Lecture 5: Logistic Regression. Neural Networks

- Logistic regression
- Comparison with generative models
- Feed-forward neural networks
- Backpropagation
- Tricks for training neural networks

Recall: Binary classification

- We are given a set of data $\langle \mathbf{x}_i, y_i \rangle$ with $y_i \in \{0, 1\}$.
- The probability of a given input $\mathbf{x}$ to have class $y = 1$ can be computed as:

$$P(y = 1|\mathbf{x}) = \frac{P(\mathbf{x}, y = 1)}{P(\mathbf{x})} = \frac{1}{1 + \exp(-a)} = \sigma(a)$$

where

$$a = \ln \frac{P(\mathbf{x}|y = 1)P(y = 1)}{P(\mathbf{x}|y = 0)P(y = 0)}$$

- σ is the sigmoid function (also called "squashing") function
- a is the log-odds of the data being class 1 vs. class 0

Recall: Modelling for binary classification

$$P(y=1|\mathbf{x}) = \sigma\left(\ln \frac{P(\mathbf{x}|y=1)P(y=1)}{P(\mathbf{x}|y=0)P(y=0)}\right)$$

- One approach is to model $P(y)$ and $P(\mathbf{x}|y)$, then use the approach above for classification (naive Bayes, Gaussian discriminants)
- This is called generative learning, because we can actually use the model to generate (i.e. fantasize) data
- Another idea is to *model directly* $P(y|\mathbf{x})$
- This is called *discriminative learning*, because we only care about discriminating (i.e. separating) examples of the two classes.
- We focus on this approach today

Error function

- Maximum likelihood classification assumes that we will find the hypothesis that maximizes the (log) likelihood of the training data:

$$\arg\max_h \log P(\langle \mathbf{x}_i, y_i \rangle_{i=1\ldots m} | h) = \arg\max_h \sum_{i=1}^{n} \log P(\mathbf{x}_i, , y_i) | h)$$

  (using the usual i.i.d. assumption)

- But if we use discriminative learning, we have *no model of the input distribution*
- Instead, we want to maximize the *conditional probability* of the labels, given the inputs:

$$\arg\max_h \sum_{i=1}^{m} \log P(y_i | \mathbf{x}_i, h)$$

The cross-entropy error function

- Suppose we interpret the output of the hypothesis, $h(\mathbf{x}_i)$, as the probability that $y_i = 1$
- Then the log-likelihood of a hypothesis h is:

$$\begin{aligned}\log L(h) &= \sum_{i=1}^{m} \log P(y_i|\mathbf{x}_i, h) = \sum_{i=1}^{m} \begin{cases} \log h(\mathbf{x}_i) & \text{if } y_i = 1 \\ \log(1 - h(\mathbf{x}_i)) & \text{if } y_i = 0 \end{cases} \\ &= \sum_{i=1}^{m} y_i \log h(\mathbf{x}_i) + (1 - y_i) \log(1 - h(\mathbf{x}_i))\end{aligned}$$

- The *cross-entropy error function* is the opposite quantity:

$$J(h) = -\left(\sum_{i=1}^{m} y_i \log h(\mathbf{x}_i) + (1 - y_i) \log(1 - h(\mathbf{x}_i))\right)$$

Logistic regression

- Suppose we represent the hypothesis itself as a logistic function of a linear combination of inputs:

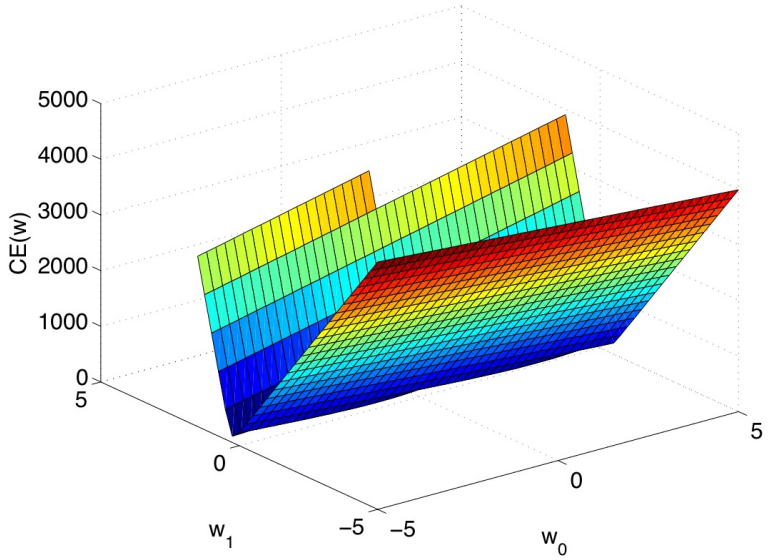
$$h(\mathbf{x} = \frac{1}{1 + \exp(\mathbf{w}^T\mathbf{x})}$$

