

Artificial Intelligence Programming

Neural Networks

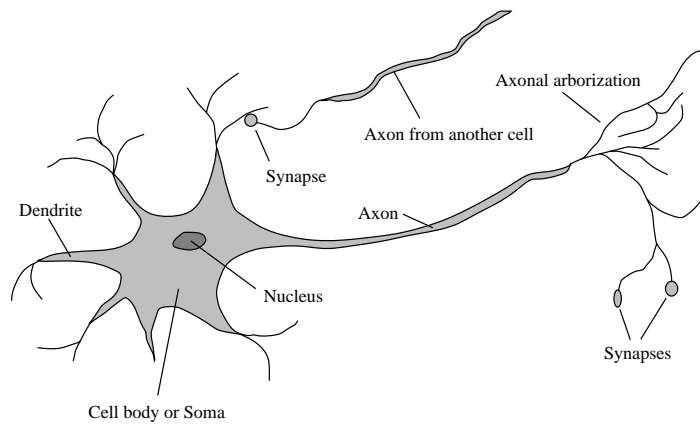
Chris Brooks

Department of Computer Science
University of San Francisco

23-2: Neural networks

- Much of what we've studied so far can be classified as *symbolic* AI.
- Focus on symbols and relations between them.
 - Search, logic, decision trees,
- The underlying assumption is that manipulation of symbols is the key requirement for intelligent behavior.
- Neural networks focus on *subsymbolic* behavior.
- Intelligent behavior emerges from the interaction of simple components.

23-3: Biology vs Computer Science



- In biological neurons, signals are received by dendrites and propagated to other neurons via the axon.
- Signaling and firing is very complex
- Thought and behavior are produced through the interaction of thousands of neurons.

23-4: Biology vs Computer Science

- Computational neural networks are related to biological neural networks primarily by analogy
 - Computational neuroscience studies the modeling of biologically plausible neurons.
- AI researchers are often more interested in developing effective algorithms.
 - As with GAs, we draw upon ideas that are successful in nature and take the parts that are useful.

