Feedforward network functions

Neural networks

Francesco Corona

Neural networks

Neural networks

We have considered models for regression and classification that comprised linear combinations of fixed basis functions

- These models have useful analytical and computational properties
- Their applicability is however limited by the curse of dimensionality

For large-scale problems, it is necessary to adapt the basis functions to data

Neural networks (cont.)

Support vector machines (SVMs) address this issue

- Basis functions that are centred on the training data points
- During training, only a subset of the functions are selected

One advantage of SVMs is that the objective function is convex

- although the training involves nonlinear optimisation
- the solution of the optimisation problem is attainable

The number of basis functions in the resulting models is generally much smaller than the number of training points, although it is often still relatively large

- Typically, it increases with the size of the training set

Relevance vector machines (RVMs) also chooses a subset from a fixed set of basis functions and this typically results in much sparser models

- Non-convex optimisation
- Probabilistic outputs

Neural networks (cont.)

An alternative approach is to fix the number of basis functions in advance

- Allow them to be adaptive

Use parametric forms for the basis functions and adapt parameter values

The most successful model of this type in the context of PR and ML is the **feed-forward neural network**, also known as the **multi-layer perceptron**

- Multiple logistic regression models (with continuous non-linearities)
- Not multiple layers of perceptrons (with discontinuous non-linearities)

Neural networks (cont.)

Often, the resulting model can be significantly more compact, and faster to evaluate, than a support vector machine with same generalisation performance

- The price of compactness, as with RVMs, is a likelihood function, the basis for network training, that is no longer a convex function of the parameters

It is often worth investing large computational resources during the training phase in order to obtain a compact model that is fast at processing new data

Neural networks (cont.)

We consider the functional form of the network model, including the parameterisation of the basis functions

We discuss the determination of network parameters in a maximum likelihood framework, which leads to a nonlinear optimisation problem

- It requires the evaluation of the derivatives of the log likelihood function with respect to network parameters
- These can be readily obtained using the technique of **error back-propagation**

Back-prop can be also used to evaluate Jacobian and Hessian matrices

Neural networks (cont.)

We discuss approaches to regularisation of network training and their relations

We consider extensions to the neural network model, and describe a general framework for modelling conditional probability distributions

- Mixture density networks

Finally, we discuss an approach to the Bayesian treatment of neural networks

Outline

Feed-forward network functions
Neural networks

Feedforward network functions

The linear models for regression and classification are based on linear combinations of fixed non-linear basis functions $\phi_j(\mathbf{x})$ and take the form

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right) \tag{1}$$

where $f(\cdot)$ is a non-linear activation function in the case of classification and the identity in the case of regression

We want to extend this model by making the basis function $\phi_j(\mathbf{x})$ depend on parameters and then allow these parameters to be adjusted during training

- along with the coefficients $\{w_j\}$

There are many ways to construct parametric non-linear basis functions

Feedforward network functions (cont.)

A basis neural network model uses basis functions that follow the same form as

$$f\Big(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\Big)$$

Each basis function is itself a non-linear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters

A neural network can be thus described as a series of functional transformations

Feedforward network functions (cont.)

First we construct M linear combinations of the **input variables** $x_1, \dots, x_D$

$$a_j = \sum_{i=1}^{D} \left(w_{ji}^{(1)} x_i \right) + w_{j0}^{(1)}, \quad j = 1, \dots, M \tag{2}$$

- parameters $w_{ji}^{(1)}$ are denoted as **weights**
- parameters $w_{j0}^{(1)}$ are denoted as **biases**
- quantities a_j are known as **activations**

The superscript (1) indicates parameters in the **first layer** of the network

Each activation is transformed using a differentiable non-linear function

$$z_j = h(a_j) \tag{3}$$

- function $h(\cdot)$ is known as **activation function**
- z_j are **outputs of the basis**, or **hidden units**

Feedforward network functions (cont.)