  This is also known as a *sigmoid neuron*
- Suppose we interpret $h(\mathbf{x})$ as $P(y = 1|\mathbf{x})$
- Then the log-odds ratio,

$$\ln\left(\frac{P(y = 1|\mathbf{x})}{P(y = 0|\mathbf{x})}\right) = \mathbf{w}^T\mathbf{x}$$

  which is linear (nice!)
- We can find the optimum weight by minimizing cross-entropy (maximizing the conditional likelihood of the data)

Cross-entropy error surface for logistic function



$$J(h) = -\left(\sum_{i=1}^{m} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i))\right)$$

Nice error surface, unique minimum, but *cannot solve in closed form*

Maximization procedure: Gradient ascent

- First we compute the gradient of $\log L(\mathbf{w})$ wrt $\mathbf{w}$:

$$\nabla \log L(\mathbf{w}) = \sum_i y_i \frac{1}{h_{\mathbf{w}}(\mathbf{x}_i)} h_{\mathbf{w}}(\mathbf{x}_i)(1 - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i$$

$$+ (1 - y_i) \frac{1}{1 - h_{\mathbf{w}}(\mathbf{x}_i)} h_{\mathbf{w}}(\mathbf{x}_i)(1 - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i(-1)$$

$$= \sum_i \mathbf{x}_i(y_i - y_i h_{\mathbf{w}}(\mathbf{x}_i) - h_{\mathbf{w}}(\mathbf{x}_i) + y_i h_{\mathbf{w}}(\mathbf{x}_i)) = \sum_i (y_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i$$

- The update rule (because we maximize) is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla \log L(\mathbf{w}) = \mathbf{w} + \alpha \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i}$$

  where $\alpha \in (0, 1)$ is a step-size or learning rate parameter
- This is called *logistic regression*

Another algorithm for optimization

- Recall Newton's method for finding the zero of a function $g : \mathbb{R} \rightarrow \mathbb{R}$
- At point w^i, approximate the function by a straight line (its tangent)
- Solve the linear equation for where the tangent equals 0, and move the parameter to this point:

$$w^{i+1} = w^i - \frac{g(w^i)}{g'(w^i)}$$

Application to machine learning

- Suppose for simplicity that the error function J has only one parameter
- We want to optimize J, so we can apply Newton's method to find the zeros of $J' = \frac{d}{dw}J$
- We obtain the iteration:

$$w^{i+1} = w^i - \frac{J'(w^i)}{J''(w^i)}$$

- Note that there is *no step size parameter*!
- This is a *second-order method*, because it requires computing the second derivative
- But, if our error function is quadratic, this will find the global optimum in one step!

Second-order methods: Multivariate setting

- If we have an error function J that depends on many variables, we can compute the *Hessian matrix*, which contains the second-order derivatives of J:

$$H_{ij} = \frac{\partial^2 J}{\partial w_i \partial w_j}$$

- The inverse of the Hessian gives the "optimal" learning rates
- The weights are updated as:

$$\mathbf{w} \leftarrow \mathbf{w} - H^{-1} \nabla_{\mathbf{w}} J$$

- This is also called Newton-Raphson method

Which method is better?

- Newton's method usually requires significantly fewer iterations than gradient descent
- Computing the Hessian requires a batch of data, so there is no natural on-line algorithm
- Inverting the Hessian explicitly is expensive, but almost never necessary
- Computing the product of a Hessian with a vector can be done in linear time (Schraudolph, 1994)

Newton-Raphson for logistic regression

- Leads to a nice algorithm called *recursive least squares*
- The Hessian has the form:

$$\mathbf{H} = \mathbf{\Phi}^T \mathbf{R} \mathbf{\Phi}$$

  where $\mathbf{R}$ is the diagonal matrix of $h(\mathbf{x}_i)(1 - h(\mathbf{x}_i))$
- The weight update becomes:

