Probabilistic Theories of the Visual Cortex

A.L. Yuille and D. Kersten

Abstract

Learning and Regression

The last lecture described two types of models for groups of neurons – the Boltzmann Machine and the Multilayer Perceptron. We described inference algorithms for these models which assumed that the model parameters were known. In this lecture we describe how to learn the parameters.

We start with Boltzmann Machines (BM). This is an unsupervised learning task since the BM has hidden states which cannot be observed. In this case, the learning algorithm must infer the states of the hidden variables in addition to learning the model parameters. This requires methods like the expectation maximization (EM) algorithm or by stochastic sampling (extension of Gibbs sampling). We also consider the special case of restricted Boltzmann machines (RBM), which greatly simplifies the inference, and which sets the stage for one type of deep learning (obtained by learning a sequence of RBM's).

Next, we consider regression. This is called supervised learning since there are no hidden variables (as variables are specified during training). This gives learning algorithms for multilayer perceptrons – namely the backpropagation algorithm.

A. Boltxmann Machines and Restricted Boltzmann Machines

Recall that the Boltzmann machine (BM) represents a group of neurons by a distribution of form:

$$P_{BM}(\vec{x},\vec{I}) = \frac{1}{Z}\exp\{\sum_{ij} T_{ij}x_i I_j + (1/2)\sum_{ij}\theta_{ij}x_i x_j\}. \tag{1}$$

The neurons with states $\{x_i\}$ are hidden units which are unobserved (but the $\{I_i\}$ are observed inputs). The term $\sum_{ij}\theta_{ij}x_i x_j$ specifies interactions between the hidden units.

The restricted Boltzmann machines (RBM) is a simplification with no interactions between the hidden units, $\theta_{ij}=0,\ \forall i,j$:

$$P_{RBM}(\vec{x}|\vec{I}) = \frac{1}{Z_T}\exp\{\sum_{ij} T_{ij}x_i I_j\} = \prod_{i=1}^{n} P(x_i|\vec{I}), \tag{2}$$

where $P(x_i|\vec{I}) = (1/Z)\exp\{x_i\sum_j T_{ij}I_j\}$. Hence the RBM distribution can be factorized which make it much easier to do inference and learning than for the BM.

This lecture will describe to do learning from both classes of models – i.e. how to estimate the model parameters/weights $\{T_{ij},\theta_{ij}\}$ given a set $\{\vec{x}^{\mu} : \mu = 1,...,N\}$ of training examples.

1) Inference on BMs and RBMs; revision: But first we need to review the inference algorithms for each model – this enables us to estimate quantities such as $\arg\max P(\vec{x}|\vec{I})$ or $\sum_{\vec{x}}\vec{x}P(\vec{x}|\vec{I})$. This is because it is impossible to do learning without an inference algorithm. Intuitively, learning a model requires adjusting the parameters of the model until it gives a good fit to the data, which requires performing inference.

Inference algorithms for BM were described in the last lecture. Recall that we described how to draw samples $\vec{x}$ from the BM distribution using Gibbs sampling (stochastic neural dynamics) or, alternatively, we could approximately estimate properties like $\sum_{\vec{x}}\vec{x}P(\vec{x}|\vec{I})$ by using mean field theory algorithms (dynamic neural network models). But these algorithms will only be effective for certain classes of BM's depending on the form of the weights $\{T_{ij}\}$. For certain BMs Gibbs sampling will take an extremely long time to converge while mean field theory will get stuck in a local minimum.

Inference for the RBM is much simpler. Because the distribution $P(\vec{x}|\vec{I})$ factorizes as $\prod P(x_i|\vec{I})$. Hence we can directly obtain samples $\vec{x}$ by independently sampling their components x_i from $P(x_i|\vec{I})$. Similarly we can compute the means $\sum_{\vec{x}} \vec{x} P(\vec{x}|\vec{I})$ by directly computing their components $\sum_{x_i} x_i P(x_i|\vec{I})$.

2) Learning BMs and RBMs: In order to do learning we must make the learning variables explicit in the BM distribution by re-expressing it as $P_{BM}(\vec{x}, \vec{I}|T, \theta)$. For simplicity of notation we set $\lambda = (T, \theta)$ and express $E(x, I) = \lambda \cdot \phi(x, I)$ (we also drop the vector notation for x and I).