The outputs or **hidden units** z_j are again linearly transformed

$$a_k = \sum_{j=1}^{M} \left(w_{kj}^{(2)} z_j \right) + w_{k0}^{(2)}, \quad k = 1, \dots, K \tag{4}$$

where K denotes the total number of **output unit activations**

- parameters $w_{kj}^{(2)}$ are denoted as **weights**
- parameters $w_{k0}^{(2)}$ are denoted as **biases**

The superscript (2) indicates parameters in the **second layer** of the network

Each output unit activation is transformed to give the network outputs y_k

$$\overbrace{y_k(\mathbf{x}, \mathbf{w}) = \sigma\Bigg(\overbrace{\sum_{j=1}^{M} \Bigg(w_{kj}^{(2)} \underbrace{h\Bigg(\underbrace{\sum_{i=1}^{D} \left(w_{ji}^{(1)} x_i \right) + w_{j0}^{(1)}}_{a_j} \Bigg)}_{z_j = h(a_j)} \Bigg) + w_{k0}^{(2)}}^{a_k} \Bigg)}^{y_k = \sigma(a_k)} \tag{5}$$

Feedforward network functions (cont.)

Output unit activations a_k are transformed by an activation function $\sigma(\cdot)$

- The choice of $\sigma(\cdot)$ is determined by the nature of the data
- More precisely, the assumed distribution of the target variables

For standard regression problems, the activation function is the identity function

$$y_k = \sigma(a_k) = a_k$$

For (multiple) binary classification problems, the activation is a logistic sigmoid

$$y_k = \sigma(a_k) = \frac{1}{1 + \exp(-a_k)}$$

For multi-class classification problems, the soft-max activation function is used

$$y_k = \sigma(a_k) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

Feedforward network functions (cont.)

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\Big(\sum_{j=1}^{M} w_{kj}^{(2)} h\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{1}\Big) + w_{k0}^{(2)}\Big)$$

Grouping together the set of weight and bias parameters into a vector $\mathbf{w}$, shows that the neural network model is simply a overall non-linear function

- from a set of input variables $\{x_i\}_{i=1}^{D}$ to a set of output variables $\{y_k\}_{k=1}^{K}$

By defining an additional input $x_0 = 1$, the bias parameter can be absorbed

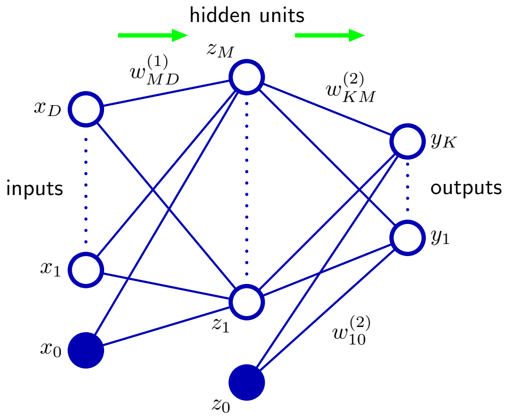
$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} \tag{6}$$

Similarly, defining $z_0 = 1$ we can absorb also the second-layer bias parameter

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\Big(\sum_{j=0}^{M} w_{kj}^{(2)} h\Big(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\Big)\Big) \tag{7}$$

Feedforward network functions (cont.)

This function can be represented in the form of a (neural?) network diagram



Input information is combined, transformed and propagated thru the network

Feedforward network functions (cont.)

The neural network model comprises two stages of processing

- In that, each resembles a perceptron model
- Hence, the misname multi-layer perceptron

The key difference is that the neural network works with continuous sigmoidal non-linearities, whereas the perceptron uses step-function non-linearities

- Feed-forward neural networks are differentiable models wrt the parameters

Feedforward network functions (cont.)

If the activation functions in all the hidden units are taken to be linear, it is always possible to find an equivalent network without hidden units

- A linear combination of linear combination is a linear combination

The network architecture we described is the most commonly used one

- It is generalisable: Additional layers, recurrencies, ...
- It can sparsified: By-passes and skip-layer connections

Feedforward network functions (cont.)

The approximation properties of feed-forward network models are widely studied

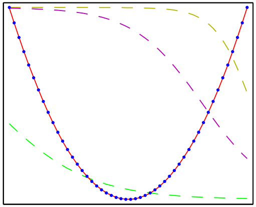
- Neural networks are said to be **universal approximators**

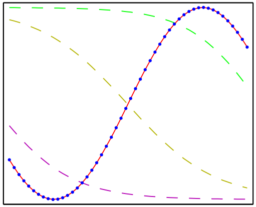
Feed-forward network models can uniformly approximate any continuous function on a compact (closed and bounded) domain to arbitrary accuracy

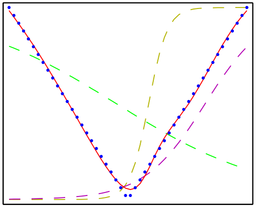
- Provided the network has a sufficiently large number of hidden units
- This is valid for a wide range of activation functions (not polynomials)

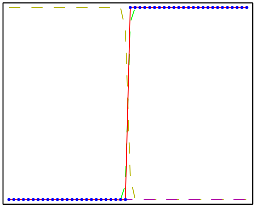
Feedforward network functions (cont.)

