Multilayer Networks

How do we get networks to do more?

Thus far, we have considered:

- PERCEPTRONS: which can only learn linearly-separable pattern classifications

How can we expand network capabilities?

- Use more complex activation functions? (won't help, really…)
- Use more layers? (will help…)
- What are the capabilities of multilayer neural networks?
- How many layers are necessary?

Multilayer networks and function approximation:

- Kolmogorov's Mapping Neural Network Existence Theorem: Given
  $f\left[0,1\right]^{n} \to \Re^{m}, f\left(\vec{x}\right) = \vec{y}$
  f can be implemented exactly by a three layer neural network with (2n+1) elements in its hidden layer
- This makes neural networks universal function approximators.

Kolmogorov's Theorem...

- Can be extended to any bounded input set
- The theorem in itself should not be surprising
- Consider function approximation via series
- One fascinating aspect is its indication that three layers are enough

The proof is good news, but....

- It gives us no idea of how to determine what the activation functions in the hidden and output layers should be

Consider multilayer perceptrons:

- Three layer perceptrons can form any convex (open or closed) decision region
- The number of hidden nodes is an upper bound on the number of sides of a decision region
- Four layer perceptrons can form any polygonal decision region
- Three layers are sufficient for bounded input sets

Multilayer perceptrons:

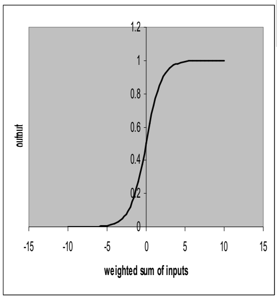
- Rosenblat knew that with the appropriate "input predicates", a layer of perceptrons could learn any categorization of input vectors
- These input predicates are the outputs of the hidden layer
- However, he had no good algorithm for training the weights into the hidden layer (finding linearly separable input predicates)

Consider a continuous perceptron...

- Note that this is a continuous approximation to a threshold…

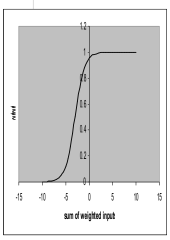
$$f\left(\vec{x}\right) = \frac{1}{1+e^{-\vec{w}^T\vec{x}}}$$

- this is called a *sigmoid*

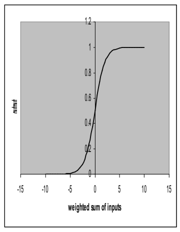


Consider three layers of continuous perceptrons:

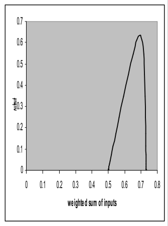
- A sum of two continuous perceptrons can form an "activation bump" in the input space



-



=



Using the bump…

- Weights in the output layer can transfer this activation bump to any output value
- Note that sigmoidal output units provide bounded output
- Linear output units can provide unbounded output

Q: How can we extend train multilayer networks?

- We will show how this is done via the backpropagation algorithm

For a moment, consider linear activation functions...

- This is like a perceptron, without the threshold

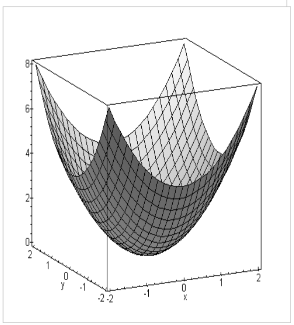
$$f\left(\vec{x}\right) = \vec{w}^T \vec{x}$$

- Several layers of these don't do much, since a sum of linear functions is another linear function

Assume there is a "correct" output, *y*

- Then, the square error is

$$E = \left(y - f(\vec{x})\right)^2 = \left(y - \vec{w}^T \vec{x}\right)^2$$



To minimize error

- By changing weights, find a point where the *gradient* is zero:

$$\nabla_{\vec{w}} E = \frac{\partial E}{\partial \vec{w}} = 0$$

- we can do this by taking steps in the negative gradient direction...

