Neural Network

Course Code: **CSC4226** Course Title: **Artificial Intelligence and Expert System**

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**Eleven (11)**	**Week No:**	**Twelve (12)**	**Semester:**	
Lecturer:	*Sajib Hasan*				*sajib.hasan@aiub.edu*

Lecture Outline

1. Perceptron : As Model of Decision-Making.
2. Perceptron for Classification
3. Limitations of Perceptron
4. Sigmoid Neuron
5. Simple Mathematical Model for a Neuron
6. Neural Network Structures
7. Backpropagation Algorithm
8. Calculating Total Error

A Simple Decision

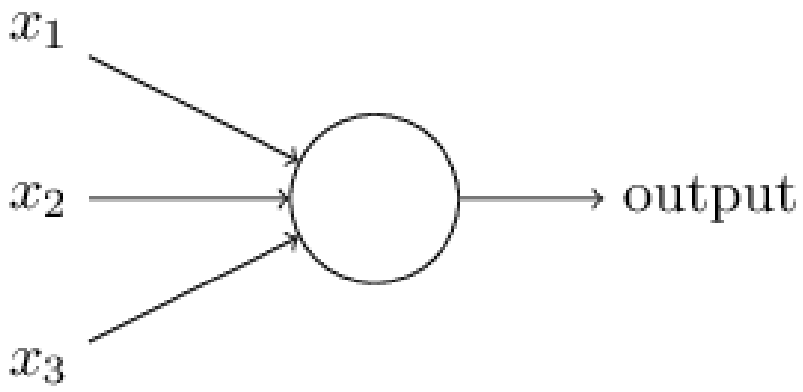
Say you want to decide whether you are going to attend a cheese festival this upcoming weekend. There are three variables that go into your decision:

1. Is the weather good?
2. Does your friend want to go with you?
3. Is it near public transportation?

We'll assume that answers to these questions are the only factors that go into your decision.

A perceptron

We can graphically represent this decision algorithm as an object that takes 3 binary inputs and produces a single binary output:



This object is called a perceptron when using the type of weighting scheme we just developed.

A Simple Decision, cont.

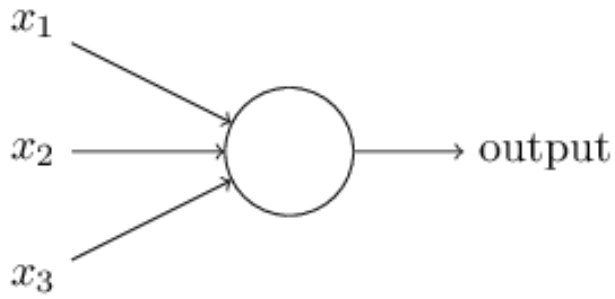
I will write the answers to these question as binary variables $\mathbf{x_i}$, with zero being the answer 'no' and one being the answer 'yes':

1. Is the weather good? $\mathbf{x_1}$
2. Does your friend want to go with you? $\mathbf{x_2}$
3. Is it near public transportation? $\mathbf{x_3}$

Now, what is an easy way to describe the decision statement resulting from these inputs.

Perceptron: Artificial Neurons

Perceptron's were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of **artificial neurons.**



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Rosenblatt proposed a simple rule to compute the output. He introduced *weights*, $w_1, w_2, \ldots$, real numbers expressing the importance of the respective inputs to the output.

The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value*.

Just like the weights, the threshold is a real number which is a parameter of the neuron.

A Simple Decision, cont.

We could determine weights $\mathbf{w_i}$ indicating how important each feature is to whether you would like to attend. We can then see if:

$$\mathbf{x1 \cdot w1 + x2 \cdot w2 + x3 \cdot w3 \geq threshold}$$

For some pre-determined threshold. If this statement is true, we would attend the festival, and otherwise we would not.

A Simple Decision, cont.

For example, if we really hated bad weather but care less about going with our friend and public transit, we could pick the weights 6, 2 and 2.

With a threshold of 5, this causes us to go if and only if the weather is good.

What happens if the threshold is decreased to 3?

What about if it is decreased to 1?

A Simple Decision, cont.

If we define a new binary variable y that represents whether we go to the festival, we can write this variable as:

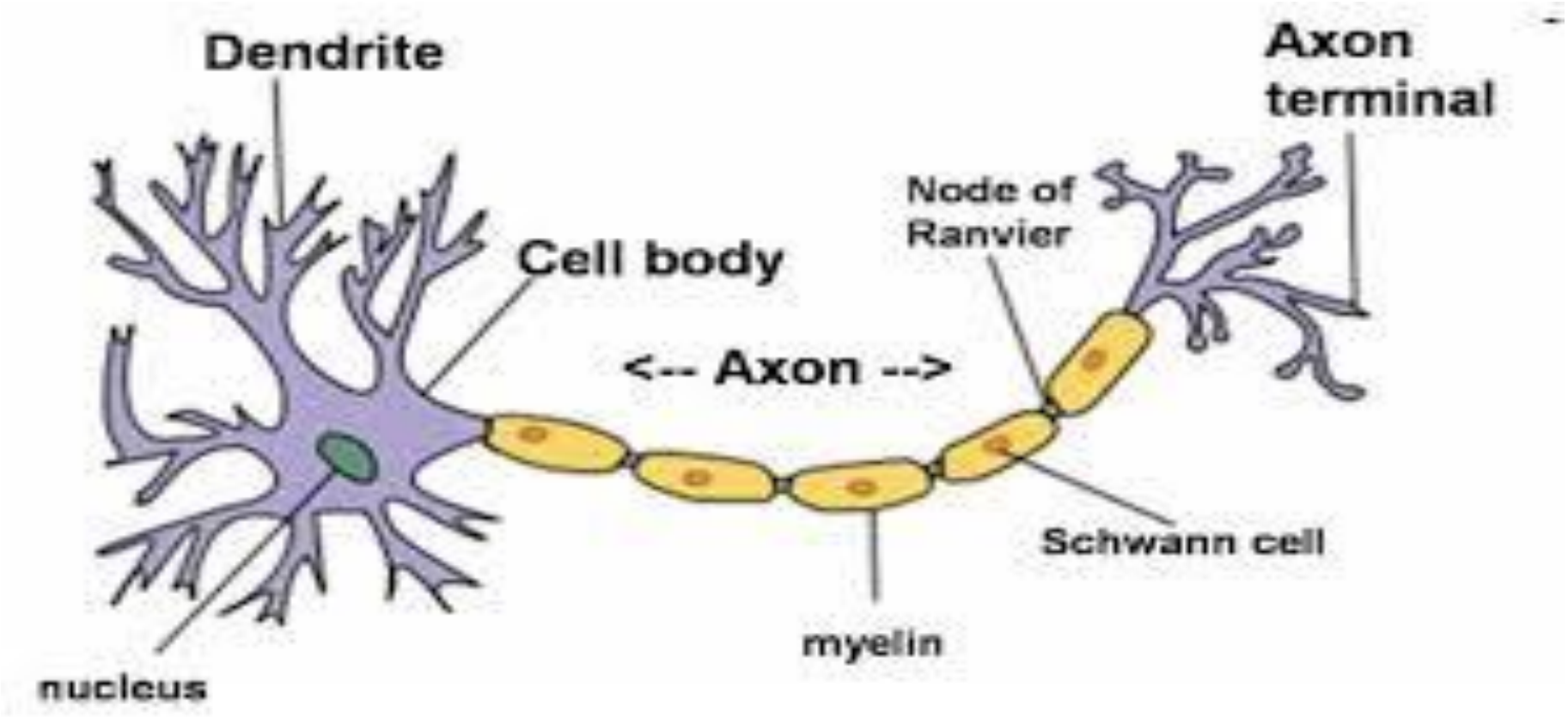
$$y = \begin{cases} 0, & x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 < \text{threshold} \\ 1, & x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 \geq \text{threshold} \end{cases}$$

