Neural networks

Course of Machine Learning
Master Degree in Computer Science
University of Rome ``Tor Vergata''

Giorgio Gambosi

a.a. 2017-2018

Perceptron

Perceptron

- Introduced in the '60s, at the basis of the neural network approach
- Simple model of a single neuron
- Hard to evaluate in terms of probability
- Works only in the case that classes are linearly separable

Definition

It corrisponds to a binary classification model where an item $\mathbf{x}$ is first transformed by a non linear function ϕ and the classified on the basis of the sign of the obtained value. That is,

$$y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}))$$

$f()$ is essentially the sign function

$$f(i) = \begin{cases} -1 & \text{if } i < 0 \\ 1 & \text{if } i \geq 0 \end{cases}$$

The resulting model is a particular generalized linear model. A special case is the one when ϕ is the identity, that is $y(\mathbf{x}) = f(\mathbf{w}^T \mathbf{x})$.

By the definition of the model, $y(\mathbf{x})$ can only be ± 1: we denote $y(\mathbf{x}) = 1$ as $\mathbf{x} \in C_1$ and $y(\mathbf{x}) = -1$ as $\mathbf{x} \in C_2$.

To each element $\mathbf{x}_i$ in the training set, a target value is then associated $t_i \in \{-1, 1\}$.

Cost function

- A natural definition of the cost function would be the number of misclassified elements in the training set
- This would result in a piecewise constant function and gradient optimization could not be applied (we would have zero gradient almost everywhere)
- A better choice is using a piecewise linear function as cost function

Cost function

We would like to find a vector of parameters $\mathbf{w}$ such that, for any $\mathbf{x}_i$, $\mathbf{w}^T\mathbf{x}_i > 0$ if $\mathbf{x}_i \in C_1$ and $\mathbf{w}^T\mathbf{x}_i < 0$ if $\mathbf{x}_i \in C_2$: in short, $\mathbf{w}^T\mathbf{x}_i t_i > 0$.

Each element $\mathbf{x}_i$ provides a contribution to the cost function as follows

1. 0 if $\mathbf{x}_i$ is classified correctly by the model
2. $-\mathbf{w}^T\mathbf{x}_i t_i > 0$ if $\mathbf{x}_i$ is misclassified

Let $\mathcal{M}$ be the set of misclassified elements. Then the cost is

$$E_p(\mathbf{w}) = -\sum_{\mathbf{x}_i \in \mathcal{M}} \mathbf{w}^T \phi(\mathbf{x}_i) t_i$$

The contribution of $\mathbf{x}_i$ to the cost is 0 if $\mathbf{x}_i \notin \mathcal{M}$ and it is a linear function of $\mathbf{w}$ otherwise

Gradient optimization

The minimum of $E_p(\mathbf{w})$ can be found through gradient descent

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}}\Big|_{\mathbf{w}^{(k)}}$$

the gradient of the cost function wrt to $\mathbf{w}$ is

$$\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}} = -\sum_{\mathbf{x}_i \in \mathcal{M}} \phi(\mathbf{x}_i) t_i$$

Then gradient descent can be expressed as

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \eta \sum_{\mathbf{x}_i \in \mathcal{M}_k} \phi(\mathbf{x}_i) t_i$$

where $\mathcal{M}_k$ denotes the set of points misclassified by the model with parameter $\mathbf{w}^{(k)}$

Gradient optimization

Online (or stochastic gradient descent): at each step, only the gradient wrt a single item is considered

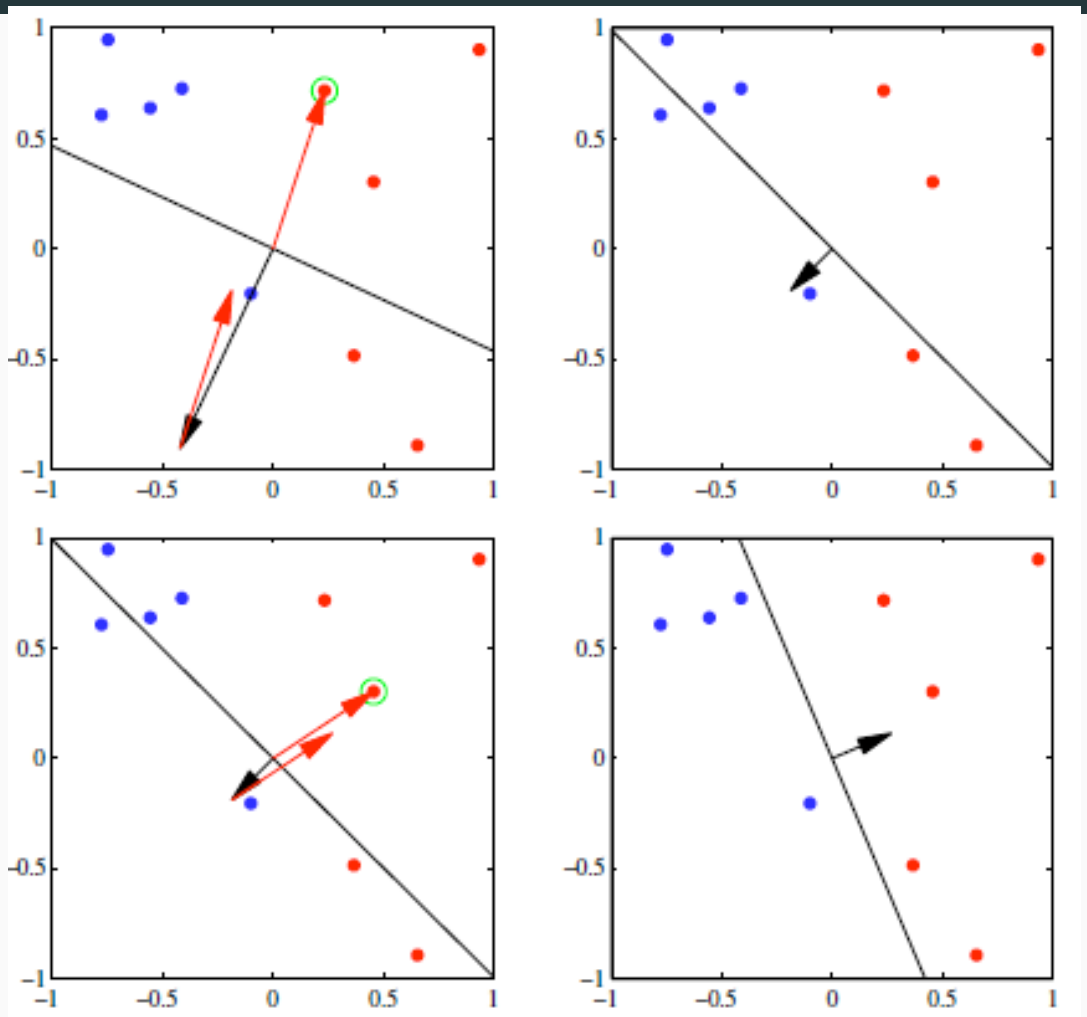
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \eta\phi(\mathbf{x}_i)t_i$$

