Neural Networks

(Russell and Norvig Chapter 20.5)

Decision trees are used a little, neural Networks a used quite a bit

Brain works pretty well, we should be able to make an algorithm that emulates the brain that also works well!

The Learning Problem

Given a set of points and labels:

(3.9, 2380, Mt. Lebanon) → A

:

(3.2, 1930, Alderdice) → B

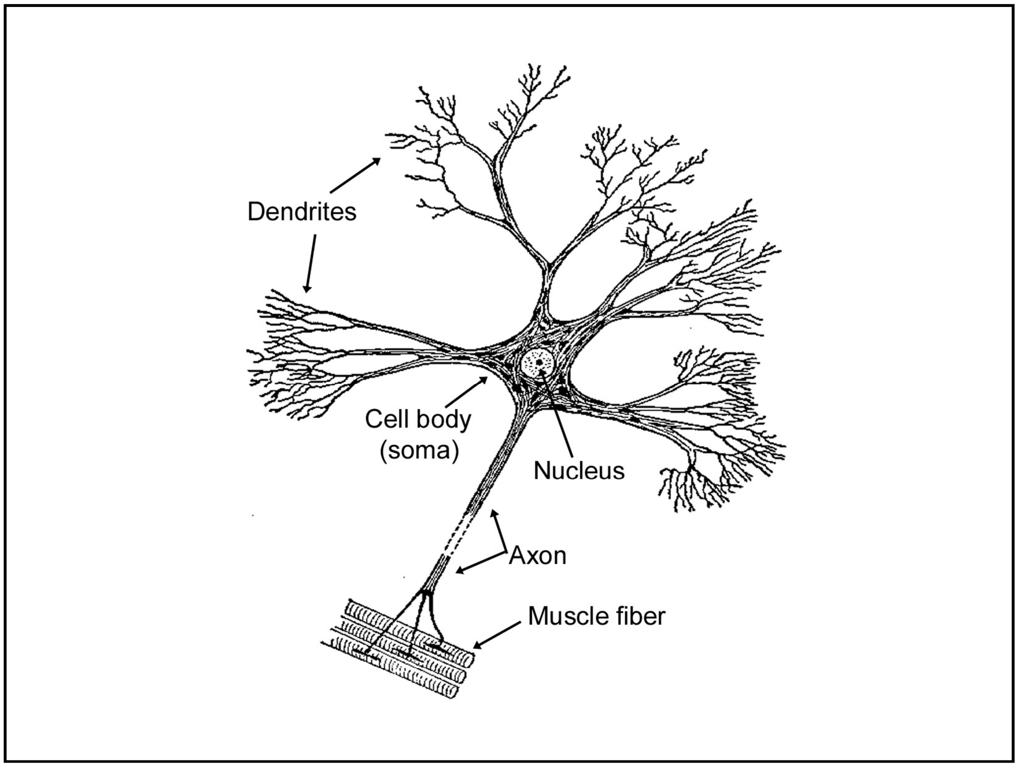
Build a classifier C that can accurately predict the label of new points:

C(3.5, 1200, Home School) = B

such that the classifier has "low error"

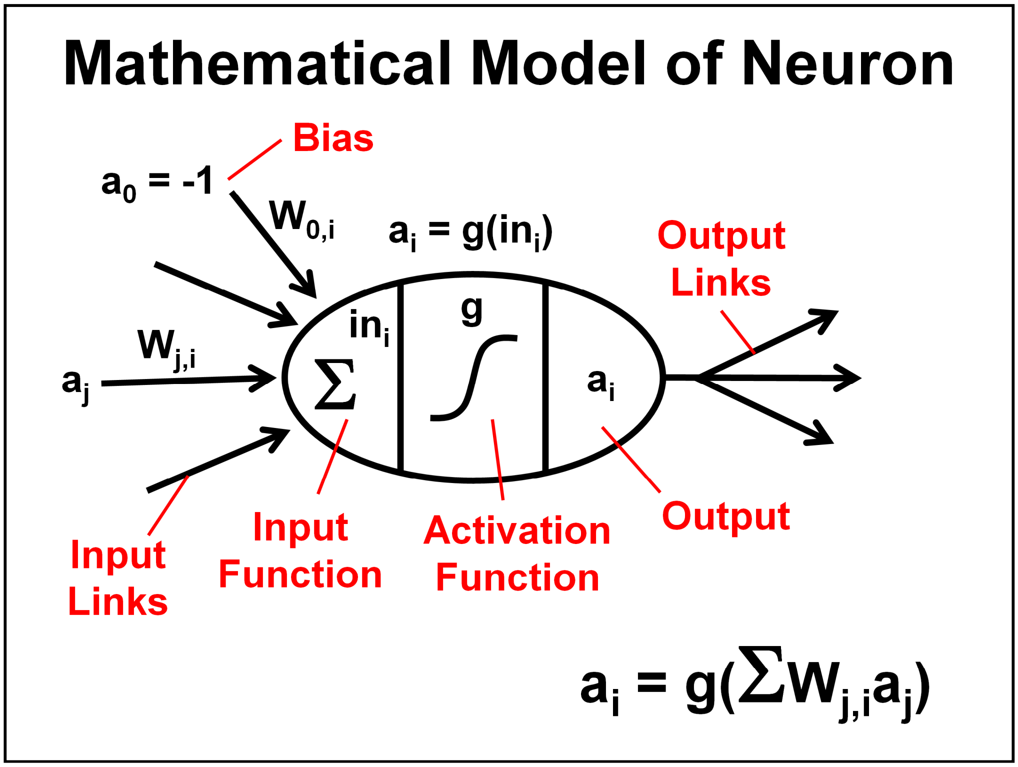
These points can be *very* high dimensional. For, example the points above (GPA, SAT, HS) could be (GPA, SAT, HS, PlaysTennis, PlaysPiano, PlaysFootball, PlaysClarinet, LikedZombieland…).