Now, if I rewrite this in terms of a dot product between the **vector of all binary inputs (x)**, a **vector of weights (w)**, and change the **threshold to the negative bias (b)**, we have:

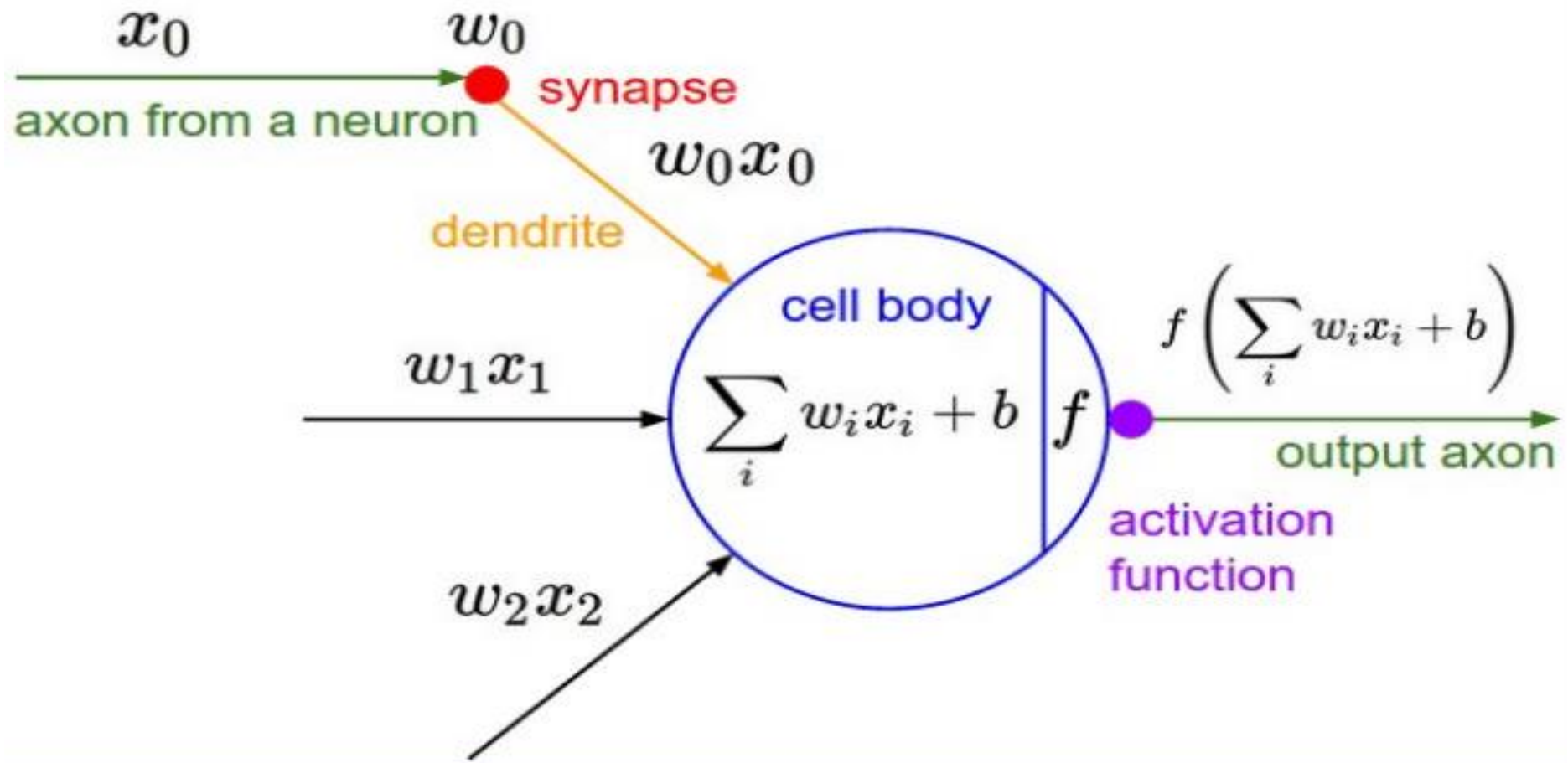
$$y = \begin{cases} 0, & x \cdot w + b < 0 \\ 1, & x \cdot w + b \geq 0 \end{cases}$$

So we found separating hyperplanes.