$N = 50$ points uniformly sampled over $(-1, +1)$, $M = 3$ hidden units

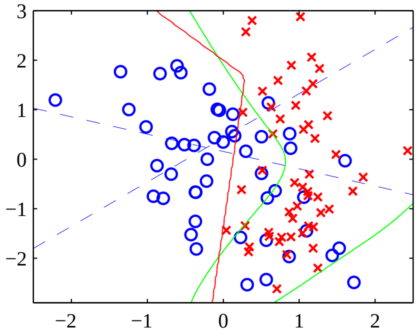
$f(x) = x^2$


$f(x) = \sin(x)$


$f(x) = |x|$


$f(x) = H(x)$


Feedforward network functions (cont.)



Feedforward network functions (cont.)

Question is, how to find suitable values for the parameters from training data?

- Both maximum likelihood- and Bayesian-type approaches

It is important to note also that multiple distinct choices of the parameter vector (the weights) can give rise to the same input-output mapping function

- We can for example interchange the values of the weights
- For two-layer networks, there are $M!2^M$ equivalent orderings

Network training
Feed-forward network functions

Network training

We have viewed neural networks as a general class of parametric nonlinear functions from a vector $\mathbf{x}$ of input variables to a vector $\mathbf{y}$ of output variables

A simple approach to the problem of determining the network parameters is to make an analogy with the early discussion on polynomial curve fitting

- Minimisation of a sum-of-squares error function

Given a training set comprising a set of input vectors $\{\mathbf{x}_n\}_{n=1}^N$, together with a corresponding set of target vectors $\{\mathbf{t}_n\}_{n=1}^N$, we can minimise the error function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n)||^2 \tag{8}$$

A more general view of network training, by giving a probabilistic interpretation

Network training (cont.)

We start with regression problems, and we consider a single target $t \in \mathbb{R}$

We assume that t has a Gaussian distribution with an $\mathbf{x}$-dependent mean given by the network output $y(\mathbf{x}, \mathbf{w})$ and precision β (inverse variance of the noise)

$$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}) \tag{9}$$

This is now a fairly restrictive assumption, but the approach can be generalised

For such a conditional distribution $p(t|\mathbf{x}, \mathbf{w})$, it suffices to take the output unit activation function to be the identity $(y_k = \sigma(a_k) = a_k$, with $k = 1)$

- the network can approximate any continuous function from $\mathbf{x}$ to y

Network training (cont.)

Given N independent identically distributed observations $\mathtt{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ along with corresponding target values $\mathtt{t} = \{t_1, \ldots, t_n\}$, we construct the corresponding likelihood function

$$p(\mathtt{t}|\mathtt{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}, \beta)$$

Taking the negative logarithm, we obtain the error function

$$\frac{\beta}{2}\sum_{n=1}^{N}\Big(y(\mathbf{x}_n, \mathbf{w}) - t_n\Big)^2 - \frac{N}{2}\ln\beta + \frac{N}{2}\ln 2\pi \qquad (10)$$

which can be used to learn the parameters $\mathbf{w}$ and β

Network training (cont.)

Consider first the determination of $\mathbf{w}$, where the maximisation of the likelihood function is equivalent to the minimisation of the sum-of-squares error function[1]

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(y(\mathbf{x}_n, \mathbf{w}) - t_n\Big)^2 \tag{11}$$

The value of $\mathbf{w}$ found by minimising $E(\mathbf{w})$ is $\mathbf{w}_{ML}$, from maximum likelihood

The nonlinearity of the network function $y(\mathbf{x}_n, \mathbf{w})$ causes the error $E(\mathbf{w})$ to be non-convex, with local maxima (local minima) of the likelihood (error function)

Having found $\mathbf{w}_{ML}$, β_{ML} can be found by minimising the negative log likelihood

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N}\Big(y(\mathbf{x}_n, \mathbf{w}_{ML}) - t_n\Big)^2 \tag{12}$$

The evaluation of β_{ML} can start only after the iterative optimisation for $\mathbf{w}_{ML}$

[1] We discard additive and multiplicative constants

Network training (cont.)