The admissions department can learn to predict your GPA! Each admitted student is a vector of attributes, and the output is their GPA when they graduate.



A neuron. Has inputs (dendrites) and outputs (axon). Some inputs are positive (make it more likely the neuron will fire), and some are negative (make it less likely the neuron will fire)

Lots of neurons in your brain.

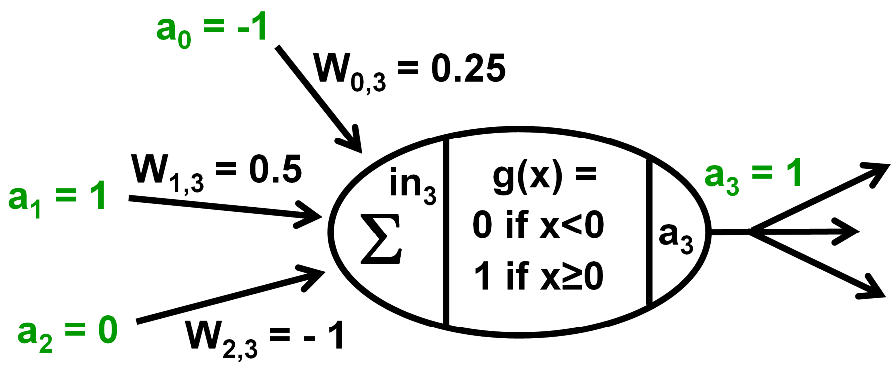


Wij is the weight from neuron i to neuron j

Activation function g is a function of the weighted sum of the input neurons

A0 doesn't come from another neuron, rather it is the bias. Bias is a constant factor and determines how much input it takes for the neuron to fire. “How hard is it to make the neruon fire”

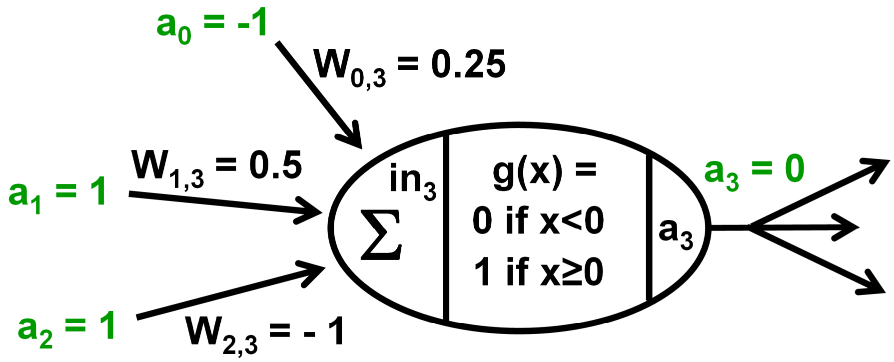
Example



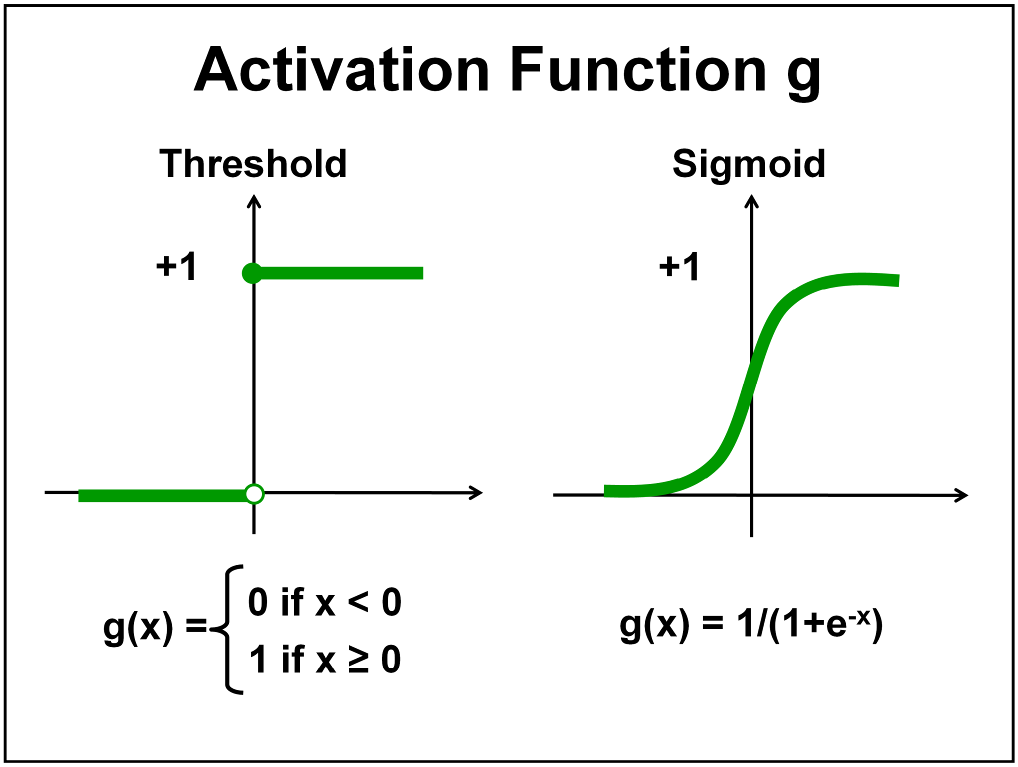
$$in_3 = (-1)(0.25) + (1)(0.5) + (0)(-1) = 0.25$$

