Squashing Functions

- Sigmoids, step functions, and other functions that force their results to be in a limited range are called “squashing functions”.
- It is generally accepted that biological neural system is based on such functions, since there are physical limits to the response level.

nndesign demos on turing

- cs /cs/cs152/matlab
- matlab
- nndtoc
- select demos:
  - 4 Perceptron Rule
  - 8 Taylor’s series (1 & 2D)
  - 9 Steepest descent, quadratic
  - 9 Steepest descent (note saddle)

Batch vs. On-Line

- Two possible methods for weight update:
  - **On-line**: Update immediately as each training sample is applied
  - **Batch**: Update only after all training samples are applied
    - This is accomplished by **averaging** the weight changes that would have been used if each sample were applied independently.

Batch vs. On-Line

- Comparisons:
  - **On-line**: Immediate reaction to a specific input; simpler implementation
  - **Batch**: Smoother, less “jerking around” of weights
- In some applications, such as **adaptive filtering**, the samples are constantly changing. On-line is typically used here. The counterpart to batch would be to use the last so-many samples.

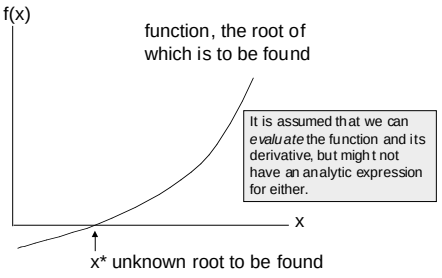
Multivariate Regression

- Regression is another approach that can be used to compute the weights for a *linear* output Adaline (no threshold step function).
- See NAS section 1.7 for derivation.
- It entails solving D+1 simultaneous linear equations, where D is the number of inputs lines.
- The theoretical end result is the same as the one derived earlier under the discussion of the MSE as a quadratic form.

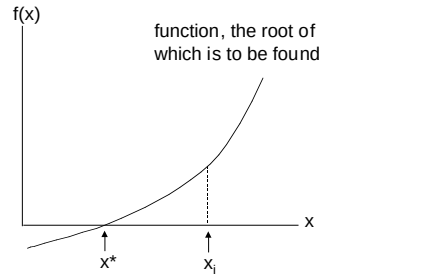
Newton’s Method

- Newton’s method is an alternate to gradient descent as an approximation method.
- Newton’s method requires first computing the inverse of the correlation matrix.
- It is faster, but can also diverge in cases where gradient descent converges, e.g. some non-quadratic surfaces.
- It is possible to combine the two methods.

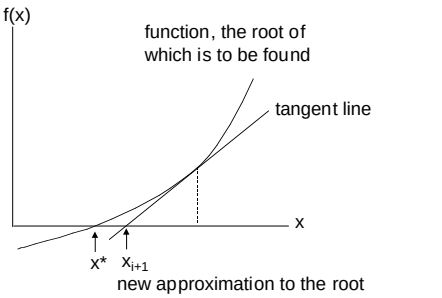
Newton's Method in 1 D



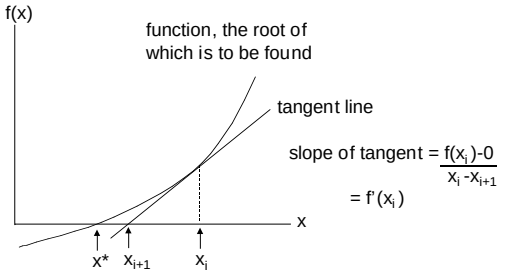
Newton's Method in 1 D



Newton's Method in 1 D



Newton's Method in 1 D



Newton's Method in 1 D

$$f'(x_i) = \frac{f(x_i)-0}{x_i - x_{i+1}}$$

Solving for x_{i+1}:

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

Our application of Newton's Method

- We want to find the root of the *derivative* J' of the MSE function J.
- So in 1D, we'd be using

$$w_{i+1} = w_i - J'(w_i)/J''(w_i)$$

where w_i is an approximation to the optimal weight.