With multiple target variables that are independent conditional on **x** and **w** with shared β, the conditional distribution of the target values is given by

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \beta^{-1}\mathbf{I}) \tag{13}$$

Following the same argument, the maximum likelihood weights $\mathbf{w}_{ML}$ are determined by minimising the sum-of-squares error function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n)||^2$$

For a number K of target variables, the noise precision is then given by

$$\frac{1}{\beta_{ML}} = \frac{1}{NK}\sum_{n=1}^{N} ||y(\mathbf{x}_n, \mathbf{w}_{ML} - \mathbf{t}_n)||^2 \tag{14}$$

Network training (cont.)

For binary classification, a single target $t \in \{0, 1\}$ with

- $t = 1$ for class $\mathcal{C}_1$
- $t = 0$ for class $\mathcal{C}_2$

A network having a single output and activation function the logistic sigmoid

$$y = \sigma(a) \equiv \frac{1}{1 + \exp(-a)}, \quad \text{so that } 0 \leq y(\mathbf{x}, \mathbf{w}) \leq 1 \tag{15}$$

We interpret $y(\mathbf{x}, \mathbf{w})$ as conditional probability $p(\mathcal{C}_1|\mathbf{x})$, $p(\mathcal{C}_2|\mathbf{x}) = 1 - y(\mathbf{x}, \mathbf{w})$

The conditional distribution of the target given the inputs is then a Bernoulli

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \Big(1 - y(\mathbf{x}, \mathbf{w})\Big)^{1-t} \tag{16}$$

Network training (cont.)

If we consider a training set of independent observations, the error function (the negative log likelihood) is the cross-entropy error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \Big(t_n \ln \big(y(\mathbf{x}_n, \mathbf{w}) + (1 - t_n) \ln \big(1 - y(\mathbf{x}_n, \mathbf{w})\big)\big)\Big) \qquad (17)$$

Network training (cont.)

If we have K separate binary classifications to perform, we can use a network having K outputs each of which has a logistic sigmoid activation function

- Associated with each output is a binary class label $t_k \in \{0, 1\}$, with $k = 1, \dots, K$

If we assume that the class labels are independent, given the input vector

- then the conditional distribution of the targets is

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k} \Big(1 - y_k(\mathbf{x}, \mathbf{w})\Big)^{1-t_k} \tag{18}$$

Taking the negative logarithm, the likelihood function gives the error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} \Big(t_{nk} \ln \Big(y_k(\mathbf{x}_n, \mathbf{w}) + (1 - t_{nk}) \ln \big(1 - y_k(\mathbf{x}_n, \mathbf{w})\big)\Big)\Big) \tag{19}$$

Network training (cont.)

If we consider a standard two-layer network, we see that the weight parameters in the first layer of the network are shared between the various outputs

- in the linear model each classification problem is solved independently

The first layer of the network can be viewed as performing a nonlinear feature extraction, and the sharing of features between the different outputs can save on computation and can also lead to improved generalization

Network training (cont.)

Multiclass classification: Inputs assigned to one of K mutually exclusive classes

- The binary target variables $t_k \in \{0,1\}$ have 1-of-K coding scheme
- The network outputs are interpreted as $y_k(\mathbf{x},\mathbf{w}) = p(t_k = 1|\mathbf{x})$
- The error function is

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n,\mathbf{w}) \tag{20}$$

The output unit activation function is given by the softmax function

$$y_k(\mathbf{x},\mathbf{w}) = \frac{\exp\left(a_k(\mathbf{x},\mathbf{w})\right)}{\sum_j \exp\left(a_j(\mathbf{x},\mathbf{w})\right)}, \quad \text{with } y_k \in [0,1] \text{ and } \sum_k y_k = 1 \tag{21}$$

Network training (cont.)

