Lecture 6. Regression

Prof. Alan Yuille

Summer 2014

Outline

1. Introduction to Regression
2. Binary Regression
3. Linear Regression; Polynomial Regression
4. Non-linear Regression; Multilayer Perceptron

1 Introduction to Regression

Consider learning directly the conditional distribution $p(y|x)$.

This is often easier than learning the likelihood function $p(x|y)$ and the prior $p(y)$ separately. This is because the space of y is typically much lower-dimensional than the space of x. For example, if x is a 10×10 image (e.g., of a face, or a non-face) then this space has an enormous number of dimensions while, by contrast, the output y takes only binary values. It is much easier to learn distributions on lower-dimensional spaces.

The task of estimating y from x is called *regression.* It has a long history. Two hundred years ago it was invented by Gauss to estimate the position of the planetoid Ceres (Gauss's father encouraged him to do work on this problem saying that there was more money in Astronomy than in Mathematics).

In this lecture we address three different types of regression problem.

(I) *Binary regression.* Here $y \in \{\pm 1\}$. We can specify a distribution to be of exponential form (non-parametric ways of doing regression are possible, but we do not have time to discuss them):

$$p(y|x;\lambda) = \frac{e^{y\lambda\cdot\phi(x)}}{e^{\lambda\cdot\phi(x)} + e^{-\lambda\cdot\phi(x)}}.$$

Note that this is of form $p(y|x;\lambda) = \frac{e^{y\lambda\cdot\phi(x)}}{Z[\lambda,x]}$, and because y is binary valued we can compute the normalization term $Z[\lambda, x] = \sum_y e^{y\lambda\cdot\phi(x)} = e^{\lambda\cdot\phi(x)} + e^{-\lambda\cdot\phi(x)}$.

(II) *Linear Regression.* Here y takes a continuous set of values (we can extend this directly to allow y to be vector-valued).

$$p(y|x,\lambda) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(1/2\sigma^2)(y-\lambda\cdot\phi(x))^2},$$

where λ includes the variance σ^2.

This model assumes that the data can be expressed as $y = \lambda\cdot\phi(x)+\epsilon$, where $\lambda\cdot\phi(x)$ is a linear predictor (i.e. it depends on linear coefficients λ) and where ϵ is a random variable (i.e. noise) Gaussianly distributed with zero mean and variance σ^2. To see this, write $p(y|x,\epsilon) = \delta(y-\lambda\cdot\phi(x)+\epsilon)$, $p(\epsilon) = \frac{1}{\sqrt{2\pi}\sigma}\exp-\frac{\epsilon^2}{2\sigma^2}$ and compute $p(y|x) = \int d\epsilon p(y|x,\epsilon)p(\epsilon)$.

(III) *Non-Linear regression.* This is an extension of binary regression.

$$p(y|x) = \frac{1}{Z}e^{M(y,g(x:\lambda))},$$

where $M(\cdot,\cdot)$ is a similarity measure and $g(x:\lambda)$ is non linear in λ. This leads to non-linear optimization problems. It is more powerful but more computationally intensive. Important cases are multi-layer perceptrons and deep networks.

In both cases (I) and (II), the parameters λ can be estimated by Maximum Likelihood (ML) and, as in previous lectures, this corresponding to minimizing a convex energy function and, in some cases (e.g., case II), there will be an analytic expression for the solution. (Sometimes it is good to add a prior $P(\lambda)$ and do MAP estimation, and sometimes a loss function can be added also). In the case (III) the maximum likelihood cannot be solved analytically. An algorithm will be necessary to solve a non-convex optimization problem.

2 Binary Regression

$$p(y|\vec{x};\vec{\lambda}) = \frac{e^{y\vec{\lambda}\cdot\vec{\phi}(\vec{x})}}{e^{\vec{\lambda}\cdot\vec{\phi}(\vec{x})} + e^{-\vec{\lambda}\cdot\vec{\phi}(\vec{x})}}.$$

Note that this corresponds to a decision rule $\hat{y} = sign(\vec{\lambda}\cdot\vec{\phi}(\vec{x}))$. Or, equivalently, $\hat{y}(\vec{x}) = \arg\max_y y\vec{\lambda}\cdot\vec{\phi}(\vec{x})$. We obtain this here by taking the log-likelihood ratio $\log\frac{p(y=1|\vec{x};\vec{\lambda})}{p(y=-1|\vec{x};\vec{\lambda})}$.

