defined as:

$$E[\omega, \nu] = \sum_i \{y_i - \sum_h \nu_{ih} \sigma(\sum_j \omega_{hj} x_j)\}^2$$

The update terms are the derivatives of the error function with respect to the parameters:

$$\Delta\omega_{hj} = -\frac{\partial E}{\partial \omega_{hj}},$$

which is computed by the chain rule, and

$$\Delta\nu_{ih} = -\frac{\partial E}{\partial \nu_{ih}},$$

which is computed directly.

By defining $r_k = \sigma(\sum_j \omega_{kj} x_j)$, $\quad E = \sum_j (y_i - \sum_k \nu_{ik} r_k)^2$, we can write

$$\frac{\partial E}{\partial \omega_{kj}} = \sum_r \frac{\partial E}{\partial r_k} \cdot \frac{\partial r_k}{\partial \omega_{kj}},$$

where

$$\frac{\partial E}{\partial r_k} = -2 \sum_j (y_i - \sum_l \nu_{il} r_l) \nu_{ik},$$

$$\frac{\partial r_k}{\partial \omega_{kj}} = x_j \sigma'(\sum_j \omega_{kj} x_j),$$

$$\sigma'(z) = \frac{d}{dz}\sigma(z) = \sigma(z)\{1 - \sigma(z)\}.$$

Hence,

$$\frac{\partial E}{\partial \omega_{hj}} = -2 \sum_j (y_i - \sum_l \nu_{il} r_l) \nu_{ik} x_k \sigma(\sum_j \omega_{kj} x_j)\{1 - \sigma(\sum_j \omega_{kj} x_j)\},$$

where $\sum_j (y_i - \sum_l \nu_{il} r_l)$ is the error at the output layer, ν_{ik} is the weight k from middle layer to output layer.

This is called *backpropagation* The error at the output layer is propagated back to the nodes at the middle layer $\sum_j (y_i - \sum_l \nu_{il} r_l)$ where it is multiplied by the activity $r_k(1 - r_k)$ at that node, and by the activity x_j at the input.

4.2.1 Variants

One variant is learning in *batch mode*, which consists in putting all data into an energy function – i.e., to sum the errors over all the training data. The weights are updated according to the equations above, by summing over all the data.

Another variant is to do *online learning.* In this variant, at each time step you select an example (x^t, y^t) at random from a dataset, or from some source that keeps inputting exmaples, and perform one iteration of steepest descent using only that datapoint. I.e. in the update equations remove the summation over t. Then you select another datapoint at random, do another iteration of steepest descent, and so on. This variant is suitable for problems in which we keep on getting new input over time.

This is called stochastic descent (or Robins-Monroe) and has some nice properties including better convergence than the *batch method* described above. This is because selecting the datapoints at random introduces an element of stochasticity which prevents the algorithm from getting stuck in a local minimum (although the theorems for this require multiplying the update – the gradiant – by a terms that decreases slowly over time).

4.3 Critical issues

One big issue is the number of hidden units. This is the main design choice since the number of input and output units is determined by the problem.

Too many hidden units means that the model will have too many parameters – the weights ω, ν – and so will fail to generalize if there is not enough training data. Conversely, too few hidden units means restricts the class of input-output functions that the multilayer perceptron can represent, and hence prevents it from modeling the data correctly. This is the classic bias-variance dilemma (previous lecture).

A popular strategy is to have a large number of hidden units but to add a *regularizer* term that penalizes the strength of the weights, This can be done by adding an additional energy term:

$$\lambda \sum_{j,j} \omega_{hj}^2 + \sum_{i,h} \nu_{ih}^2$$

This term encourages the weights to be small and maybe even to be zero, unless the data says otherwise. Using an L^1-norm penalty term is even better for this.

Still, the number of hidden units is a question and in practice some of the most effective multilayer perceptrons are those in which the structure was hand designed (by trial and error).

4.4 Relation to Support Vector Machines

In a perceptron we get $y_i = \sum_h \nu_{ih} z_h$ at the output and at the hidden layer we get $z_h = \sum_j \sigma(\sum_h \omega_{hj} x_j)$ from the input layer.

Support Vector Machines (SVM) can also be represented in this way.

$$y = sign(\sum_{\mu} \alpha_\mu y_\mu \vec{x}_\mu \cdot \vec{x}),$$

with $\vec{x}_\mu \cdot \vec{x} = z_\mu$ the hidden units response, i.e, $y = sign(\sum_\mu \alpha_\mu y_\mu z_\mu)$.

An advantage of SVM is that the number of hidden units is given by the number of support vectors. $\{\alpha_\mu\}$ is specified by minimizing the primal problem, and there is a well defined algorithm to perform this minimization.