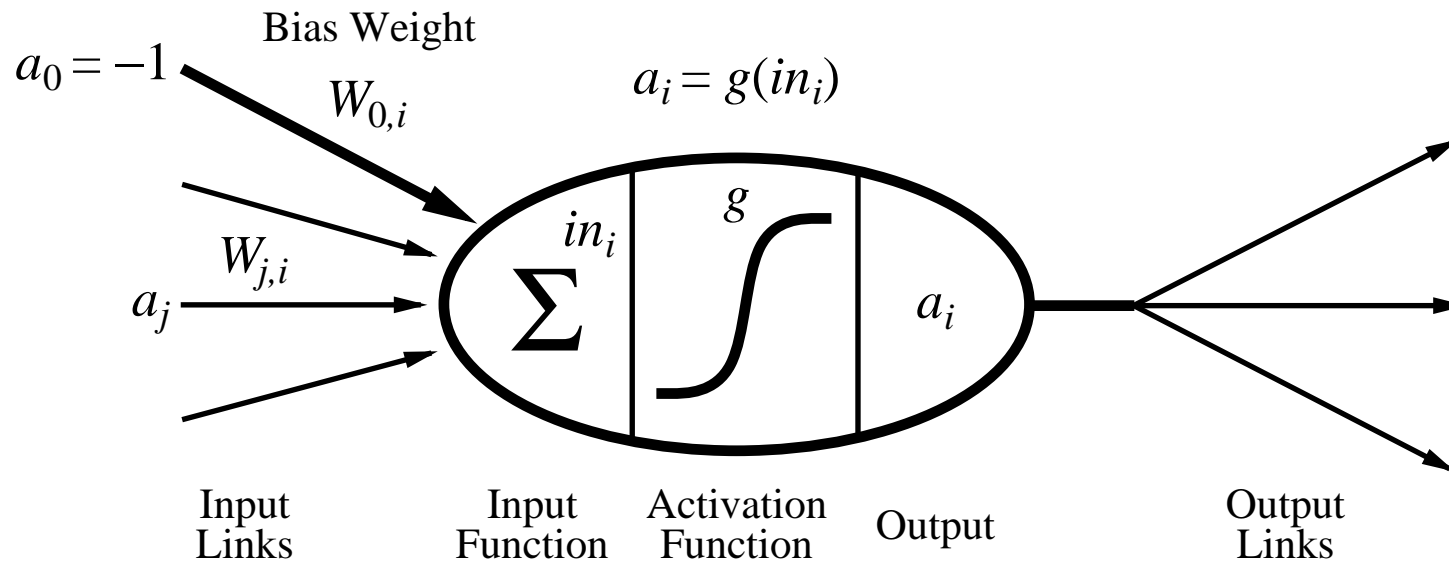
23-5: Computational Neural Networks

- Neural networks are composed of nodes.
- These nodes are connected by links
 - Abstraction of axons
- Each link has an associated weight that indicates the strength of the signal.
- Each node has a nonlinear activation function
 - Governs node's output as a function of the weighted sum of its inputs.

23-6: Appropriate tasks for neural learning

- Many attribute-value pairs.
- Real-valued inputs
- Real or discrete target value
- Noisy or error-containing data
- Long training time OK
- Fast evaluation of test cases needed
- Ability of humans to understand the learned hypothesis is not important.

23-7: Computational Neural Networks

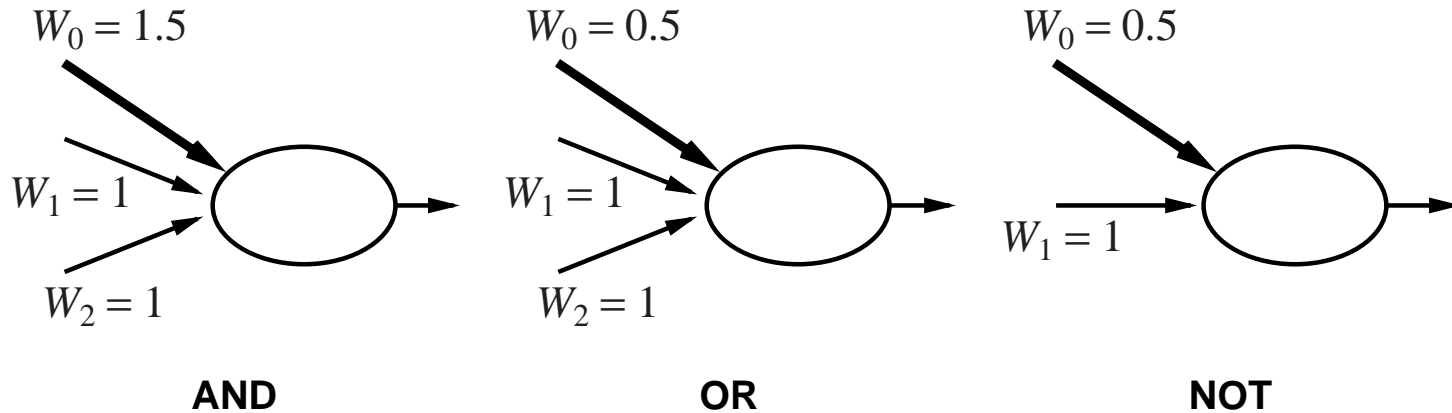


- Bias unit is used to control the *threshold value*
 - How strong the weighted input signal must be for the node to fire.

23-8: Activation functions

- Any nonlinear function can be used in principle.
- Two most common functions are:
 - Step function (threshold function) - Outputs 1 if input positive, zero otherwise.
 - Sigmoid/logistic function: $\frac{1}{1+e^{-x}}$.
 - Continuously differentiable
 - Rapid change near threshold, gradual at extremes.

23-9: Examples



- Neural nets can easily be built to perform some standard logical operations using the threshold activation function.
- Change the threshold depending on the function needed

23-10: Types of nodes

- We can distinguish between three types of nodes:
 - Input nodes
 - Output nodes
 - Hidden nodes
- We can also distinguish between types of networks
 - Feed-forward networks: signals flow in one direction, no cycles.
 - Recurrent networks: Cycles in signal propagation
- We'll focus primarily on feedforward networks

