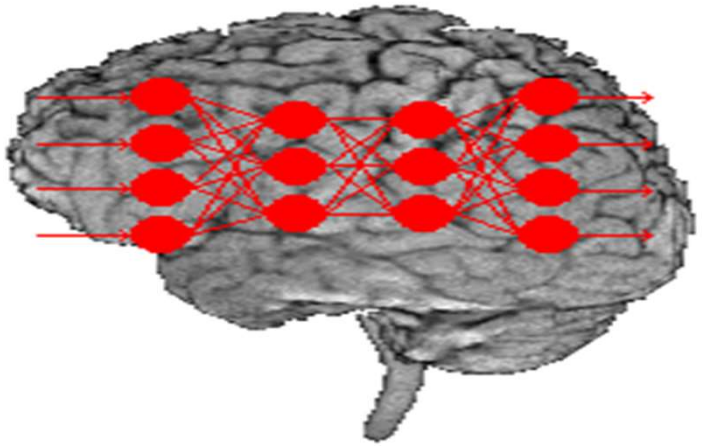
Artificial Neural Networks

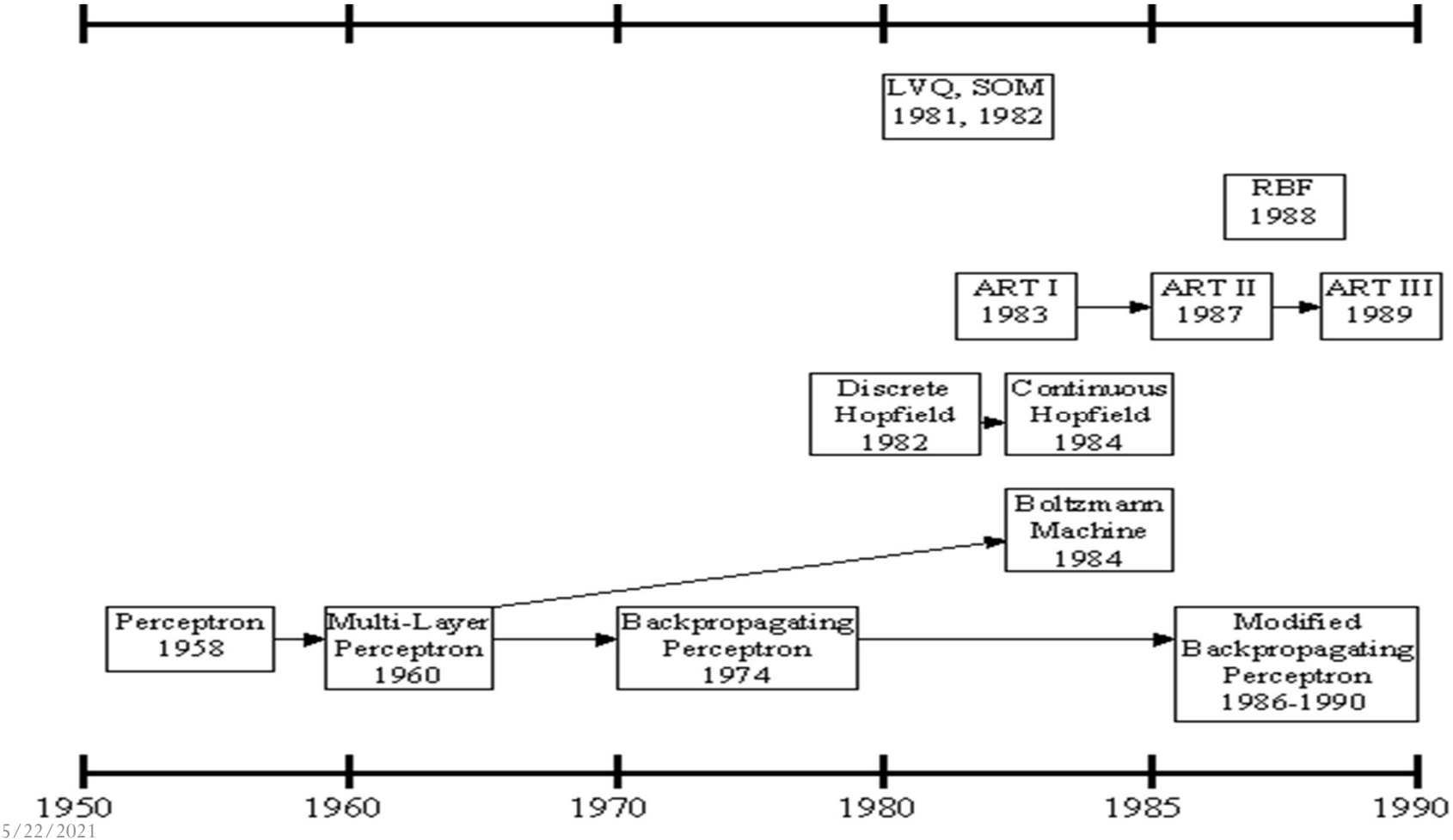


Dr.Ravi Kumar Chandu

Agenda

- History of Artificial Neural Networks
- What is an Artificial Neural Networks?
- How it works?