where $\mathbf{x}_i \in \mathcal{M}_k$

The method works by circularly iterating on all elements and applying the above formula.

Initialize $\mathbf{w}^0$
k := 0 **repeat**
  $k := k + 1$
  $i := (k \bmod n) + 1$
  $y := f(\mathbf{w}^T\phi(\mathbf{x}_i))t_i$
  if $y > 0$ **then** $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)}$
  else $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \eta\phi(\mathbf{x}_i)t_i$
until all elements are well classified

Gradient optimization



In black, decision boundary and corresponding parameter vector $\mathbf{w}$; in red misclassified item vector $\phi(\mathbf{x}_i)$, added by the algorithm to the parameter vector as $\eta\phi(\mathbf{x}_i)$

Gradient optimization

At each step, if $\mathbf{x}_i$ is well classified then $\mathbf{w}^{(k)}$ is unchanged; else, its contribution to the cost is modified as follows

$$\begin{aligned} -(\mathbf{w}^{(k+1)})^T \phi(\mathbf{x}_i) t_i &= -(\mathbf{w}^{(k)})^T \phi(\mathbf{x}_i) t_i - \eta(\phi(\mathbf{x}_i) t_i)^T \phi(\mathbf{x}_i) t_i \\ &= -(\mathbf{w}^{(k)})^T \phi(\mathbf{x}_i) t_i - \eta ||\phi(\mathbf{x}_i)||^2 \\ &< -(\mathbf{w}^{(k)})^T \phi(\mathbf{x}_i) t_i \end{aligned}$$

This contribution is decreasing, however this does not guarantee the convergence of the method, since the cost function could increase due to some other element becoming misclassified if $\mathbf{w}^{(k+1)}$ is used

Perceptron convergence theorem

It is possible to prove that, in the case the classes are linearly separable, the algorithm converges to the correct solution in a finite number of steps.

Let $\hat{\mathbf{w}}$ be a solution (that is, it discriminates C_1 and C_2): if $\mathbf{x}_{k+1}$ is the element considered at iteration $(k+1)$ and it is misclassified, then

$$\mathbf{w}^{(k+1)} - \alpha\hat{\mathbf{w}} = (\mathbf{w}^{(k)} - \alpha\hat{\mathbf{w}}) + \eta\phi(\mathbf{x}_{k+1})t_{k+1}$$

where $\alpha > 0$ is a constant, to be specified later

Perceptron convergence theorem

By squaring left and right expressions of the above formula, we get

$$\left|\left|\mathbf{w}^{(k+1)} - \alpha\hat{\mathbf{w}}\right|\right|^2 =$$

$$\left|\left|\mathbf{w}^{(k)} - \alpha\hat{\mathbf{w}}\right|\right|^2 + \eta^2 \left|\left|\phi(\mathbf{x}_{k+1})\right|\right|^2 + 2\eta(\mathbf{w}^{(k)} - \alpha\hat{\mathbf{w}})^T\phi(\mathbf{x}_{k+1})t_{k+1} =$$

$$\left|\left|\mathbf{w}^{(k)} - \alpha\hat{\mathbf{w}}\right|\right|^2 + \eta^2 \left|\left|\phi(\mathbf{x}_{k+1})\right|\right|^2 + 2\eta(\mathbf{w}^{(k)})^T\phi(\mathbf{x}_{k+1})t_{k+1} - 2\eta\alpha\hat{\mathbf{w}}^T\phi(\mathbf{x}_{k+1})t_{k+1}$$

Since $\mathbf{x}_{k+1}$ was misclassified by hypothesis, $(\mathbf{w}^{(k)})^T\phi(\mathbf{x}_{k+1})t_{k+1} < 0$ and

$$\left|\left|\mathbf{w}^{(k+1)} - \alpha\hat{\mathbf{w}}\right|\right|^2 < \left|\left|\mathbf{w}^{(k)} - \alpha\hat{\mathbf{w}}\right|\right|^2 + \eta^2 \left|\left|\phi(\mathbf{x}_{k+1})\right|\right|^2 - 2\eta\alpha\hat{\mathbf{w}}^T\phi(\mathbf{x}_{k+1})t_{k+1}$$