Suppose we have training data $\{I^\mu : \mu = 1, ..., N\}$. Maximum likelihood estimating λ from the marginal distribution $P(I|\lambda) = \sum_x P(I, x|\lambda)$, which requires summing out the hidden variables x (we need hidden variables x^μ for each training example I^μ). Then ML estimates λ^*:

$$\lambda^* = \arg\max_\lambda \prod_{\mu=1}^{N} \sum_{x^\mu} P(x^\mu, I^\mu|\lambda). \tag{3}$$

This type of estimation problem is usually addressed by the estimation-minimization (EM) algorithm. This introduces additional variables $\{q^\mu(x^\mu) : \mu = 1, ..., N\}$, which are distributions over the hidden variables $\{x^\mu : \mu = 1, ..., N\}$. EM proceeds by estimating the q and the λ in alternation by the following two steps:

$$q_\mu^{t+1} = P(x^\mu|I^\mu, \lambda^{t+1}),\ \mu = 1, ..., N$$
$$\lambda^{t+1} \text{ solves } (1/N)\sum_\mu \sum_{x^\mu} q_\mu(x^\mu)\phi(x^\mu, I^\mu) = \sum_{x,I} \phi(x, I)\frac{e^{\lambda\cdot\phi(x,I)}}{\sum_{x,I} e^{\lambda\cdot\phi(x,I)}}. \tag{4}$$

This algorithm is guaranteed to converge to a local minimum of the maximum likelihood criterion.

It is not easy to perform these steps for the Boltsmann Machine. But we can estimate them by using stochastic sampling. Firstly, we can perform the first step by using Gibbs sampling to get samples from $P(x^\mu|I^\mu, \lambda^{t+1})$ and hence estimate q_μ^{t+1}.

To calculate the second step is more complicated. The equation can be formulated as finding the minimum of the (convex) function $\log Z[\lambda] - (1/N)\lambda \cdot \sum_\mu \sum_{x^\mu} q_\mu(x^\mu)\phi(x^\mu, I)$, where $\log Z[\lambda] = \sum_{x,I} e^{\lambda\cdot\phi(x,I)}$. This can be performed by steepest descent (or by generalized iterative scaling) using update equations such as:

$$\lambda^{t+1} = \lambda^t + \Delta\{(1/N)\sum_\mu \sum_{x^\mu} q^\mu(x^\mu)\phi(x^\mu, I^\mu) - \sum_{x,I}\phi(x, I)\frac{e^{\lambda\cdot\phi(x,I)}}{\sum_{x,I} e^{\lambda\cdot\phi(x,I)}}\}. \tag{5}$$

The terms on the right-hand side can be estimated by stochastic sampling.

Hence performing EM for the BM is complex (and there are further approximations not mentioned here). Estimating the parameters λ is much easier for the RBM. The sampling can be done directly (ie. without needing MCMC like Gibbs).

Note: we can also learn by doing Gibbs sampling for the combination (h, λ), This requires specifying a prior distribution $P(\lambda)$ (which could be the uniform distribution). Then sampling alternatively from $P(\lambda|I, x)$ and $P(x|\lambda, I)$ (this is an extension of Gibbs sampling).

Note: Deep belief networks can be constructed by stacking RBMs on top of each other so that the output of one RBM is the input to the next RMB. The RBMs are learnt one after the other in a greedy manner (then there is a final stage where the parameters/weights of all layers are adjusted). This will be described in later lectures.

B. Regression

Now consider the case where the training data includes input and output pairs. 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 encourage him to do work on this problem saying that there was more money in Astronomy than in Mathematics).

Note: the notation is changed in this section. x is used as input and y as output.

We consider three types of regression problems.

(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 $x = \lambda\cdot\phi(x) + \epsilon$, 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.

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).

(III) Non-Linear regression – multi-layer perceptron.

The general form of non-linear regression assumes that $y = f(x,\lambda) + \epsilon$, where $f(x,\lambda)$ is a non-linear function of λ and ϵ may be Gaussian or non-Gaussian.

An important case is multi-layer perceptons.

1) Binary Regression:

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

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

To perform ML on this model we need to minimize:

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

where $\mathcal{X} = \{(x_i,y_i)\text{"}i=1,...,N\}$ is the training dataset.

