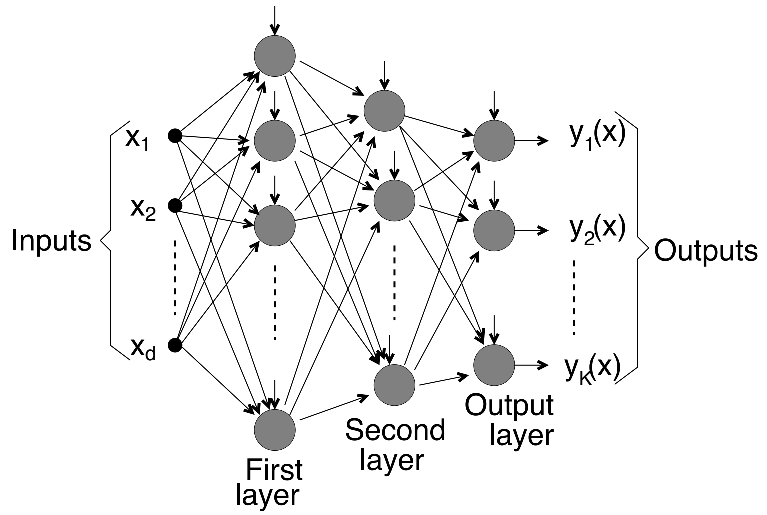
University of Cambridge
Engineering Part IIB

Module 4F10: Statistical Pattern Processing

Handout 8: Multi-Layer Perceptrons



Mark Gales
mjfg@eng.cam.ac.uk
Michaelmas 2013

Introduction

In the next two lectures we will look at Multi-Layer Perceptrons (MLPs) which are more powerful than the Single-Layer models which construct linear decision boundaries. It will be seen that MLPs can be trained as a discriminative model to yield class posteriors.

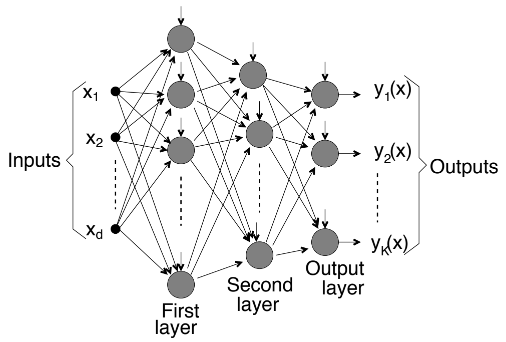
MLPs are classified as a type of Artificial Neural Network: the computation is performed using a set of (many) simple units with weighted connections between them. Furthermore there are learning algorithms to set the values of the weights and the same basic structure (with different weight values) is able to perform many tasks.

In this and the following lecture we will consider

- Overall structure of multi-layer perceptrons
- Decision boundaries that they can form
- Training Criteria
- Networks as posterior probability estimators
- Basic Error back-propagation training algorithm
- Improved training methods
- Deep Neural networks

Multi-Layer Perceptron

From the previous lecture we need a multi-layer perceptron to handle the XOR problem. More generally multi-layer perceptrons allow a neural network to perform arbitrary mappings.

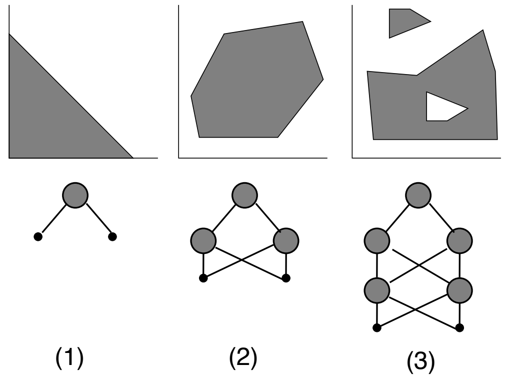


A 2-hidden layer neural network is shown above. The aim is to map an input vector $\boldsymbol{x}$ into an output $\boldsymbol{y}(\boldsymbol{x})$. The layers may be described as:

- Input layer: accepts the data vector or pattern;
- Hidden layers: one or more layers. They accept the output from the previous layer, weight them, and pass through a, normally, non-linear activation function.
- Output layer: takes the output from the final hidden layer weights them, and possibly pass through an output non-linearity, to produce the target values.

Possible Decision Boundaries

The nature of the decision boundaries that may be produced varies with the network topology. Here only threshold (see the single layer perceptron) activation functions are used.



There are three situations to consider

1. Single layer: this is able to position a hyperplane in the input space (the SLP).
2. Two layers (one hidden layer): this is able to describe a decision boundary which surrounds a single convex region of the input space.
3. Three layers (two hidden layers): this is able to to generate arbitrary decision boundaries

Note: any decision boundary can be approximated arbitrarily closely by a two layer network having sigmoidal activation functions if there are enough hidden units.

Number of Hidden Units

From the previous slide we can see that the number of hidden layers determines the decision boundaries that can be generated. In choosing the number of layers the following considerations are made.

- Multi-layer networks are harder to train than single layer networks.
- A two layer network (one hidden) with sigmoidal activation functions can model any decision boundary.

Two layer networks are most commonly used in pattern recognition (the hidden layer having sigmoidal activation functions).

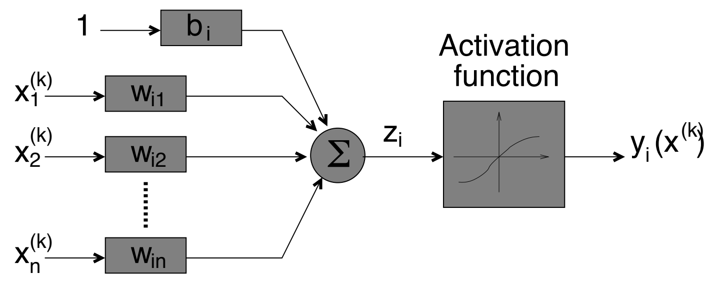
How many units to have in each layer?

- The number of output units is often determined by the number of output classes.
- The number of inputs is determined by the number of input dimensions
- The number of hidden units is a design issue. The problems are:
  - too few, the network will not model complex decision boundaries;
  - too many, the network will have poor generalisation.