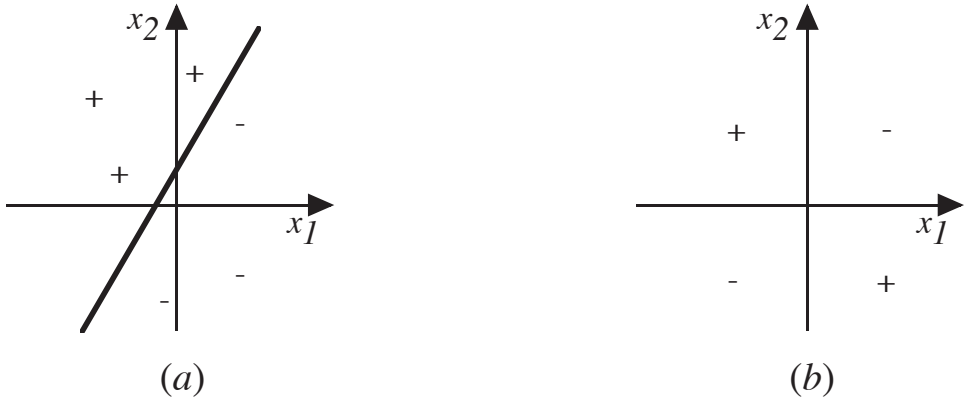
$$\mathbf{w} \leftarrow (\mathbf{\Phi}^T \mathbf{R} \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{R} (\mathbf{\Phi}\mathbf{w} - \mathbf{R}^{-1}(\mathbf{\Phi}\mathbf{w} - \mathbf{y})$$

Logistic vs. Gaussian Discriminant

- Comprehensive study done by Ng and Jordan (2002)
- If the Gaussian assumption is correct, as expected, Gaussian discriminant is better
- If the assumption is violated, Gaussian discriminant suffers from bias (unlike logistic regression)
- In practice, Gaussian discriminant, tends to converge quicker to a less helpful solution
- Note that they optimize *different error function*!

From neurons to networks

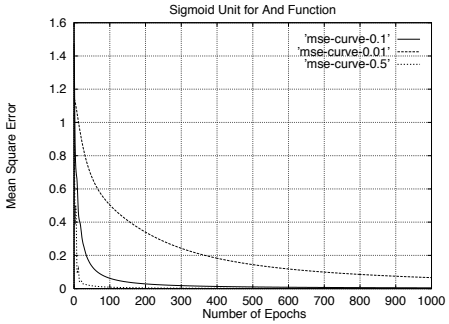
- Logistic regression can be used to learn *linearly separable* problems

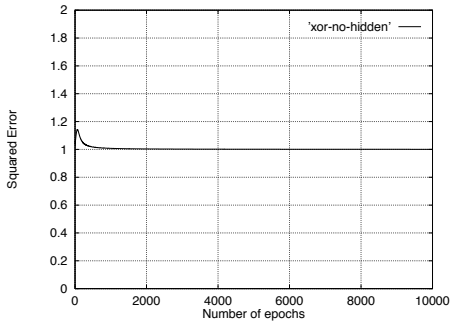


(a) (b)

- If the instances are not linearly separable, the function cannot be learned correctly (e.g. XOR)
- One solution is to provide fixed, non-linear basis functions instead of inputs (like we did with linear regression)
- Today: learn such non-linear functions from data

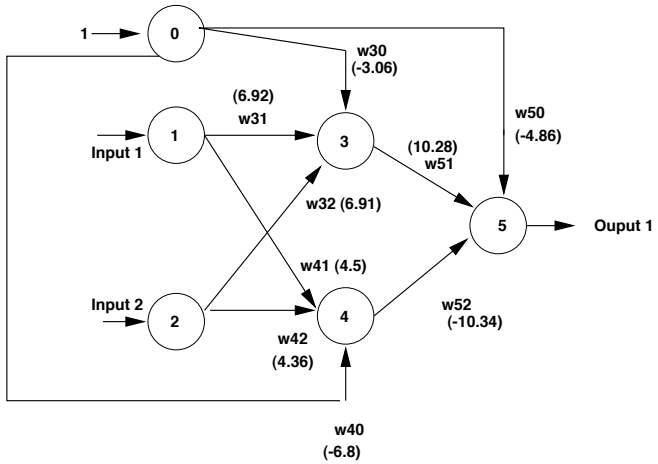
Example: Logical functions of two variables