To perform ML on this model we need to minimize:

$$F(\vec{\lambda}) = -\sum_{i=1}^{N}\log p(y_i|\vec{x}_i;\vec{\lambda}) = -\sum_{i=1}^{N} y_i\lambda\cdot\vec{\phi}(\vec{x}_i) + \sum_{i=1}^{N}\log\{e^{\vec{\lambda}\cdot\vec{\phi}(\vec{x})} + e^{-\vec{\lambda}\cdot\vec{\phi}(\vec{x})}\},$$

where $\mathcal{X} = \{(\vec{x}_i, y_i) : i = 1,...,N\}$ is the training dataset.

It can be checked that $F(\vec{\lambda})$ is a convex function of $\vec{\lambda}$ (compute the Hessian, then use Cauchy-Schwartz to show it is positive semi-definite.).

The gradient of $F(\vec{\lambda})$ with respect to $\vec{\lambda}$ can be computed to be:

$$\frac{\partial F}{\partial \vec{\lambda}} = -\sum_{i=1}^{N} y_i \vec{\phi}(\vec{x}_i) + \sum_{i=1}^{N} \sum_{y \in \{\pm 1\}} y \vec{\phi}(\vec{x}_i) p(y|\vec{x}_i, \vec{\lambda}). \tag{1}$$

Hence the ML estimate – at $\hat{\vec{\lambda}}$, such that $\frac{\partial F}{\partial \vec{\lambda}}(\hat{\vec{\lambda}}) = 0$ – balances the statistics of the data (first term in equation (1) –with the model statistics (second term in equation (1) where the expected over x is based on the data (i.e. regression only learns a probability model for y and not for x).

Usually we cannot solve equation (1) analytically to solve for $\hat{\lambda}$. Instead, we can solve for λ by doing steepest descent (is there an analogy to GIS? check! yes, easy to derive one). I.e.

$$\begin{aligned} \vec{\lambda}^{t+1} &= \vec{\lambda}^{t} - \Delta\{-\log p(y|\vec{x}, \vec{\lambda})\} \\ &= \vec{\lambda}^{t} - \Delta\{-\sum_{i=1}^{N} y_i \vec{\phi}(\vec{x}_i) + \sum_{i=1}^{N} \sum_{y \in \{\pm 1\}} y \vec{\phi}(x_i) p(y|\vec{x}_i, \vec{\lambda})\}. \end{aligned}$$

2.1 Special Case of Binary Regression: the Artificial Neuron

An artificial model of a neuron is obtained by setting $\vec{\phi}(\vec{x}) = (x_1, ..., x_M)$, where the x_i are scalar values. This is illustrated in figure (1). In this case $\vec{\lambda} \cdot \vec{\phi}(\vec{x}) = \sum_{i=1}^{M} \lambda_i x_i$. The x_i are thought of as the input to the neuron and their strength in weighted by the synaptic strength λ_i. The weighted inputs are summed and at the cell body, the soma, the artificial neuron fires with probability $p(y = 1|\vec{x})$, given by:

$$p(y|\vec{x}) = \frac{e^{y\vec{\lambda} \cdot \vec{\phi}(\vec{x})}}{e^{\vec{\lambda} \cdot \vec{\phi}(\vec{x})} + e^{-\vec{\lambda} \cdot \vec{\phi}(\vec{x})}}$$

This is called integrate-and-fire. In practice, we can add another term λ_0 to the summation which acts as a threshold for firing (i.e. $\vec{\phi}(\vec{x}) = (1, x_1, ..., x_M)$ and $\vec{\lambda} = (\lambda_0, \lambda_1, ..., \lambda_M)$). In this case, $\vec{\lambda} \cdot \vec{\phi}(\vec{x}) > 0$, i.e., $\sum_{i=1}^{M} \lambda_i x_i > -\lambda_0$

3 Gaussian Linear Regression

Now, consider continuous linear regression with a Gaussian model. Here y is a scalar output (a continuous number).