Summarising, there is a natural choice of both output unit activation function and matching error function, according to the type of problem being solved

- For regression, we use linear outputs and a sum-of-squares error
- For (multiple independent) binary classifications, we use logistic sigmoid outputs and a cross-entropy error function
- For multiclass classifications, we use softmax outputs with the corresponding multiclass cross-entropy error function

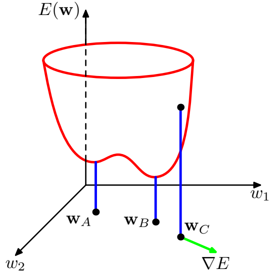
For binary classification problems, we can use either a network with single logistic sigmoid output, or alternatively we can use a network with two outputs having a softmax output activation function

Parameter optimisation

Network training

Parameter optimisation

The task of finding a weight vector **w** minimising the chosen function $E(\mathbf{w})$



Error function is a surface over weight space

A small step from **w** to $\mathbf{w} + \delta\mathbf{w}$ leads to a change in the function $\delta E \simeq \delta\mathbf{w}^T \nabla E(\mathbf{w})$

- Vector ∇E is the local gradient at **w**
- Vector $\nabla E(\mathbf{w})$ points in the direction of greatest rate of increase of $E(\mathbf{w})$

Function $E(\mathbf{w})$ is smooth continuous in **w**

- Its smallest value is at a point in the weight space where gradient vanishes

$$\nabla E(\mathbf{w}) = 0$$

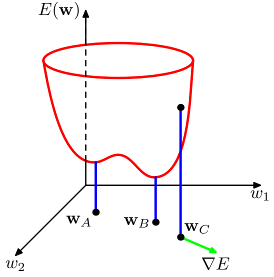
Points where $\nabla E(\mathbf{w}) = 0$ are stationary (minima, maxima, and saddle points)

- We search for a vector **w** such that $E(\mathbf{w})$ takes its smallest value

Parameter optimisation (cont.)

The error function typically has a nonlinear dependence on weights and biases

- Typically, there are many points in weight space at which the gradient either vanishes or is numerically very small



A minimum that corresponds to the smallest value of $E(\mathbf{w})$ for any weight vector is a **global minimum**

Any other minima corresponding to higher values of the error function $E(\mathbf{w})$ are said to be **local minima**

There is no hope of finding an analytical solution to the equation $\nabla E(\mathbf{w}) = 0$

- Iterative numerical procedures (widely studied, lots of literature)

Parameter optimisation (cont.)

Most techniques involve choosing some initial value $\mathbf{w}^{(0)}$ for the weight vector and then moving through weight space in a succession of steps of the form

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{22}$$

Many algorithms use gradient information, and therefore require that the value of $\nabla E(\mathbf{w})$ is evaluated after each step at the new weight vector $\mathbf{w}^{(\tau+1)}$

To appreciate the importance of gradient information, we analyse a local quadratic approximation to the error fuction based on a Taylor expansion

Local quadratic approximation
Network training

Local quadratic approximation

Consider the Taylor expansion of $E(\mathbf{w})$ around some point $\hat{\mathbf{w}}$ in weight space

$$E(\mathbf{w}) \simeq E(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})^T \mathbf{b} + \frac{1}{2}(\mathbf{w} - \hat{\mathbf{w}})^T \mathbf{H}(\mathbf{w} - \hat{\mathbf{w}}) \tag{23}$$

Vector $\mathbf{b}$ is defined to be the gradient of E evaluated at $\hat{\mathbf{w}}$

$$\mathbf{b} \equiv \nabla E \Big|_{\mathbf{w}=\hat{\mathbf{w}}} \tag{24}$$

The Hessian matrix $\mathbf{H} = \nabla\nabla E$ has elements

$$(\mathbf{H})_{ij} \equiv \frac{\partial E}{\partial w_i \partial w_j}\Big|_{\mathbf{w}=\hat{\mathbf{w}}} \tag{25}$$

The corresponding local approximation of the gradient is given by

$$\nabla E \simeq \mathbf{b} + \mathbf{H}(\mathbf{w} - \hat{\mathbf{w}}) \tag{26}$$

Reasonable approximation of the error and its gradient for points $\mathbf{w}$ close to $\hat{\mathbf{w}}$

Local quadratic approximation (cont.)

For a $\mathbf{w}^\star$ that is a minimum of a quadratic approximation of the error function

- There is no linear term: $(\mathbf{w}-\mathbf{w}^\star)^T\mathbf{b}=0$, because $\nabla E=0$ at $\mathbf{w}^\star$

$$E(\mathbf{w}) \simeq E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w}-\mathbf{w}^\star)^T\mathbf{H}(\mathbf{w}-\mathbf{w}^\star) \tag{27}$$

- The Hessian is evaluated at point $\mathbf{w}^\star$

To interpret this, consider the eigenvalue equation for the Hessian matrix

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{28}$$

The eigenvectors $\mathbf{u}_i$ form a complete orthogonal set such that $\mathbf{u}_i^T\mathbf{u}_i = \delta_{ij}$

Local quadratic approximation (cont.)