Neuron: Biological Model



Neuron:
Mapped to Computational Model



Neuron: Mathematical Model

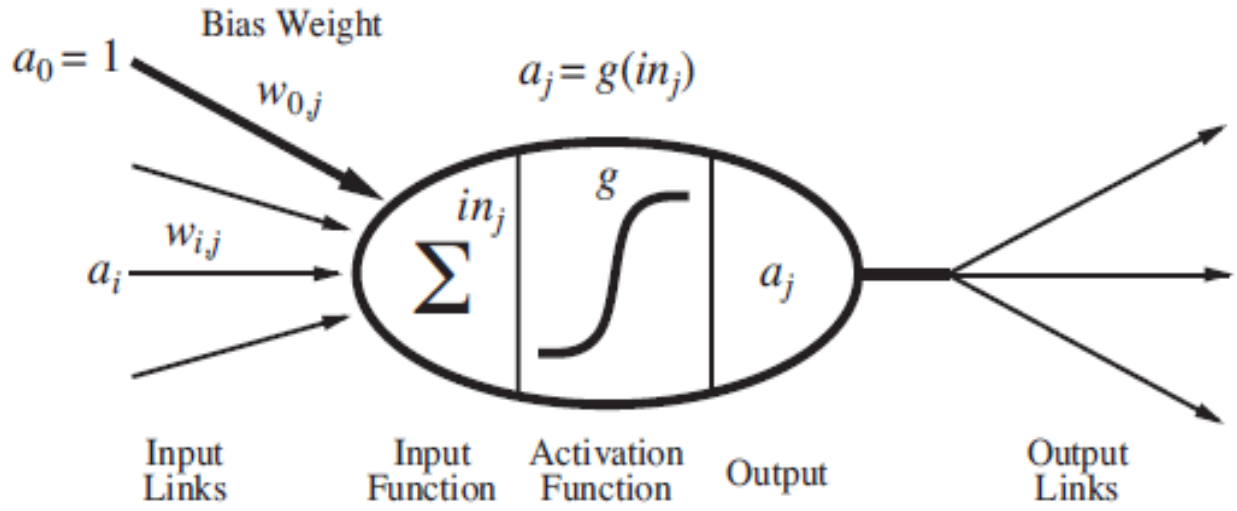
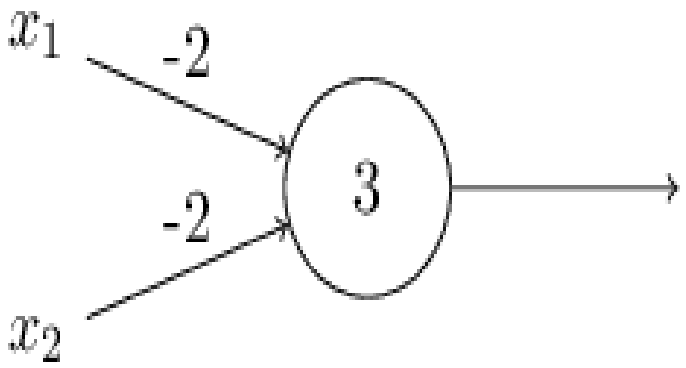


Figure 18.19 A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.

Perceptron: Logical Function

Another way perceptron can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as AND, OR, and NAND.

Suppose we have a perceptron with two inputs, each with weight −2, and an overall bias of 3. Here's our perceptron:



Then we see that input 00 produces output 1, since (−2)∗0+(−2)∗0+3=3 is positive

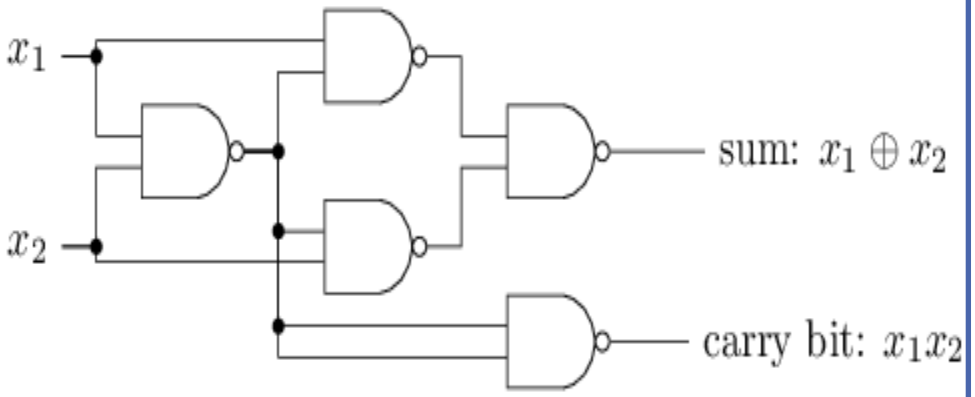
Similar calculations show that the inputs 01 and 10 produce output 1.

But the input 11 produces output 0, since (−2)∗1+(−2)∗1+3=−1 is negative.

And so our perceptron implements a NAND gate!

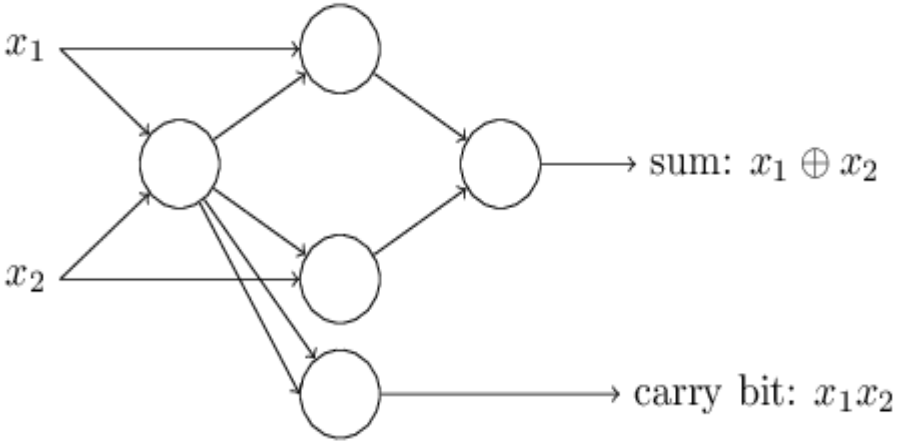
Perceptron: Logical Function, Cond

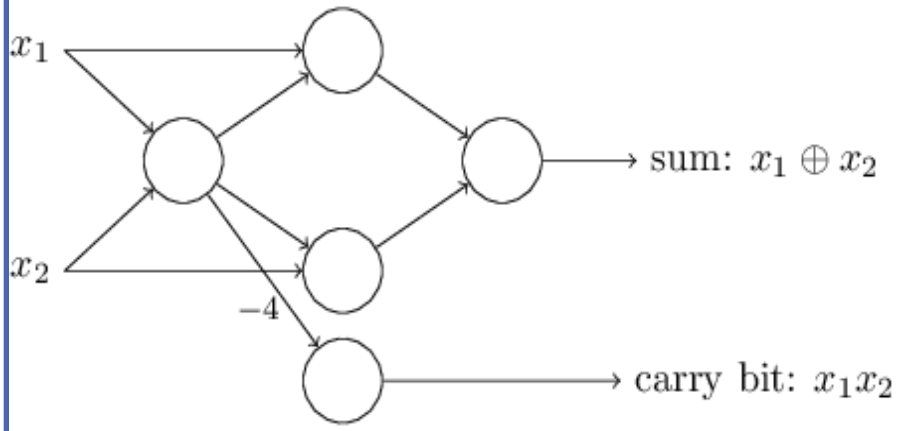
We can use NAND gates to build a circuit which adds two bits, x_1 and x_2



The adder example demonstrates how a network of perceptrons can be used to simulate a circuit containing many NAND gates.