Hidden Layer Perceptron

The form of the hidden, and the output, layer perceptron is a generalisation of the single layer perceptron from the previous lecture. Now the weighted input is passed to a general activation function, rather than a threshold function.

Consider a single perceptron. Assume that there are n units at the previous level $(k-1)$ - $x_j^{(k)} = y_j(\boldsymbol{x}^{(k-1)})$



The output from the perceptron, $y_i(\boldsymbol{x})$ may be written as

$$y_i(\boldsymbol{x}^{(k)}) = \phi(z_i) = \phi(\boldsymbol{w}_i'\boldsymbol{x}^{(k)} + b_i) = \phi(\sum_{j=1}^{n} w_{ij}x_j^{(k)} + b_i)$$

where $\phi()$ is the activation function.

We have already seen one example of an activation function the threshold function. Other forms are also used in multi-layer perceptrons.

Note: the activation function is not necessarily non-linear. However, if linear activation functions are used much of the power of the network is lost.

Activation Functions

There are a variety of non-linear activation functions that may be used. Consider the general form

$$y_i(\boldsymbol{x}) = \phi(\boldsymbol{w}_i'\boldsymbol{x} + b_i) = \phi(z_i)$$

and there are n units, perceptrons, for the current level.

- Heaviside (or step) function:

$$y_i(\boldsymbol{x}) = \begin{cases} 0, & z_i < 0 \\ 1, & z_i \geq 0 \end{cases}$$

  These are sometimes used in threshold units, the output is binary (gives a discriminant function).

- Sigmoid (or logistic regression) function:

$$y_i(\boldsymbol{x}) = \frac{1}{1 + \exp(-z_i)}$$

  The output is continuous, $0 \leq y_i(\boldsymbol{x}) \leq 1$.

- Softmax (or normalised exponential or generalised logistic) function:

$$y_i(\boldsymbol{x}) = \frac{\exp(z_i)}{\Sigma_{j=1}^{n} \exp(z_j)}$$

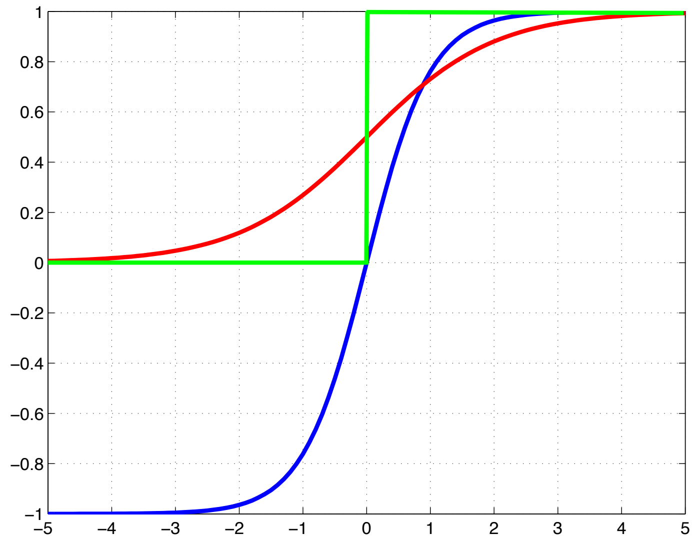
  The output is positive and the sum of all the outputs at the current level is 1, $0 \leq y_i(\boldsymbol{x}) \leq 1$.

- Hyperbolic tan (or tanh) function:

$$y_i(\boldsymbol{x}) = \frac{\exp(z_i) - \exp(-z_i)}{\exp(z_i) + \exp(-z_i)}$$

  The output is continuous, $-1 \leq y_i(\boldsymbol{x}) \leq 1$.

Activation Functions (cont)



Graph shows:

- step activation function (green)
- sigmoid activation function (red)
- tanh activation function (blue)

Sigmoid, or softmax, often used at output layers as sum-to-one constraint is enforced.

Training Criteria

A variety of training criteria may be used. Assuming we have supervised training examples

$$\{\{\boldsymbol{x}_1, \boldsymbol{t}_1\} \ldots, \{\boldsymbol{x}_n, \boldsymbol{t}_n\}\}$$

Some standard examples are:

- Least squares error: one of the most common training criteria.

$$\begin{aligned} E &= \frac{1}{2}\sum_{p=1}^{n} ||\boldsymbol{y}(\boldsymbol{x}_p) - \boldsymbol{t}_p)||^2 \\ &= \frac{1}{2}\sum_{p=1}^{n}\sum_{i=1}^{K} (y_i(\boldsymbol{x}_p) - t_{pi})^2 \end{aligned}$$

  This may be derived from considering the targets as being corrupted by zero-mean Gaussian distributed noise.

- Cross-Entropy for two classes: consider the case when t is binary (and softmax output). The expression is

$$E = -\sum_{p=1}^{n} (t_p \log(y(\boldsymbol{x}_p)) + (1 - t_p)\log(1 - y(\boldsymbol{x}_p)))$$

  This expression goes to zero with the "perfect" mapping - compare logistic regression training.

- Cross-Entropy for multiple classes: the above expression becomes (again softmax output)

$$E = -\sum_{p=1}^{n}\sum_{i=1}^{K} t_{pi}\log(y_i(\boldsymbol{x}_p))$$

  The minimum value is now non-zero, it represents the entropy of the target values.

Posterior Probabilities

Consider the multi-class classification training problem with

- d-dimensional feature vectors: $\boldsymbol{x}$;
- K-dimensional output from network: $\boldsymbol{y}(\boldsymbol{x})$;
- K-dimensional target: $\boldsymbol{t}$.

We would like the output of the network, $\boldsymbol{y}(\boldsymbol{x})$, to approximate the posterior distribution of the set of K classes. So

$$y_i(\boldsymbol{x}) \approx P(\omega_i|\boldsymbol{x})$$

Consider training a network with:

- 1-out-of$K$ coding, i.e.

$$t_i = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \omega_i \\ 0 & \text{if } \boldsymbol{x} \notin \omega_i \end{cases}$$

The network will act as a d-dimensional to K-dimensional mapping.

Can we interpret the output of the network?