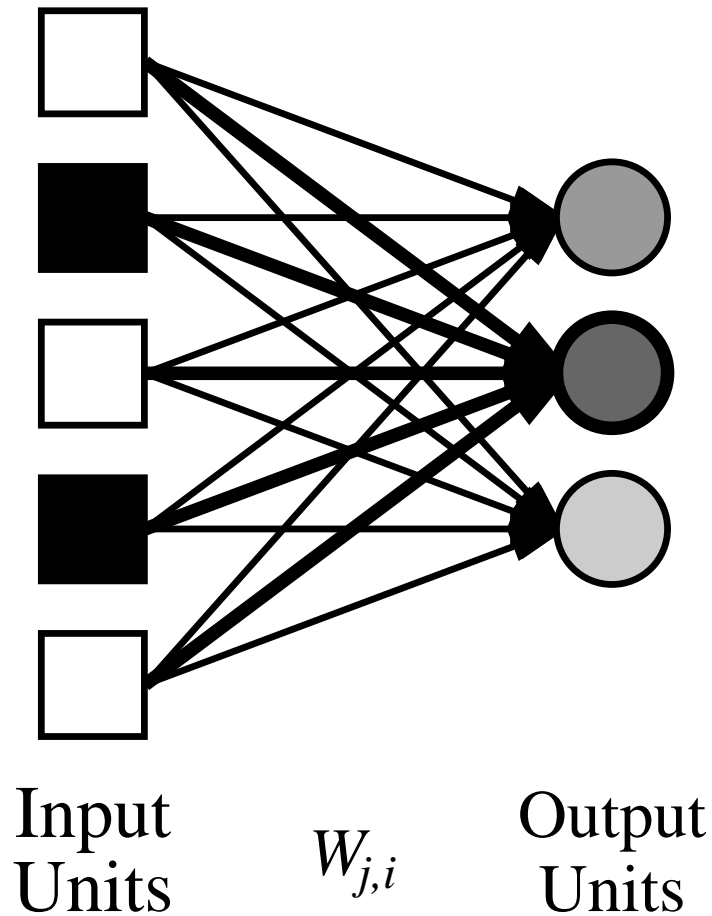
23-11: NNs as Function Approximators

- Feedforward NNs fall into a family of algorithms called *nonlinear function approximators*
- The output of a NN is a function of its inputs
- Nonlinear activation function allows the representation of complex functions.
- By adjusting weights, we change the function being represented
- NNs are often used to efficiently approximate complex functions from data.

23-12: Classification with Neural Networks

- NNs also perform classification very well.
- Map inputs into one or more outputs.
- Output range is split into discrete “classes”
- Very useful for learning tasks where “what to look for” is not known
 - Face recognition, handwriting recognition, driving a car

23-13: 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.

23-14: Perceptrons

- Output is thresholded weighted sum of the inputs.
- Threshold firing function used here.
- $o(x_1, \dots, x_n) = 1$ if $w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0$
- - 1 otherwise

23-15: Representational power of perceptrons

- Output of net: $\sum_{j=0}^n W_j i_j > 0$
- Perceptrons are capable of representing any linearly separable function.
- Unfortunately, many common functions (XOR, parity) are not linearly separable.
- In the early days of AI, perceptrons were popular, due to the fact that their weights could be easily learned.

23-16: Perceptron Training Algorithm

```
inputs = in1, in2, ..., inj
weights = w1, w2, ..., wn - initialize each to rand(-0.5, 0.5)
output = o
training examples: t1 = (tin1, to1), t2 = (tin2, to2) , ...

do
  for t in training examples
    inputs = tin
    o = compute net's output with current weights
    E = to - o
    for w in weight
      w = w + alpha * tin[i] * E
while notConverged
```

23-17: Perceptron Learning

- Intuition: If the output signal is too high, weights need to be reduced.
 - “Turn down” weights that contributed to output.
 - Weights with zero input are not affected.
- If output is too low, weights need to be increased.
 - “Turn up” weights that contribute to output.
 - Again, zero-input weights are not affected.
- Learning is really an optimization of weights.
- Alternatively, we are doing a hill-climbing search through weight space.

23-18: Example of learning AND

Step function - step at 0

Two inputs, plus a bias input set at -0.2

Learning rate = 0.1

Initial weights: 0.3, -0.1

23-19: Gradient Descent and the Delta rule

- What about cases where we can't learn the function exactly?
 - Function is not linearly separable
- In this case, we want to perform as well as 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.

23-20: Gradient Descent and the Delta rule

- We can visualize this as a search through a space of weights
- Defining E in this way gives a parabolic space with a single global minimum (for linear units).
- 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?