And because NAND gates are universal for computation, it follows that perceptrons are also universal for computation.





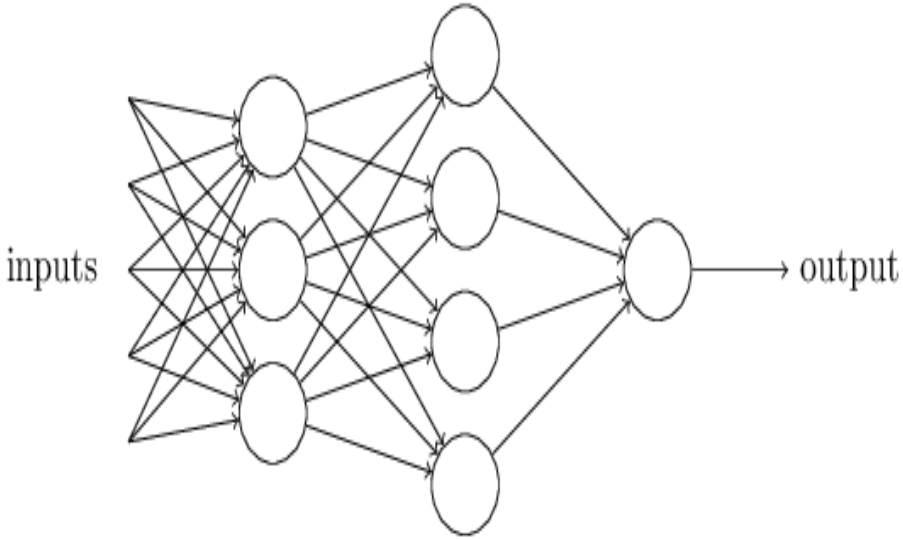
A Network of Perceptrons

In this network, the first column of perceptrons - what we'll call the first *layer* of perceptrons - is making three very simple decisions, by weighing the input evidence.

Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making.

In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer.

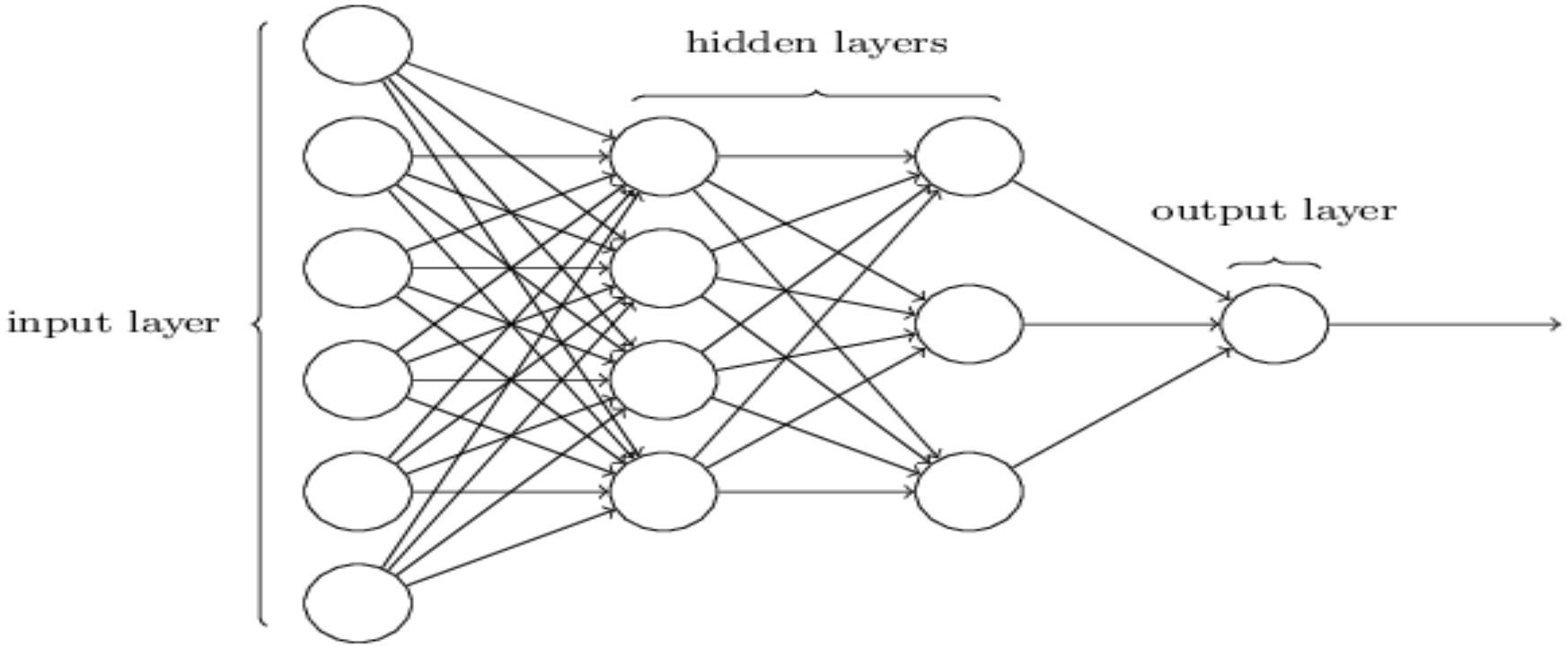
And even more complex decisions can be made by the perceptron in the third layer.