History of Artificial Neural Networks



History of the Artificial Neural Networks

- 1943 McCulloch-Pitts neurons
- 1949 Hebb's law
- 1958 Perceptron (Rosenblatt)
- 1960 Adaline, better learning rule (Widrow, Huff)
- 1969 Limitations (Minsky, Papert)
- 1972 Kohonen nets, associative memory

History of the Artificial Neural Networks

- 1977 Brain State in a Box (Anderson)
- 1982 Hopfield net, constraint satisfaction
- 1985 ART (Carpenter, Grossfield)
- 1986 Backpropagation (Rumelhart, Hinton, McClelland)
- 1988 Neocognitron, character recognition (Fukushima)

Artificial Neural Network

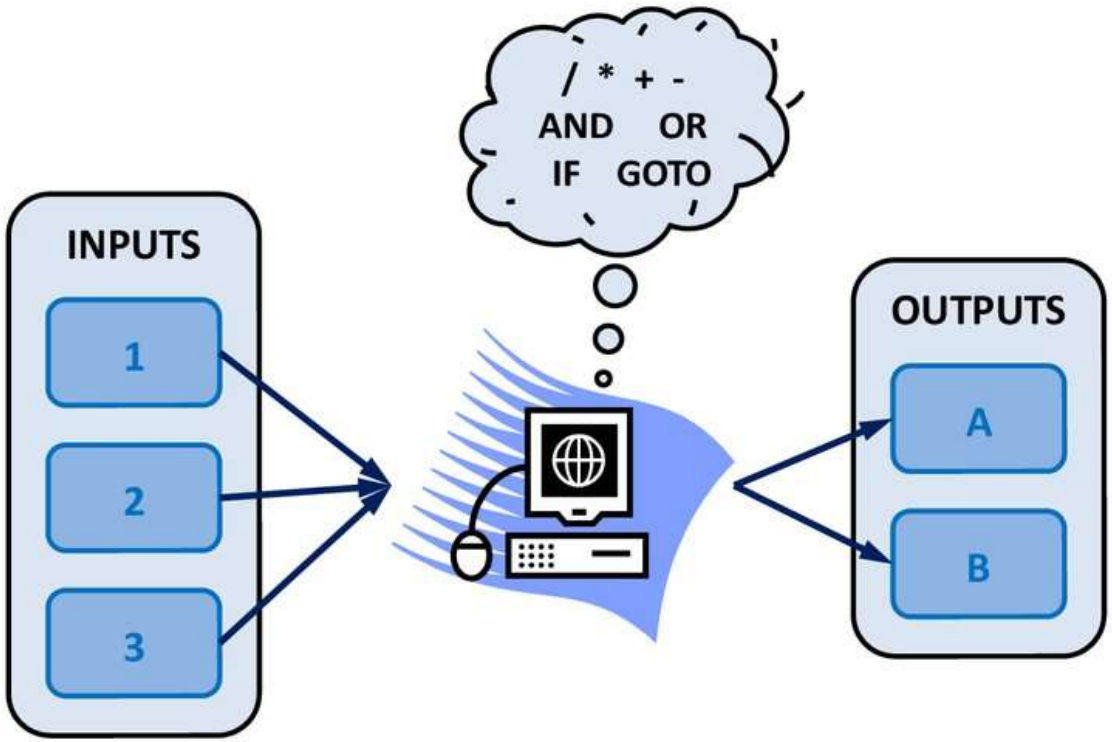
- An artificial neural network consists of a pool of simple processing units which communicate by sending signals to each other over a large number of weighted connections.

A set of major aspects of a parallel distributed model include:

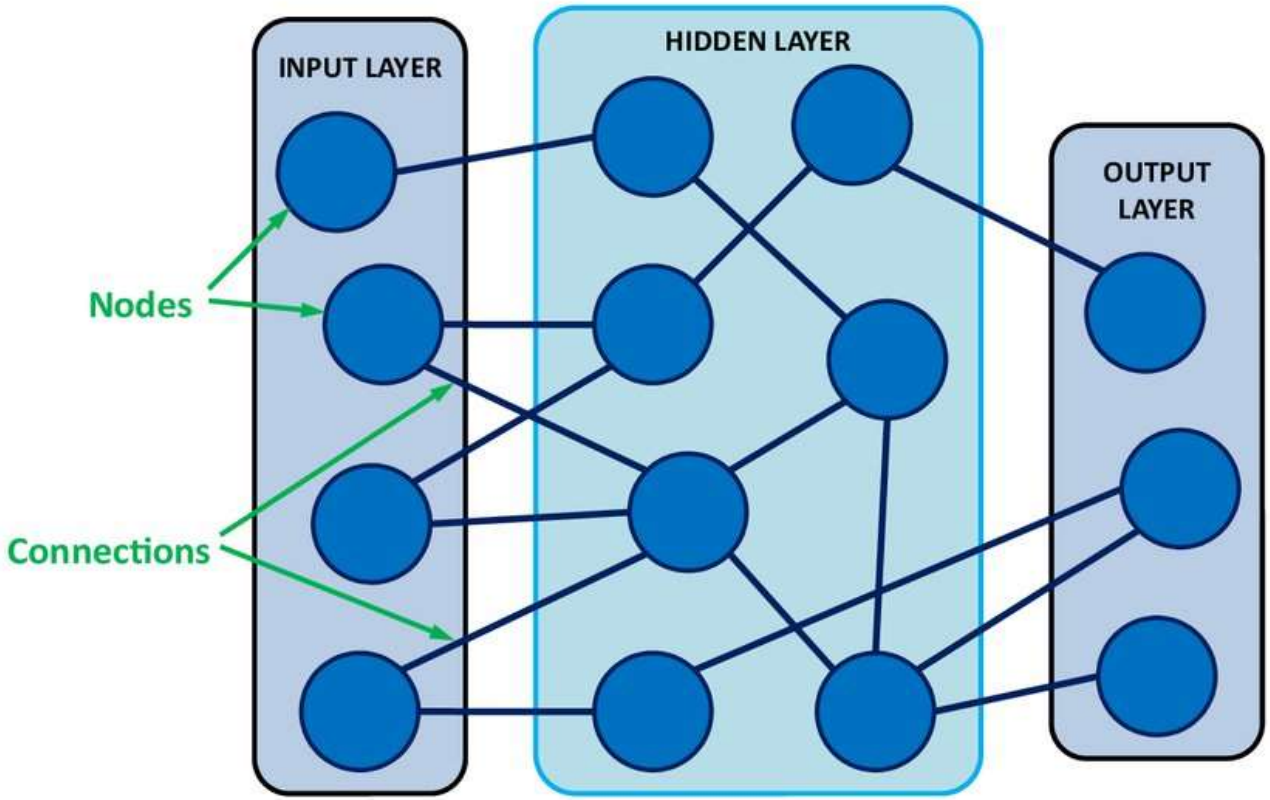
- a set of processing units (cells).
- a state of activation for every unit, which equivalent to the output of the unit.
- connections between the units. Generally each connection is defined by a weight.
- a propagation rule, which determines the effective input of a unit from its external inputs.
- an activation function, which determines the new level of activation based on the effective input and the current activation.
- an external input for each unit.
- a method for information gathering (the learning rule).
- an environment within which the system must operate, providing input signals and _ if necessary _ error signals.