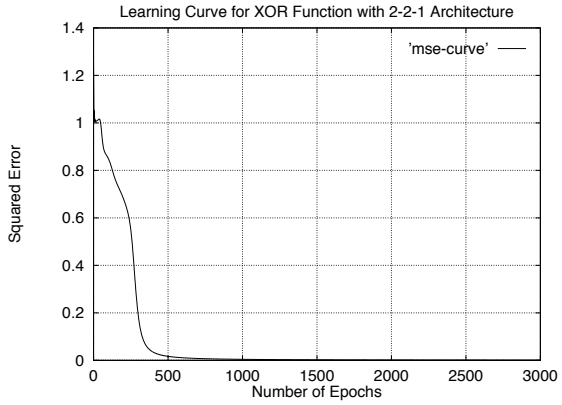


- One sigmoid neuron can learn the AND function (left) but not the XOR function (right)
- In order to learn discrimination in data sets that are not linearly separable, we need *networks of sigmoid units*

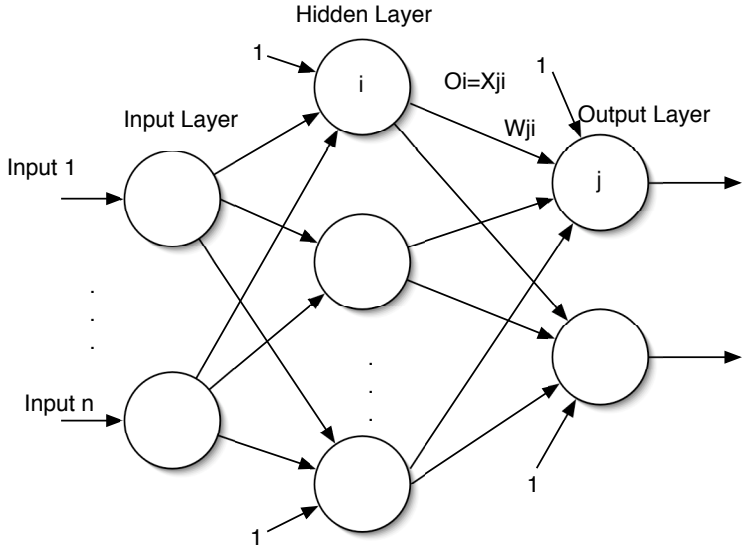
Example: A network representing the XOR function



Input1	Input2	o3	o4	Ouput 1
0	0	0.04	0.001	0.011
0	1	0.98	0.08	0.99
1	0			
1	1			



Feed-forward neural networks

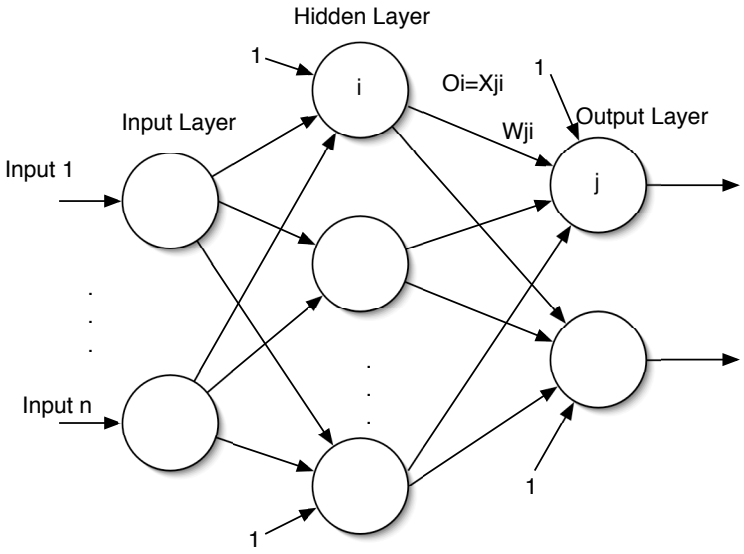


- A collection of units (neurons) with sigmoid or sigmoid-like activations, arranged in *layers*
- Layers 0 is the *input layer*, and its units just copy the inputs (by convention)
- The last layer, K, is called *output layer*, since its units provide the result of the network
- Layers $1, \ldots K-1$ are usually called *hidden layers* (their presence cannot be detected from outside the network)

Why this name?