In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

The architecture of neural networks

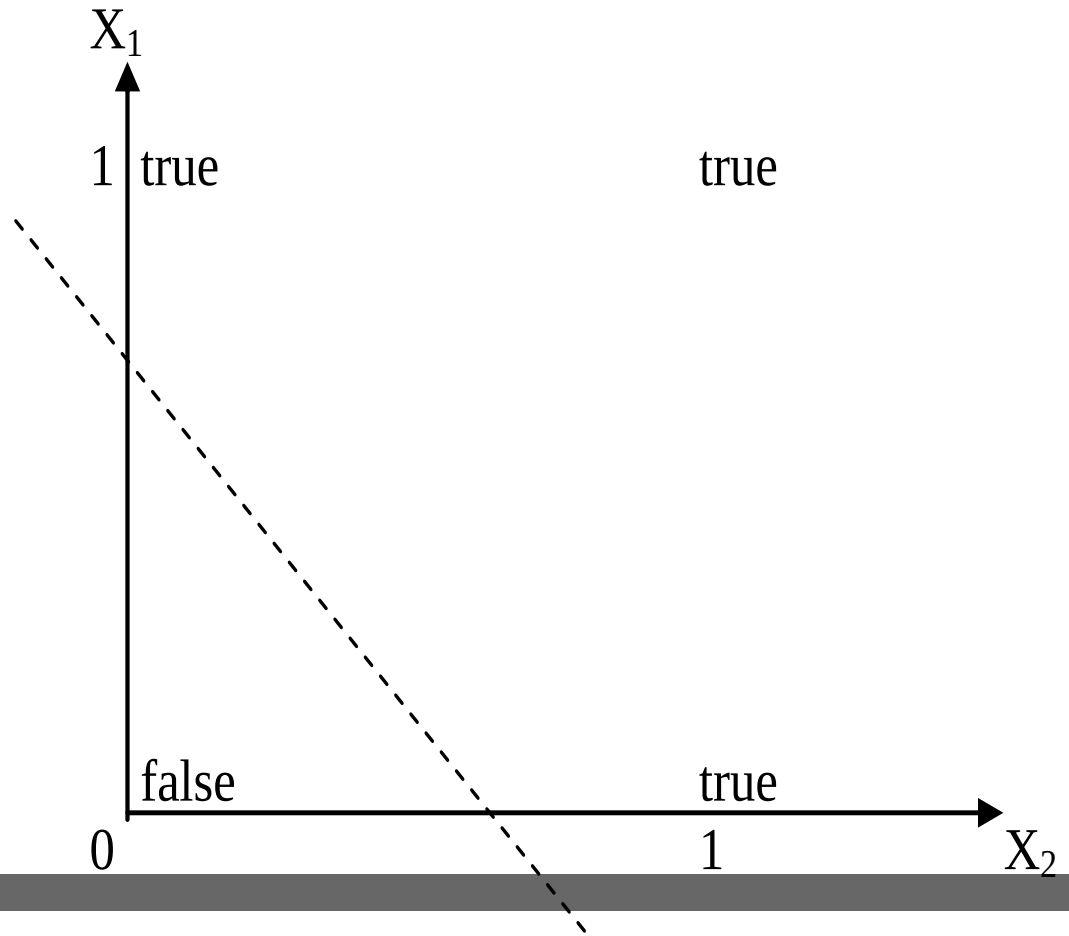
The input and outputs are typically represented as their own neurons, with the other neurons named hidden layers



Perceptron: Classifier

- ✓The Perceptron is used for **binary classification**.
- ✓First train a Perceptron for a classification task.
  - ✓Find suitable weights in such a way that the training examples are correctly classified.
  - ✓Geometrically try to find a hyper-plane that separates the examples of the two classes.
- ✓The Perceptron can only model linearly separable classes.
- ✓**When the two classes are not linearly separable, it may be desirable to obtain a linear separator that minimizes the mean squared error.**
- ✓Given training examples of classes C_1, C_2 train the Perceptron in such a way that :
  - ✓If the output of the Perceptron is +1 then the input is assigned to class C_1
  - ✓If the output is -1 then the input is assigned to C_2

Boolean function OR – Linearly separable



Perceptron: Limitations

- ✓The Perceptron can only model linearly separable functions,
  - ✓those functions which can be drawn in 2-dim graph and single straight-line separates values in two part.
- ✓Boolean functions given below are linearly separable:
  - ✓AND
  - ✓OR
- ✓It cannot model XOR function as it is nonlinearly separable.
  - ✓When the two classes are not linearly separable, it may be desirable to obtain a linear separator that minimizes the mean squared error.

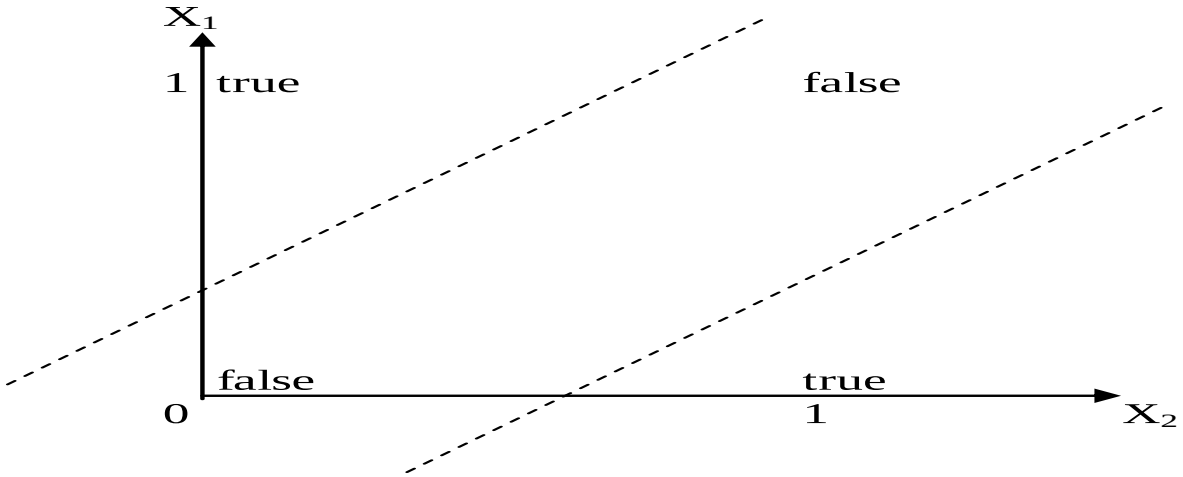
XOR – Nonlinearly separable function

- ✓A typical example of non-linearly separable function is the XOR that computes the logical exclusive or..
- ✓This function takes two input arguments with values in {0,1} and returns one output in {0,1},
- ✓Here 0 and 1 are encoding of the truth values false and true,
- ✓The output is true if and only if the two inputs have different truth values.
- ✓XOR is nonlinearly separable function which can not be modeled by Perceptron.
- ✓For such functions we must use multi layer feed-forward network.