	Brain	Computer
Processing Elements	10^{10} neurons	10^{8} transistors
Element Size	10^{-6} m	10^{-6} m
Energy Use	30 W	30 W (CPU)
Processing Speed	10^{2} Hz	10^{12} Hz
Style Of Computation	Parallel, Distributed	Serial, Centralized
Energetic Efficiency	10^{-16} joules/opn/sec	10^{-6} joules/opn/sec
Fault Tolerant	Yes	No
Learns	Yes	A little

Conventional Computer Model



Neural Network As A Computer Model



Computers vs. Natural Neural Networks

- Precise design, highly constrained, not very adaptive or fault tolerant, Centralized control, deterministic, basic switching times $\backsim 10^{-9}$
- Massively parallel, highly adaptive and fault tolerant, self configuring, self repairing, noisy, stochastic, basic switching time $\backsim 10^{-3}$ sec

Appropriate Problems for Neural Network Learning

- Instances are represented by many attribute-value pairs (e.g., the pixels of a picture. ALVINN [Mitchell, p. 84]).
- The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.
- The training examples may contain errors.
- Long training times are acceptable.
- Fast evaluation of the learned target function may be required.
- The ability for humans to understand the learned target function is not important.

Why Artificial Neural Networks?

- Can be viewed as one approach towards understanding brain/building intelligent machines
- Computational architectures inspired by brain Computational methods for ‘learning’ dependencies in data stream
  e.g. Pattern Recognition, System identification
- Characteristics: Emergent properties, learning, self adaptation
- Modeling Biology?
  Mathematically purified neurons!!

Why Artificial Neural Networks?

- There are two basic reasons why we are interested in building artificial neural networks (ANNs):

- **Technical viewpoint:** Some problems such as character recognition or the prediction of future states of a system require massively parallel and adaptive processing.

- **Biological viewpoint:** ANNs can be used to replicate and simulate components of the human (or animal) brain, thereby giving us insight into natural information processing.

What are neural networks used for?

Classification: Assigning each object to a known specific class

Clustering: Grouping together objects similar to each other

Pattern Association: Presenting of an input sample triggers the generation of specific output pattern

Function approximation: Constructing a function generating almost the same outputs from input data as the modeled process

Optimization: Optimizing function values subject to constraints

Forecasting: Predicting future events on the basis of past history

Control: Determining values for input variables to achieve desired values for output variables

What is an Artificial Neural Network?

"A parallel distributed information processor made up of simple processing units that has a propensity for acquiring problem solving knowledge through experience "

- A large network of interconnected units
- Each unit has simple input-output mapping
- Each interconnection has numerical weight attached to it
- Output of unit depends on outputs and connection weights of units connected to it
- 'Knowledge' resides in the weights
- Problem solving ability is often through learning

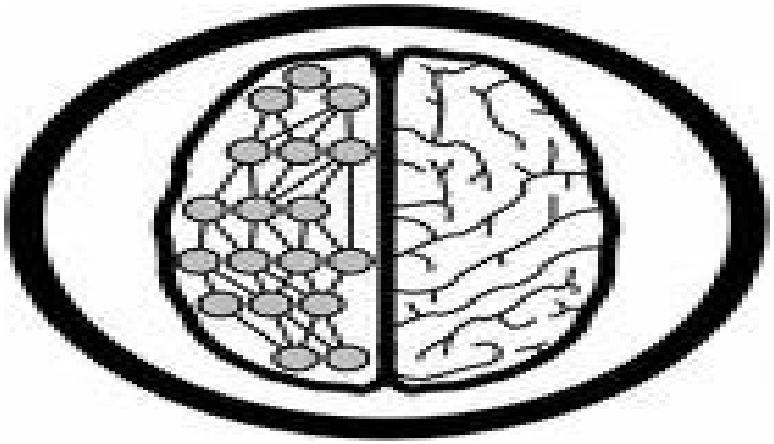
An architecture inspired by the structure of Brain

Artificial Neural Networks

- The “building blocks” of neural networks are the **neurons**.
  - In technical systems, we also refer to them as **units** or **nodes**.
- Basically, each neuron
  - receives **input** from many other neurons.
  - changes its internal state (**activation**) based on the current input.
  - sends **one output signal** to many other neurons, possibly including its input neurons (recurrent network).