The in3 is the weighted sum of all the inputs to the neuron

Example



$$in_3 = (-1)(0.25) + (1)(0.5) + (1)(-1) = -0.75$$



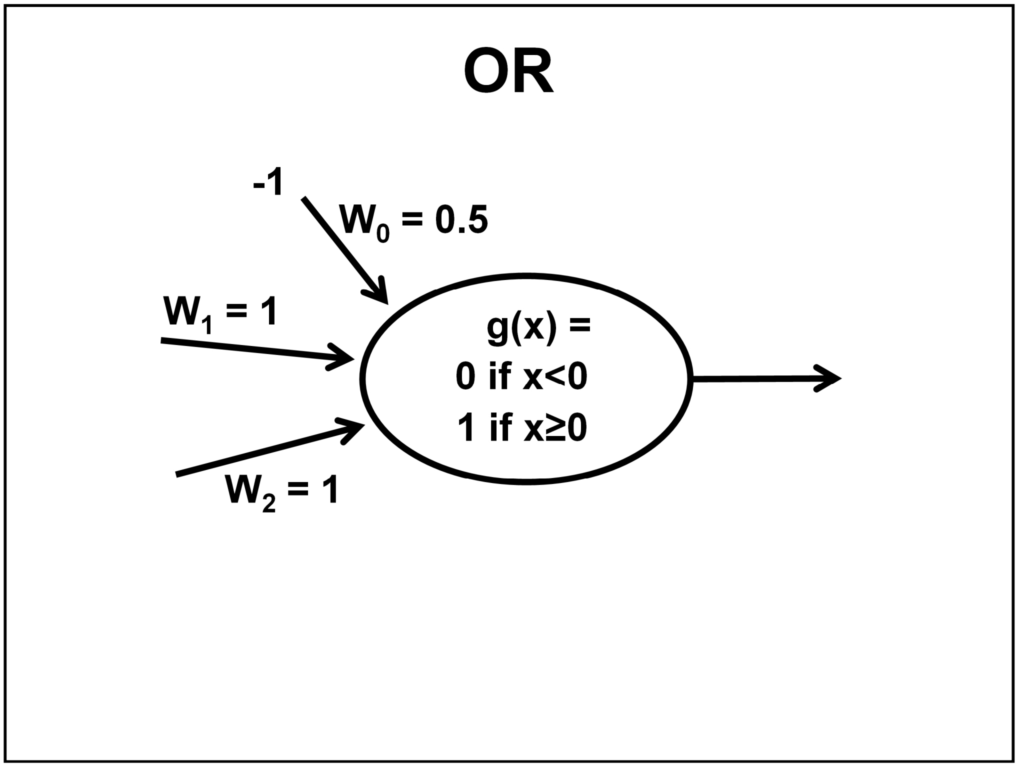
We can *always* center this at 0, because of the bias term. Note if we didn't have bias (b) we'd just have it centered at whatever the bias was.

Threshold activation functions are very common.

The sigmoid is continuous and differentiable. This is really nice, as we will see later.

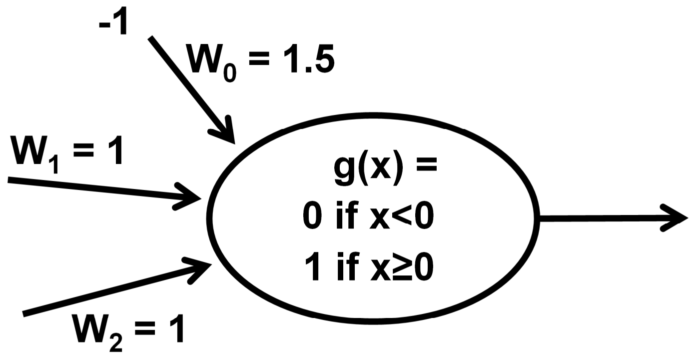
These two functions are very similar…almost always practically equal

We are assuming all neurons use the same activation function

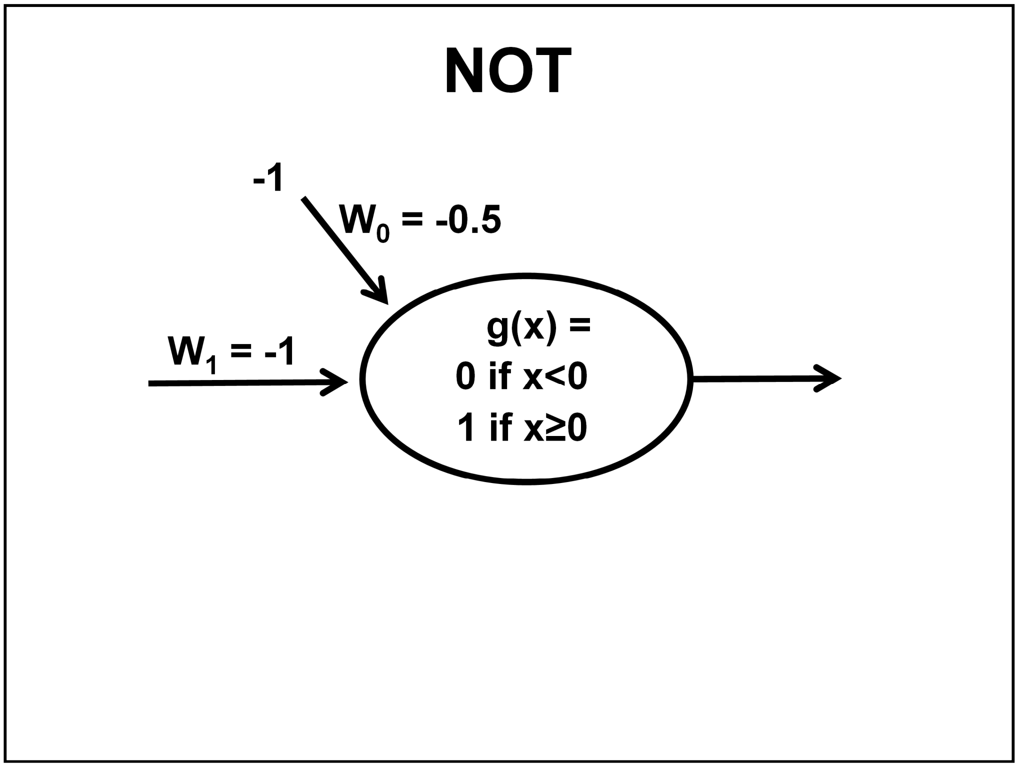


OR is like adding– you need at least one.

