

Artificial Intelligence Programming

More Neural Networks

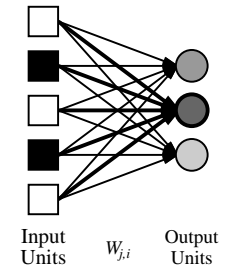
Chris Brooks

Department of Computer Science
University of San Francisco

24-2: Neural networks

- 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.

24-3: Perceptrons



- Single-layer networks (perceptrons) are the simplest form of NN.
- Easy to understand, but computationally limited.
- Each input unit is directly connected to one or more output units.

24-4: Delta rule

- The appeal of perceptrons is the ability to automatically learn their weights in a supervised fashion.
- The weight updating rule is known as the *delta rule*.
- $\Delta w_i = \alpha \sum_{d \in D} (t_d - o_d) x_{id}$
- Where D is the training set, t_d is expected output and o_d is actual output.

24-5: Example

- Let's suppose we want to learn the majority function with three inputs
- Firing fn: $O(i) = 1$ if $\sum w_j B_j > 0, 0$ otherwise
- bias: -0.1
- $\alpha = 0.1$
- initially, all weights 0

inputs			output
1	0	0	0
0	1	1	1
1	1	0	1
1	1	1	1
0	0	1	0
1	0	1	1
0	0	0	0
0	1	0	0

24-6: Example

inputs	expected output	w1	w2	w3	actual output	new weights
1 0 0	0	0	0	0	0	0 0
0 1 1	1					
1 1 0	1					
1 1 1	1					
0 0 1	0					
1 0 1	1					
0 0 0	0					
0 1 0	0					

24-7: Example

inputs	expected output	w1	w2	w3	actual output	new	weights
1 0 0	0	0	0	0	0	0	0 0 0
0 1 1	1	0	0	0	0	0	0.1 0.1
1 1 0	1						
1 1 1	1						
0 0 1	0						
1 0 1	1						
0 0 0	0						
0 1 0	0						

24-8: Example

inputs	expected output	w1	w2	w3	actual output	new	weights
1 0 0	0	0	0	0	0	0	0 0 0
0 1 1	1	0	0	0	0	0	0.1 0.1
1 1 0	1	0	0.1	0.1	0	0.1	0.2 0.1
1 1 1	1						
0 0 1	0						
1 0 1	1						
0 0 0	0						
0 1 0	0						

24-9: Example

inputs	expected output	w1	w2	w3	actual output	new	weights
1 0 0	0	0	0	0	0	0	0 0 0
0 1 1	1	0	0	0	0	0	0.1 0.1
1 1 0	1	0	0.1	0.1	0	0.1	0.2 0.1
1 1 1	1	0.1	0.2	0.1	1	0.1	0.2 0.1
0 0 1	0						
1 0 1	1						
0 0 0	0						
0 1 0	0						

24-10: Example

inputs	expected output	w1	w2	w3	actual output	new	weights
1 0 0	0	0	0	0	0	0	0 0 0
0 1 1	1	0	0	0	0	0	0.1 0.1
1 1 0	1	0	0.1	0.1	0	0.1	0.2 0.1
1 1 1	1	0.1	0.2	0.1	1	0.1	0.2 0.1
0 0 1	0	0.1	0.2	0.1	0	0.1	0.2 0.1
1 0 1	1						
0 0 0	0						
0 1 0	0						

24-11: Example

inputs	expected output	w1	w2	w3	actual output	new	weights
1 0 0	0	0	0	0	0	0	0 0 0
0 1 1	1	0	0	0	0	0	0.1 0.1
1 1 0	1	0	0.1	0.1	0	0.1	0.2 0.1
1 1 1	1	0.1	0.2	0.1	1	0.1	0.2 0.1
0 0 1	0	0.1	0.2	0.1	0	0.1	0.2 0.1
1 0 1	1	0.1	0.2	0.1	1	0.1	0.2 0.1
0 0 0	0						
0 1 0	0						

24-12: Example

inputs	expected output	w1	w2	w3	actual output	new	weights
1 0 0	0	0	0	0	0	0	0 0 0
0 1 1	1	0	0	0	0	0	0.1 0.1
1 1 0	1	0	0.1	0.1	0	0.1	0.2 0.1
1 1 1	1	0.1	0.2	0.1	1	0.1	0.2 0.1
0 0 1	0	0.1	0.2	0.1	0	0.1	0.2 0.1
1 0 1	1	0.1	0.2	0.1	1	0.1	0.2 0.1
0 0 0	0						
0 1 0	0						

24-13: Example

inputs	expected output	w1	w2	w3	actual output	new weights
1 0 0	0	0	0	0	0	0 0 0
0 1 1	1	0	0	0	0	0 0.1 0.1
1 1 0	1	0	0.1	0.1	0	0.1 0.2 0.1
1 1 1	1	0.1	0.2	0.1	1	0.1 0.2 0.1
0 0 1	0	0.1	0.2	0.1	0	0.1 0.2 0.1
1 0 1	1	0.1	0.2	0.1	1	0.1 0.2 0.1
0 0 0	0	0.1	0.2	0.1	0	0.1 0.2 0.1
0 1 0	0	0.1	0.2	0.1	1	0.1 0.1 0.1

- At this point, the network is trained properly.
 - In many cases, we would need more iterations to converge on a solution.

24-14: Gradient Descent and the Delta rule

- Recall that perceptrons are only able to perfectly learn *linearly separable* functions.
 - This is their representational bias.
- In other cases, we can still learn as much as is possible.
- We'll interpret this to mean minimizing the sum of squared error.
 - $E = 1/2 \sum (t_d - o_d)^2$ for d in training set.