How do ANNs work?

- An artificial neural network (ANN) is either a hardware implementation or a computer program which strives to simulate the information processing capabilities of its biological exemplar. ANNs are typically composed of a great number of interconnected artificial neurons. The artificial neurons are simplified models of their biological counterparts.
- ANN is a technique for solving problems by constructing software that works like our brains.



Consider humans

- Neuron - the basic computing unit
- Brain is a highly organized structure of networks of interconnected neurons

Neuron switching time	0.001 second
Number of neurons	10^{11} (100 billion)
Connections per neuron	10^4 to 10^5
Total synapses	10^{15}
Scene recognition time	0.1 second
Processing	Much parallel computation

Properties of artificial neural networks(ANN's)

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

When to consider neural networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

Application Examples

- Speech phoneme recognition
- Image Classification
- Financial prediction
- Handwriting recognition
- Computer vision

Advantages of ANNs

- Distributed representation
- Representation and processing of fuzziness
- Highly parallel and distributed action
- Speed and fault-tolerance

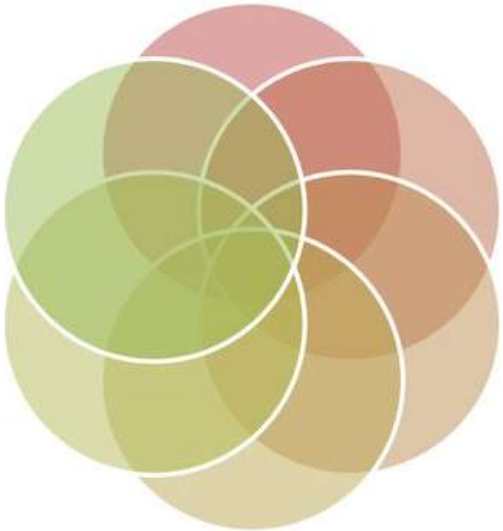
How do our brains work?

- The Brain is A massively parallel information processing system.
- Our brains are a huge network of processing elements. A typical brain contains a network of 10 billion neurons.

Characteristics of a Biological Brain

parallel computing;

Learning is based only on local information



learning ability and generalization;

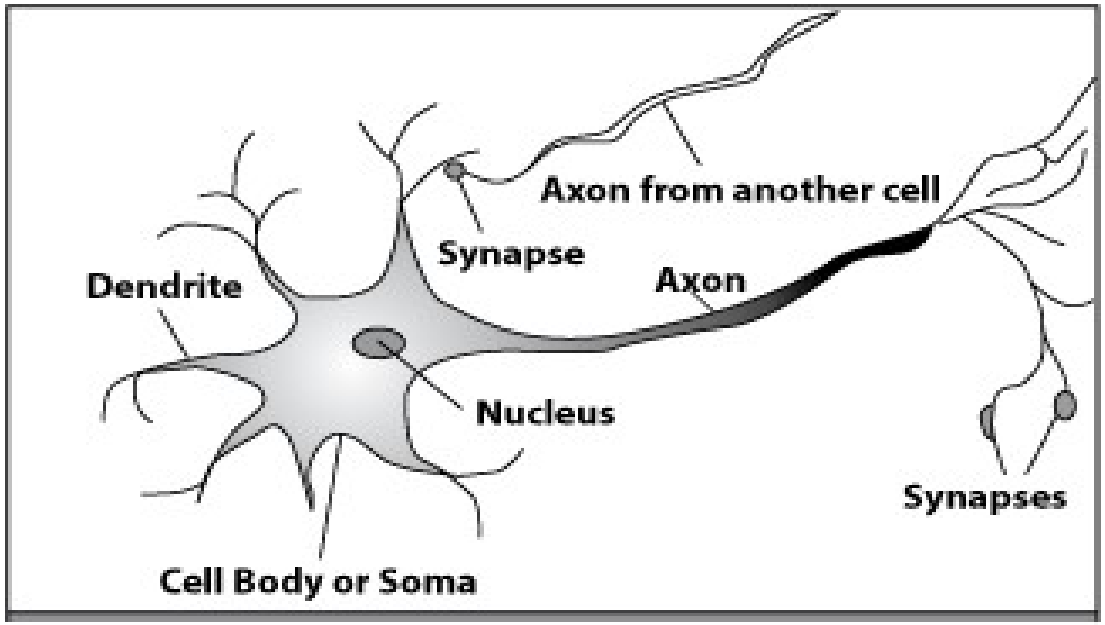
Learning is constant and usually unsupervised

Connections get reorganized based on experience

Performance degrades if some units are removed (i.e. some nerve cells die)

How do our brains work?