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

The gradient of $F(\lambda)$ can be computed to be:

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

Hence the ML estimate – at $\hat{\lambda}$ such that $\frac{\partial F}{\partial \lambda}(\hat{\lambda}) = 0$ – balances the statistics of the data (left hand side of equation (-B1) –with the model statistics (right hand size of equation (-B1) 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 (-B1) 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.

$$\lambda^{t+1} = \lambda^t - \Delta\{-\sum_{i=1}^N y_i \phi(x_i) + \sum_{i=1}^N \sum_{y \in \{\pm 1\}} y \phi(x_i) p(y|x_i, \lambda)\}.$$

2) Special Case of Binary Regression: the Artificial Neuron: An artificial model of a neuron is obtained by setting $\phi(x) = (x_1, ..., x_n)$, where the x_i are scalar values. This is illustrated in figure (1). In this case $\lambda \cdot \phi(x) = \sum_{i=1}^n \lambda_i x_i$. 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 given by $p(y = 1|x)$. This is called integrate-and-fire. In practice, we can can another term λ_0 to the summation which acts as a threshold for firing (i.e. $\phi(x) = (1, x_1, ..., x_n)$ and $\lambda = (\lambda_0, \lambda_1, ..., \lambda_n)$.

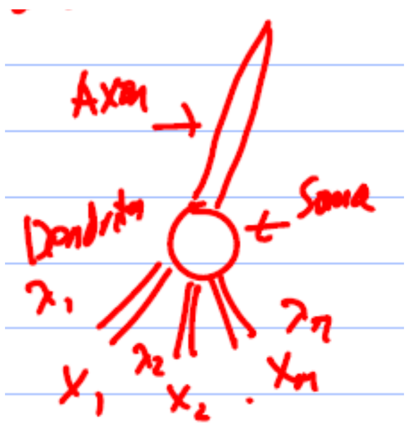

Fig. 1. An artificial model of a neuron. The inputs are $x_1, ..., x_n$ at the dendrites, the synaptic strengths are $\lambda_1, ..., \lambda_n$, the cell body (soma) calculates the weighted sum of the inputs $\sum_{i=1}^n \lambda_i x_i$ and fires a spike down the axon with probability $p(y = 1|x)$.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

3) Linear Regression: Now consider linear regression. Here y is a scalar output (a continuous number). Straightforward to extend this to vector-valued outputs.

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

ML estimation minimizes:

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

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

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

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

where T denotes vector transpose and $^{-1}$ denotes matrix inverse. To see this, write $F(\lambda)$ using coordinate summation for the dot product terms – i.e. $(y_i - \lambda \cdot \phi(x_i))^2 = (y_i - \sum_a \lambda_a \phi_a(x_i))^2$. 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 $\hat{\sigma}$:

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

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

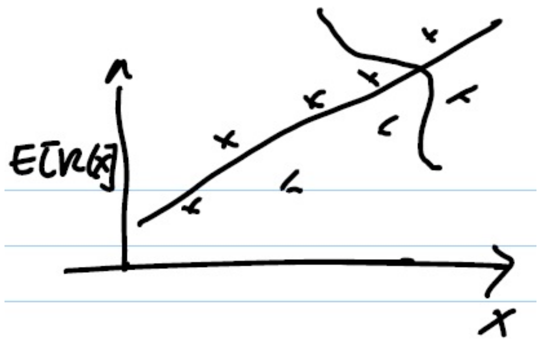

Fig. 2. Regression seeks to fit an estimator function, here a straight line, to the data – $y = \omega_0 + \omega_1 x + \epsilon$.

4) Background to Linear Regression Example:

In general, we express the dependent variable (y) as a function of the input (independent variable).

$y = f(x) + \epsilon$, where y : output , $f(x)$: unknown function of input, ϵ: random noise.

Want to approximate $f(x)$ by an estimator $g(x|\theta), \theta$- unknown parameters.

Standard assumption: $\epsilon \sim N(0, \sigma^2)$, where $N(\mu, \sigma^2)$ is a Gaussian with mean μ and variance σ^2. This corresponds to a distribution $p(y|x) = N(g(x|\theta), \sigma^2)$.

Use ML to learn the parameter θ. $p(y|x) \propto p(y|x)p(x)$, $\mathcal{X} = \{x^t, y^t\}_{t=1}^N$.