Perceptron convergence theorem

Let γ be the minimum value of the signed dot product of $\hat{\mathbf{w}}$ with $\phi(\mathbf{x}_i)$ for some element $\mathbf{x}_i$, where the sign depends on the class of $\mathbf{x}_i$

$$\gamma = \min_i \left(\hat{\mathbf{w}}^T \phi(\mathbf{x}_i) t_i\right) = \min_i \left|\hat{\mathbf{w}}^T \phi(\mathbf{x}_i)\right| > 0$$

Let δ be the length of the longest $\phi(\mathbf{x}_i)$

$$\delta^2 = \max_i \ ||\phi(\mathbf{x}_i)||^2$$

Then,

$$\left|\left|\mathbf{w}^{(k+1)} - \alpha\hat{\mathbf{w}}\right|\right|^2 < \left|\left|\mathbf{w}^{(k)} - \alpha\hat{\mathbf{w}}\right|\right|^2 + \eta^2\delta^2 - 2\eta\alpha\gamma$$

Perceptron convergence theorem

By setting

$$\alpha = \frac{\eta\delta^2}{\gamma}$$

we get

$$\left|\left|\mathbf{w}^{(k+1)} - \alpha\hat{\mathbf{w}}\right|\right|^2 < \left|\left|\mathbf{w}^{(k)} - \alpha\hat{\mathbf{w}}\right|\right|^2 - \eta^2\delta^2$$

As can be seen, the squared distance between $\mathbf{w}^{(k+1)}$ and $\hat{\mathbf{w}}$ decreases at each step of an amount greater than $\eta^2\delta^2$

Perceptron convergence theorem

Iterating the above properties on all steps,

$$\left|\left|\mathbf{w}^{(k+1)} - \alpha\hat{\mathbf{w}}\right|\right|^2 < \left|\left|\mathbf{w}^{(0)} - \alpha\hat{\mathbf{w}}\right|\right|^2 - (k+1)\eta^2\delta^2$$

Note that, after

$$\overline{k} = \frac{\left|\left|\mathbf{w}^{(0)} - \alpha\hat{\mathbf{w}}\right|\right|^2}{\eta^2\delta^2} - 1$$

steps we get

$$\left|\left|\mathbf{w}^{(0)} - \alpha\hat{\mathbf{w}}\right|\right|^2 - (k+1)\eta^2\delta^2 = 0$$

So, after at most $\overline{k}$ updates of $\mathbf{w}$, a decision boundary has been derived

Perceptron convergence theorem

Setting $\mathbf{w}^{(0)} = \mathbf{0}$, we have

$$\overline{k} = \frac{\alpha^2}{\eta^2 \delta^2} ||\hat{\mathbf{w}}||^2 - 1 = \frac{\delta^2}{\gamma^2} ||\hat{\mathbf{w}}||^2 - 1 = \frac{\max_i ||\phi(\mathbf{x}_i)||^2}{(\min_i (\hat{\mathbf{w}}^T \phi(\mathbf{x}_i)))^2} ||\hat{\mathbf{w}}||^2 - 1$$

The number of required step is large if $\min_i (\hat{\mathbf{w}}^T \phi(\mathbf{x}_i))$ is small, that is if there exists some $\mathbf{x}_i$ such that $\phi(\mathbf{x}_i)$ is (almost) orthogonal to $\hat{\mathbf{w}}$.

Multilayer networks

- Up to now, only models with a single level of parameters to be learned were considered.
- The model has a generalized linear model structure such as $y = f(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}))$: model parameters are directly applied to input values.
- More general classes of models can be defined by means of sequences of transformations applied on input data, corresponding to multilayered networks of functions.

Multilayer network structure: first layer

For any d-dimensional input vector $\mathbf{x} = (x_1, \ldots, x_d)$, the first layer of a neural network derives $m_1 > 0$ activations $a_1^{(1)}, \ldots, a_{m_1}^{(1)}$ through suitable linear combinations of $x_1, \ldots, x_d$

$$a_j^{(1)} = \sum_{i=1}^{d} w_{ji}^{(1)} x_i + w_{j0}^{(1)} = \overline{\mathbf{x}}^T \mathbf{w}_j^{(1)}$$

where M is a given, predefined, parameter and $\overline{\mathbf{x}} = (1, x_1, \ldots, x_d)^T$.

Multilayer network structure: first layer

Each activation $a_j^{(1)}$ is tranformed by means of a non-linear activation function h_1 to provide a vector $\mathbf{z}^{(1)} = (z_1^{(1)}, \ldots, z_{m_1}^{(1)})^T$ as output from the layer, as follows

$$z_j^{(1)} = h_1(a_j^{(1)}) = h_1(\overline{\mathbf{x}}^T \mathbf{w}_j^{(1)})$$

here h_1 is some approximate threshold function, such as a sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

or a hyperbolic tangent

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1}{1 + e^{-2x}} - \frac{1}{1 + e^{2x}} = \sigma(2x) - \sigma(-2x)$$

Observe that this corresponds to defining m_1 units, where unit j implements a GLM on $\mathbf{x}$ to derive $z_j^{(1)}$.