$$y = \vec{\lambda} \cdot \vec{\phi}(\vec{x}) + \epsilon,$$

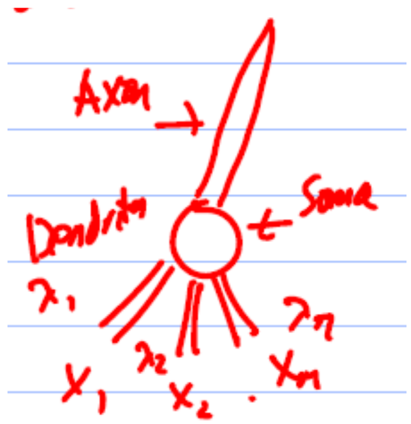

Figure 1: An artificial model of a neuron. The inputs are $x_1, ..., x_M$ at the dendrites, the synaptic strengths are $\lambda_1, ..., \lambda_M$, the cell body (soma) calculates the weighted sum of the inputs $\sum_{i=1}^M \lambda_i x_i$ and fires a spike down the axon with probability $p(y = 1|x)$. This provides input to another neuron..

where ϵ is a noise term which we can model with Gaussian: $p(\epsilon) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\epsilon^2}{2\sigma^2}}$ The probability distribution of y is

$$p(y|\vec{x}, \vec{\lambda}) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y - \vec{\lambda} \cdot \vec{\phi}(\vec{x}))^2}{2\sigma^2}}.$$

Maximum Likelihood (ML) estimation minimizes:

$$F(\vec{\lambda}, \sigma) = \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \vec{\lambda} \cdot \vec{\phi}(\vec{x}_i))^2 + N \log \sqrt{2\pi}\sigma.$$

We can minimize, and obtain analytic expressions for $\hat{\vec{\lambda}}$ and $\hat{\sigma}^2$ by differentiating $F(\vec{\lambda}, \sigma)$ with respect to $\vec{\lambda}$ and σ and setting the derivatives to be zero.

This gives an analytic solution for $\hat{\vec{\lambda}}$:

$$-\frac{1}{\sigma^2} \sum_{i=1}^{N} (y_i - \vec{\lambda} \cdot \vec{\phi}(\vec{x}_i)) \vec{\phi}(\vec{x}_i) = 0$$

$$\hat{\vec{\lambda}} = \{\sum_{i=1}^{N} \phi(\vec{x}_i) \phi(\vec{x}_i)^T\}^{-1} \sum_{i=1}^{N} y_i \vec{\phi}(\vec{x}_i),$$

where T denotes vector transpose and $^{-1}$ denotes matrix inverse. To see this, write $F(\vec{\lambda})$ using coordinate summation for the dot product terms – i.e. $(y_i - \vec{\lambda} \cdot \vec{\phi}_b(\vec{x}_i))^2 = (y_i - \sum_a \vec{\lambda}_a \vec{\phi}_a(\vec{x}_i))^2$. Then the solution is $\hat{\lambda}_a = \{\sum_{i=1}^N \phi_a(x_i)\phi_b(x_i)\}^{-1} \sum_{i=1}^N y_i \phi_a(x_i)$, where we are taking the inverse of the matrix with row and column entries indexed by a and b.

We also get an analytic solution for $\hat{\sigma}$:

$$\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{\vec{\lambda}} \cdot \vec{\phi}(\vec{x}_i)).$$

Hence we use MLE to estimate the regression coefficients $\hat{\vec{\lambda}}$ and the variance $\hat{\sigma}^2$.

This can be generalized to allow for vector-valued output.

3.1 L^1 Variant of Linear Regression

The linear regression model assumes that the noise (i.e. ϵ) is additive zero-mean Gaussian. But we know that Gaussians are non-robust to outliers. An alternative, which also leads to a convex ML estimation problem, is to use a Laplacian distribution. This replaces the quadratic, or L^2, term in the exponent of the Gaussian by a modulus, or L^1, term.

So set $y = \vec{\lambda} \cdot \vec{\phi}(\vec{x}) + \epsilon$ where $P(\epsilon) = \frac{1}{2\sigma} e^{-|\epsilon|/\sigma}$. Here σ is a parameter of the model, it is not a standard deviation. (It is given by the integral $\int_0^\infty e^{-\epsilon/\sigma} d\epsilon = [-\sigma e^{-\epsilon/\sigma}]_0^\infty = \sigma$).

