Artificial Intelligence Programming
Neural Networks II

Chris Brooks

Department of Computer Science
University of San Francisco
A network is composed of layers of nodes
- input, hidden, output layers in feedforward nets
An input is applied to the input units
The resulting output is the value of the function the net computes.
Recall that perceptron networks can be trained using the Delta rule.

\[ w_i = w_i + \alpha(t - o)x_i \]

This is a form of gradient descent.

We’re using the error \((t - o)\) to determine the direction of the gradient.

\(\alpha\) determines the size of the “step” we take down the gradient.
As with GAs, part of the challenge with neural nets is representing your problem.

Face recognizer uses 128 x 120 inputs - one for each pixel.

Other common choices:
- Binary values
- Boolean attributes
- Real-valued numbers
While perceptrons have the advantage of a simple learning algorithm, their computational limitations are a problem.

What if we add another “hidden” layer?

Computational power increases

- With one hidden layer, can represent any continuous function
- With two hidden layers, can represent any function

Problem: How to find the correct weights for hidden nodes?
20-4: Multilayer Network Example

Output units

\[ a_i \]

\[ W_{j,i} \]

Hidden units

\[ a_j \]

\[ W_{k,j} \]

Input units

\[ a_k \]
Backpropagation is an extension of the perceptron learning algorithm to deal with multiple layers of nodes.

Nodes use sigmoid activation function

\[ g(\text{input}_i) = \frac{1}{1 + e^{-\text{input}_i}}. \]

\[ g'(\text{input}_i) = g(\text{input}_i)(1 - g(\text{input}_i)) \]

We will still “follow the gradient”, where \( g' \) gives us the gradient.
Notation:

- $a_j$ - output of the jth hidden unit.
- $o_i$ - output of the ith output unit.
- $t_i$ - output for the ith training example
- Output error for output node $i$: $(t_i - o_i)$
- Delta rule for output node $i$:
  $$\delta_i = (t_i - o_i) \ast g(input_i) \ast (1 - g(input_i))$$
- Weight updating (hidden-output):
  $$W_{j,i} = W_{j,i} + \alpha \ast a_j \ast \delta_i$$
Updating input-hidden weights:

Idea: each hidden node is responsible for a fraction of the error in $\delta_i$.

Divide $\delta_i$ according to the strength of the connection between the hidden and output node.

For each hidden node $j$

$\delta_j = g(input)(1 - g(input)) \sum_{i \in outputs} W_{j,i}\delta_i$

Update rule for input-hidden weights:

$W_{k,j} = W_{k,j} + \alpha \cdot input_k \cdot \delta_j$
The whole algorithm can be summed up as:

While not done:
    for d in training set
        Apply inputs of d, propagate forward.
        for node $i$ in output layer
            $\delta_i = \text{output} \times (1 - \text{output}) \times (t_{\text{exp}} - \text{output})$
        for each hidden node
            $\delta_j = \text{output} \times (1 - \text{output}) \times \sum W_{k,i} \delta_i$

Adjust each weight

$W_{j,i} = W_{j,i} + \alpha \times \delta_i \times \text{input}_j$
Stopping conditions

When to stop training?
- Fixed number of iterations
- Total error below a set threshold
- Convergence - no change in weights
20-10: Comments on Backpropagation

- Also works for multiple hidden layers
- Backpropagation is only guaranteed to converge to a local minimum
  - May not find the absolute best set of weights
- Low initial weights can help with this
  - Makes the network act more linearly - fewer minima
- Can also use random restart - train multiple times with different initial weights.
A common extension to backpropagation is the addition of a momentum term. 
- Carries the algorithm through minima and plateaus.

Idea: remember the “direction” you were going in, and by default keep going that way.

Mathematically, this means using the second derivative.
Implementing momentum typically means remembering what update was done in the previous iteration.

Our update rule becomes:

\[ \Delta w_{ji}(n) = \alpha \Delta_j x_{ji} + \alpha \Delta w_{ji}(n - 1) \]

To consider the effect, imagine that our new delta is zero (we haven’t made any improvement)

Momentum will keep the weights “moving” in the same direction.

Also gradually increases step size in areas where gradient is unchanging.

△ This speeds up convergence,
One problem with backpropagation is that every node contributes to the output of a solution.

This means that all weights must be tuned in order to minimize global error.

Noise in one portion of the data can have an impact on the entire output of the network.

Also, training times are long.

Radial Basis function nets provide a solution to this.
20-14: Radial Basis Function networks

6 Intuition: Each node in the network will represent a portion of the input space.

6 Responsible for classifying examples that fall “near” it.

6 Vanilla approach: For each training point $< x_i, f(x_i) >$, create a node whose “center” is $x_i$.

6 The output of this node for a new input $x$ will be $W \ast \phi(|x - x_i|)$.

6 Where $W$ is the weight, and $\phi = \exp(-\frac{x^2}{2\sigma^2})$.

6 $\phi$ is a basis function.
20-15: Radial Basis Function networks

6 Each node has a “zone of influence” where it can classify nearby examples.

6 Training due to misclassification will only affect nodes that are near the misclassified example.

6 Also, network is single-layer.

6 Weights can be trained by writing a matrix equation:

- $\Phi W = t$
- $W = \Phi^{-1}t$

6 Inverting a matrix is a much faster operation than training with backpropagation.
So far, we’ve talked only about feedforward networks.

- Signals propagate in one direction
- Output is immediately available
- Well-understood training algorithms

There has also been a great deal of work done on recurrent neural networks.

- At least some of the outputs are connected back to the inputs.
This is a single-layer recurrent neural network.

Notice that it looks a bit like an S-R latch.
A Hopfield network has no special input or output nodes.

Every node receives an input and produces an output.

Every node connected to every other node.

Typically, threshold functions are used.

Network does not immediately produce an output.
  - Instead, it oscillates.

Under some easy-to-achieve conditions, the network will eventually stabilize.

Weights are found using simulated annealing.
20-19: Hopfield networks

- Hopfield networks can be used to build an **associative memory**
- A portion of a pattern is presented to the network, and the net “recalls” the entire pattern.
- Useful for letter recognition
- Also for optimization problems
- Often used to model brain activity
20-20: Neural nets - summary

- Key idea: simple computational units are connected together using weights.
- Globally complex behavior emerges from their interaction.
- No direct symbol manipulation
- Straightforward training methods
- Useful when a machine that approximates a function is needed
  - No need to understand the learned hypothesis