5) Log likelihood: (alternative formulation):

$$\mathcal{L}(\theta|X) = \log \prod_{t=1}^N p(x^t, y^t) = \sum_{t=1}^N \log p(y^t|x^t) + \sum_{t=1}^N \log p(x^t).$$

$$\mathcal{L}(\theta|X)_i = -N \log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2} \sum_{t=1}^N [y^t - g(x^t|\theta)]^2$$

Max w.r.t. θ is equivalent to min. $[y^t - g(x^t|\theta)]^2$. This is the least square estimates (Gaussian distribution $\rightarrow$ quadratic minimization).

6) Linear Regression Examples: Linear regression : $g(x^t|\omega_1, \omega_0) = \omega_1 x^t + \omega_0$

Differentiate energy (quadratic function) w.r.t. ω_1, ω_0 gives two equations.

$\sum_t y^t = N\omega_0 + \omega_1 \sum_t x^t$

$\sum_t y^t x^t = \omega_0 \sum_t x^t + \omega_1 \sum_t (x^t)^2$

Expressed in linear algebra form as $\underline{A\omega} = \underline{z}$

$$\underline{A} = \begin{bmatrix} N & \sum_t x^t \\ \sum_t x^t & \sum_t (x^t)^2 \end{bmatrix}, \underline{\omega} = \begin{bmatrix} \omega_0 \\ \omega_1 \end{bmatrix}, \underline{z} = \begin{bmatrix} \sum_t y^t \\ \sum_t y^t x^t \end{bmatrix}$$

Solved to give $\underline{\omega} = \underline{A}^{-1}\underline{z}$

More generally, Polynomial Regression.

$g(x^t|\omega_k, \dots, \omega_2, \omega_1, \omega_0) = \omega_k (x^t)^k + \cdots + \omega_1 x^t + \omega_0$

k+1 parameters $\omega_k, \dots, \omega_0$

Diff. energy - gives k+1 linear equ's in k+1 variables.

$\underline{A\omega} = \underline{z}$

Can write $\underline{A} = \underline{D}^T \underline{D}, \underline{z} = \underline{D}^T \underline{y}$

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

solve to get $\underline{\omega} = (D^T D)^{-1} D^T \underline{y}$

Must adjust the complexity of the model to the amount of data available.
Complexity of polynomial regression is number of parameters k.
Need to pick k to give best generalization error.

7) Nonlinear Regression: Multilevel Perceptrons: 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.

Multilayer perceptrons were developed to address the limitations of perceptrons – 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, $sign(\underline{a} \cdot \underline{x}) \in \{\pm 1\}$. So replace it by a graded, or *soft*, output $\sigma_T(\underline{a} \cdot \underline{x}) = \frac{1}{1+e^{-(\underline{a}\cdot\underline{x})/T}}$, see figure (3). This makes the output a differentiable function of the weights $\underline{a}$. Note : $\sigma_T(\underline{a} \cdot \underline{x}) \rightarrow$ step function as $T \rightarrow 0$. (II) Introduce hidden units, or equivalently, multiple layers, see figure (4).

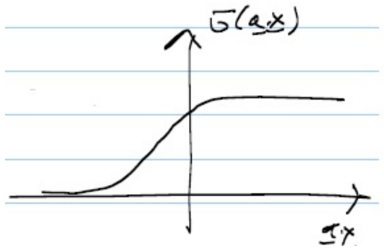

Fig. 3. The sigmoid function of $(\vec{a} \cdot \vec{x})$ tends to 0 for small $(\vec{a} \cdot \vec{x})$ and tends to 1 for large $(\vec{a} \cdot \vec{x})$. As T tends to 0, the sigmoid tends to a step function – i.e. 1 if $(\vec{a} \cdot \vec{x}) > 0$ and 0 if $(\vec{a} \cdot \vec{x}) < 0$.

$h_a = \sigma(\sum_i \tau_{ai} x_i)$

$\omega_\alpha = \sigma(\sum_b \Omega_{\alpha b} h_b)$

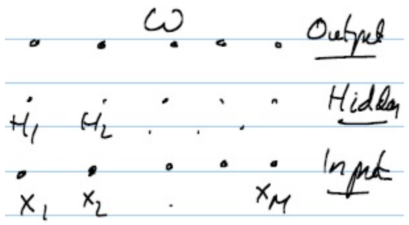

Fig. 4. A multi-layer perceptron with input x's, hidden units h's, and outputs y's.