Multilayer network structure: inner layers

Vector $\mathbf{z}^{(1)}$ provides an input to the next layer, where m_2 hidden units compute a vector $\mathbf{z}^{(2)} = (z_1^{(2)}, \ldots, z_{m_2}^{(1)})^T$ by first performing linear combinations of the input values

$$a_k^{(2)} = \sum_{i=1}^{m_1} w_{ki}^{(2)} a_i^{(1)} + w_{k0}^{(2)} = (\overline{\mathbf{z}}^{(1)})^T \overline{\mathbf{w}}_k^{(2)}$$

and then applying function h_2, as follows

$$z_k^{(2)} = h_2((\overline{\mathbf{z}}^{(1)})^T \overline{\mathbf{w}}_k^{(2)})$$

Multilayer network structure: inner layers

The same structure can be repeated for each inner layer, where layer r has m_r units which, from input vector $\mathbf{z}^{(r-1)}$, derive output vector $\mathbf{z}^{(r-1)}$ through linear combinations

$$a_k^{(r)} = (\overline{\mathbf{z}}^{(r-1)})^T \overline{\mathbf{w}}_k^{(r)}$$

and non linear transformation

$$z_k^{(r)} = h_r((\overline{\mathbf{z}}^{(r-1)})^T \overline{\mathbf{w}}_k^{(r)})$$

Multilayer network structure: output layer

For what concerns the last layer, say layer t, an output vector $\mathbf{y} = \mathbf{z}^{(t)}$ is again produced by means of m_t output units by first performing linear combinations on $\mathbf{z}^{(t-1)}$

$$a_k^{(t)} = (\overline{\mathbf{z}}^{(t-1)})^T \overline{\mathbf{w}}_k^{(t)}$$

and then applying function h_t

$$y_k = z_k^{(t)} = h_t((\overline{\mathbf{z}}^{(t-1)})^T \overline{\mathbf{w}}_k^{(t)})$$

where:

- h_t is the identity function in the case of regression
- h_t is a sigmoid in the case of binary classification
- h_t is a softmax in the case of multiclass classification

3 layer networks

A sufficiently powerful model is provided in the case of 3 layers (input, hidden, output).

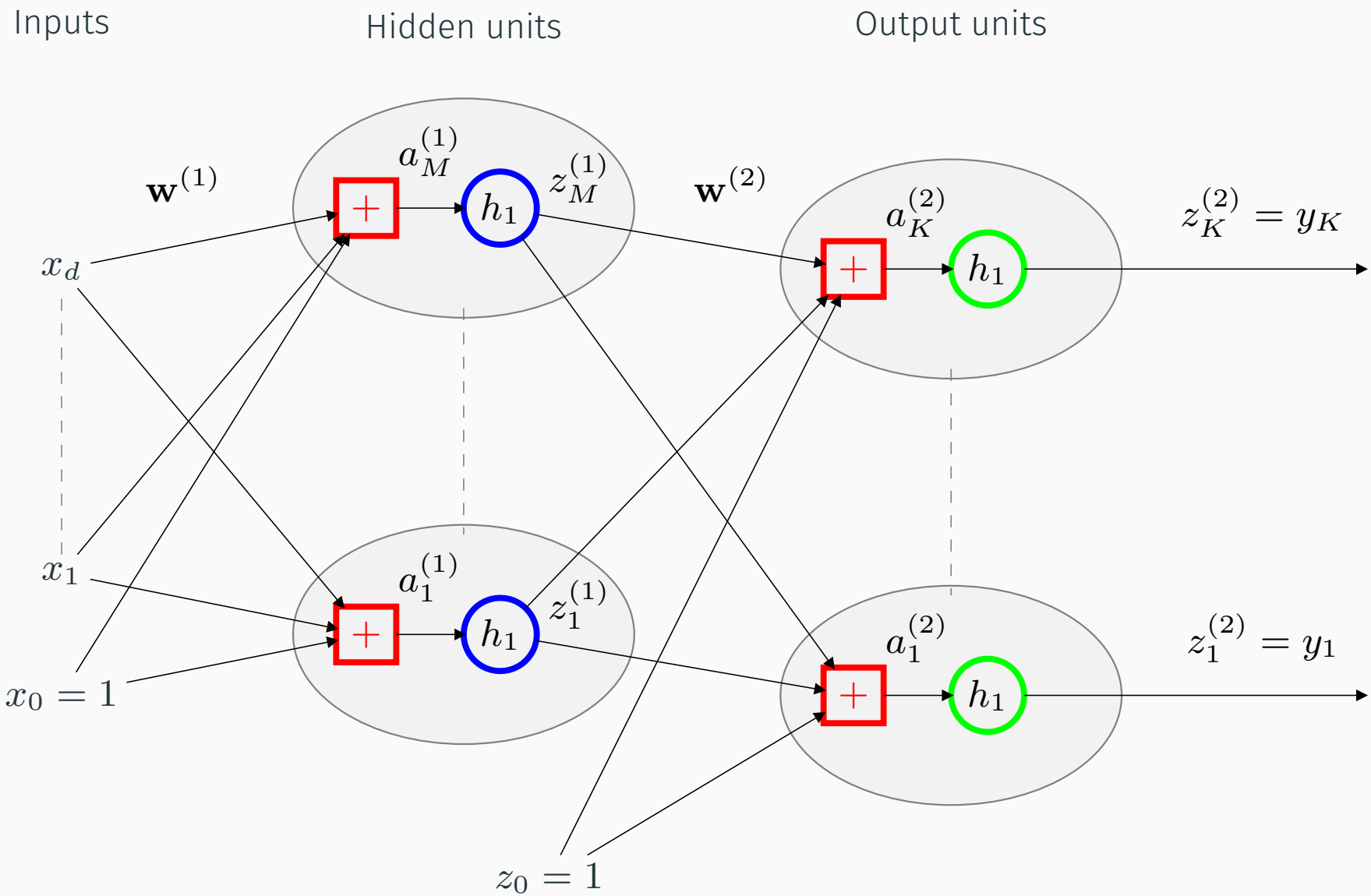
For example, applying this model for K-class classification corresponds to the following overall network function for each y_k, $k = 1, \ldots, K$

$$y_k = \sigma \left(\sum_{j=1}^{M} w_{kj}^{(2)} h \left(\sum_{i=1}^{d} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right)$$

where the number M of hidden units is a model structure parameter.