AND

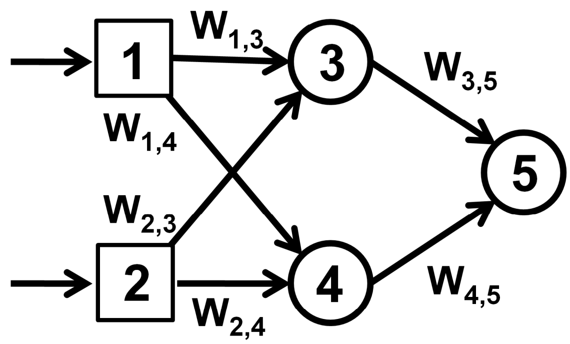


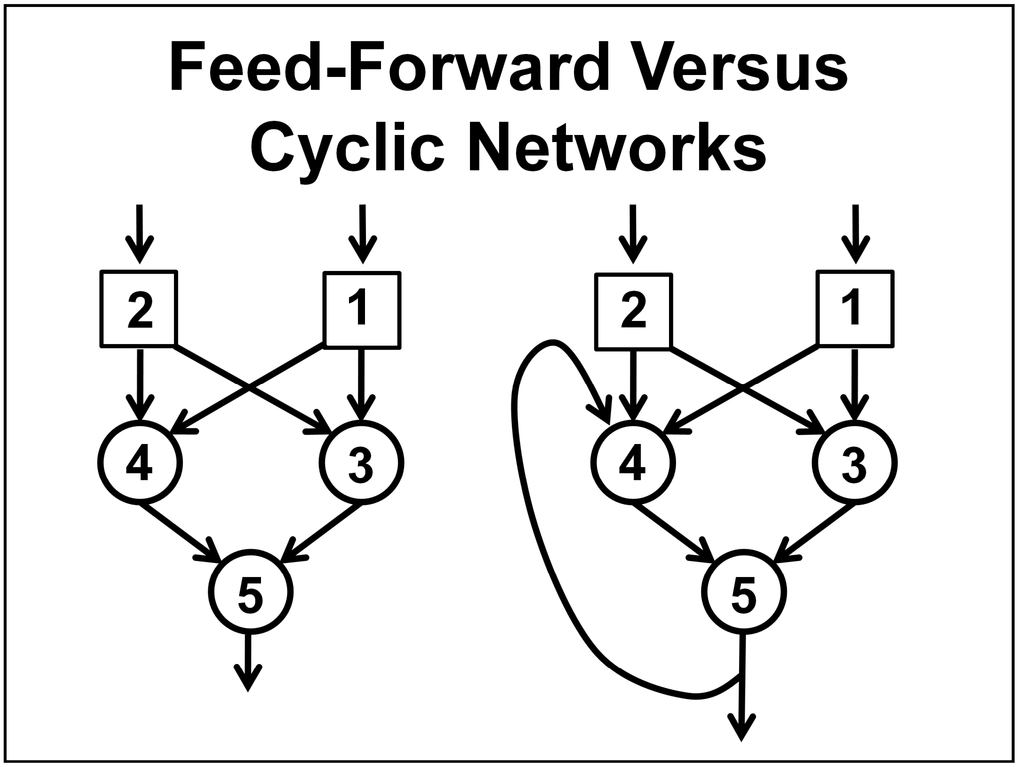
Here you need both.



Try to make an XOR. Actually, you can't, and we'll tell you why later.

Neural Networks

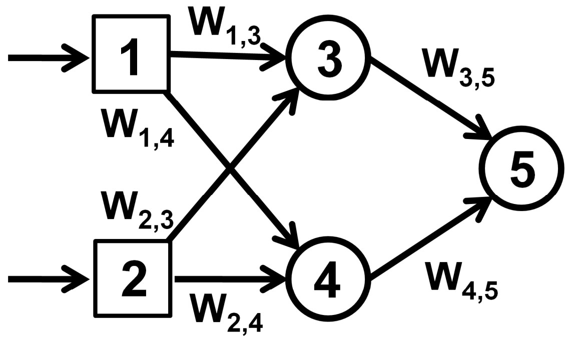




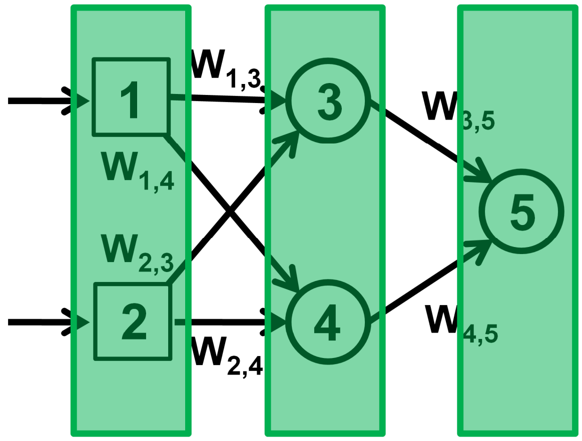
Feed forward neural networks do not have any cycles.