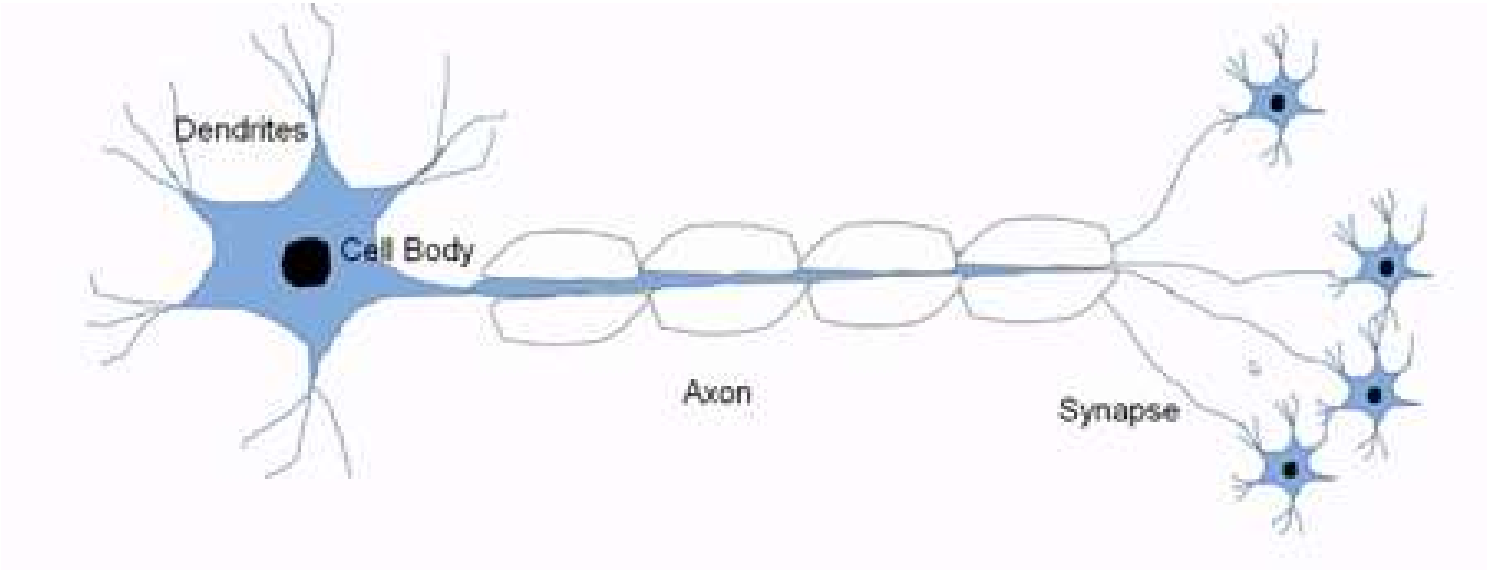
- Biological Neuron - A processing element



Dendrites: Input
Soma/Cell body: Processor
Synaptic: Link
Axon: Output

How do our brains work?

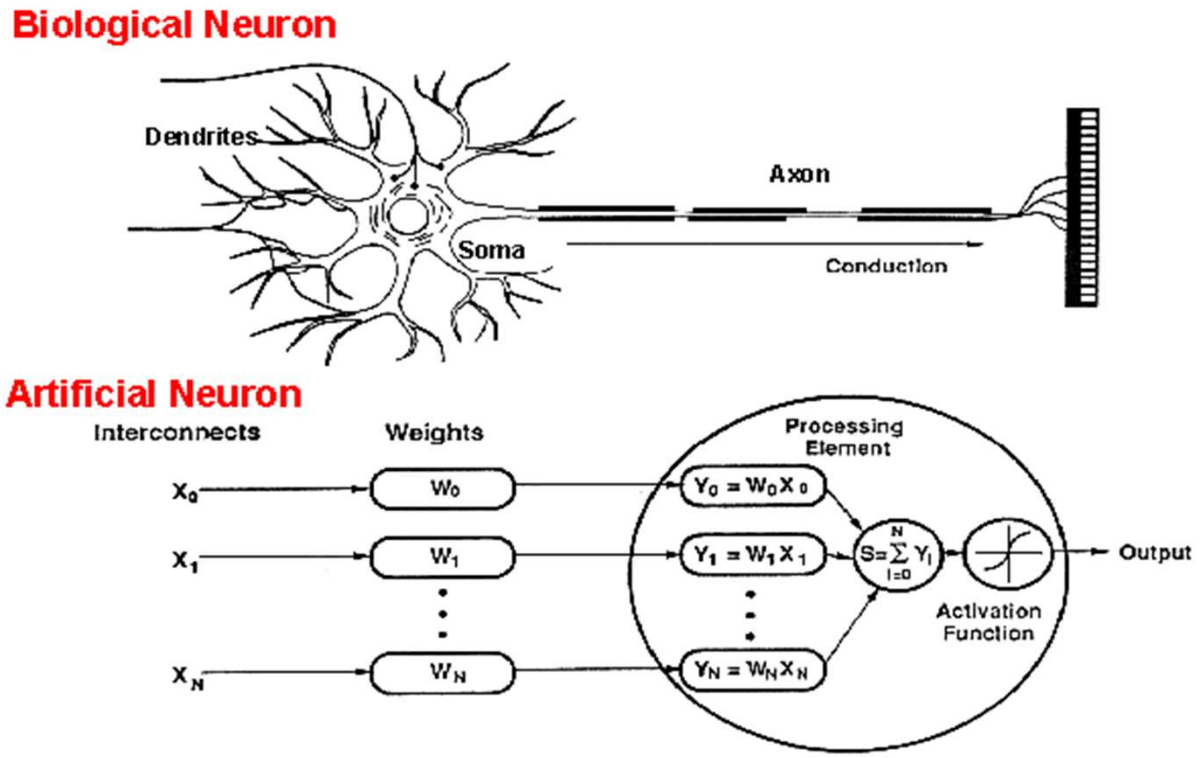
- dendrites: nerve fibres carrying electrical signals to the cell
- cell body: computes a non-linear function of its inputs
- axon: single long fiber that carries the electrical signal from the cell body to other neurons
- synapse: the point of contact between the axon of one cell and the dendrite of another, regulating a chemical connection whose strength affects the input to the cell.