The resulting network can be seen as a GLM where base functions are not predefined wrt to data, but are instead parameterized by coefficients in $\mathbf{w}^{(1)}$.

3 layer networks



Approximating functions with neural networks

Neural networks, despite their simple structure, are sufficient powerful models to act as universal approximators.

It is possible to prove that any continuous function can be approximated, at any by means of two-layered neural networks with sigmoidal activation functions. The approximation can be indefinitely precise, as long as a suitable number of hidden units is defined.

3-layered neural network training

Maximum likelihood and neural networks

The training phase of a neural network implies learning the values of all parameters from a training set $(\mathbf{X}, \mathbf{t}) = \{(\mathbf{x}_1, \mathbf{t}_1), (\mathbf{x}_2, \mathbf{t}_2), \ldots, (\mathbf{x}_n, \mathbf{t}_n)\}$. In the case of 3-layered networks, this corresponds to learning $\mathbf{w} = \mathbf{w}^{(1)} \cup \mathbf{w}^{(2)}$.

As usual, learning can be performed by minimizing some cost function, in dependance of the problem considered and the assumed probabilistic model.

In the case of maximum likelihood, the minimization of the cost function is equivalent to the maximization of the likelihood of the training set, given the model and its parameters.

ML and regression in the case $K = 1$

Probabilistic model: for each element $(\mathbf{x}_i, t_i)$ of the training set, the value $y_i = y(\mathbf{x}_i, \mathbf{w})$ returned by the network is normally distributed around the target value t_i with variance σ^2 to be determined.

This is equivalent to assuming that, given an element $\mathbf{x}$, its unknown target value t is normally distributed around the returned value $y = y(\mathbf{x}, \mathbf{w})$ with same variance σ^2: that is,

$$p(t|\mathbf{x}, \mathbf{w}, \sigma^2) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \sigma^2)$$

ML and regression in the case $K = 1$

The likelihood of the training set is

$$L(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{i=1}^{n} p(t_i|\mathbf{x}_i, \mathbf{w}, \sigma^2) = \prod_{i=1}^{n} \mathcal{N}(t_i|y(\mathbf{x}_i, \mathbf{w}), \sigma^2)$$

and the log-likelihood

$$\begin{aligned} l(\mathbf{t}|\mathbf{X}, \mathbf{x}, \sigma^2) &= \log L(\mathbf{t}|\mathbf{x}, \mathbf{w}, \sigma^2) \\ &= -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y(\mathbf{x}_i, \mathbf{w}) - t_i)^2 \end{aligned}$$

ML and regression in the case $K = 1$

As well known, maximizing the log-likelihood wrt $\mathbf{w}$ is equivalent to minimizing the cost function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(y(\mathbf{x}_i, \mathbf{w}) - t_i)^2$$

Differently than in the case of linear regression, $y(\mathbf{x}, \mathbf{w})$ is now not linear and, in general, has several local minima.

The variance can be estimated once maximum (or at least local maximum) likelihood values are available by setting the corresponding derivative to zero

$$\frac{\partial l(\mathbf{t}|\mathbf{X}, \mathbf{w}_{ML}, \sigma^2)}{\partial \sigma^2} = 0$$

thus obtaining

$$\sigma^2_{ML} = \frac{1}{n}\sum_{i=1}^{n}(y(\mathbf{x}_i, \mathbf{w}_{ML}) - t_i)^2$$

ML and regression in the case $K > 1$

Probabilistic model: if we assume that target variables are conditionally independent given $\mathbf{x}$ and $\mathbf{w}$, with common variance σ^2, the overall distribution of $\mathbf{t}$ given $\mathbf{x}$ is a multivariate gaussian with mean $\mathbf{y} = \mathbf{y}(\mathbf{x}, \mathbf{w})$: that is,

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \sigma^2) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \sigma^2\mathbf{I}) = \prod_{j=1}^{K} \mathcal{N}(t_j|y_j(\mathbf{x}, \mathbf{w}), \sigma^2)$$

This results in the training set likelihood,

$$L(\mathbf{T}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{i=1}^{n} p(\mathbf{t}_i|\mathbf{x}_i, \mathbf{w}, \sigma^2) = \prod_{i=1}^{n}\prod_{j=1}^{K} \mathcal{N}(t_{ij}|y_j(\mathbf{x}_i, \mathbf{w}), \sigma^2)$$

and log-likelihood

$$l(\mathbf{T}|\mathbf{X}, \mathbf{w}, \sigma^2) = -\frac{nK}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}\sum_{j=1}^{K}(y_j(\mathbf{x}_i, \mathbf{w}) - t_{ij})^2$$

ML and regression in the case $K > 1$

The cost function is now

$$E(\mathbf{w}) = \sum_{i=1}^{n} \sum_{j=1}^{K} (y_j(\mathbf{x}_i, \mathbf{w}) - t_{ij})^2$$

and the variance, given a maximum likelihood estimation $\mathbf{w}_{ML}$ of the parameters, can now be estimated as

$$\sigma^2_{ML} = \frac{1}{nK} \sum_{i=1}^{n} ||\mathbf{y}(\mathbf{x}_i, \mathbf{w}_{ML}) - \mathbf{t}_i||^2$$