Newton's Method in N dimensions

- In one dimension:
  $w_{i+1} = w_i - J'(w_i)/J''(w_i)$
- In N dimensions, the analogous formula is

  $w_{i+1} = w_i - \nabla J(w_i)\, A^{-1}(w_i)$

  where A is the Hessian matrix, the matrix of 2nd partial derivatives:
  $A_{ij} = \partial/\partial w_i\, \partial/\partial w_j\, J$
- This formula can also be derived using Taylor's series.

Newton's Method in N dimensions

- For the special case of a *quadratic*, the Hessian matrix doesn't depend on the particular value of w.
- In this case, Newton's method arrives at the root in 1 step.
- In general, the Hessian will need to be estimated, and multiple steps will be required.

Demo of Newton's Method

- Use nndtoc, chapter 9
  - Newton's method
  - Steepest descent (for comparison)
  - Note that both methods can get stuck on the saddle, but Newton's does so more readily.

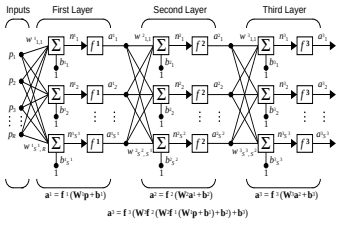
Multi-Level Networks

- Generally much more versatile than single neurons
- No linear separability requirement.
- Training is less obvious and potentially more time consuming.

Multi-Level Networks

- Several varieties, the most common of which is known as:
  - MLP (Multi-Level Perceptron, although Multi-Level Adaline would be more accurate)
  - Backpropagation Network (alluding to a common method of training these networks)

MLP Network



Note that sometimes the input is counted as a "layer".

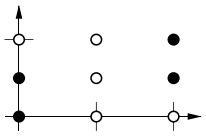
The real layers other than the output are called "hidden" layers.

Bias

b_i = bias

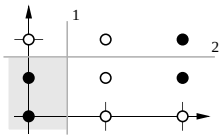
= negative of threshold

Multi-Layer Example



Design a network by hand that implements this decision problem.

Elementary Decision Boundaries



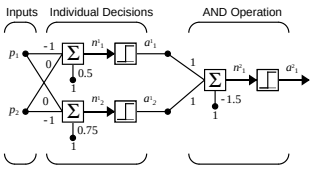
First Boundary:

$$a_1^1 = hardlim(\begin{bmatrix}-1 & 0\end{bmatrix}\mathbf{p} + 0.5)$$

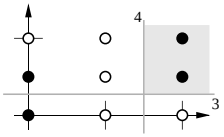
Second Boundary:

$$a_2^1 = hardlim(\begin{bmatrix}0 & -1\end{bmatrix}\mathbf{p} + 0.75)$$

First Subnetwork



Elementary Decision Boundaries



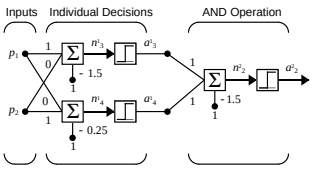
Third Boundary:

$$a_3^1 = hardlim(\begin{bmatrix}1 & 0\end{bmatrix}\mathbf{p} - 1.5)$$

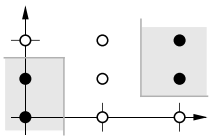
Fourth Boundary:

$$a_4^1 = hardlim(\begin{bmatrix}0 & 1\end{bmatrix}\mathbf{p} - 0.25)$$

Second Subnetwork



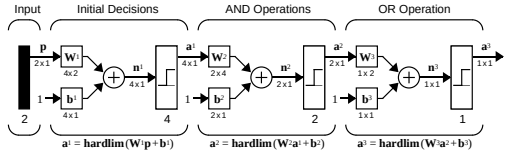
Total Network



$$\mathbf{W}^1 = \begin{bmatrix}-1 & 0\\ 0 & -1\\ 1 & 0\\ 0 & 1\end{bmatrix} \quad \mathbf{b}^1 = \begin{bmatrix}0.5\\ 0.75\\ -1.5\\ -0.25\end{bmatrix}$$

$$\mathbf{W}^2 = \begin{bmatrix}1 & 1 & 0 & 0\\ 0 & 0 & 1 & 1\end{bmatrix} \quad \mathbf{b}^2 = \begin{bmatrix}-1.5\\ -1.5\end{bmatrix}$$