How do our brains work?

- A neuron is connected to other neurons through about *10,000 synapses*
- A neuron receives input from other neurons. Inputs are combined
- Once input exceeds a critical level, the neuron discharges a spike - an electrical pulse that travels from the body, down the axon, to the next neuron(s)
- The axon endings almost touch the dendrites or cell body of the next neuron.
- Transmission of an electrical signal from one neuron to the next is effected by neurotransmitters.
- Neurotransmitters are chemicals which are released from the first neuron and which bind to the Second
- This link is called a synapse. The strength of the signal that reaches the next neuron depends on factors such as the amount of neurotransmitter available

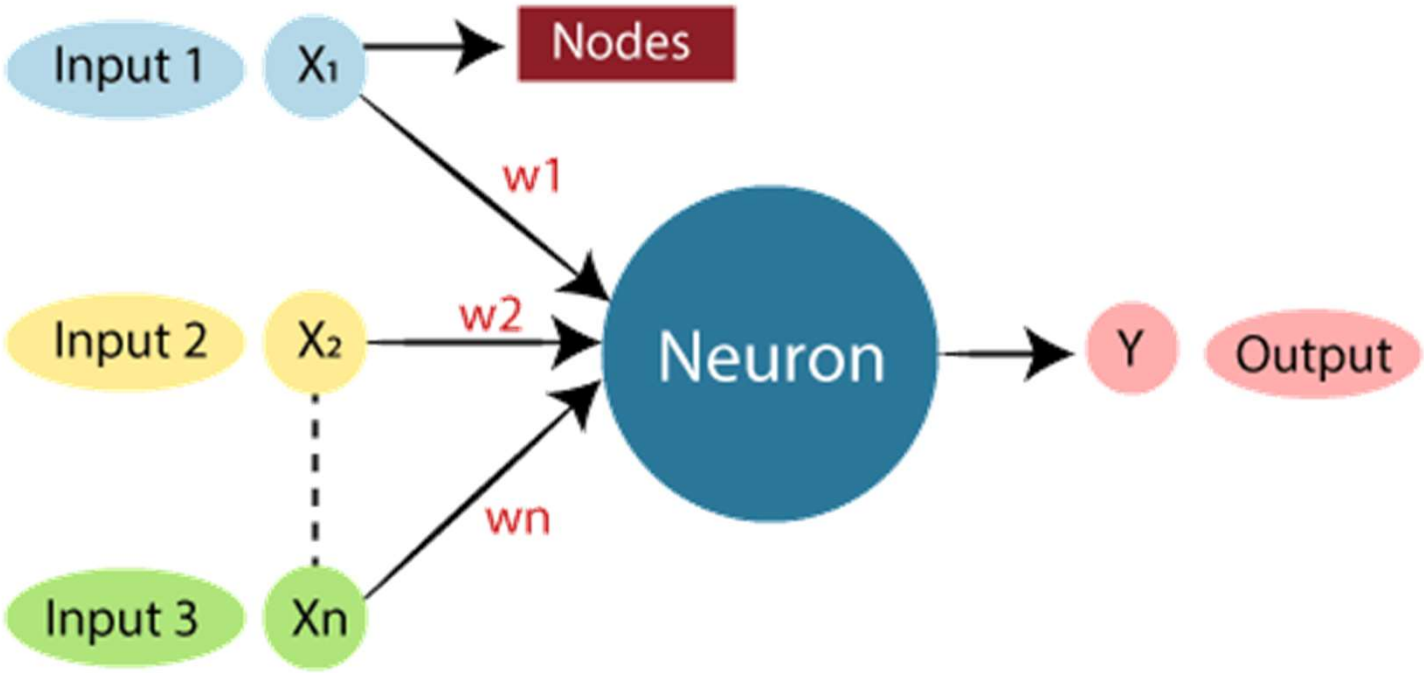
How do ANNs work?