ML and regression in the case $K > 1$

For each element $(\mathbf{x}, \mathbf{t})$ of the training set, the derivative of the error function $E(\mathbf{W})$ wrt the linear combination $a_k^{(2)} = (\overline{\mathbf{w}}_k^{(2)})^T \mathbf{z}^{(1)}$ computed by the k-th output unit is

$$\frac{\partial E(\mathbf{W})}{\partial a_k^{(2)}} = \frac{\partial}{\partial a_k^{(2)}} \left(\frac{1}{2} \sum_{j=1}^{K} (y_j - t_j)^2 \right)$$

In linear regression, $y_j = a_j^{(2)}$ for all $j = 1, \ldots, K$, hence

$$\frac{\partial E(\mathbf{W})}{\partial a_k^{(2)}} = \frac{\partial}{\partial a_k^{(2)}} \left(\frac{1}{2} \sum_{i=1}^{K} (a_j^{(2)} - t_j)^2 \right) = a_k^{(2)} - t_k = y_k - t_k$$

Let $y(\mathbf{x}, \mathbf{W}) = p(C_1|\mathbf{x})$: we refer to a probabilistic model where the conditional probability of the target value, given the feature values, is distributed according to a Bernoulli

$$p(t|\mathbf{x}, \mathbf{W}) = y(\mathbf{x}, \mathbf{W})^t (1 - y(\mathbf{x}, \mathbf{W}))^{1-t}$$

The likelihood of the traning set is then

$$L(\mathbf{t}|\mathbf{X}, \mathbf{W}) = \prod_{i=1}^{n} y(\mathbf{x}_i, \mathbf{W})^{t_i} (1 - y(\mathbf{x}_i, \mathbf{W}))^{1-t_i}$$

with log-likelihood

$$l(\mathbf{t}|\mathbf{X}, \mathbf{W}) = \sum_{i=1}^{n} (t_i \ln y_i + (1 - t_i) \ln(1 - y_i))$$

The cost function is $E(\mathbf{W}) = -l(\mathbf{t}|\mathbf{X}, \mathbf{W})$ and its derivative wrt $a_k^{(2)}$

$$\frac{\partial E(\mathbf{W})}{\partial a_k^{(2)}} = -t_k \frac{1}{y_k} \frac{\partial y_k}{\partial a_k^{(2)}} + (1 - t_k) \frac{1}{1 - y_k} \frac{\partial y_k}{\partial a_k^{(2)}}$$

Since $y_k = \sigma(a_k^{(2)})$, we get

$$\frac{\partial y_k}{\partial a_k^{(2)}} = \frac{\partial \sigma(a_k^{(2)})}{\partial a_k^{(2)}} = \sigma(a_k^{(2)})(1 - \sigma(a_k^{(2)})) = y_k(1 - y_k)$$

As a consequence,

$$\begin{aligned}\frac{\partial E(\mathbf{W})}{\partial a_k^{(2)}} &= -t_k \frac{1}{y_k} y_k(1-y_k) + (1-t_k)\frac{1}{1-y_k} y_k(1-y_k) \\ &= -t_k(1-y_k) + (1-t_k)y_k \\ &= y_k - t_k\end{aligned}$$

Again, as in the case of regression, the derivative is equal to the difference between value computed by the network and corresponding target.

It is possible, also in this case, to prove that

$$\frac{\partial E(\mathbf{W})}{\partial a_k^{(2)}} = y_k - t_k$$

Parameter optimization

Iterative methods to minimize $E(\mathbf{w})$

The error function $E(\mathbf{w})$ is usually quite hard to minimize:

- there exist many local minima
- for each local minimum there exist many equivalent minima
  - any permutation of hidden units provides the same result
  - changing signs of all input and output links of a single hidden unit provides the same result

  hence, if the network has M hidden units, for each local minimum there exists a set of $M!2^M$ equivalent minima

Analytical approaches to minimization cannot be applied: resort to iterative methods (possibly comparing results from different runs).

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \Delta\mathbf{w}^{(k)}$$

Gradient descent

At each step, two stages:

1. the derivatives of the error functions wrt all weights are evaluated at the current point
2. weights are adjusted (resulting into a new point) by using the derivatives

Batch gradient descent

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\Big|_{\mathbf{w}^{(k)}}$$

- at each step the weight vector is moved in the direction of greatest decrease
- the whole training set is used at each step: quite expensive
- more efficient methods can be applied (conjugate gradient, quasi-newton)

On-line gradient descent

We explot the property that the error function is the sum of a collection of terms, each characterizing the error corresponding to each observation

$$E(\mathbf{w}) = \sum_{i=1}^{n} E_i(\mathbf{w})$$

the update is based on one training set element at a time

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\partial E_i(\mathbf{w})}{\partial \mathbf{w}}\Big|_{\mathbf{w}^{(k)}}$$

- at each step the weight vector is moved in the direction of greatest decrease only wrt the error for a specific data element
- only one training set element is used at each step: less expensive at each step (more steps may be necessary)
- makes it possible to escape from local minima