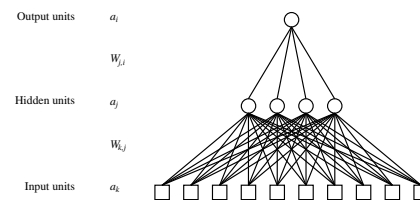
24-15: Gradient Descent and the Delta rule

- We can visualize this as a hillclimbing search through a space of weights
- Defining E in this way gives a parabolic space with a single global minimum.
- The delta rule adjusts the weights so that E is reduced at each step.
- By following the gradient in this space, we find the combination of weights that minimizes error.
- Use *unthresholded* output - what is the real number computed by the weighted sum of inputs?

24-16: Multilayer Networks

- 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?

24-17: Multilayer Network Example



24-18: Backpropagation

- Backpropagation is an extension of the perceptron learning algorithm to deal with multiple layers of nodes.
- Nodes use sigmoid activation function, rather than the step function
 - $g(input_i) = \frac{1}{1+e^{-input_i}}$.
 - $g'(input_i) = g(input_i)(1 - g(input_i))$ (good news here - to compute g' , we just need g)
- We will still "follow the gradient", where g' gives us the gradient.

24-19: Backpropagation

- Notation:
 - a_j - output of the j th hidden unit.
 - o_i - output of the i th output unit.
 - t_i - output for the i th training example
 - Output error for output node i : $(t_i - o_i)$
 - Delta rule for output node i :
$$\delta_i = (t_i - o_i) * g(input_i) * (1 - g(input_i))$$
 - Weight updating (hidden-output): $W_{j,i} = W_{j,i} + \alpha * a_j * \delta_i$
 - Same as the perceptron rule, except that we use g' rather than just the difference between expected and actual.

24-20: Backpropagation

- Updating input-hidden weights:
- Idea: each hidden node is responsible for a fraction of the error in δ_i .
- Divide δ_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:
- $W_{k,j} = W_{k,j} + \alpha * input_k * \delta_j$

24-21: Backpropagation Algorithm

- 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
 $\delta_i = output * (1 - output) * (t_{exp} - output)$
 for each hidden node
 $\delta_j = output * (1 - output) * \sum W_{k,i} \delta_i$
 Adjust each weight
 $W_{j,i} = W_{j,i} + \alpha * \delta_i * input_j$

24-22: Theory vs Practice

- In the definition of backpropagation, a single update for all weights is computed for all data points at once.
 - Find the update that minimizes total sum of squared error.
- Guaranteed to converge in this case.
- Problem: This is often computationally space-intensive.
 - Requires creating a matrix with one row for each data point and inverting it.
- In practice, updates are done incrementally instead.

24-23: Stopping conditions

- Unfortunately, incremental updating is not *guaranteed* to converge.
- Also, convergence can take a long time.
- When to stop training?
 - Fixed number of iterations
 - Total error below a set threshold
 - Convergence - no change in weights

24-24: 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.

24-25: Momentum

- Since backpropagation is a hillclimbing algorithm, it is susceptible to getting stuck in plateaus
 - Areas where local weight changes don't produce an improvement in the error function.
- 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.

24-26: Momentum

- Implementing momentum typically means remembering what update was done in the previous iteration.
- Our update rule becomes:
- $\Delta w_{ji}(n) = \alpha \Delta_j x_{ji} + \beta \Delta w_{ji}(n-1)$
- 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,

24-27: Design issues

- As with GAs, one difficulty with neural nets is determining how to *encode* your problem
 - Inputs must be 1 and 0, or else real-valued numbers.
 - Same for outputs
- Symbolic variables can be given binary encodings
- More complex concepts may require care to represent correctly

24-28: Design issues

- Like some of the other algorithms we've studied, neural nets have a number of parameters that must be tuned to get good performance.
 - Number of layers
 - Number of hidden units
 - Learning rate
 - Initial weights
 - Momentum term
 - Training regimen
- These may require trial and error to determine
- Alternatively, you could use a GA or simulated annealing to figure them out.

24-29: Radial Basis Function networks

- 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.

24-30: Radial Basis Function networks

- Intuition: Each node in the network will represent a portion of the input space.
- Responsible for classifying examples that fall "near" it.
- Vanilla approach: For each training point $\langle x_i, f(x_i) \rangle$, create a node whose "center" is x_i .
- The output of this node for a new input x will be $W * \phi(|x - x_i|)$
- Where W is the weight, and $\phi = \exp(-\frac{x^2}{2\sigma^2})$
- ϕ is a *basis function*.

24-31: Radial Basis Function networks

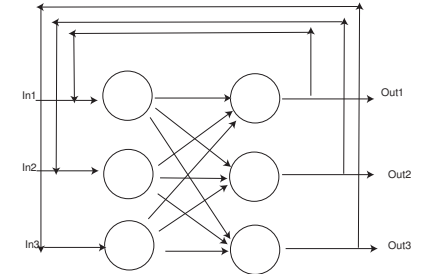
- Each node has a "zone of influence" where it can classify nearby examples.
- Training due to misclassification will only affect nodes that are near the misclassified example.
- Also, network is single-layer.
- Weights can be trained by writing a matrix equation:
 - $\Phi \mathbf{W} = \mathbf{t}$
 - $\mathbf{W} = \Phi^{-1} \mathbf{t}$
- Inverting a matrix is a much faster operation than training with backpropagation.

24-32: Recurrent NNs

- 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.

24-33: Recurrent NNs

- This is a single-layer recurrent neural network



- Notice that it looks a bit like an S-R latch.

24-34: Hopfield networks

- 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.

24-35: 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

24-36: 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