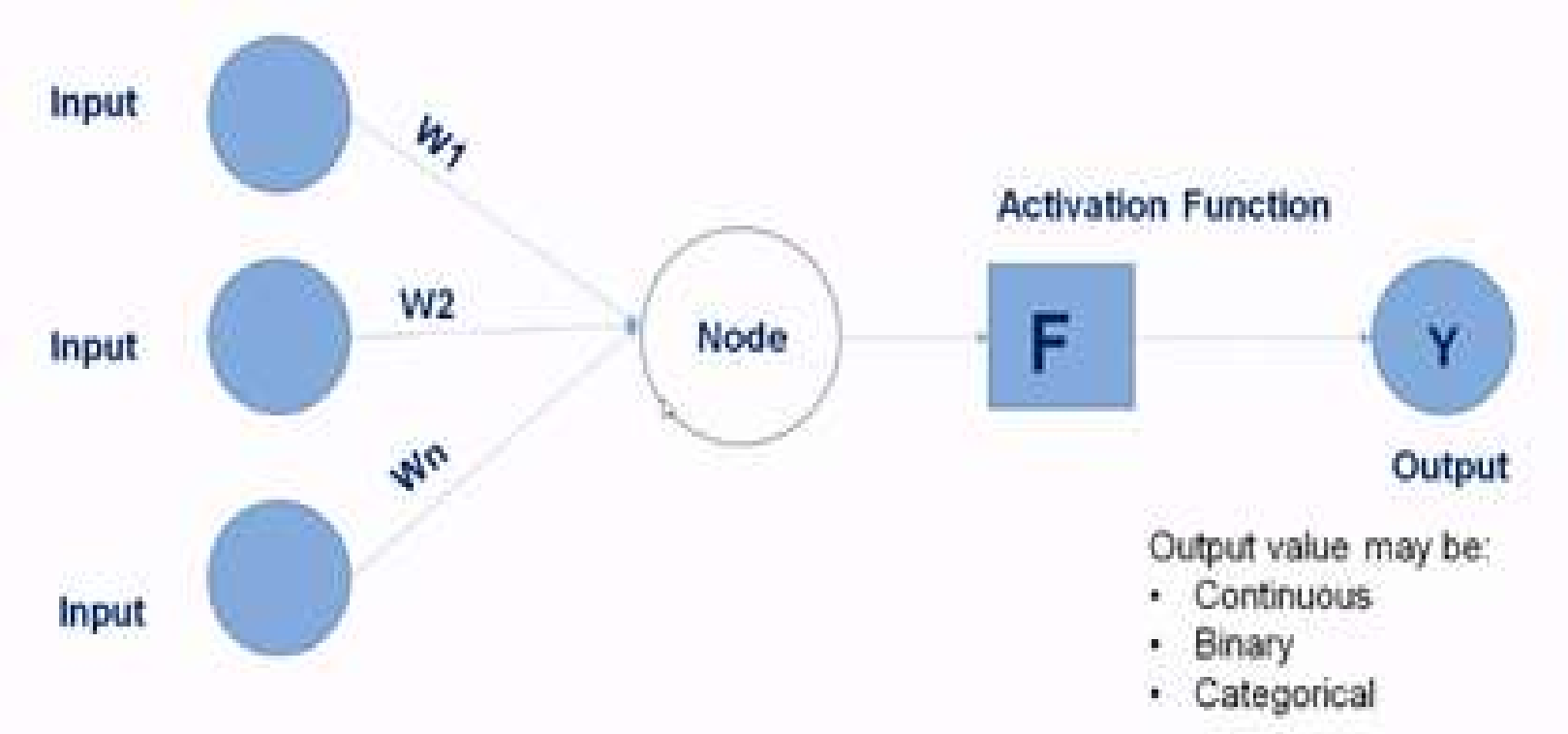
- An artificial neuron is an imitation of a human neuron.

Biological Neuron Network BNN	Artificial Neural Network ANN
Soma	Node
Dendrites	Input
Synapse	Weights or Interconnections
Axon	Output

Artificial Neuron

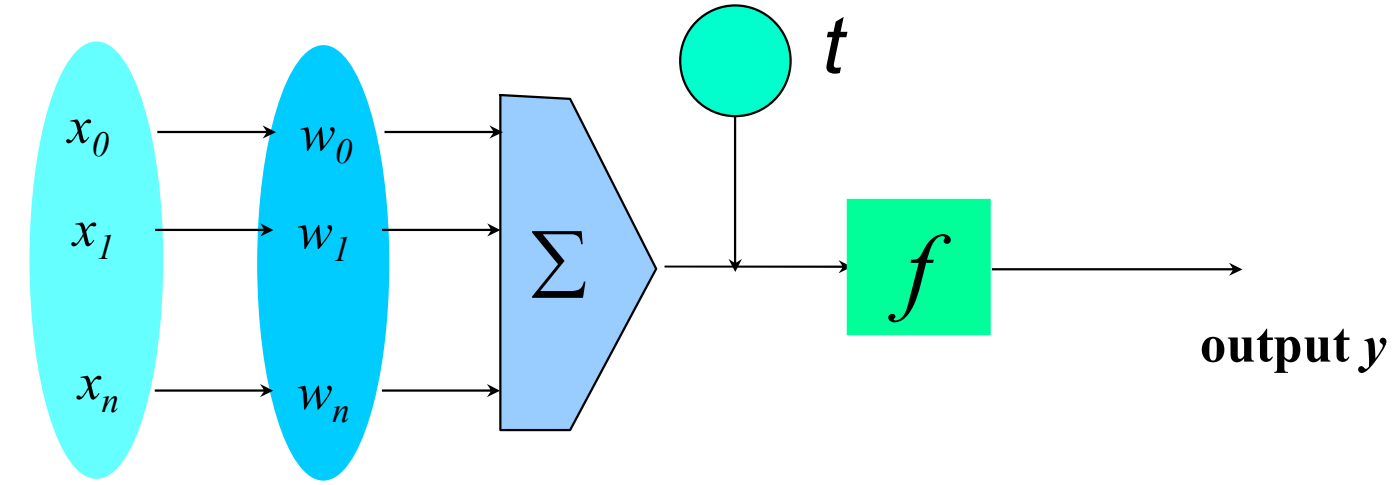


Perceptron



Artificial Neuron is also called as perceptron.

A Neuron (= a perceptron)



