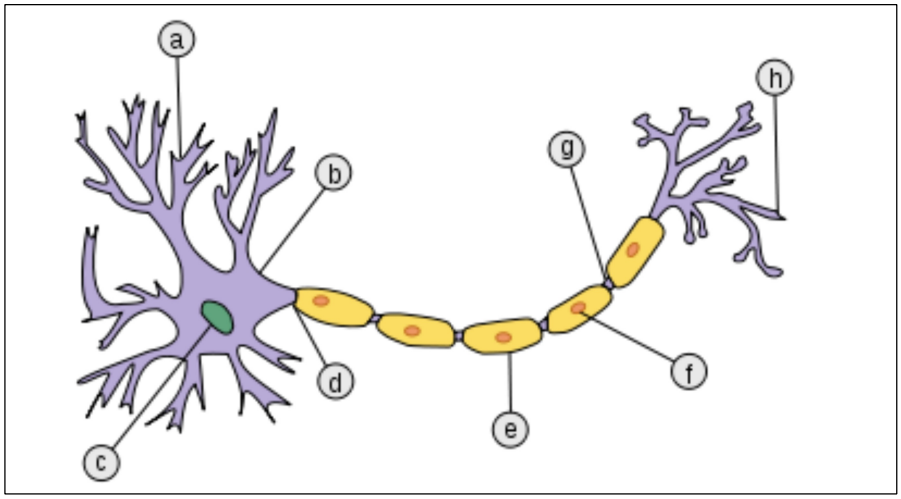
Three eras

The history of neural networks falls into three distinct eras:

1. The Perceptron era: 1958-1969
2. The Backpropagation era: 1986-1995
3. The Deep Learning era: 2011-present

These notes will cover the first two eras. The next set of notes will get us started on the third era.

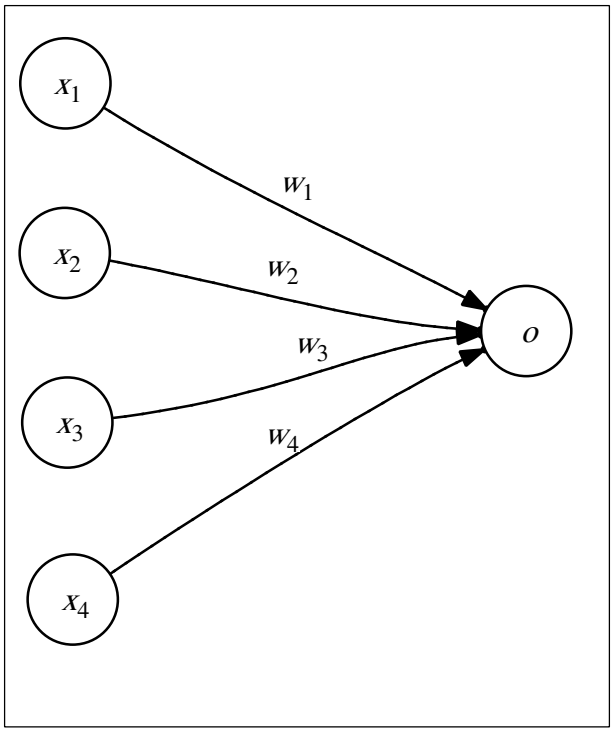
Biological Neural Networks



- Neurons receive inputs from other connected nerve cells via their dendrites (a).
- When the neuron receives a critical level of inputs it fires.
- When the neuron fires it sends an impulse down the axon (e) to its synapses (h).
- When the neuron fires it releases neurotransmitters from its synapses that travel across the synaptic gap to the dendrites of nearby neurons.
- Synaptic connections can be excitory or inhibitory, making neighbors either more or less likely to fire.

Perceptrons

The Perceptron was the first model of a neuron to find wide-spread application. The Perceptron is based on a simplified model of a neuron developed by McCulloch and Pitts in 1943. The first actual implementation of a Perceptron came in a 1958 machine developed by Frank Rosenblatt.