XOR – Nonlinearly separable function

Input		Output
X_1	X_2	X_1 XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

These two classes (true and false) cannot be separated using a line. Hence XOR is nonlinearly separable.



Sigmoid Neurons

Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem.

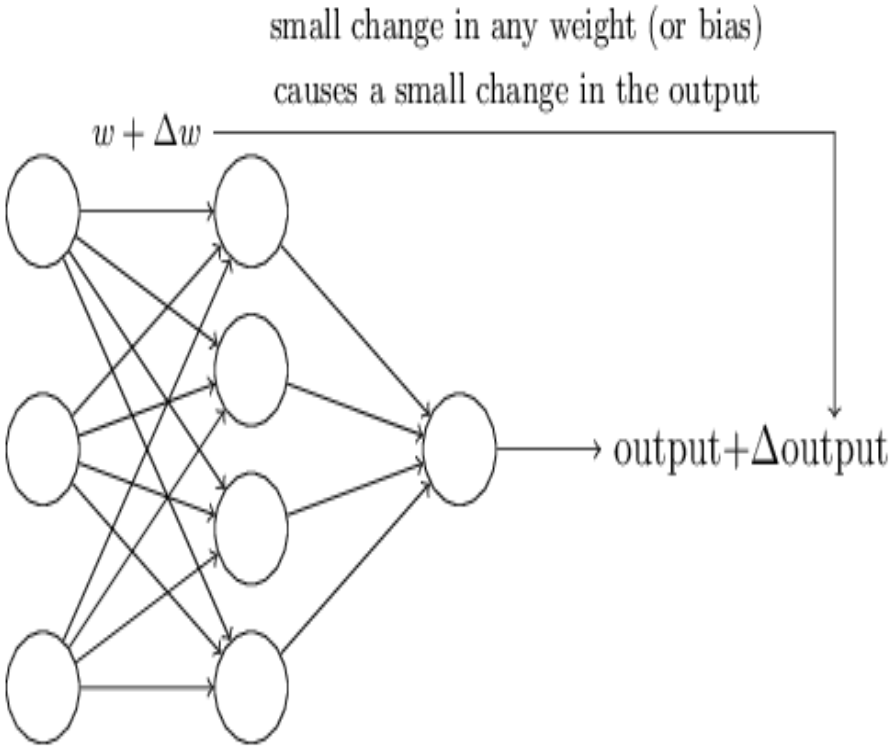
For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit.

And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit.

To see how learning might work, suppose we make a small change in some weight (or bias) in the network.

What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network

Sigmoid Neurons, Cond



If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want.

The problem is that this isn't what happens when our network contains perceptrons.

A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip,
say from 0 to 1.

That flip may then cause the behavior of the rest of the network to completely change in some very complicated way.

Sigmoid Neurons, Cond

We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron.

Sigmoid neurons are like perceptron but modified so that small changes in their weights and bias cause only a small change in their output.

That's the crucial fact which will allow a network of sigmoid neurons to learn.

Just like a perceptron, the sigmoid neuron has inputs**, x_1,x_2,….** But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1.

Also just like a perceptron, the sigmoid neuron has weights for each input, **w_1,w_2,…,** and an overall bias**, b**.

But the output is not 0 or 1.

Instead, it's **$\sigma(w \cdot x+b)$**, where
σ is called the **sigmoid function.**

Sigmoid Function

Sigmoid Function is defined by

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

This perfectly mimics logistic regression, and in fact uses the logit function to do so.

In the neural network literature, the logit function is called the sigmoid function, thus leading to the name sigmoid neuron for a neuron that uses its logic.

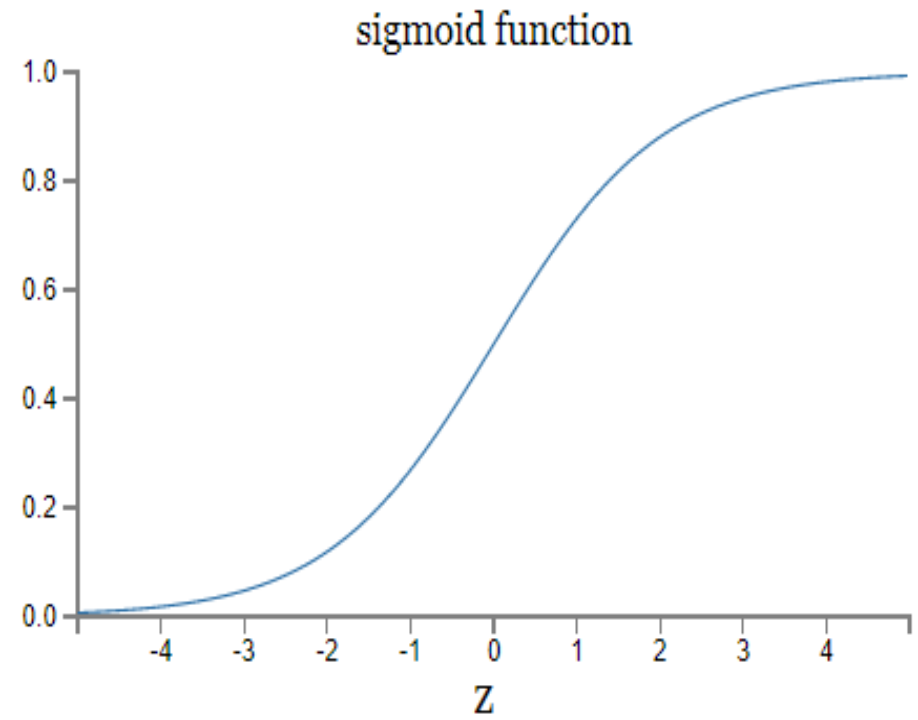
The output of a sigmoid neuron with inputs $x_1, x_2, \ldots$, weights $w_1, w_2, \ldots$, and bias b is