- In *feed-forward networks* the outputs of units in layer k become inputs for units in layers with index $> k$
- There are no cross-connections between units in the same layer
- There are no backward ("recurrent") connections from layers downstream
- Typically, units in layer k provide input *only* to units in layer $k + 1$
- In *fully connected networks*, all units in layer k are connected to all units in layer $k + 1$

Notation



- $w_{j,i}$ is the weight on the connection from unit i to unit j
- By convention, $x_{j,0} = 1, \forall j$
- The output of unit j, denoted o_j, is computed using a sigmoid: $o_j = \sigma(\mathbf{w}_j^T \mathbf{x}_j)$ where $\mathbf{w}_j$ is the vector of weights on the connections *entering* unit j and $\mathbf{x}_j$ is the vector of inputs to unit j
- By the definition of the connections, $x_{j,i} = o_i$

Computing the output of the network

- Suppose that we want the network to make a prediction for instance $\langle \mathbf{x}, y \rangle$
- In a feed-forward network, this can be done in one *forward pass*:

  For layer $k = 1$ to K

  1. Compute the output of all neurons in layer k:

$$o_j \leftarrow \sigma(\mathbf{w}_j^T \mathbf{x}_j), \forall j \in \text{Layer } k$$

  2. Copy these outputs as inputs to the next layer:

$$x_{j,i} \leftarrow o_i, \forall i \in \text{Layer } k, \forall j \in \text{Layer } k+1$$

Expressiveness of feed-forward neural networks

- A single sigmoid neuron has the same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR
- Every Boolean function can be represented by a network with a single hidden layer, but might require a number of hidden units that is exponential in the number of inputs
- Every bounded continuous function can be approximated with arbitrary precision by a network with one, sufficiently large hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Learning in feed-forward neural networks

- Usually, the network structure (units and interconnections) is specified by the designer
- The learning problem is finding a *good set of weights*
- The answer: *gradient descent*, because the form of the hypothesis formed by the network, $h_{\mathbf{w}}$, is
  - Differentiable! Because of the choice of sigmoid units
  - Very complex! Hence, direct computation of the optimal weights is not possible

Backpropagation algorithm

- Just gradient descent over all weights in the network!
- We put together the two phases described above:
  1. **Forward pass:** Compute the outputs of all units in the network, $o_k, k = N+1, \ldots N+H+1$, going in increasing order of the layers
  2. **Backward pass:** Compute the δ_k updates described before, going from $k = N+H+1$ down to $k = N+1$ (in decreasing order of the layers)
  3. Update to all the weights in the network:

$$w_{i,j} \leftarrow w_{i,j} + \alpha_{i,j}\delta_i x_{i,j}$$

For computing probabilities, drop the $o(1-o)$ terms from the δ

Backpropagation algorithm

1. Initialize all weights to small random numbers.
2. Repeat until satisfied:
   (a) Pick a training example
   (b) Input example to the network and compute the outputs o_k
   (c) For each output unit k, $\delta_k \leftarrow o_k(1-o_k)(y_k - o_k)$
   (d) For each hidden unit l

$$\delta_l \leftarrow o_l(1-o_l) \sum_{k \in outputs} w_{lk}\delta_k$$

   (e) Update each network weight: $w_{ij} \leftarrow w_{ij} + \alpha_{ij}\delta_j x_{ij}$
   - x_{ij} is the input from unit i into unit j (for the output neurons, these are signals received from the hidden layer neurons)
   - α_{ij} is the learning rate or step size

Backpropagation variations

- The previous version corresponds to incremental (stochastic) gradient descent
- An analogous batch version can be used as well:
  - Loop through the training data, accumulating weight changes
  - Update weights
- One pass through the data set is called **epoch**
- Algorithm can be easily generalized to predict probabilities, instead of minimizing sum-squared error
- Generalization is easy for other network structures as well.

Convergence of backpropagation

- Backpropagation performs gradient descent over *all* the parameters in the network
- Hence, if the learning rate is appropriate, the algorithm is guaranteed to converge to a *local minimum* of the cost function
  - NOT the global minimum
  - Can be much worse than global minimum
  - There can be MANY local minima (Auer et al, 1997)
- Partial solution: **restarting** = train multiple nets with different initial weights.
- In practice, quite often the solution found is very good

Adding momentum

On the t-th training sample, instead of the update:

$$\Delta w_{ij} \leftarrow \alpha_{ij}\delta_j x_{ij}$$

we do:

$$\Delta w_{ij}(t) \leftarrow \alpha_{ij}\delta_j x_{ij} + \beta \Delta w_{ij}(t-1)$$