Cycles make things much more difficult. With cycles whether 4 fires or not would potentially not converge. Luckily we almost always do feed forward neural networks

Terminology

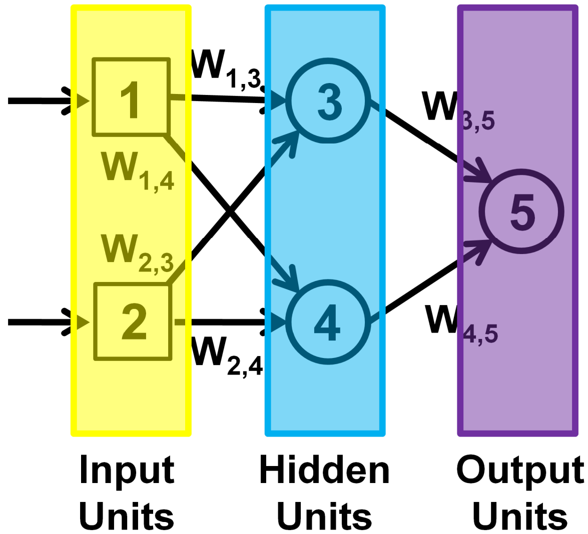


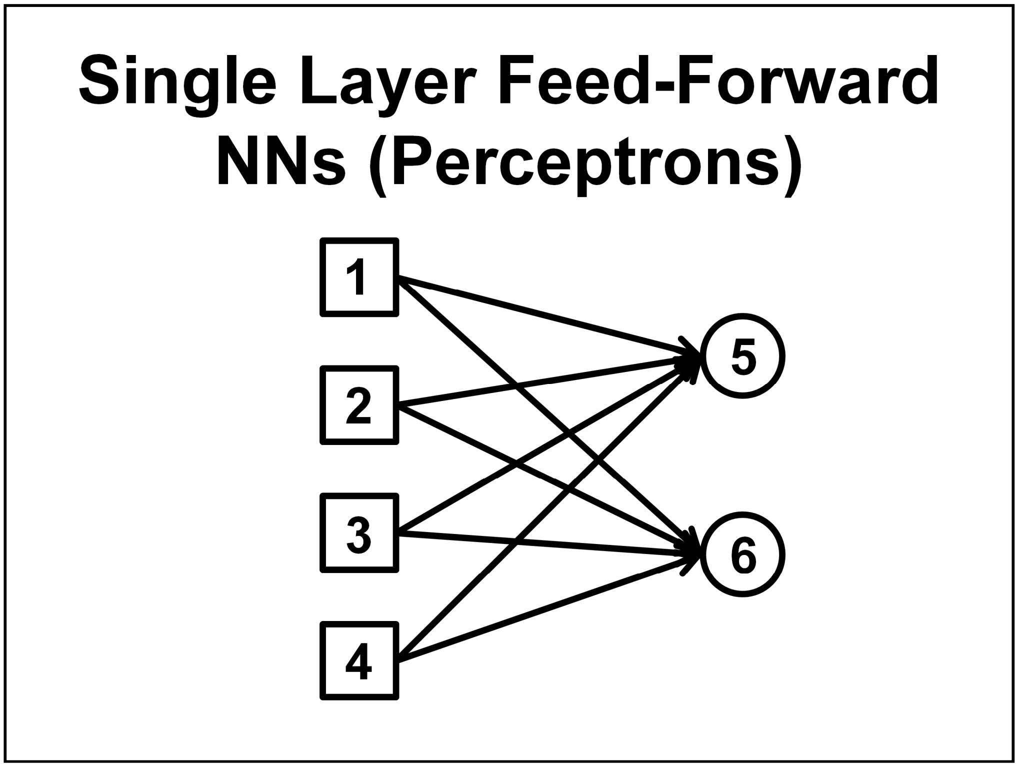
Terminology



Layers

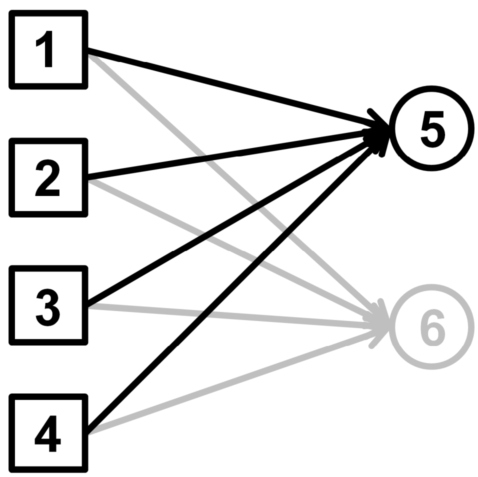
Terminology





Only one layer of weights. Slightly confusing name, as there are two layers of neurons (the input layer and the output layer.

Single Layer Feed-Forward NNs (Perceptrons)

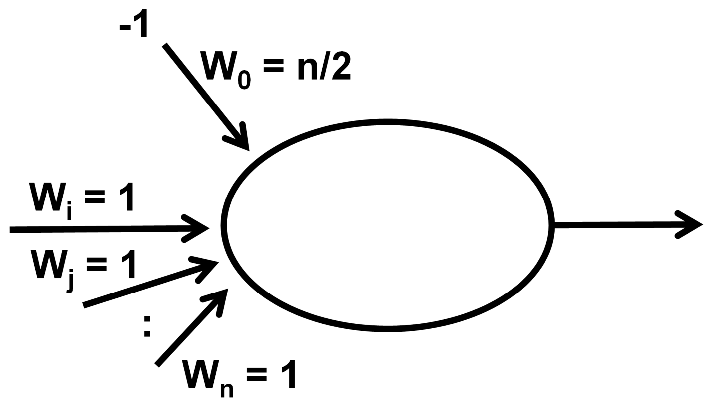


Only need to analyze with a single output unit.