We can expand $(\mathbf{w} - \mathbf{w}^\star)$ as a linear combination of the eigenvectors

$$\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i \tag{29}$$

This can be regarded as equivalent to a change of coordinate system

- With origin $\mathbf{w}^\star$ and axes rotated to align with the eigenvectors
- Rotation is through a orthogonal matrix whose columns are $\{\mathbf{u}_i\}$

Substituting $\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i$ into $E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^T \mathbf{H}(\mathbf{w} - \mathbf{w}^\star)$ and using $\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i$ and $\mathbf{u}_i^T \mathbf{u}_i = \delta_{ij}$, the error function can be written as

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2} \sum_i \lambda_i \alpha_i^2 \tag{30}$$

Local quadratic approximation (cont.)

Matrix $\mathbf{H}$ is said to be positive definite if and only if

$$\mathbf{v}^T\mathbf{H}\mathbf{v} > 0, \quad \text{for all } \mathbf{v} \tag{31}$$

Because the eigenvectors $\{\mathbf{u}_i\}$ form a complete set, an arbitrary $\mathbf{v}$

$$\mathbf{v} = \sum_i c_i \mathbf{u}_i \tag{32}$$

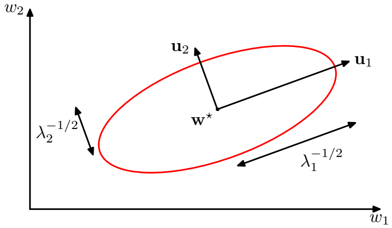
From $\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i$ and $\mathbf{u}_i^T\mathbf{u}_i = \delta_{ij}$ and manipulations and rearrangments

$$\mathbf{v}^T\mathbf{H}\mathbf{v} = \sum_i c_i^2 \lambda_i \tag{33}$$

and the Hessian is positive definite iff all its eigenvalues are positive

Local quadratic approximation (cont.)

In the new coordinate system, whose basis vectors are given by eigenvectors $\mathbf{u}_i$ of the Hessian matrix, contours of constant E are ellipses centred on the origin



In the neighbourhood of a minimum $\mathbf{w}^\star$, the error function can be approximated by a quadratic

The eigenvectors have lengths that are inversely proportional to the square roots of the corresponding eigenvalues λ_i

For a $1D$ weight space, a stationary point $w^\star$ is a minimum if $\partial^2 E / \partial w^2|_{w^\star} > 0$

In D-dimensions, the Hessian matrix, evaluated at $\mathbf{w}^\star$ should be positive definite

Use of gradient information

Network training

Use of gradient information

In the quadratic approximation to the error function

$$E(\mathbf{w}) \simeq E(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})^T \mathbf{b} + \frac{1}{2}(\mathbf{w} - \hat{\mathbf{w}})^T \mathbf{H}(\mathbf{w} - \hat{\mathbf{w}})$$

the error surface is specified by the quantities **b** and **H**

There is a total of $W(W+3)/2$ independent elements (because matrix **H** is symmetric), where W is the dimensionality of **w** (total number of parameters)

The location of the minimum of this quadratic approximation therefore depends on $\mathcal{O}(W^2)$ parameters, and we should not expect to be able to locate the minimum until we have gathered $\mathcal{O}(W^2)$ independent pieces of information

If we do not make use of gradient information, we would expect to have to perform $\mathcal{O}(W^2)$ function evaluations, each of which would require $\mathcal{O}(W)$ steps

- Thus, the computational effort needed to find the minimum using such an approach would be $\mathcal{O}(W^3)$

Use of gradient information (cont.)

Now compare this with an algorithm that makes use of the gradient information

Because each evaluation of ∇E brings W items of information, we might hope to find the minimum of the function in $\mathcal{O}(W)$ gradient evaluations

As we shall see, by using error backpropagation, each such evaluation takes only $\mathcal{O}(W)$ steps and so the minimum can now be found in $\mathcal{O}(W^2)$ steps

For this reason, the use of gradient information forms the basis of practical algorithms for training neural networks

Gradient descent optimisation

Network training

Gradient descent optimisation

The basic approach to using gradient information is to choose weight updates $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$ to comprise a small step in direction of negative gradient

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) \tag{34}$$

where the quantity $\eta > 0$ is commonly known as the **learning rate**

After each step, gradient is re-evaluated for a new weight and process repeated

- The error function is defined with respect to the whole data set
- At each step, the whole set is processed to evaluate the gradient
- $\Longrightarrow$ **Batch learning**

At each step the weight vector is moved in the direction of the greatest rate of decrease of the error function, **gradient descent** or **steepest descent** approach

Gradient descent optimisation (cont.)