Backpropagation

Backpropagation

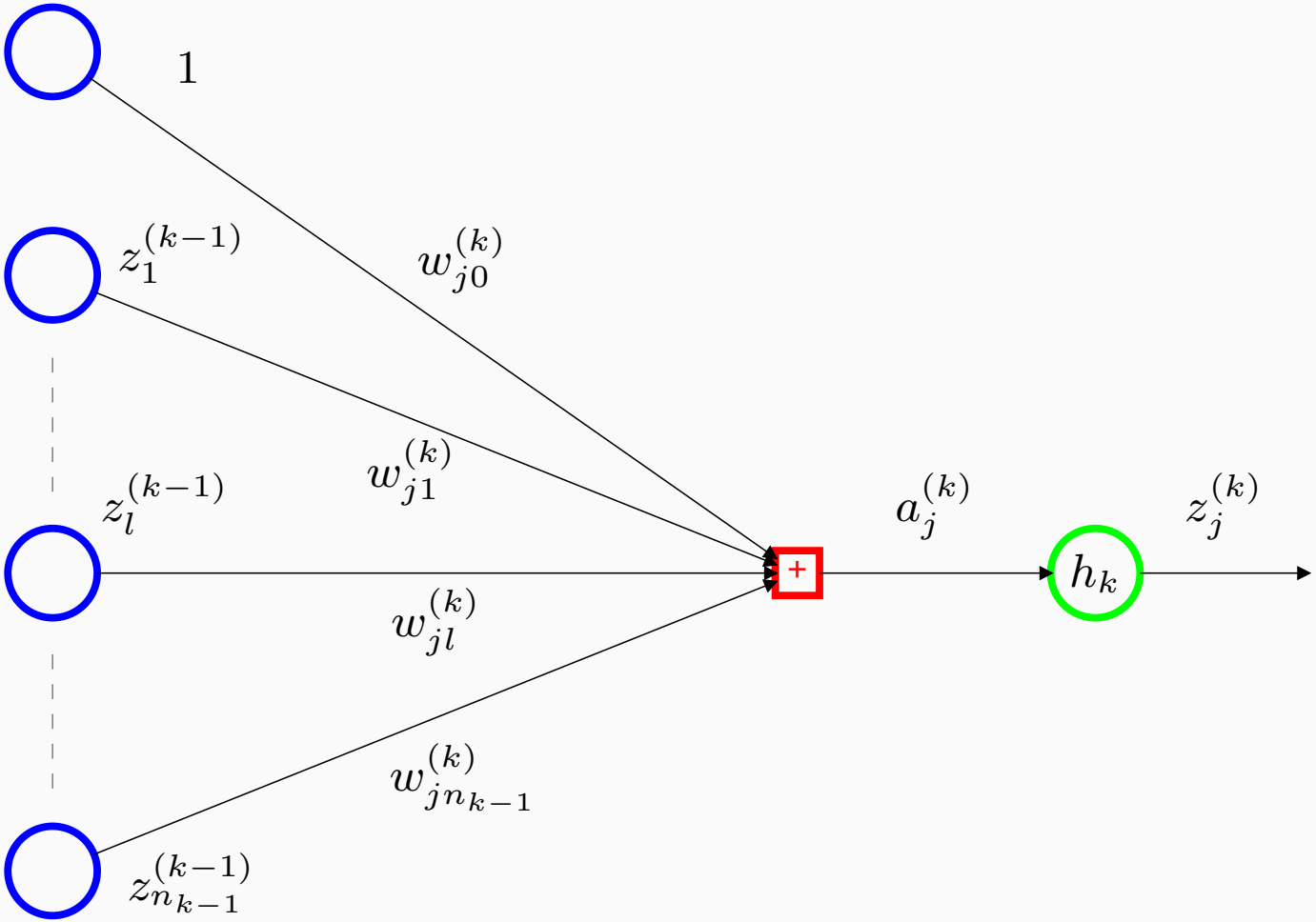
Algorithm applied to evaluate derivatives of the error wrt all weights

It can be interpreted in terms of backward propagation of a computation in the network, from the output towards input units.

It provides an efficient method to evaluate derivatives wrt weights. It can be applied also to compute derivatives of output wrt to input variables, to provide evaluations of the Jacobian and the Hessian matrices at a given point.

Backpropagation

Assume a feed-forward neural network with arbitrary topology and differentiable activation functions and error function.



Backpropagation

- All variables z_i could be either an input variable to the network or the output from a unit in the preceding layer
- The variable a_j could also be direclty returned (h_k being the identity function)

Assumption on the error function: it may be expressed, given a training set, as the sum of the errors corresponding to single elements of the training set

$$E(\mathbf{w}) = \sum_{i=1}^{n} E_i(\mathbf{w})$$

If E_i is differentiable, so is E, with derivative given by the sum of the derivatives of functions E_i.

Backpropagation

- Assume that, for each element $(\mathbf{x}_i, \mathbf{t}_i)$ of the training set, the feature values $\mathbf{x}_i$ have been given as input to the network and both the activation values for each unit and the output values are available: this step is denoted as forward propagation
- We wish to evaluate the derivative of E_i wrt to parameter $w_{jl}^{(k)}$, which associates a weight to the contribution of $z_l^{(k-1)}$ to the unit computing $a_j^{(k)}$
- E_i is a function of $w_{jl}^{(k)}$ only through the following sum

$$a_j^{(k)} = \sum_{r=1}^{m} w_{jr}^{(k)} z_r^{(k-1)}$$

Backpropagation

Let us define $\delta_j^{(k)}$ as follows:

$$\delta_j^{(k)} = \frac{\partial E_i}{\partial a_j^{(k)}}$$

Since

$$\frac{\partial a_j^{(k)}}{\partial w_{jl}^{(k)}} = \frac{\partial}{\partial w_{jl}^{(k)}} \sum_{r=1}^{m} w_{jr}^{(k)} z_r^{(k-1)} = z_l^{(k-1)}$$

it results

$$\frac{\partial E_i}{\partial w_{jl}^{(k)}} = \delta_j^{(k)} z_l^{(k-1)}$$

To compute the derivatives of E_i wrt to all parameters, it is necessary to compute $\delta_j^{(k)}$ for all network units.