23-21: Gradient Descent and the Delta rule

- Gradient descent requires us to follow the steepest slope down the error surface.
- We consider the derivative of E with respect to each weight
- After derivation, we find that the updating rule (called the Delta rule) is:
- $$\Delta w_i = \alpha \sum_{d \in D} (t_d - o_d) x_{id}$$
- Where D is the training set, t_d is expected output and o_d is actual output.

23-22: Incremental learning

- Often it is not practical to compute a global weight change for the entire training set.
- Instead, we want to update weights incrementally.
 - Observe one piece of data, then update
- Our update rule is then: $w_i = \alpha(t - o)x_i$
 - Like the perceptron learning rule, except that unthresholded output is used.
- Smaller step size (α) typically used
- No theoretical guarantees of convergence

23-23: 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?

23-24: Multilayer Network Example

Output units

a_i

$W_{j,i}$

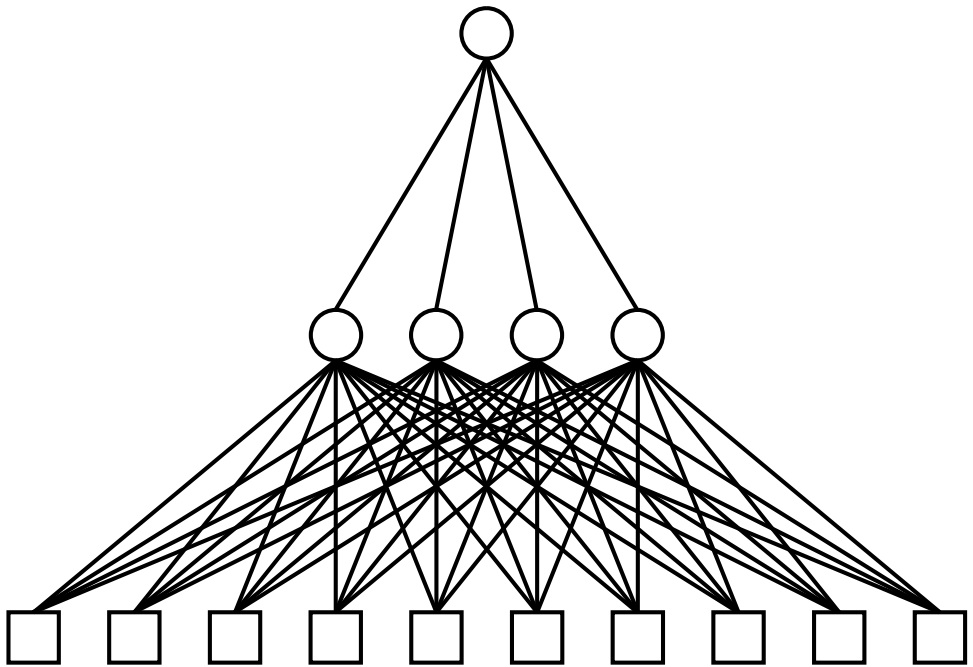
Hidden units

a_j

$W_{k,j}$

Input units

a_k



23-25: Backpropagation

- Backpropagation is an extension of the perceptron learning algorithm to deal with multiple layers of nodes.
- Nodes use sigmoid activation function
 - $g(input_i) = \frac{1}{1+e^{-input_i}}$.
 - $g'(input_i) = g(input_i)(1 - g(input_i))$
- We will still “follow the gradient”, where g' gives us the gradient.

23-26: Backpropagation

- Notation:

- $h_w(x)$ - vector of network outputs
- y - desired output for a training example
- 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 :
$$\Delta_i = (t_i - o_i) * g(input_i) * (1 - g(input_i))$$
- Weight updating (hidden-output): $W_{j,i} = W_{j,i} + \alpha * a_j * \Delta_i$

23-27: 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.
- $Delta_j = g(input)(1 - g(input)) \sum_i W_{j,i} \Delta_i$
- Update rule for input-hidden weights:
- $W_{k,j} = W_{k,j} + \alpha * input_k * \Delta_j$

23-28: 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

$$W_{j,i} = W_{j,i} + \alpha * \Delta_i * input_j$$

23-29: Stopping conditions

- When to stop training?
 - Fixed number of iterations
 - Total error below a set threshold
 - Convergence - no change in weights

23-30: 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