What functions can be represented by a perceptron?

The Majority Function

Outputs 1 if and only if at least half of its n inputs are 1:



If this were a decision tree, it'd have to be really big. Isn't that nice?

A threshold preceptron returns 1 if and only if:

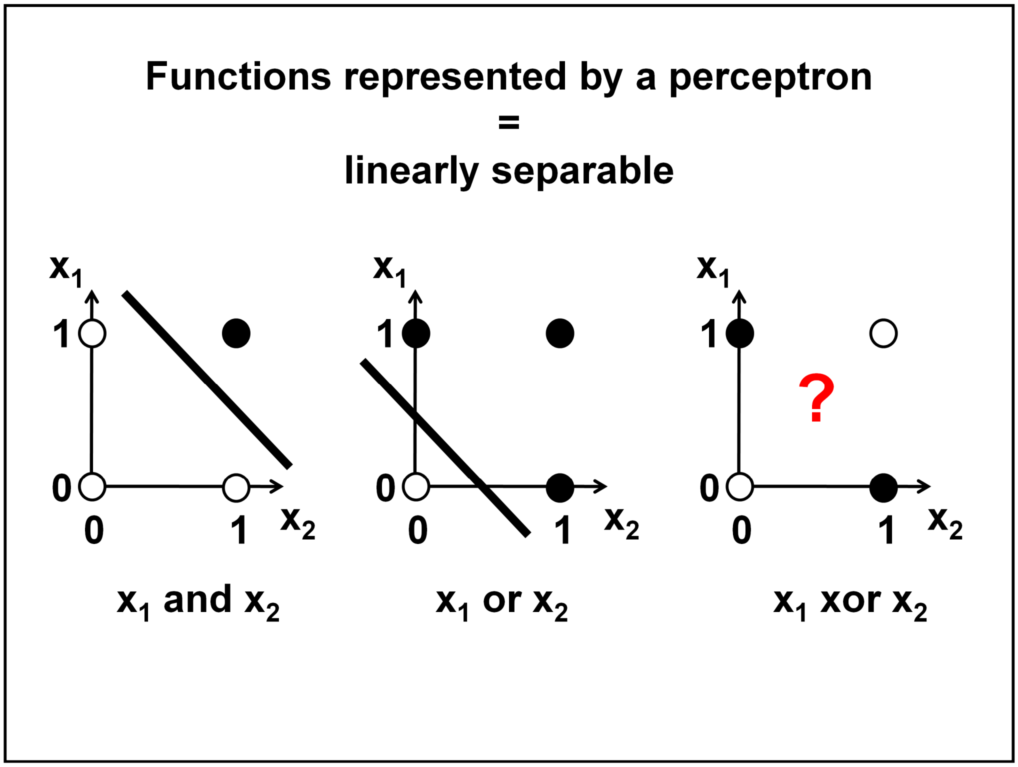
$$\sum_{j=0}^{n} W_j x_j > 0$$

A threshold preceptron returns 1 if and only if:

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 0$$

This equation defines a half space, so the perceptron returns 1 if and only if the input is on one side of a hyperplane

A vector of weights dot product with a vector of inputs.



Linearly separable means all negative and positive examples can be separated by a line (in higher dimensions, separable by a hyperplane)

And *this* is why you can’t do XOR, as it is not linearly separable. Therefore a single threshold perceptron cannot do XOR

However, if you have multiple perceptrons, you can do a lot more.

Learning in Perceptrons

Learning algorithm that will fit a threshold perceptron to any linearly separable function

Idea: adjust the weights to minimize some notion of error

We will use the notion of squared error

For a single example with input x and true output y:

$$E = (1/2)Err^2 = (1/2)(y - h_W(x))^2$$

Where $h_W(x)$ is the output of the perceptron

We're doing local search on the weights, trying to minimize the squared error.

Minimizing Squared Error

Recall that: $E = (1/2)Err^2 = (1/2)(y - h_W(x))^2$

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j}$$

$$= Err \times \frac{\partial}{\partial W_j}\left[y - g\left(\sum_{k=0}^{n} W_k x_k\right) \right]$$

$$= -Err \times g'(in) \times x_j$$

Perceptron learning rule:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

We can derive the equation to minimize the squared error. If the error (not squared error) is positive, we'll make the weight increase. If the error is negative (you're overshooting), you decrease the weights.

This is why using a g which has continuous partial derivatives in respect to the weights is useful.

The black box tells you how to update the weights when learning. g' is the derivative of g.

Multi-Layer Feed-Forward NNs

Can represent more complex functions

With a single sufficiently large hidden layer can represent any continuous function with arbitrary accuracy

BIG QUESTION: What structure to use for a given problem?

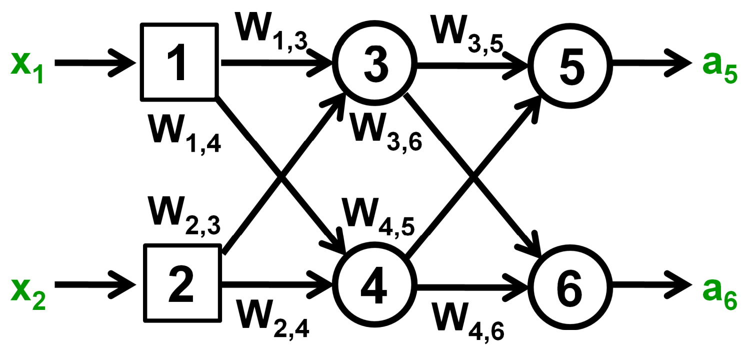
Hidden layer is the layer between the input layer and the output layer.

Just guess a structure.

Often people guess a structure, learn the weights, then go back and revise if the error is bad.

Structure learning is another interesting problem.