This gives:

$$P(y|\vec{x}; \vec{\lambda}, \vec{\sigma}) = \frac{1}{2\sigma} e^{-|y - \vec{\lambda} \cdot \vec{\phi}(\vec{x})|/\sigma}.$$

Estimating $\vec{\lambda}$ and σ by ML corresponds to minimizing the expression:

$$-\sum_{i=1}^{N} \log p(y_i|x; \vec{\lambda}) = \frac{1}{\sigma}\sum_{i=1}^{N} |y_i - \vec{\lambda} \cdot \vec{\phi}(\vec{x}_i)| + N \log(2\sigma).$$

$$\hat{\vec{\lambda}} = \arg\min_{\vec{\lambda}} \sum_{i=1}^{n} |y_i - \vec{\lambda} \cdot \vec{\phi}(\vec{x}_i)|,$$

which requires minimizing a convex function which can be done by steepest descent or a variety of iterative algorithms. The ML estimate of σ is given by (after differentiating the ML criterion with respect to σ and setting the derivative to be 0):

$$\hat{\vec{\sigma}} = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{\vec{\lambda}} \cdot \vec{\phi}(\vec{x}_i)|.$$

This requires more computation than linear regression – because we can no longer solve for λ analytically (i.e. by linear algebra) and instead must do steepest descent (but this can be done very fast nowadays).

It is generally far more robust to outliers than the linear model, which assumes that the noise is Gaussian. This is because the L^1 norm penalizes errors by their magnitude while the L^2 norm (used in Gaussians in the exponent) penalizes errors by the square of their magnitude, which makes it much more sensitive to outliers (which will adjust the parameters of the model by avoiding these huge penalties – imagine how your behaviour would change if you paid a large penalty, like a month in jail, for parking a car in the wrong place).

3.2 Linear regression examples

Let's define a regression model in a d-dimensional space given by:

$$\vec{\lambda} \cdot \vec{\phi}(\vec{x}) = \omega_0 + \omega_1 x_1 + \cdots + \omega_d x_d = \sum_{j=1}^{d} \omega_j x_j + \omega_0$$

$$\vec{\lambda} = (\omega_0, \omega_1, \cdots, \omega_d), \qquad \vec{\phi}(\vec{x}) = (1, x_1, \cdots, x_d)$$

In one of the simplest cases we can have a linear model

$$y = \omega_1 x + \omega_0 + \epsilon$$

with 1-dimensional x. In this case the dataset is $\mathcal{X}_N = \{(x^i, y^i) : i = 1..N\}$. Here y^i doesn't denote a class label, but a desired continuous value, and i is the number of data point in $\mathcal{X}_N$.

The error of the model given the dataset is:

$$E(\omega_1, \omega_0 | \mathcal{X}_N) = \sum_{i=1}^{N} \{y^i - (\omega_1 x^i + \omega_0)\}^2$$

Minimizing the error with respect to the model parameters: $\dfrac{\partial E}{\partial \omega_1} = 0$, $\dfrac{\partial E}{\partial \omega_0} = 0$. The solution is:

$$\begin{aligned} \hat{\omega}_1 &= \frac{\sum_{i=1}^{N} x^i y^i - \overline{x}\,\overline{y} N}{\sum_{i=1}^{N} (x^i)^2 - N\overline{x}^2)} \\ \hat{\omega}_0 &= \overline{y} \sum_{i=1}^{N} (x^i)^2 \overline{x} \sum_{i=1}^{N} x^i y^i \overline{\sum_{i=1}^{N} (x^i)^2 - N\overline{x}^2} \end{aligned}$$

where $\overline{x} = \sum_{i=1}^{N} x_i / N$ and $\overline{y} = \sum_{i=1}^{N} y_i / N$ denote the means.

A "richer" model can be used, but too high an order follow the data too closely, as shown in Figure (2). E.g, a second order model is $\vec{\lambda}\vec{\phi}(\vec{x}) = \omega_2 x x + \omega_1 x + \omega_0 + \epsilon$.

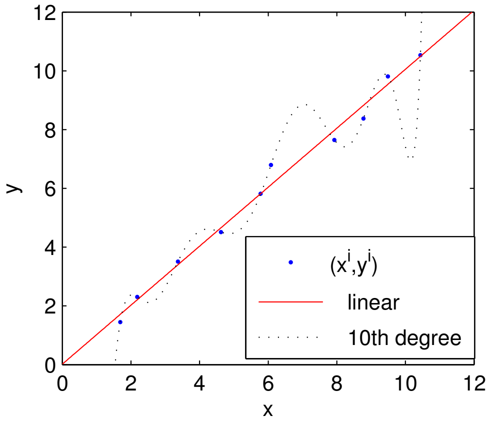

Figure 2: Regression with two models: linear and 10-th order polynomial. The richer model may be overfitting the data, i.e, will not generalize to new data generated from the same distribution.