A couple of assumptions will be made

- errors from each output independent (not true of softmax)
- error is a function of the magnitude difference of output and target $E = \Sigma_{i=1}^{K} f(|y_i(\boldsymbol{x}) - t_i|)$

Posterior Probabilities (cont)

In the limit of infinite training data,

$$\mathcal{E}\{E\} = \sum_{i=1}^{K} \int \sum_{\boldsymbol{t}} f(|y_i(\boldsymbol{x}) - t_i|) P(\boldsymbol{t}|\boldsymbol{x}) p(\boldsymbol{x}) d\boldsymbol{x}$$

As we are using the 1-out-of-K coding

$$P(\boldsymbol{t}|\boldsymbol{x}) = \prod_{i=1}^{K} \left(\sum_{j=1}^{K} \delta(t_i - \delta_{ij}) P(\omega_j|\boldsymbol{x}) \right)$$

where

$$\delta_{ij} = \begin{cases} 1, & (i = j) \\ 0, & \text{otherwise} \end{cases}$$

This yields

$$\mathcal{E}\{E\} = \sum_{i=1}^{K} \int \left(f(1 - y_i(\boldsymbol{x})) P(\omega_i|\boldsymbol{x}) + f(y_i(\boldsymbol{x}))(1 - P(\omega_i|\boldsymbol{x})) \right) p(\boldsymbol{x}) d\boldsymbol{x}$$

Differentiating and equating to zero, for least squared, cross-entropy solutions are

$$y_i(\boldsymbol{x}) = P(\omega_i|\boldsymbol{x})$$

- this is a discriminative model.

Some limitations exist for this to be valid:

- an infinite amount of training data, or knowledge of correct joint distribution for $p(\boldsymbol{x}, \boldsymbol{t})$;
- the topology of the network is "complex" enough that final error is small;
- the training algorithm used to optimise the network is good - it finds the global maximum.

Compensating for Different Priors

The approach described at the start of the course was to use Bayes' law to obtain the posterior probability for generative models.

$$P(\omega_j|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\omega_j)P(\omega_j)}{p(\boldsymbol{x})}$$

where the priors class priors, $P(\omega_j)$, and class conditional densities, $p(\boldsymbol{x}|\omega_j)$, are estimated separately. For some tasks the two use different training data (for example for speech recognition, the language model and the acoustic model).

How can this difference in priors from the training and the test conditions be built into the neural network framework where the posterior probability is directly calculated? (This is applicable to all discriminative models)

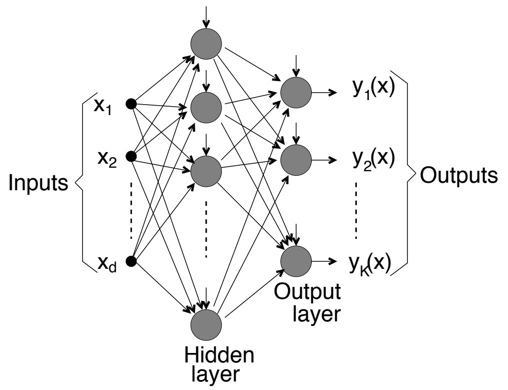
Again using Bayes' law

$$p(\boldsymbol{x}|\omega_j) \propto \frac{P(\omega_j|\boldsymbol{x})}{P(\omega_j)}$$

Thus if posterior is divided by the training data prior a value proportional to the class-conditional probability can be obtained. The standard form of Bayes' rule may now be used with the appropriate, different, prior.

Error Back Propagation

So far the training of MLPs has not been discussed. Originally interest in MLPs was limited due to issues in training. These problems were partly addressed with the development of the error back propagation algorithm.



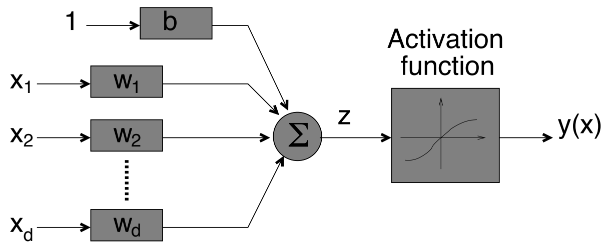
A single hidden layer network is shown above. Using sigmoidal activation functions arbitrary decision boundaries may be obtained with this network topology.

The error back propagation algorithm is based on gradient descent. Hence the activation function must be differentiable. Thus threshold and step units will not be considered. We need to be able to compute the derivative of the error function with respect to the weights of all layers.

All gradients in the next few slides are evaluated at the current model parameters.

Single Layer Perceptron

Rather than examine the multi-layer case instantly, consider the following single layer perceptron.



We would like to minimise (for example) the square error between the target of the output, t_p, and the current output value $y(\boldsymbol{x}_p)$. Assume that the training criterion is least squares and the activation function is known to be a sigmoid function. The cost function may be written as

$$E = \frac{1}{2}\sum_{p=1}^{n}(y(\boldsymbol{x}_p) - t_p)'(y(\boldsymbol{x}_p) - t_p)) = \sum_{p=1}^{n} E^{(p)}$$

To simplify notation, we will only consider a single observation $\boldsymbol{x}$ with associated target values t and current output from the network $y(\boldsymbol{x})$. The error with this single observation is denoted E.

How does the error change as $y(\boldsymbol{x})$ changes?

$$\frac{\partial E}{\partial y(\boldsymbol{x})} = y(\boldsymbol{x}) - t$$

But we are not interested in $y(\boldsymbol{x})$

How do we find the effect of varying the weights?