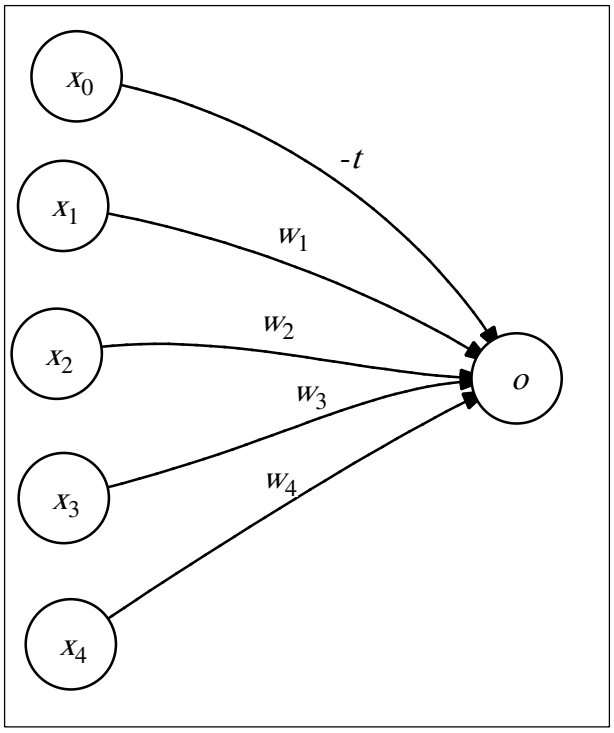


- The Perceptron consists of a set of input units x_1 through x_n connected to an output unit o.
- In the simplest version of the Perceptron each input unit is either on (1) or off (0).
- The output unit receives the weighted sum of the input from the input units:

$$\sum_{i=1}^{n} w_i x_i$$

- If the weighted sum of the inputs exceeds the output unit's *threshold value*, t, the output unit outputs a value of 1. Otherwise, it outputs a value of 0.

An alternative architecture you will also come across involves the use of a special *bias unit*. Instead of forcing the weighted sum of the inputs to the output unit to excede a threshold t we can instead introduce an extra input unit x_0 whose value is always set to 1 and connect that extra bias unit to the output with connection having a weight of $-t$. We then set the threshold on the output unit to 0. The bias unit forces the weighted sum of the inputs from the other units to have to excede t to get the output to turn on.



Supervised Learning

The Perceptron learns to potentially map a set of inputs to correct outputs by adjusting its weights with a *learning algorithm*.

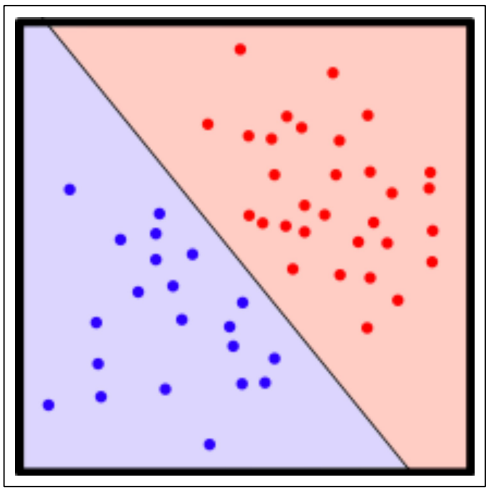
$$\Delta\ w_i = \eta\,(\,d - o\,)\,x_i$$

Here d is the desired output and η is a learning rate parameter.

The Perceptron learning algorithm works by repeatedly presenting inputs to the network and computing an output for each input. We then use the learning rule to adjust the weights after each input round.

Limitations

The simple single layer Perceptron can only learn mappings in which the inputs with a desired output of 0 can be *linearly separated* from the inputs with a desired output of 1.