There are other approaches such as **conjugate gradient** and **quasi-Netwon** methods, which are much more robust and fast than simple gradient descent

- Unlike gradient descent, these algorithms have the property that the error function always decreases at each iteration unless the weight vector has arrived at a local or global minimum

Gradient descent optimisation (cont.)

There is an **on-line** or **sequential learning** approach that has proven to be valid

- Error functions are based on maximum likelihood for iid observations
- The error functions are sums of terms, one from each observation

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{35}$$

On-line gradient descent or **sequential** or **stochastic gradient descent** makes an update to the weight vector based on one observation at a time, such that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}) \tag{36}$$

Error back-propagation

Neural networks

Error back-propagation

Error back-propagation is an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network

It consists of a local message passing scheme where information is sent alternately forwards and backwards through the network

Error back-propagation (cont.)

Most training algorithms involve an iterative procedure for minimisation of an error function, with adjustments to the weights made in a sequence of steps

At each such step, we can distinguish between two distinct stages

First stage: Derivatives of the error function wrt weights must be evaluated

- The contribution of back-propagation is in providing an efficient method for evaluating such derivatives
- Because it is at this stage that errors are propagated backwards through the network, we use the term backpropagation for derivatives evaluation

Second stage: Derivatives are used to compute adjustments to the weights

- The simplest such technique involves gradient descent

It is important to recognise that the two stages are distinct

Evaluation of error-function derivatives

Error back-propagation

Evaluation of error-function derivatives

We derive the back-propagation algorithm for a general network with arbitrary feed-forward topology and arbitrary differentiable nonlinear activation functions

- For broad class of error functions

The results are illustrated using a simple layered network structure having a single layer of sigmoidal hidden units together with a sum-of-squares error

Many error functions comprise a sum of terms, one for each training point

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{37}$$

We consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in $E(\mathbf{w})$

Evaluation of error-function derivatives (cont.)

For a linear model with outputs y_k as linear combinations of the inputs x_i

$$y_k = \sum_i w_{ki} x_i \tag{38}$$

For an observation n and $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$, the error function contribution

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \tag{39}$$

The gradient of the error function with respect to the weights w_{ji} is given by

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni} \tag{40}$$

It is a local computation, involving a product of an error and the input variable

- Logistic sigmoid activation functions and cross-entropy error function, and softmax activation functions and cross-entropy error function share this

Evaluation of error-function derivatives (cont.)

In a general feed-forward net, each unit computes a weighted sum of its inputs

$$a_j = \sum_i w_{ji} z_i \tag{41}$$

- z_i the activation of a unit, the input, that sends a connection to unit j
- w_{ji} is the weight of that connection

The result of the sum is transformed by a non-linear activation function $h(\cdot)$

$$z_j = h(a_j) \tag{42}$$

- z_j is the activation of unit j

Evaluation of error-function derivatives (cont.)

For each input observation, the calculation of all activations of all hidden and output units can be seen as a forward flow of information through the network

- **Forward propagation**

Evaluation of error-function derivatives (cont.)

Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}

- The error E_n depend on weight w_{ji} via the summed input a_j to unit j

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i \tag{43}$$

- $\delta_j \equiv \frac{\partial a_j}{\partial w_{ji}}$
- $z_i \equiv \frac{\partial a_j}{\partial w_{ji}}$

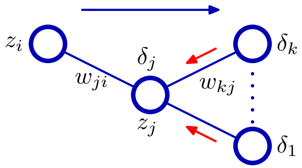
The derivative is obtained by multiplying the δ for the unit at the output end of the weight by z for the unit at the input end of the weight (for the bias $z = 1$)

Evaluation of error-function derivatives (cont.)

Because $\delta_k = y_k - t_k$ for usual output-input activation functions

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{44}$$

The sum rolls over all units k to which unit j sends connection



Back-propagation formula

$$\delta_j = h'(a_j) \sum_k w_{kj}\delta_k \tag{45}$$