Represent the full output as:

$$y_\alpha = \sigma(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i))$$

. This is a differentiable function of the parameters Ω, τ.

8) Teaching a Multilayer Perceptrons: Train the system using a set of labelled samples$\{(\underline{x}_\mu, \underline{y}_\mu) : \mu =$1 to N$\}$. Here $\underline{x}_\mu = (x_1^\mu, \ldots, x_\mu^\mu)$ the input units, and $\underline{y}_\mu = (y_1^\mu, \ldots, y_N^\mu)$ are the output units.

Note: we allow multiple classes (i.e. not joint two classes).

We can treat this as a regression problem where we assume additive Gaussian noise. This gives a quadratic energy function:

$$E[\langle\Omega\rangle, \langle\tau\rangle] = \frac{1}{N} \sum_{\mu=1}^{N} \sum_{\alpha=1}^{M} \{y_\alpha^\mu - \sigma(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu))\}^2$$

$E[\langle\Omega\rangle, \langle\tau\rangle]]$ is a non-convex function of $\langle\Omega\rangle \& \langle\tau\rangle$.

We can attempt to minimize it by steepest descent:
Iterate $\frac{\delta \Omega}{\delta t} = -\frac{\delta E}{\delta \Omega}, \frac{\delta \tau}{\delta t} = -\frac{\delta E}{\delta \tau}$. See next section.

9) Multilayer Perceptrons: Learning Algorithms: The update equations for steepest descent are messy algebrally, but not impractical (see Alypadin). You can calculate (σ' is the derivative of σ):

$$\frac{\partial E}{\partial \Omega_{\alpha c}} = \frac{-2}{N} \sum_{\alpha=1}^{M} \{y_\alpha^\mu - \sigma(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu))\} \sigma'(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu)) \sigma(\sum_i \tau_{ci} x_i^\mu). \quad (6)$$

So the update rule for the weights Ω which connect the hidden units to the output units depends only on the difference between the states of those units – the $\{y_\alpha^\mu - \sigma(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu))\}$ term – multiplied by terms which depend on the states of the hidden units only. Hence it depends only on 'local information' that is available to the hidden and the output units. This is neurally plausible (if you care about neurons). The gradient will be zero if the states of the hidden units predict the states of the outputs perfectly – i.e. when $\{y_\alpha^\mu - \sigma(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu))\} = 0$.

You can also calculate:

$$\frac{\partial E}{\partial \tau_{cj}} = \frac{-2}{N} \sum_{\mu=1}^{N} \sum_{\alpha=1}^{M} \{y_\alpha^\mu - \sigma(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu))\} \sigma'(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu)) \Omega_{\alpha c} \sigma'(\sum_i (\tau_{ci} x_i^\mu) x_i^\mu. \quad (7)$$

This requires the error between the output and hidden units – the $\{y_\alpha^\mu - \sigma(\sum_b \Omega_{\alpha b} \sigma(\sum_i \tau_{bi} x_i^\mu))\}$ – to be propagated backwards (hence the name 'backpropagation' for the algorithm) using the weights $\Omega_{\alpha c}$ to give an error term for the weights between the input and hidden units. This was considered a big problem at the time – called the credit assignment problem – how to reward units and weights at the first layer of a network? It is easier to see how to reward the last layer of a network because it is related to the output and so there is a direct measure of how well they are doing.

There is no guarantee that the updates will converge to a global minimum. Indeed, it is very hard to prove anything about multilevel perceptrons - except that, with a sufficient number of hidden units, they can represent any input output function.

10) Variants of Multilayer Perceptrons: 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, τ – 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 class bias-variance dilemma.

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:

$$\lambda \sum_{\alpha b} \Omega_{\alpha b}^2 + \sum_{bi} \tau_{bi}^2$$

This term encourages the eights to be small and maybe even to be zero – using an L^1 penalty term is even better for this.

This extra energy terms modifies the update rules slightly by introducing extra update terms $-\lambda \Omega_{\alpha b}$ and $-\lambda \tau_{bi}$ which will encourage the weights to take small values unless the data says otherwise.

Another variant is to do *online learning.* In this variant, at each time step you select an example (x^μ, y^μ) 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 μ. Then you select another datapoint at random, do another iteration of steepest descent, and so on,

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).