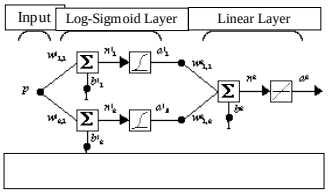
$$\mathbf{W}^3 = \begin{bmatrix}1 & 1\end{bmatrix} \quad \mathbf{b}^3 = \begin{bmatrix}-0.5\end{bmatrix}$$



Demo nnd11f

- Shows a simple 2-level network:
  - 1 input, 1 output
  - 2 neurons in first layer, with 1 weight and 1 bias each, logsig activation function
  - 1 neuron in output layer, with 2 weights and 1 bias
  - output activation function selectable from: purelin (identity), tansig, logsig
- Plot is network output vs. input

Function Approximation Demo nnd11f



$$f^1(n) = \frac{1}{1+e^{-n}}$$

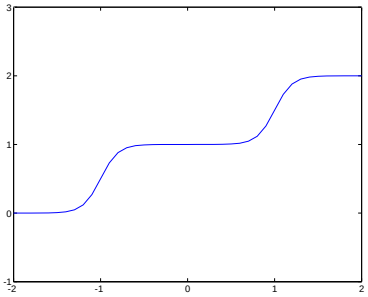
$$f^2(n) = n$$

Nominal Parameter Values

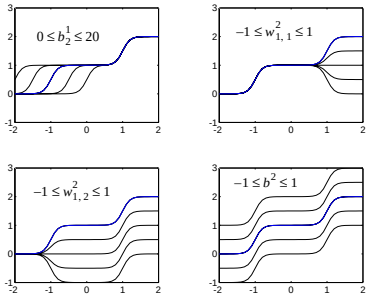
$w^1_{1,1} = 10 \quad w^1_{2,1} = 10 \quad b^1_1 = -10 \quad b^1_2 = 10$

$w^2_{1,1} = 1 \quad w^2_{1,2} = 1 \quad b^2 = 0$

Nominal Response



Parameter Variations



Demo nnd11gn (generalization)

- Trains a 2-level logsig/linear network using gradient descent.
- The samples are sets of points (input, output)
- Vary
  - The complexity of the training input
  - The number of neurons in the first level
- Conclusions?

How to Train a MLP?

- With a single neuron, it is not too hard to see how to adjust the weights based upon the error values. We've already seen a couple of ways.
- With a multi-layer network, it is less obvious. For one thing, what is the "error" for the neurons in non-final layers? Without these, we don't know how to adjust.
- This is called the "credit assignment" problem (maybe should be "blame assignment").

Backpropagation

- Werbos, in his Harvard PhD thesis in 1974 found a method.
- Rumelhart and McClelland, in 1985, discovered the method, presumably independently, and popularized it under the current name.

Backpropagation

- The technique is gradient descent, as explained for Adalines.
- However, the computation of the gradient is less clear.

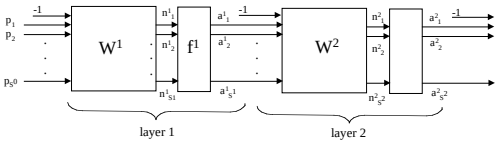
Backpropagation Training Cycle

- **Forward propagation**: Derive the activation values (the inputs to the activation functions) at each neuron, and the final output.
- **Compute the error** in the output.
- **Backpropagate** the error through the network to get "sensitivities" at each neuron. (The gradient approximation is derivable from the sensitivities.)
- Use the sensitivities to **derive weight changes**.
- Apply the weight changes.

Backpropagation Training Cycle

- Backpropagate is a lot like forward propagate.
- Sensitivities are used instead of signal values.
- The sensitivities are the partial derivatives of the MSE with respect to the activation values.
- Basically both are iterated matrix multiplications.

Multilayer Network



For m = 1, 2, …, number of layers:

S^m = size of layer m output vector

a^m = layer m output vector = layer m+1 input vector

($a^m_0 = -1$)

n^m = layer m net activation vector

f^m = layer m activation functions

W^m = layer m weight matrix

Forward Propagation

$$\mathbf{a}^0 = \mathbf{p}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m)$$

$$m = 1, 2, \ldots, M\text{-}1$$

$$\mathbf{a} = \mathbf{a}^M$$