SLP Training (cont)

Calculate effect of changing z on the error using the chain rule

$$\frac{\partial E}{\partial z} = \left(\frac{\partial E}{\partial y(\boldsymbol{x})}\right)\left(\frac{\partial y(\boldsymbol{x})}{\partial z}\right)$$

However what we really want is the change of the error with respect to the weights (the parameters that we want to learn).

$$\frac{\partial E}{\partial w_i} = \left(\frac{\partial E}{\partial z}\right)\left(\frac{\partial z}{\partial w_i}\right)$$

The error function therefore depends on the weight as

$$\frac{\partial E}{\partial w_i} = \left(\frac{\partial E}{\partial y(\boldsymbol{x})}\right)\left(\frac{\partial y(\boldsymbol{x})}{\partial z}\right)\left(\frac{\partial z}{\partial w_i}\right)$$

Noting that (the bias term b can be treated as the $d+1$ element)

$$\frac{\partial y(\boldsymbol{x})}{\partial z} = y(\boldsymbol{x})(1 - y(\boldsymbol{x}))$$

$$\frac{\partial E}{\partial w_i} = (y(\boldsymbol{x}) - t)y(\boldsymbol{x})(1 - y(\boldsymbol{x}))x_i$$

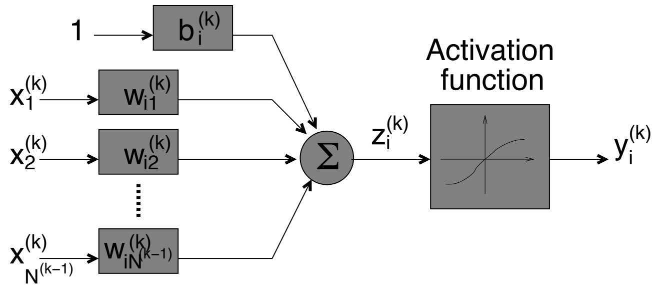
This has been computed for a single observation. We are interested in terms of the complete training set. We know that the total errors is the sum of the individual errors, so

$$\boldsymbol{\nabla} E = \sum_{p=1}^{n}(y(\boldsymbol{x}_p) - t_p)y(\boldsymbol{x}_p)(1 - y(\boldsymbol{x}_p))\tilde{\boldsymbol{x}}_p$$

So for a single layer we can use gradient descent schemes to find the "best" weight values. We can also apply the above to compute the derivatives wrt the weights for the final hidden to output layer for an MLP.

Error Back Propagation Algorithm

Now consider a particular node, i, of hidden layer k. Using the previously defined notation, the input to the node is $\boldsymbol{x}^{(k)}$ and the output $y_i^{(k)}$.



From the previous section it is possible to derive the rate of change of the error function with the weights of the output layer. Need to examine the rate of change with the k^{th} hidden layer weights.

The parameters of the network for layer k can be written as

$$\tilde{\mathbf{W}}^{(k)} = \left[\begin{array}{cc} \mathbf{W}^{(k)} & \boldsymbol{b}^{(k)} \end{array}\right]$$

$\tilde{\mathbf{W}}^{(k)}$ is an $N^{(k)} \times (N^{(k-1)} + 1)$ matrix

- output from previous layer is the input to the next layer

Using this fact allows the SLP training expression to be extended to allow training of an MLP. [The forms of the maths behind this are given at the end of these notes].

Error Back Propagation Summary

To calculate $\boldsymbol{\nabla} E^{(p)}\big|_{\boldsymbol{\theta}[\tau]}$ ($\boldsymbol{\theta}[\tau]$ is the set of "current" (training epoch τ) values of the weights) we use the following algorithm.

1. Apply the input vector $\boldsymbol{x}_p$ to the network and use the feed forward matrix equations to propagate the input forward through the network. For all layers this yields $\boldsymbol{y}^{(k)}$ and $\mathbf{z}^{(k)}$.
2. Compute $\frac{\partial E}{\partial \boldsymbol{y}(\boldsymbol{x})}\Big|_{\boldsymbol{\theta}[\tau]}$ (the gradient at the output layer).
3. Using the back-propagation formulae back-propagate the δs back through the network, layer by layer and hence the partial derivatives for each weight.

Having obtained the derivatives of the error function with respect to the weights of the network, we need a scheme to optimise the value of the weights.

The obvious choice is gradient descent

Gradient Descent

Having found an expression for the gradient, gradient descent may be used to find the values of the weights.

Initially consider a batch update rule. Here

$$\tilde{\boldsymbol{w}}_i^{(k)}[\tau+1] = \tilde{\boldsymbol{w}}_i^{(k)}[\tau] - \eta \frac{\partial E}{\partial \tilde{\boldsymbol{w}}_i^{(k)}}\Bigg|_{\boldsymbol{\theta}[\tau]}$$

where $\boldsymbol{\theta}[\tau] = \{\tilde{\mathbf{W}}^{(1)}[\tau], \ldots, \tilde{\mathbf{W}}^{(L+1)}[\tau]\}$, $\tilde{\boldsymbol{w}}_i^{(k)}[\tau]$ is the i^{th} row of $\tilde{\mathbf{W}}^{(k)}$ at training epoch τ and

$$\frac{\partial E}{\partial \tilde{\boldsymbol{w}}_i^{(k)}}\Bigg|_{\boldsymbol{\theta}[\tau]} = \sum_{p=1}^{n} \frac{\partial E^{(p)}}{\partial \tilde{\boldsymbol{w}}_i^{(k)}}\Bigg|_{\boldsymbol{\theta}[\tau]}$$

If the total number of weights in the system is N then all N derivatives may be calculated in $\mathcal{O}(N)$ operations with memory requirements $\mathcal{O}(N)$.

However in common with other gradient descent schemes there are problems as:

- we need a value of η that achieves a stable, fast descent;
- the error surface may have local minima, maxima and saddle points.

This has lead to refinements of gradient descent.

Training Schemes

On the previous slide the weights were updated after all n training examples have been seen. This is not the only scheme that may be used.

- Batch update: the weights are updated after all the training examples have been seen. Thus

$$\tilde{\boldsymbol{w}}_i^{(k)}[\tau+1] = \tilde{\boldsymbol{w}}_i^{(k)}[\tau] - \eta \left(\sum_{p=1}^{n} \left. \frac{\partial E^{(p)}}{\partial \tilde{\boldsymbol{w}}_i^{(k)}} \right|_{\boldsymbol{\theta}[\tau]} \right)$$

- Sequential update: the weights are updated after every sample. Now

$$\tilde{\boldsymbol{w}}_i^{(k)}[\tau+1] = \tilde{\boldsymbol{w}}_i^{(k)}[\tau] - \eta \left. \frac{\partial E^{(p)}}{\partial \tilde{\boldsymbol{w}}_i^{(k)}} \right|_{\boldsymbol{\theta}[\tau]}$$

  and we cycle around the training vectors.
  There are some advantages of this form of update.
  - It is not necessary to store the whole training database. Samples may be used only once if desired.
  - They may be used for online learning
  - In dynamic systems the values of the weights can be updated to "track" the system.