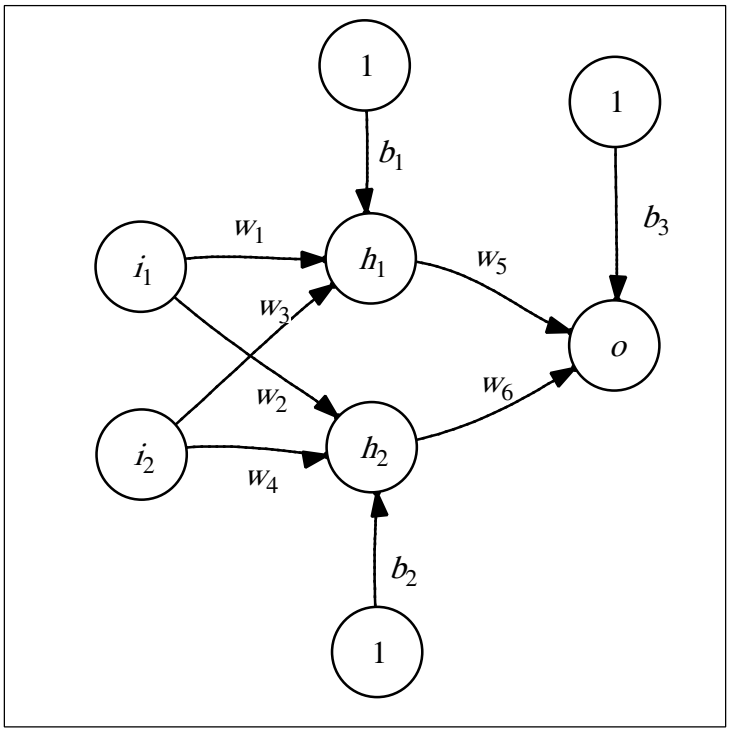


In their 1969 book *Perceptrons* Marvin Minsky and Seymour Papert pointed out this limitation of the Perceptron. The book caused interest in the Perceptron model to decline quickly. Interest in neural networks only picked back up again by the mid 1980s with the development of multilayered neural networks with nonlinear activation functions and the backpropagation learning algorithm.

Backpropagation

Interest in neural networks picked back up again in 1986 with the publication of a paper by Rumelhart, Hinton, and Williams. This paper introduced the backpropagation algorithm for training multi-layer neural networks.

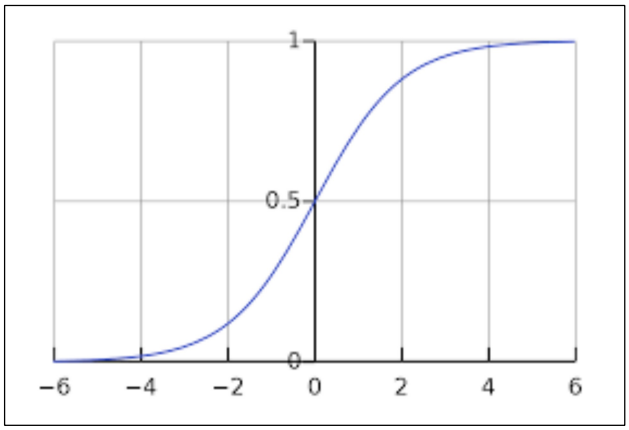
Here is an example of a simple two-layer network:



The three units labled with 1s are bias units. These units always output a value of 1. The combination of the bias unit and the weight of the connection connecting it to a unit in the network is to set the threshold for the unit.

Both the *hidden units* (h_1 and h_2) and the output unit o compute their values by taking the weighted sums of their inputs and then passing that weighted sum to an *activation function* to compute the unit's value. The most common activation function used in the backpropagation era was the *sigmoid activation function*:

$$a(x) = \frac{1}{1 + e^{-x}}$$



In the *backpropagation algorithm* we seek to find values for the weights in a network that minimize the

errors produced by the network. The first step in the algorithm is to compute the errors produced by the existing network:

1. Construct a set of input, target pairs. Each pair consists of an input vector containing values for the input units and a target value that we want the network to produce for that particular input.

2. For each input target pair ($\vec{i}_j$, d_j) we compute the square of the difference between the actual output o_j and the desired output d_j: $(o_j - d_j)^2$

3. The sum of all of these terms over all of the elements of the training set is the *loss function*:

$$L(\vec{w}) = \frac{1}{n}\sum_{j=1}^{n}(o_j - d_j)^2$$

The loss function is a function of the vector of weights in the network, since these weights determine the outputs given a fixed set of inputs to the network. Our goal is to find a value for the weight vector that minimizes the loss. Here the weight vector includes both the w weights and the bias b weights.

To minimize $L(\vec{w})$ we use the *gradient descent algorithm*. Here are the key features of this algorithm:

1. We start by computing the gradient of the loss function: $\nabla L(\vec{w}) = \left(\frac{\partial L(\vec{w})}{\partial w_1}, \frac{\partial L(\vec{w})}{\partial w_2}, \ldots, \frac{\partial L(\vec{w})}{\partial w_n}, \frac{\partial L(\vec{w})}{\partial b_1}, \ldots, \frac{\partial L(\vec{w})}{\partial b_3}\right)$. The gradient computes the direction of steepest increase of the loss function in weight space.