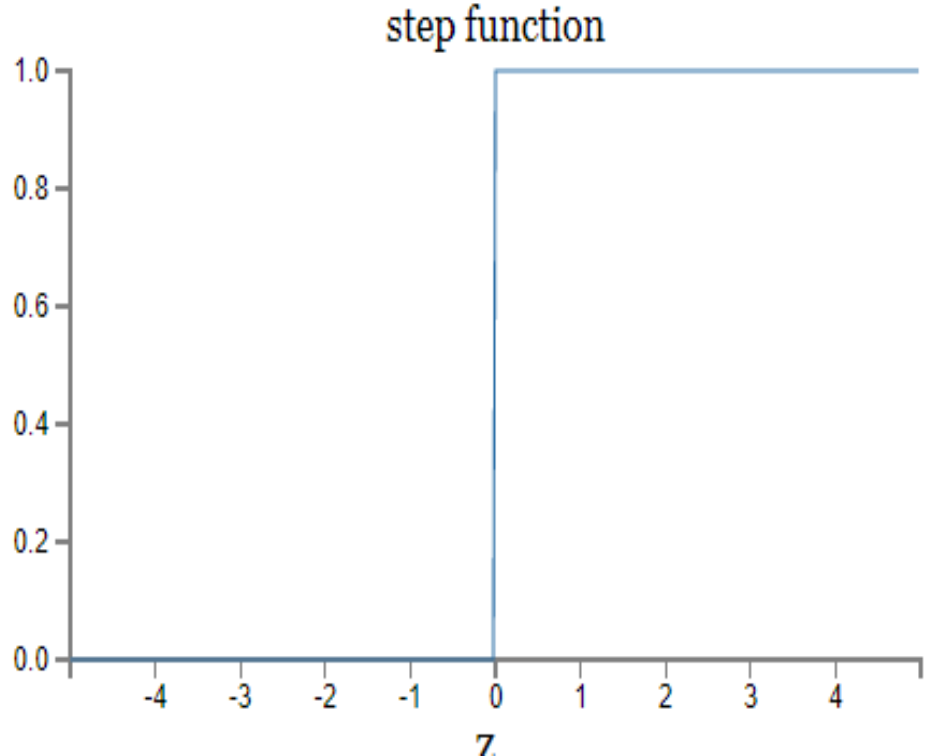
$$\sigma(x \cdot w + b)$$

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

Sigmoid Function

Sigmoid Function is a smoothed-out version of a step function





Activation Functions

Activation Function derive the output form weighted sum

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

In the sigmoid neuron example, the choice of what function to use to go from **x · w + b** to an output is called the activation function.

Using a logistic, or sigmoid, activation function has some benefits in being able to easily take derivatives and the interpret them using logistic regression.

Other choices have certain benefits that have recently grown in popularity. Some of these include:

1. hyperbolic tan: tanh(z) = 2σ(2x) − 1
2. rectified linear unit: ReLU(z) = max(0, z)
3. leaky rectified linear unit
4. maxout

The Backpropagation Algorithm

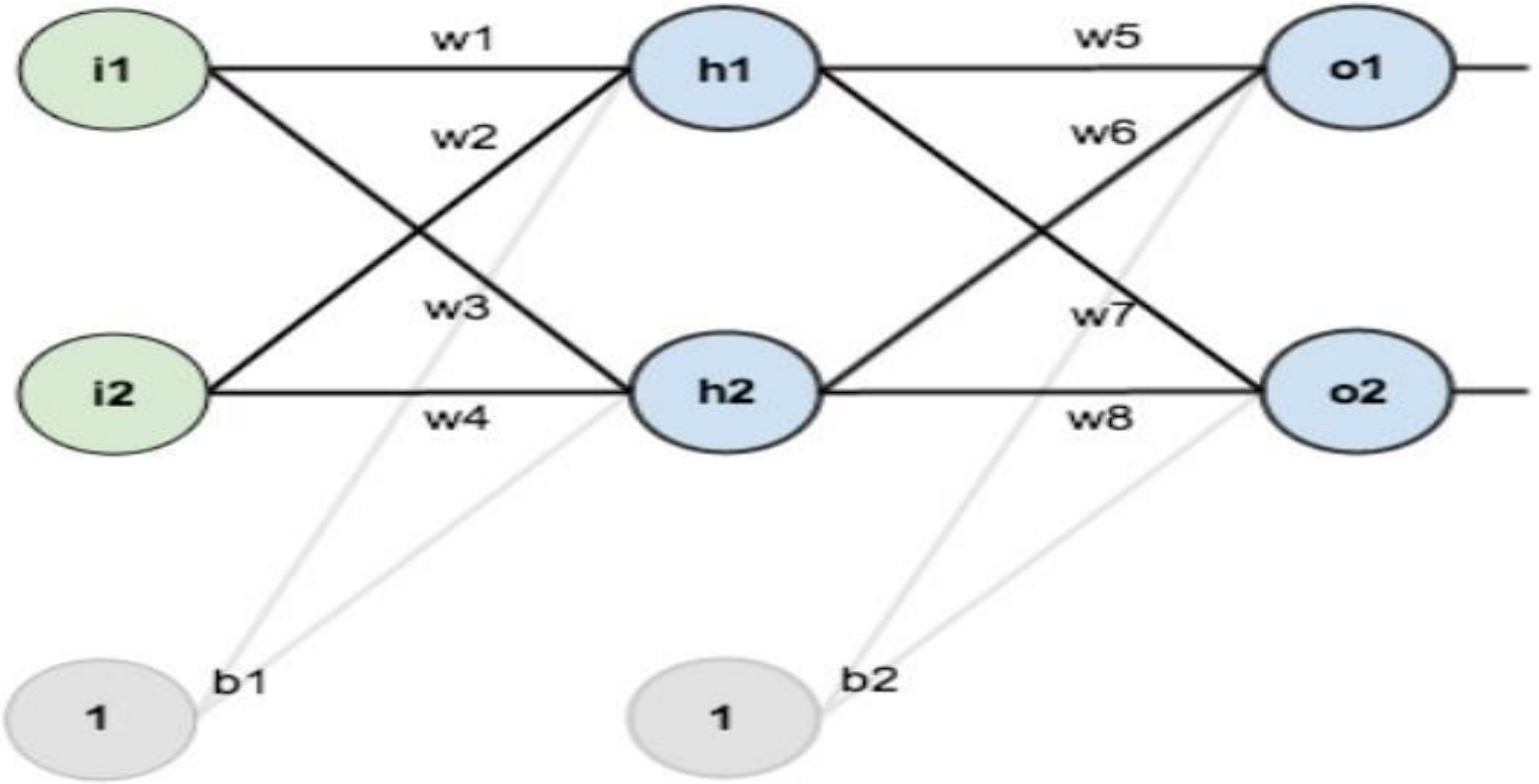
✓The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

✓The most basic method of training a neural network is trial and error.