Gradient Descent

- With respect to a weight:

$$\frac{\partial E}{\partial w_i} = \frac{\partial\left[\left(y - f(\vec{x})\right)^2\right]}{\partial w_i} = -2\left(y - f(\vec{x})\right)\frac{\partial\left[f(\vec{x})\right]}{\partial w_i}$$

- assume

$$f\left(\vec{x}\right) = F\left(\vec{w}^T \vec{x}\right)$$

Then...

$$\frac{\partial E}{\partial w_i} = -2\left(y - f(\vec{x})\right)\frac{\partial\left[f(\vec{x})\right]}{\partial w_i} =$$

$$-2\left(y - f(\vec{x})\right)\frac{\partial\left[F(\vec{w}^T \vec{x})\right]}{\partial\left(\vec{w}^T \vec{x}\right)}\frac{\partial\left[\vec{w}^T \vec{x}\right]}{\partial w_i} =$$

$$-2\left(y - f(\vec{x})\right)\frac{\partial\left[F(\vec{w}^T \vec{x})\right]}{\partial\left(\vec{w}^T \vec{x}\right)} x_i$$

For linear activation...

$$F(whatever) = whatever, \qquad \therefore$$

$$\frac{\partial E}{\partial w_i} = -2\left(y - f(\vec{x})\right) x_i$$

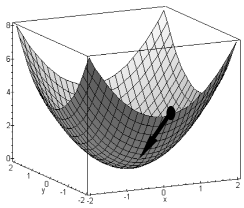
Gradient Descent

$$\Delta w_i = -c\frac{\partial E}{\partial w_i} =$$

$$c\left(y - f(\vec{x})\right)\frac{\partial\left[F(\vec{w}^T\vec{x})\right]}{\partial\left(\vec{w}^T\vec{x}\right)}x_i$$

$$\Delta\vec{w} = -c\nabla_{\vec{w}}E =$$

$$c\left(y - f(\vec{x})\right)\frac{\partial\left[F(\vec{w}^T\vec{x})\right]}{\partial\left(\vec{w}^T\vec{x}\right)}\vec{x}$$



For linear activation...

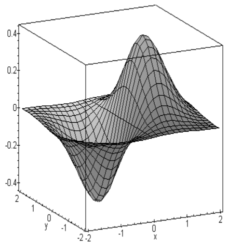
- It's (essentially) the perceptron learning law...

$$\Delta w_i = -c\frac{\partial E}{\partial w_i} = c\left(y - f(\vec{x})\right)x_i$$

$$\Delta\vec{w} = -c\nabla_{\vec{w}}E = c\left(y - f(\vec{x})\right)\vec{x}$$

Consider nonlinear activation functions

- Which we need 3 layers of for interesting nets...
- The square error in the weight space is now a multi-modal function
- However, we can still use gradient descent



The Generalized Delta Rule

- We can take the derivative of the square error with respect to any weight in the network . . . .

$$\frac{\partial E}{\partial w_i} = -2\left(y - f(\vec{x})\right)\frac{\partial\left[F(\vec{w}^T\vec{x})\right]}{\partial\left(\vec{w}^T\vec{x}\right)}x_i$$

The Backpropagation Algorithm

- is the computer implementation of the generalized delta rule
- it gets its name from the way deltas propagate backwards through the network
- appropriate deltas can be derived for any number of layers

Advantages of Backpropagation

- It is founded in the calculus
- It is highly effective in a broad class of problems
- Calculations are entirely local to each neuron
- Computer implementation is painfully easy

Problems with Backpropagation

- it is gradient descent over a multimodal surface, therefore
- it can get stuck on local minima
- it can be slow
- every weight is updated every cycle
- it must take small steps...
- it is only approximate gradient descent in the mean square error space

Next time....

- A good derivation of BP, to give…
- Computer implementation of backprop
- Modifications to make backprop work better