3.3 More abstract example

In linear regression we have a linear model $\vec{\lambda} \cdot \vec{\phi}(x) = \omega_1 x + \omega_0$. Differentiating the energy (a quadratic function) w.r.t. ω_1, ω_0 and setting the derivatives to zero gives two equations.

$$\sum_i y^i = N\omega_0 + \omega_1 \sum_i x^i$$

$$\sum_i y^i x^i = \omega_0 \sum_i x^i + \omega_1 \sum_i (x^i)^2$$

Expressed in linear algebra form as $\mathbf{A}\vec{\omega} = \vec{d}$

$$\mathbf{A} = \left[\begin{array}{cc} N & \sum_t x^i \\ \sum_t x^i & \sum_t (x^i)^2 \end{array} \right],$$

$$\vec{\omega} = \left[\begin{array}{c} \omega_0 \\ \omega_1 \end{array} \right],$$

$$\vec{d} = \left[\begin{array}{c} \sum_t y^i \\ \sum_t y^i x^i \end{array} \right]$$

solved to give $\vec{\omega} = \mathbf{A}^{-1}\vec{d}$.

More generally, for Polynomial Regression the model is

$$g(x^i|\omega_k, \ldots, \omega_2, \omega_1, \omega_0) = \omega_K (x^i)^K + \cdots + \omega_1 x^i + \omega_0$$

with $K+1$ parameters $\omega_K, \ldots, \omega_0$.

Differentiating the energy gives $K+1$ linear equations in $K+1$ variables.

$$\mathbf{A}\vec{\omega} = \vec{d},$$

where $\mathbf{A}$ is a $(k+1) \times (k+1)$ dimensional matrix with components $A_{kl} = \sum_i (x^i)^k (x^i)^l$ and $\vec{d}$ is a $(k+1)$ dimensional vector with components $d_k = \sum_i y^i (x^i)^k$.

We can write $\mathbf{A} = \mathbf{D}^T\mathbf{D}$, where $\mathbf{D}$ is an $N \times (K+1)$ dimensional matrix with components $D_{il} = (x^i)^l$, and $\vec{d} = \mathbf{D}^T\vec{y}$, where $\vec{y}$ is an N-dimensional vector with components $y_i = y^i$. We can express $\mathbf{D}$ and $\vec{y}$ by:

$$\mathbf{D} = \begin{bmatrix} 1 & x_1 & \ldots & x_1^k \\ 1 & x_2 & \ldots & x_2^k \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix}, \vec{y} = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^N \end{bmatrix}$$

and solve to get $\vec{\omega} = (\mathbf{D}^T\mathbf{D})^{-1}\mathbf{D}^T\vec{r}$.

We must adjust the complexity of the model to the amount of data available. The complexity of polynomial regression is number of parameters k, so one needs to pick k to give best generalization error.

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).

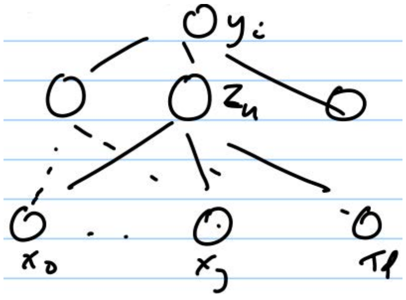

Figure 3: Idealized neuron implementing a perceptron.

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 × (desired output – actual output)× 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 $\vec{\omega}$.

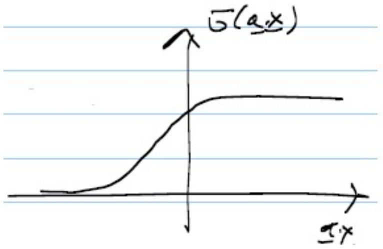

Figure 4: The sigmoid function of $(\vec{\omega}_h \cdot \vec{x})$ tends to 0 for small $(\vec{a} \cdot \vec{x})$ and tends to 1 for large $(\vec{a} \cdot \vec{x})$.

(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. $y_i = \sigma(\vec{\nu}_i^T \vec{z})$.

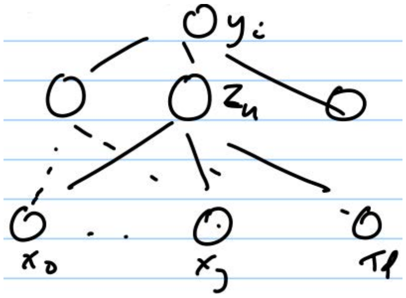

Figure 5: A multi-layer perceptron with input x's, hidden units z's, and outputs y's.

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.