Backpropagation

Let us first consider the output, that is $z_j^{(k)} = y_j$.

As observed before, in this case we have

$$\delta_j^{(k)} = \frac{\partial E_i}{\partial a_j^{(k)}} = y_j - t_j$$

Hidden unit.

- any change of $a_j^{(k)}$ has effect on E_i by inducing changes for all variables $a_l^{(k+1)}$
- the effect on E_i is a function of the sum of the effect of the change of $a_j^{(k)}$ on all variables $a_r^{(k+1)}$

$$\delta_j^{(k)} = \frac{\partial E_i}{\partial a_j^{(k)}} = \sum_{r=1}^{n_{k+1}} \frac{\partial E_i}{\partial a_r^{(k+1)}} \frac{\partial a_r^{(k+1)}}{\partial a_j^{(k)}} = \sum_{r=1}^{n_{k+1}} \delta_r^{(k+1)} \frac{\partial a_r^{(k+1)}}{\partial a_j^{(k)}}$$

Since by definition

$$a_r^{(k+1)} = \sum_l w_{rl}^{(k+1)} z_l^{(k)}$$

$$z_j^{(k)} = h_k(a_j^{(k)})$$

it results

$$\frac{\partial a_r^{(k+1)}}{\partial a_j^{(k)}} = \frac{\partial a_r^{(k+1)}}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial a_j^{(k)}} = w_{rj}^{(k+1)} h_k'(a_j^{(k)})$$

and

$$\delta_j^{(k)} = h_k'(a_j^{(k)}) \sum_{r=1}^{n_{k+1}} \delta_r^{(k+1)} w_{rj}^{(k+1)}$$

Backpropagation

$$\delta_j^{(k)} = h_k'(a_j^{(k)}) \sum_{r=1}^{n_{k+1}} \delta_r^{(k+1)} w_{rj}^{(k+1)}$$

can the be evaluated if the following are known

- $w_{rj}^{(k+1)}$, $r = 1, \ldots, k+1$: this are assumed as known for any single back propagation step
- $a_j^{(k)}$: this is computed, during forward propagation, from the current $\mathbf{w}$ and the input values
- $\delta_r^{(k+1)}$, $r = 1, \ldots, k+1$: these can be computed from the network output and the target values by applying a backward propagation of the values from the last to the first network layers (that is, in opposite sense wrt to the output computation)

Backpropagation

Example of backpropagation on a 3-layered network:

1. The feature values $\mathbf{x}_i$ of a training set item are provided as input to the network: all values $a_j^{(1)}$, $a_j^{(2)}$, $z_j^{(1)}$, $z_j^{(2)} = y_j$ are derived and made available

2. Starting from output and target values, the δ values for each output variables is derived, as $\delta_j^{(2)} = y_j - t_j$

3. For each hidden unit, the corresponding δ value is computed, as

$$\delta_j^{(1)} = h_1'(a_j^{(1)}) \sum_{i=1}^{K} w_{ij}^{(2)} \delta_i^{(2)} = h_1'(a_j^{(1)}) \sum_{i=1}^{K} w_{ij}^{(2)} (y_j - t_j)$$

which, in the usual case $h_1(x) = \sigma(x)$, results into

$$\delta_j^{(1)} = \sigma(a_j^{(1)})(1 - \sigma(a_j^{(1)})) \sum_{i=1}^{K} w_{ij}^{(2)} (y_j - t_j) = z_j^{(1)}(1 - z_j^{(1)}) \sum_{i=1}^{K} w_{ij}^{(2)} (y_j - t_j)$$

4. For each parameter $w_{jl}^{(k)}$, where $k = 1, 2$, the value of the derivative of the function error wrt $w_{jl}^{(k)}$ at the current value $\mathbf{w}$ of all weights is computed as

$$\frac{\partial E_i}{\partial w_{jl}^{(k)}} = \delta_j^{(k)} z_l^{(k-1)}$$

which results into

$$\frac{\partial E_i}{\partial w_{jl}^{(2)}} = z_l (y_j - t_j)$$

$$\frac{\partial E_i}{\partial w_{jl}^{(1)}} = x_l z_j (1 - z_j) \sum_{i=1}^{K} w_{ij}^{(2)} (y_j - t_j)$$

Backpropagation

Iterate the preceding steps on all items in the training set (or a subset of them). In fact, since

$$E(\mathbf{w}) = \sum_{i=1}^{n} E_i(\mathbf{w})$$

it is

$$\frac{\partial E}{\partial w_{jl}^{(k)}} = \sum_{i=1}^{n} \frac{\partial E_i}{\partial w_{jl}^{(k)}}$$

This provides an evaluation of $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ at the current point $\mathbf{w}$.

Backpropagation

Once $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ is known, a single step of gardient descent can be performed

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\Big|_{\mathbf{w}^{(i)}}$$

The whole process can be made more efficient through on-line descent, that is by considering a single training set element at a time.

Computational efficiency of backpropagation

A single evaluation of error function derivatives requires $O(|\mathbf{w}|)$ steps

Alternative approach: finite differences. Perturb each weight w_{ij} in turn and approximate the derivative as follows

$$\frac{\partial E_i}{\partial w_{ij}} = \frac{E_i(w_{ij} + \varepsilon) - E_i(w_{ij} - \varepsilon)}{2\varepsilon} + O(\varepsilon^2)$$

This requires $O(|\mathbf{w}|)$ steps for each weight, hence $O(|\mathbf{w}|^2)$ steps overall.