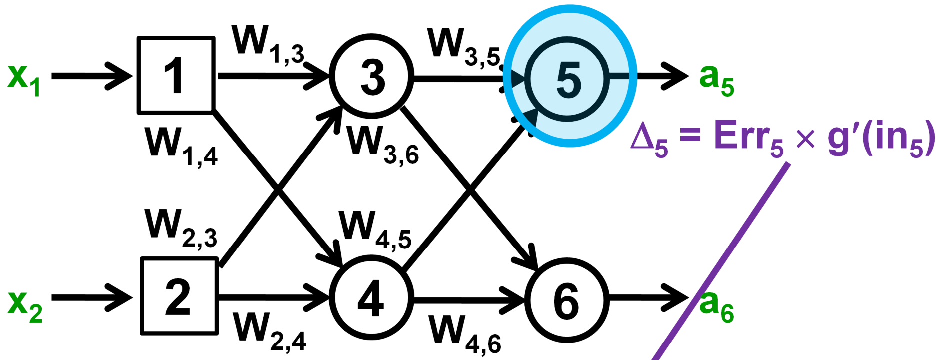
Learning Weights



The error will now be a vector: Err = (y_1-a_5, y_2-a_6)

We will treat each component separately, so let Err_i be the component associated with a_i

Learning Weights

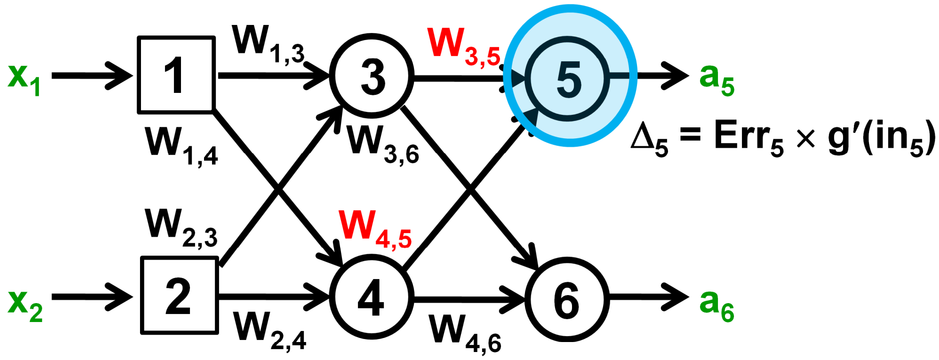


The output unit is like a perceptron, so we update its weight like we did for perceptrons:

$$W_{3,5} \leftarrow W_{3,5} + \alpha \times \boxed{Err_5 \times g'(in_5)} \times a_3$$

$$W_{4,5} \leftarrow W_{4,5} + \alpha \times Err_5 \times g'(in_5) \times a_4$$

Learning Weights

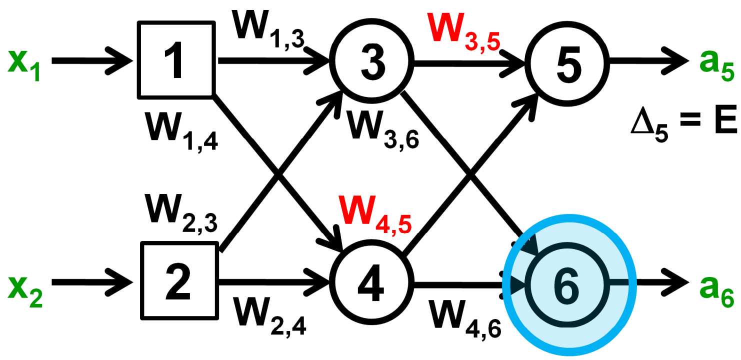


The output unit is like a perceptron, so we update its weight like we did for perceptrons:

$$W_{3,5} \leftarrow W_{3,5} + \alpha \times \Delta_5 \times a_3$$

$$W_{4,5} \leftarrow W_{4,5} + \alpha \times \Delta_5 \times a_4$$

Learning Weights



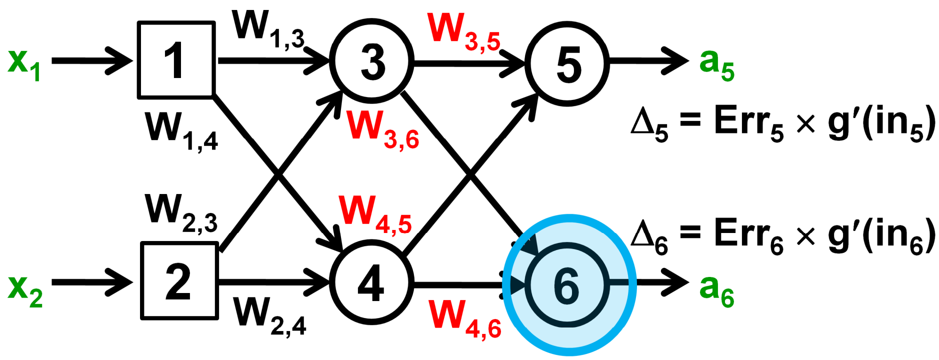
$\Delta_5 = Err_5 \times g'(in_5)$

The output unit is like a perceptron, so we update its weight like we did for perceptrons:

$$W_{3,5} \leftarrow W_{3,5} + \alpha \times \Delta_5 \times a_3$$

$$W_{4,5} \leftarrow W_{4,5} + \alpha \times \Delta_5 \times a_4$$

Learning Weights

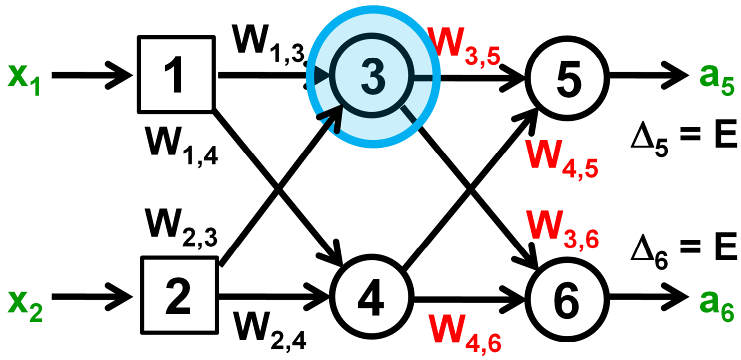


The output unit is like a perceptron, so we update its weight like we did for perceptrons:

$$W_{3,6} \leftarrow W_{3,6} + \alpha \times \Delta_6 \times a_3$$

$$W_{4,6} \leftarrow W_{4,6} + \alpha \times \Delta_6 \times a_4$$

Learning Weights



$\Delta_5 = Err_5 \times g'(in_5)$

$\Delta_6 = Err_6 \times g'(in_6)$

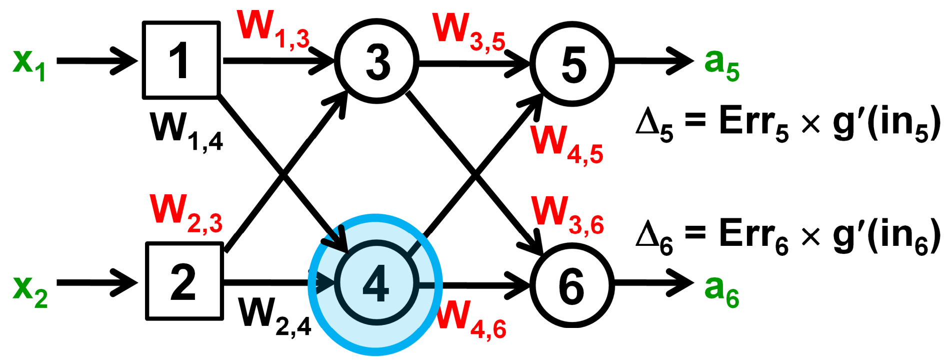
How do we define the error at unit 3?

$$\Delta_3 = g'(in_3)(W_{3,5}\Delta_5 + W_{3,6}\Delta_6)$$

$$W_{1,3} \leftarrow W_{1,3} + \alpha \times \Delta_3 \times a_1 \qquad W_{2,3} \leftarrow W_{2,3} + \alpha \times \Delta_3 \times a_2$$

Okay, so since we know what the output of 5 and 6 should have been we can calculate that easily. But what about unit 3?

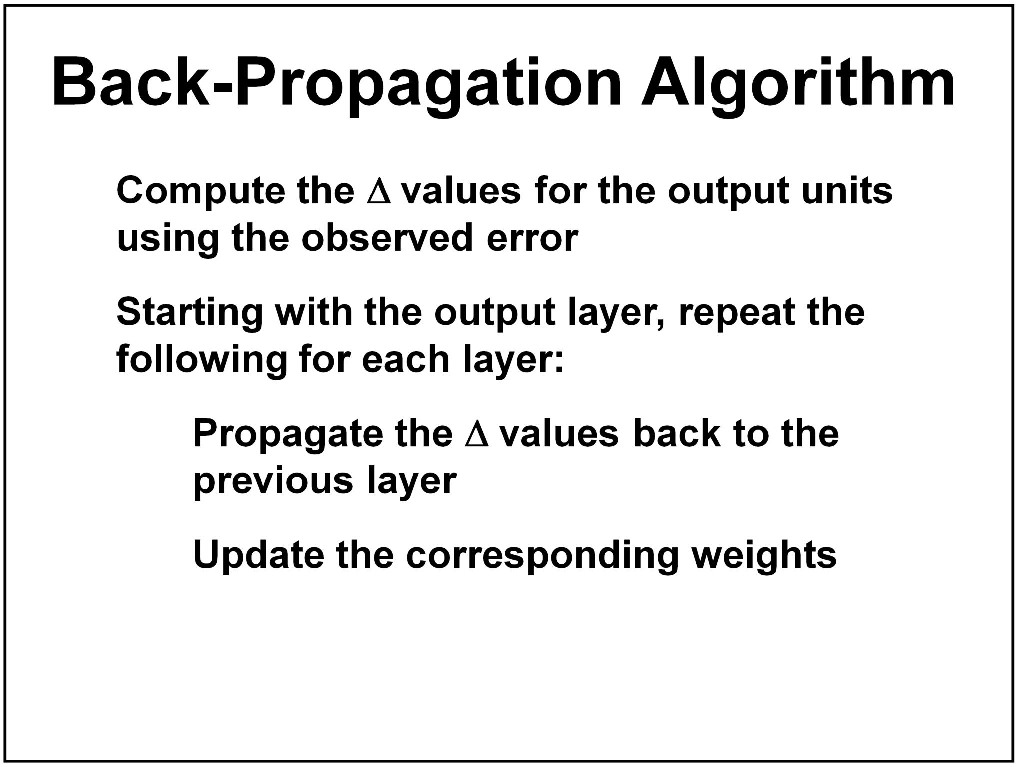
Learning Weights



How do we define the error at unit 4?

$$\Delta_4 = g'(in_4)(W_{4,5}\Delta_5 + W_{4,6}\Delta_6)$$

$W_{1,4} \leftarrow W_{1,4} + \alpha \times \Delta_4 \times a_1$ $\quad$ $W_{2,4} \leftarrow W_{2,4} + \alpha \times \Delta_3 \times a_4$



This is the back-propagation algorithm. The idea is that you are back-propagating the errors from the units in the end layers. Then once you can calculate the error, you can learn the weights.