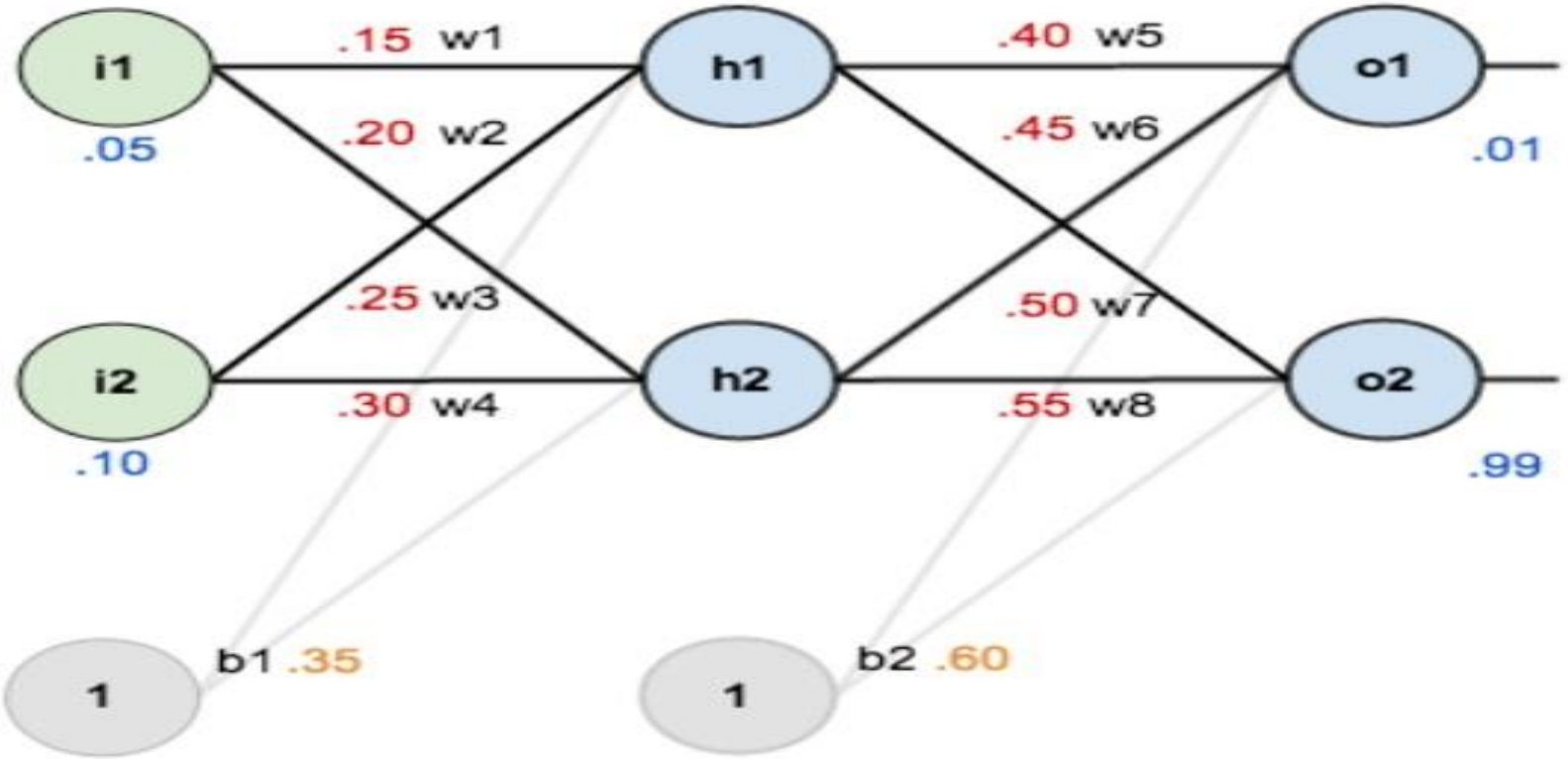
✓If the network isn't behaving the way it should, change the weighting of a random link by a random amount. If the accuracy of the network declines, undo the change and make a different one.

✓It takes time, but the trial and error method does produce results.

Sample Neural Network to Calculate Total Error Using Forward Pass Backpropagation



Sample Neural Network to Calculate Total Error Using Forward Pass Backpropagation



Total net input for h1

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

Sigmoid function to get the output of h1

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

$$out_{h2} = 0.596884378$$

Repeat this process for the output layer neurons,

using the output from the hidden layer neurons as inputs.

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

$$out_{o2} = 0.772928465$$

Calculating the Total Error

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

The total error for the neural network is the sum of these errors:

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

References

1. Chapter 18: Learning From Examples , Pages 727-34“Artificial Intelligence: A Modern Approach,” by Stuart J. Russell and Peter Norvig
2. “Neural Computing Theory and Practice,” by Philip D. Wasserman .
3. “Neural Network Design,” by Martin T. Hagan, Howard B. Demuth, Mark H. Beale
4. http://euler.stat.yale.edu/~tba3/stat665/lectures/lec12/lecture12.pdf
5. http://neuralnetworksanddeeplearning.com/chap1.html

Books

1. “Artificial Intelligence: A Modern Approach,” by Stuart J. Russell and Peter Norvig.
2. "Artificial Intelligence: Structures and Strategies for Complex Problem Solving", by George F. Luger, (2002)
3. "Artificial Intelligence: Theory and Practice", by Thomas Dean.
4. "AI: A New Synthesis", by Nils J. Nilsson.
5. “Programming for machine learning,” by J. Ross Quinlan,
6. “Neural Computing Theory and Practice,” by Philip D. Wasserman, .
7. “Neural Network Design,” by Martin T. Hagan, Howard B. Demuth, Mark H. Beale, .
8. “Practical Genetic Algorithms,” by Randy L. Haupt and Sue Ellen Haupt.
9. “Genetic Algorithms in Search, optimization and Machine learning,” by David E. Goldberg.
10. "Computational Intelligence: A Logical Approach", by David Poole, Alan Mackworth, and Randy Goebel.
11. “Introduction to Turbo Prolog”, by Carl Townsend.