The second term is called **momentum**

Advantages:

- Easy to pass small local minima
- Keeps the weights moving in areas where the error is flat
- Increases the speed where the gradient stays unchanged

Disadvantages:

- With too much momentum, it can get out of a global maximum!
- One more parameter to tune, and more chances of divergence

Choosing the learning rate

- Backprop is VERY sensitive to the choice of learning rate
  - Too large $\Rightarrow$ divergence
  - Too small $\Rightarrow$ VERY slow learning
  - The learning rate also influences the ability to escape local optima
- Very often, different learning rates are used for units inn different layers
- It can be shown that *each unit* has its own optimal learning rate

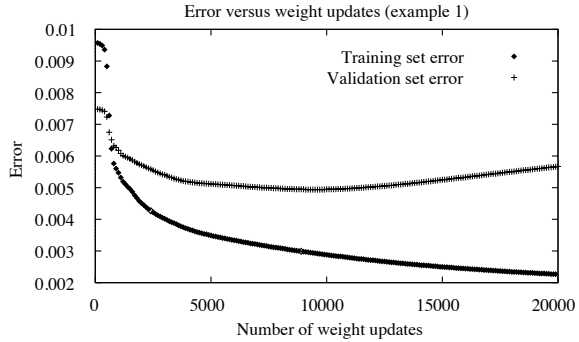
Adjusting the learning rate: Delta-bar-delta

- Heuristic method, works best in batch mode (though there have been attempts to make it incremental)
- The intuition:
  - If the gradient direction is stable, the learning rate should be increased
  - If the gradient flips to the opposite direction the learning rate should be decreased
- A running average of the gradient and a separate learning rate is kept for each weight
- If the new gradient and the old average have the same sign, increase the learning rate by a constant amount
- If they have opposite sign, decay the learning rate exponentially

How large should the network be?

- Overfitting occurs if there are too many parameters compared to the amount of data available
- Choosing the number of hidden units:
  - Too few hidden units do not allow the concept to be learned
  - Too many lead to slow learning and overfitting
  - If the n inputs are binary, $\log n$ is a good heuristic choice
- Choosing the number of layers
  - Always start with one hidden layer
  - Never go beyond 2 hidden layers, unless the task structure suggests something different

Overtraining in feed-forward networks



- Traditional overfitting is concerned with the number of parameters vs. the number of instances
- In neural networks there is an additional phenomenon called *overtraining* which occurs when weights take on large magnitudes, pushing the sigmoids into saturation
- Effectively, as learning progresses, the network has more actual parameters
- *Use a validation set to decide when to stop training!*

Finding the right network structure

- *Destructive methods* start with a large network and then remove (prune) connections
- *Constructive methods* start with a small network (e.g. 1 hidden unit) and add units as required to reduce error

Destructive methods

- Simple solution: consider removing each weight in turn (by setting it to 0), and examine the effect on the error
- Weight decay: give each weight a chance to go to 0, unless it is needed to decrease error:

$$\Delta w_j = -\alpha_j \frac{\partial J}{\partial w_j} - \lambda w_j$$

  where λ is a decay rate
- Optimal brain damage:
  - Train the network to a local optimum
  - Approximate the saliency of each link or unit (i.e., its impact on the performance of the network), using the Hessian matrix
  - Greedily prune the element with the lowest saliency

  This loop is repeated until the error starts to deteriorate

Constructive methods

- Dynamic node creation (Ash):
  - Start with just one hidden unit, train using backprop
  - If the error is still high, add another unit in the same layer and repeat

  Only one layer is constructed
- Meiosis networks (Hanson):
  - Start with just one hidden unit, train using backprop
  - Compute the variance of each weight during training
  - If a unit has one or more weights of high variance, it is split into two units, and the weights are perturbed

  The intuition is that the new units will specialize to different functions.
- Cascade correlation: Add units to correlate with the residual error

Example applications

- Speech phoneme recognition [Waibel] and synthesis [Nettalk]
- Image classification [Kanade, Baluja, Rowley]
- Digit recognition [Bengio, Bouttou, LeCun et al - LeNet]
- Financial prediction
- Learning control [Pomerleau et al]

When to consider using neural networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued, or a vector of values
- Possibly noisy data
- Training time is not important
- Form of target function is unknown
- Human readability of result is not important
- The computation of the output based on the input has to be fast