2. The negative of the gradient computes the direction of steepest decrease of the loss. Since our goal is to minimize the loss, we take a small step in that direction:

$$\vec{w}_{k+1} = \vec{w}_k - \eta \nabla L(\vec{w}_k)$$

3. This basic step repeats until we have reached an acceptably small value for the loss.

One technical issue we have to overcome is that although technically the loss function is a function of the weights, the dependence of the loss on the weights is rather complex. The loss function

$$L(\vec{w}) = \frac{1}{n}\sum_{j=1}^{n}(o_j - d_j)^2$$

depends first of all on the outputs o_j. Those outputs in turn depend in a complicated way on both the inputs and weights in the network. Here are some of the details:

$$o_j(\vec{w}) = a(w_5\, h_1 + w_6\, h_2 + b_3)$$

$$h_1 = a(w_1\, i_{1,j} + w_2\, i_{2,j} + b_1)$$

$$h_2 = a(w_3\, i_{1,j} + w_4\, i_{2,j} + b_2)$$

Because the weights are buried deep inside these expressions we have to use the chain rule for partial derivatives to compute the necessary derivatives. Here are some examples:

$$\frac{\partial L(\vec{w})}{\partial w_5} = \frac{1}{n}\sum_{j=1}^{n} \frac{\partial}{\partial w_5}(o_j - d_j)^2 = \frac{1}{n}\sum_{j=1}^{n} 2\,(o_j - d_j)\,\frac{\partial\, o_j}{\partial w_5}$$

$$\frac{\partial\, o_j}{\partial w_5} = a'(w_5\, h_1 + w_6\, h_2 + b_3)\, h_1$$

$$\frac{\partial L(\vec{w})}{\partial w_1} = \frac{1}{n}\sum_{j=1}^{n} \frac{\partial}{\partial w_1}(o_j - d_j)^2 = \frac{1}{n}\sum_{j=1}^{n} 2\,(o_j - d_j)\,\frac{\partial o_j}{\partial w_1}$$

$$\frac{\partial o_j}{\partial w_1} = a'(w_5\, h_1 + w_6\, h_2 + b_3)\,(w_5\,\frac{\partial h_1}{\partial w_1} + w_6\,\frac{\partial h_2}{\partial w_1})$$

$$\frac{\partial h_1}{\partial w_1} = a'(w_1\, i_{i,j} + w_2\, i_{2,j} + b_1)\, i_{1,j}$$

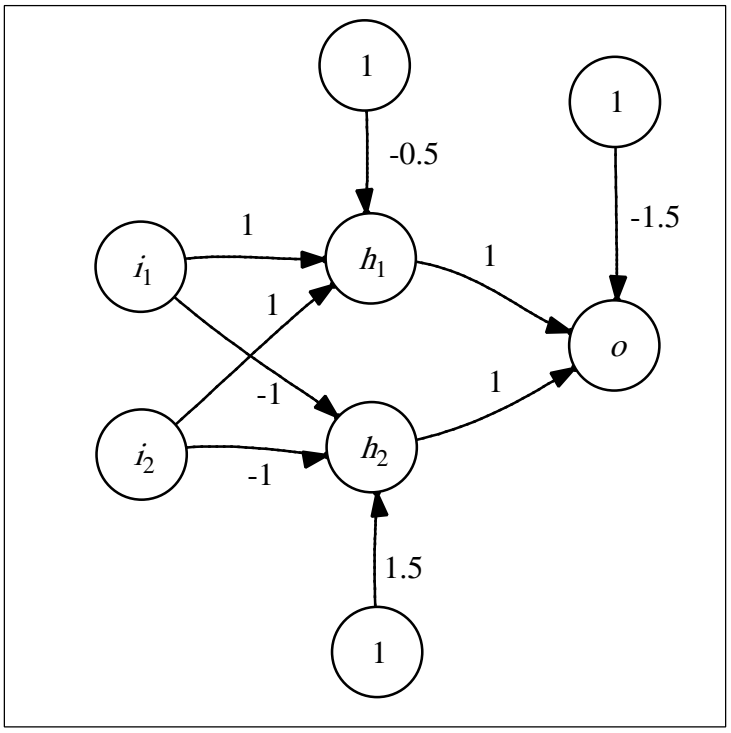
$$\frac{\partial h_2}{\partial w_1} = a'(w_3\, i_{1,j} + w_4\, i_{2,j} + b_2)\, 0 = 0$$

XOR example

Backpropagation was an important development because it made it easy to build multi-layer networks, which in turn made it possible for neural networks to solve problems that Perceptrons were unable to solve. An example of an early success was training a network to solve the XOR problem.

a	b	$a \wedge b$
1	1	0
1	0	1
0	1	1
0	0	0

Here is a set of weights for our simple two layer network that produces the correct results:



Another way to solve the XOR problem without the use of bias units is to use three hidden units instead of two: