CS4811 Neural Network Learning Algorithms

From: Stuart Russell and Peter Norvig
Artificial Intelligence a Modern Approach
Prentice Hall Series in Artificial Intelligence, 2003, 2010.

Single perceptron learning

The following is a gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g. For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.

function PERCEPTRON-LEARNING(*examples, network*)
returns a perceptron hypothesis

inputs:
  examples, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output y
  network, a perceptron with weights W_j, $j = 0, \ldots n$ and activation function g
repeat
  for each e **in** *examples* **do**
    in $\leftarrow \sum_{j=0}^{n} W_j \; x_j[e]$ // Compute the weighted sum.
    $err \leftarrow y[e] - g(in)$ // Compute the error.
    $W_j \leftarrow W_j + c \times Err \times g'(in) \times x_j[e]$ // Adjust the weights.
until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS(*network*)

Note that $x_1, \ldots, x_n$ are the real inputs and x_0 is the bias input which is always 1. We'll take $g'(in)$ to be 1 for simplicity.

The stopping criterion can be a combination of the following:

- Convergence: The algorithms stops when every example is classified correctly.
- Number of iterations: The algorithm stops when a preset iteration limit is reached. This puts a time limit in case the network does not converge.
- Inadequate progress; The algorithm stops when the maximum weight change is less than a preset ϵ value. The procedure can find a minimum squared error solution even when the minimum error is not zero.

The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(*examples, network*)
returns a neural network

inputs:
  examples, a set of examples, each with input vector **x** and output vector **y**.
  network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables:Δ, a vector of errors, indexed by network node

for each weight $w_{i,j}$ in *network* do
  $w_{i,j} \leftarrow$ a small random number
repeat
  for each example **(x,y) in** *examples* **do**
    /* Propagate the inputs forward to compute the outputs. */
    for each node i in the input layer **do**   // Simply copy the input values.
      $a_i \leftarrow x_i$
    for $l = 2$ to L **do**   // Feed the values forward.
      for each node j in layer l **do**
        $in_j \leftarrow \sum_i w_{i,j}\, a_i$
        $a_j \leftarrow g(in_j)$
    for each node j in the output layer **do**   // Compute the error at the output.
      $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
    /* Propagate the deltas backward from output layer to input layer */
    for $l = L - 1$ **to** 1 **do**
      for each node i in layer l **do**
        $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$ // "Blame" a node as much as its weight.
    /* Update every weight in network using deltas. */
    for each weight $w_{i,j}$ in *network* **do**
      $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ // Adjust the weights.
until some stopping criterion is satisfied

return *network*