In practice forms of batch training or an intermediate between batch and sequential training are often used.

Refining Gradient Descent

There are some simple techniques to refine standard gradient descent. First consider the learning rate η. We can make this vary with each iteration. One of the simplest rules is to use

$$\eta[\tau+1] = \begin{cases} 1.1\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) < E(\boldsymbol{\theta}[\tau-1]) \\ 0.5\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) > E(\boldsymbol{\theta}[\tau-1]) \end{cases}$$

In words: if the previous value of $\eta[\tau]$ decreased the value of the cost function, then increase $\eta[\tau]$. If the previous value of $\eta[\tau]$ increased the cost function ($\eta[\tau]$ too large) then decrease $\eta[\tau]$.

It is also possible to add a momentum term to the optimisation (common in MLP estimation). The update formula is

$$\tilde{\boldsymbol{w}}_i^{(k)}[\tau+1] = \tilde{\boldsymbol{w}}_i^{(k)}[\tau] + \boldsymbol{\Delta}\tilde{\boldsymbol{w}}_i^{(k)}[\tau]$$

where

$$\boldsymbol{\Delta}\tilde{\boldsymbol{w}}_i^{(k)}[\tau] = -\eta[\tau+1] \left. \frac{\partial E}{\partial \tilde{\boldsymbol{w}}_i^{(k)}} \right|_{\boldsymbol{\theta}[\tau]} + \alpha[\tau]\boldsymbol{\Delta}\tilde{\boldsymbol{w}}_i^{(k)}[\tau-1]$$

The use of the momentum term, $\alpha[\tau]$:

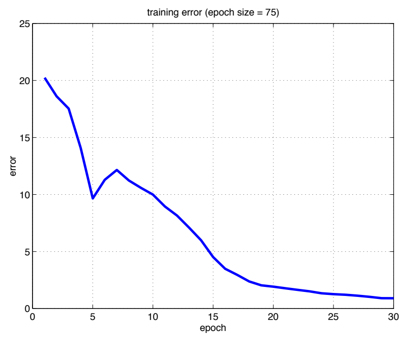
- smooths successive updates;
- helps avoid small local minima.

Unfortunately it introduces an additional tunable parameter to set. Also if we are lucky and hit the minimum solution we will overshoot.

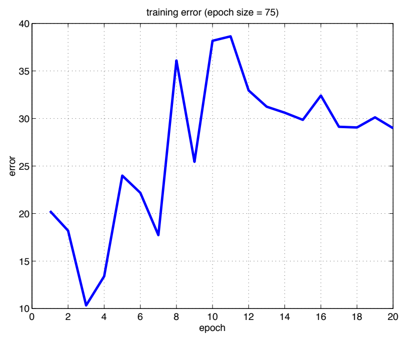
Iris Data

Again using the Iris data (only petal information) with:

- least squared error training
- tanh hidden, softmax output layer activation functions
- data transformed - zero mean +/-5 range
- gradient descent (with momentum)
- batch update, $\eta = 0.1$, (momentum $\alpha = 0.8$)



3 hidden units



100 hidden units

Plots show stability issues, in this case as the number of hidden units is increased.

How to set learning rates to get stable training?

Quadratic Approximation

Gradient descent makes use of first-order derivatives of the error function. What about higher order techniques?

Consider the vector form of the Taylor series

$$\begin{aligned} E(\boldsymbol{\theta}) &= E(\boldsymbol{\theta}[\tau]) + (\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])'\mathbf{g} \\ &\quad + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])'\mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau]) + \mathcal{O}(\boldsymbol{\theta}^3) \end{aligned}$$

where

$$\mathbf{g} = \boldsymbol{\nabla} E(\boldsymbol{\theta})\big|_{\boldsymbol{\theta}[\tau]}$$

and

$$(\mathbf{H})_{ij} = h_{ij} = \left.\frac{\partial^2 E(\boldsymbol{\theta})}{\partial w_i \partial w_j}\right|_{\boldsymbol{\theta}[\tau]}$$

Ignoring higher order terms we find

$$\boldsymbol{\nabla} E(\boldsymbol{\theta}) = \mathbf{g} + \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])$$

Equating this to zero we find that the value of $\boldsymbol{\theta}$ at this point $\boldsymbol{\theta}[\tau+1]$ is

$$\boldsymbol{\theta}[\tau+1] = \boldsymbol{\theta}[\tau] - \mathbf{H}^{-1}\mathbf{g}$$

This gives us a simple update rule. The direction $\mathbf{H}^{-1}\mathbf{g}$ is known as the Newton direction.

Problems with the Hessian

In practice the use of the Hessian is limited.

1. The evaluation of the Hessian may be computationally expensive as $\mathcal{O}(N^2)$ parameters must be accumulated for each of the n training samples.
2. The Hessian must be inverted to find the direction, $\mathcal{O}(N^3)$. This gets very expensive as N gets large.
3. The direction given need not head towards a minimum - it could head towards a maximum or saddle point. This occurs if the Hessian is not positive-definite i.e.

$$\mathbf{v}'\mathbf{H}\mathbf{v} > 0$$

   for all $\mathbf{v}$.
4. If the surface is highly non-quadratic the step sizes may be too large and the optimisation becomes unstable.

Approximations to the Hessian are commonly used.

The simplest approximation is to assume that the Hessian is diagonal. This ensures that the Hessian is invertible and only requires N parameters.

The Hessian may be made positive definite using

$$\tilde{\mathbf{H}} = \mathbf{H} + \lambda\mathbf{I}$$

If λ is large enough then $\tilde{\mathbf{H}}$ is positive definite.

QuickProp

An interesting approximate second-order optimisation approach is quickprop. This makes some assumptions about the nature of the problem

- error surface is quadratic in nature
- the change of slope of error function wrt to a weight may be treated independently of all other weights (diagonal Hessian)

This yields an interesting update rule. Consider the surface for a single weight denoted by θ

$$\begin{aligned} E(\theta) &\approx E(\theta[\tau]) + b(\theta - \theta[\tau]) + a(\theta - \theta[\tau])^2 \\ \frac{\partial E(\theta)}{\partial \theta} &\approx b + 2a(\theta - \theta[\tau]) \end{aligned}$$

The following information is then used to find a and b

- the update difference, $\Delta\theta[\tau-1]$, and gradient, $g[\tau-1]$, at iteration $\tau-1$
- the gradient at iteration τ is $g[\tau]$
- after applying the unknown offset $\Delta\theta[\tau]$ the gradient should be zero (a minimum)

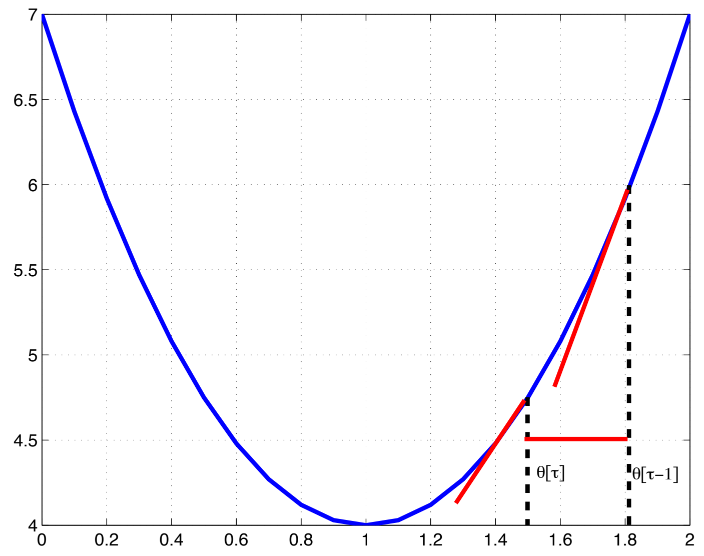
The following equalities are obtained

$$g[\tau-1] = b - 2a\Delta\theta[\tau-1], \quad 0 = b + 2a\Delta\theta[\tau], \quad g[\tau] = b$$

and solving gives

$$\Delta\theta[\tau] = \frac{g[\tau]}{g[\tau-1] - g[\tau]}\Delta\theta[\tau-1]$$

QuickProp (cont)



The operation of quick-prop is illustrated above.

- The assumed quadratic error surface is shown in blue
- The statistics for quickprop are shown in red

The new-estimate will lie exactly at the minimum of the quadratic approximation

$$\theta[\tau+1] = 1$$

Regularisation

One of the major issues with training neural networks is how to ensure generalisation. One commonly used technique is weight decay. A regulariser may be used. Here

$$\Omega = \frac{1}{2}\sum_{i=1}^{N} w_i^2$$

where N is the total number of weights in the network. A new error function is defined

$$\tilde{E} = E + \nu\Omega$$

Using gradient descent on this gives

$$\boldsymbol{\nabla}\tilde{E} = \boldsymbol{\nabla}E + \nu\boldsymbol{w}$$

The effect of this regularisation term Ω penalises very large weight terms. From empirical results this has resulted in improved performance.

Rather than using an explicit regularisation term, the "complexity" of the network can be controlled by training with noise.

For batch training we replicate each of the samples multiple times and add a different noise vector to each of the samples. If we use least squares training with a zero mean noise source (equal variance ν in all the dimensions) the error function may be shown to have the form

$$\tilde{E} = E + \nu\Omega$$

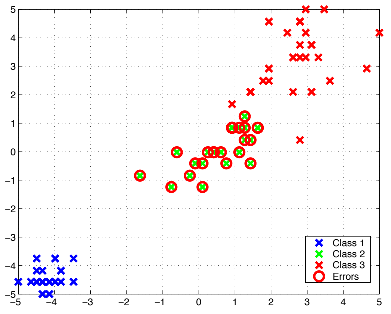
This is a another form of regularisation.

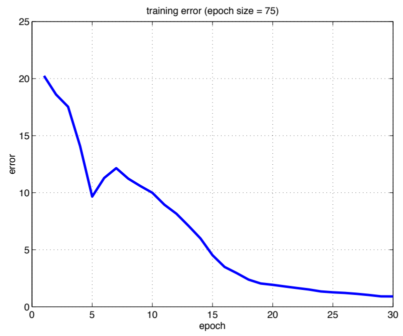
Iris Data Example

Again using the Iris data (only petal data) with:

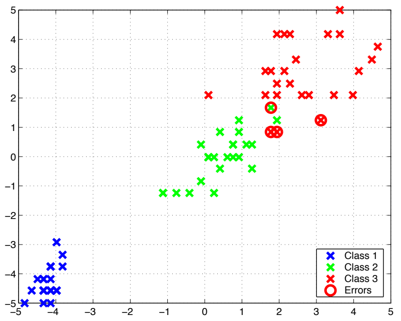
- 3-unit hidden-layer
- tanh hidden, softmax output layer activation functions
- data transformed - zero mean +/-5 range
- batch update, $\eta = 0.1$, $\alpha = 0.8$

Data split into 75 training samples, 75 test samples.





Performance: training data above (1/75) test error below (4/75)



Deep Neural Networks

For many years research into neural networks typically examined configurations with few (typically one or two) hidden layers and relatively small number of parameters. The reasons for this were:

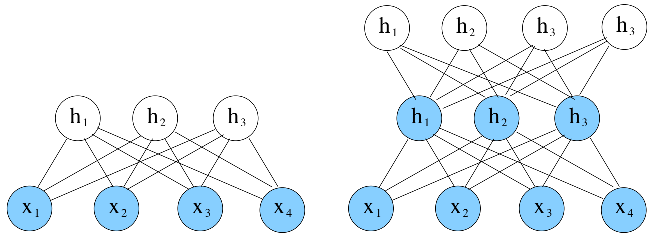
1. a small number of layers with sufficient units could approximate any decision boundary;
2. computational issues with using large numbers of model parameters (and layers);
3. initialisation of the network parameters (and optimisation approaches).

Recent research typically uses larger numbers of hidden layers - for this reason the MLPs are sometimes refered to as Deep Neural Networks.

Network Initialisation

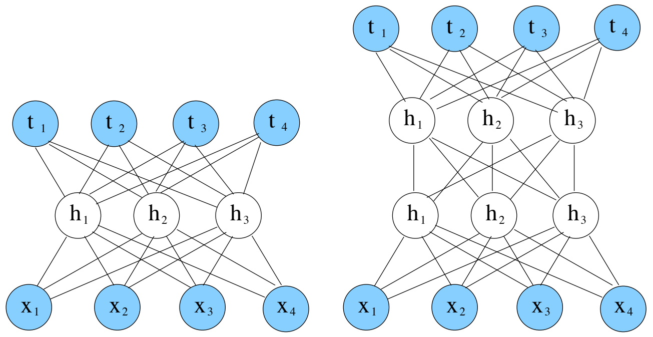
For the optimisation approaches described, the network parameters for each layer must be initialised. Three approaches that are used are:

1. random: the parameters are randomly distributed, usually samples drawn from a zero mean Gaussian distribution. This was the standard approach for many years;
2. restricted Boltzmann machine: this initialises the network parameters using the generative RBM. This involves the layer-by-layer generation of an RBM where the number of elements matches the number required in the final network.



This is sometimes refered to as generative pre-training. If the hidden layers are not further trained, so only the output layer is discrimanitively trained (single layer training), the network is sometimes referred to as a deep belief network.

3. layer-by-layer training: rather than initialising all the parameters in one go, hidden layers of the network are incrementally added. After each new layer is added (using random initial values) the network is then trained.



The number of iterations when each layer is added is often limited to reduce training time. This is sometimes called discriminative pre-training.

After the network has been initialised standard error-back propagation can be used. In the context of deep neural networks this is called fine-tuning.

Hidden Layer Configurations & Training

The arguments for larger numbers of layers are:

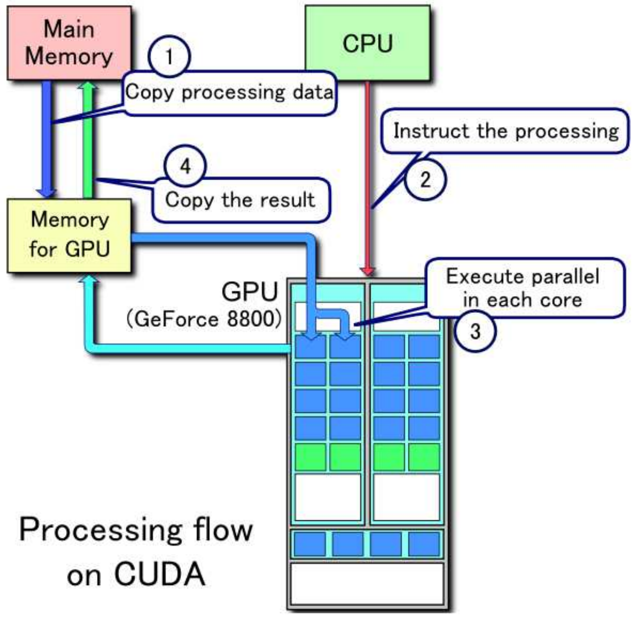
- though a single layer in theory (with a sigmoid activation function) allows any form of decision boundary to be modelled, in practice there is only finite training data, thus the number of single layer units is bounded;
- appropriate network initialisation should allow many layers to be trained without gradients "vanishing".

For Deep Neural Networks many hidden layers (often > 5) are used. These have been shown to yield better performance having a large number of hidden units in one or two layers.

In addition for training large systems stochastic gradient descent is used. Here a sequential (example-by-example), or mini-batch (small number of samples), update is used. However rather than using observations in a fixed order the data is randomly ordered after updates using all the data has been completed.

Graphical Processor Units

Compute power (based on CPUs) has continued to improve over the last decade. This has enabled more complicated networks to be examined. Recently Graphical Processor Units (GPUs) have also been applied to training MLPs.

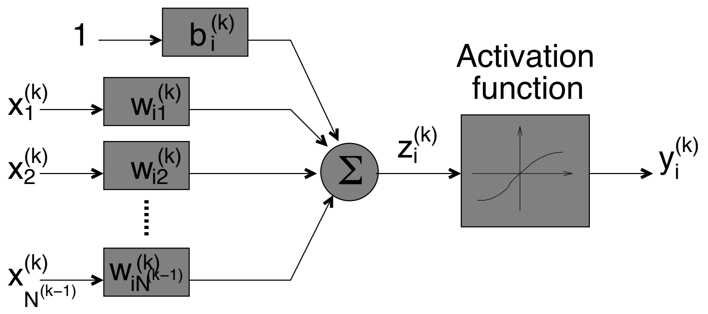


GPUs were originally designed for manipulating and creating images. They have a highly parallelized architecture (manipulate pixels in parallel). Training neural networks also maps well to this sort of architecture - these are increasingly used in many large-scale training of deep neural networks.

Reference - Error Back Propagation

Consider a multi-layer perceptron with:

- d-dimensional input data;
- L hidden layers ($L+1$ layer including the output layer);
- $N^{(k)}$ units in the k^{th} level;
- K-dimensional output.



The following notation will be used

- $\boldsymbol{x}^{(k)}$ is the input to the k^{th} layer
- $\tilde{\boldsymbol{x}}^{(k)}$ is the extended input to the k^{th} layer

$$\tilde{\boldsymbol{x}}^{(k)} = \begin{bmatrix} \boldsymbol{x}^{(k)} \\ 1 \end{bmatrix}$$

- $\mathbf{W}^{(k)}$ is the weight matrix of the k^{th} layer. By definition this is a $N^{(k)} \times N^{(k-1)}$ matrix.

Notation (cont)

- $\tilde{\mathbf{W}}^{(k)}$ is the weight matrix including the bias weight of the k^{th} layer. By definition this is a $N^{(k)} \times (N^{(k-1)}+1)$ matrix.

$$\tilde{\mathbf{W}}^{(k)} = \begin{bmatrix} \mathbf{W}^{(k)} & \boldsymbol{b}^{(k)} \end{bmatrix}$$

- $\mathbf{z}^{(k)}$ is the $N^{(k)}$-dimensional vector defined as

$$\mathbf{z}^{(k)} = \tilde{\mathbf{W}}^{(k)}\tilde{\boldsymbol{x}}^{(k)}$$

- $\boldsymbol{y}^{(k)}$ is the output from the k^{th} layer, so

$$y_j^{(k)} = \phi(z_j^{(k)})$$

All the hidden layer activation functions are assumed to be the same $\phi()$. Initially we shall also assume that the output activation function is also $\phi()$.

The following matrix notation feed forward equations may then used for a multi-layer perceptron with input $\boldsymbol{x}$ and output $\boldsymbol{y}(\boldsymbol{x})$.

$$\begin{aligned} \boldsymbol{x}^{(1)} &= \boldsymbol{x} \\ \boldsymbol{x}^{(k)} &= \boldsymbol{y}^{(k-1)} \\ \mathbf{z}^{(k)} &= \tilde{\mathbf{W}}^{(k)}\tilde{\boldsymbol{x}}^{(k)} \\ \boldsymbol{y}^{(k)} &= \boldsymbol{\phi}(\mathbf{z}^{(k)}) \\ \boldsymbol{y}(\boldsymbol{x}) &= \boldsymbol{y}^{(L+1)} \end{aligned}$$

where $1 \le k \le L+1$.

The target values for the training of the networks will be denoted as $\boldsymbol{t}$ for training example $\boldsymbol{x}$.

Error Back Propagation Algorithm

Need to calculate $\frac{\partial E}{\partial \tilde{w}_{ij}^{(k)}}$ for all layers, k, and all rows and columns of $\tilde{\mathbf{W}}^{(k)}$. Applying the chain rule

$$\frac{\partial E}{\partial \tilde{w}_{ij}^{(k)}} = \frac{\partial E}{\partial z_i^{(k)}} \frac{\partial z_i^{(k)}}{\partial \tilde{w}_{ij}^{(k)}} = \delta_i^{(k)} \tilde{x}_j^{(k)}$$

where

$$\frac{\partial E}{\partial z_i^{(k)}} = \delta_i^{(k)}$$

and the δ's are sometimes known as the individual "errors" (that are back-propagated).
For the output nodes the evaluation of δ_i is straightforward as we saw for the single layer perceptron.
To evaluate the δ_i's for hidden layers

$$\delta_i^{(k)} = \sum_m \left[\frac{\partial E}{\partial z_m^{(k+1)}} \frac{\partial z_m^{(k+1)}}{\partial z_i^{(k)}} \right]$$

where it is assumed that only the units in layer $k+1$ are connected to units in layer k, or

$$\delta_i^{(k)} = y_i^{(k)}(1 - y_i^{(k)}) \sum_m \tilde{w}_{mi}^{(k+1)} \delta_m^{(k+1)}$$

Note that all that is being done here is evaluating the differentials of the error at the output with respect to the weights throughout the network by using the chain rule for partial derivatives.

Matrix Formulation

In matrix notation we can write

$$\frac{\partial E}{\partial \tilde{\mathbf{W}}^{(k)}} = \boldsymbol{\delta}^{(k)} \tilde{\boldsymbol{x}}^{(k)\prime}$$

We need to find a recursion for $\boldsymbol{\delta}^{(k)}$.

$$\begin{aligned} \boldsymbol{\delta}^{(k)} &= \left(\frac{\partial E}{\partial \mathbf{z}^{(k)}} \right) \\ &= \left(\frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{z}^{(k)}} \right) \left(\frac{\partial E}{\partial \mathbf{z}^{(k+1)}} \right) \\ &= \left(\frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{z}^{(k)}} \right) \left(\frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{y}^{(k)}} \right) \boldsymbol{\delta}^{(k+1)} \end{aligned}$$

But we know from the forward recursions

$$\frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{y}^{(k)}} = \frac{\partial \mathbf{z}^{(k+1)}}{\partial \boldsymbol{x}^{(k+1)}} = \mathbf{W}^{(k+1)\prime}$$

This yields the recursion

$$\boldsymbol{\delta}^{(k)} = \boldsymbol{\Lambda}^{(k)} \mathbf{W}^{(k+1)\prime} \boldsymbol{\delta}^{(k+1)}$$

Matrix Formulation (cont)

Define the activation derivative matrix for layer k as

$$\boldsymbol{\Lambda}^{(k)} = \frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{z}^{(k)}} = \begin{bmatrix} \frac{\partial y_1^{(k)}}{\partial z_1^{(k)}} & \frac{\partial y_2^{(k)}}{\partial z_1^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_1^{(k)}} \\ \frac{\partial y_1^{(k)}}{\partial z_2^{(k)}} & \frac{\partial y_2^{(k)}}{\partial z_2^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_2^{(k)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1^{(k)}}{\partial z_{N^{(k)}}^{(k)}} & \frac{\partial y_2^{(k)}}{\partial z_{N^{(k)}}^{(k)}} & \cdots & \frac{\partial y_{N^{(k)}}^{(k)}}{\partial z_{N^{(k)}}^{(k)}} \end{bmatrix}$$

This has given a matrix form of the backward recursion for the error back propagation algorithm.

We need to have an initialisation of the backward recursion. This will be from the output layer (layer $L+1$)

$$\begin{aligned} \boldsymbol{\delta}^{(L+1)} &= \frac{\partial E}{\partial \mathbf{z}^{(L+1)}} \\ &= \left(\frac{\partial \mathbf{y}^{(L+1)}}{\partial \mathbf{z}^{(L+1)}}\right)\left(\frac{\partial E}{\partial \mathbf{y}^{(L+1)}}\right) \\ &= \boldsymbol{\Lambda}^{(L+1)}\left(\frac{\partial E}{\partial \mathbf{y}(\boldsymbol{x})}\right) \end{aligned}$$

$\boldsymbol{\Lambda}^{(L+1)}$ is the activation derivative matrix for the output layer.