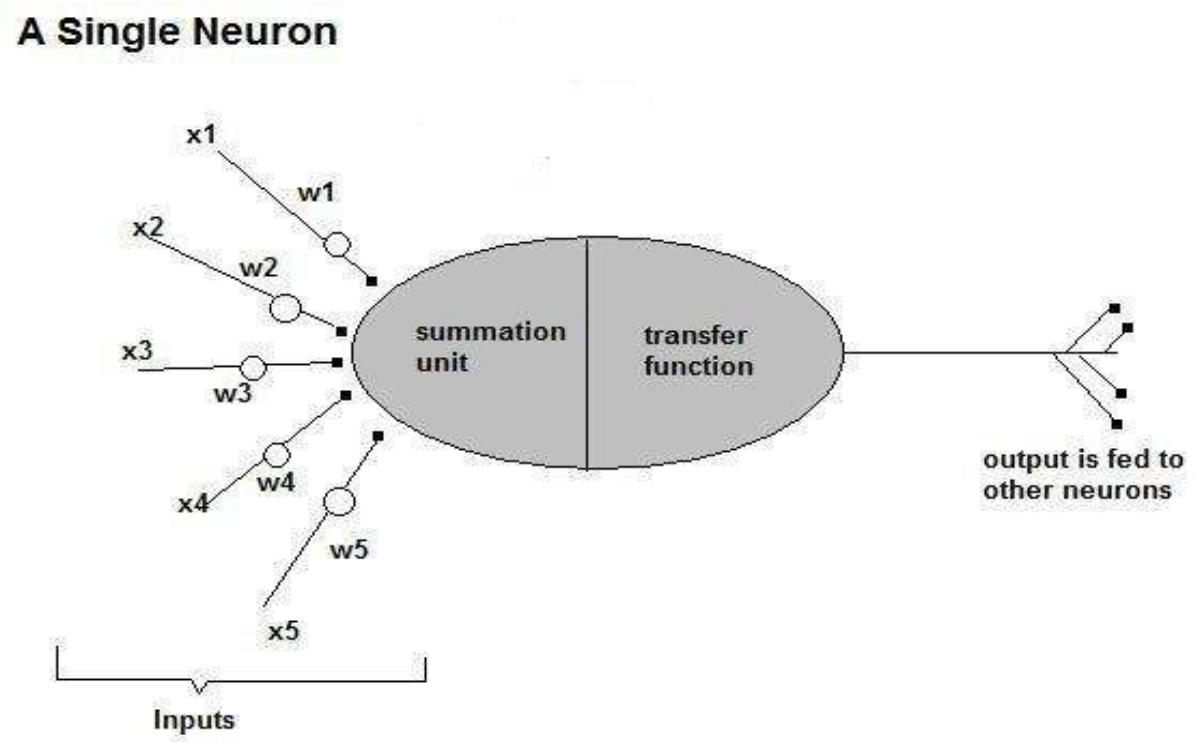
- The *n*-dimensional input vector **x** is mapped into variable y by means of the scalar product and a nonlinear function mapping

Perceptron

- Basic unit in a neural network
- Linear separator
- Parts
  - N inputs, $x_1 \ldots x_n$
  - Weights for each input, $w_1 \ldots w_n$
  - A bias input x_0 (constant) and associated weight w_0
  - Weighted sum of inputs, $y = w_0x_0 + w_1x_1 + \ldots + w_nx_n$
  - A threshold function or activation function,
    - i.e 1 if y > t, -1 if y <= t

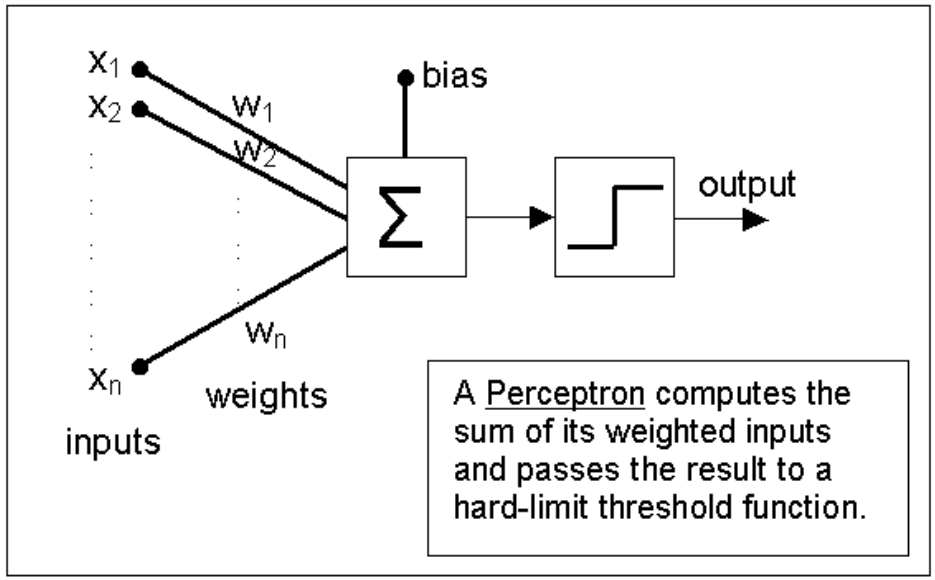
How do ANNs work?

- Now, let us have a look at the model of an artificial neuron.



Perceptron

- Perceptron is a type of artificial neural network (ANN)

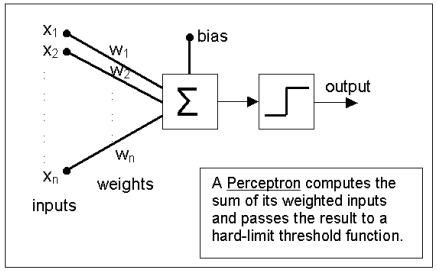


Perceptron - Operation

- It takes a vector of real-valued inputs, calculates a linear combination of these inputs, then output 1 if the result is greater than some threshold and -1 otherwise

$$R = w_0 + w_1 x_1 + w_2 x_2, \cdots, w_n x_n = w_0 + \sum_{i=1}^{n} w_i x_i$$

$$o = sign(R) = \begin{cases} +1; & \text{if} \quad R > 0 \\ -1, & \text{otherwise} \end{cases}$$



Perceptron – Decision Surface

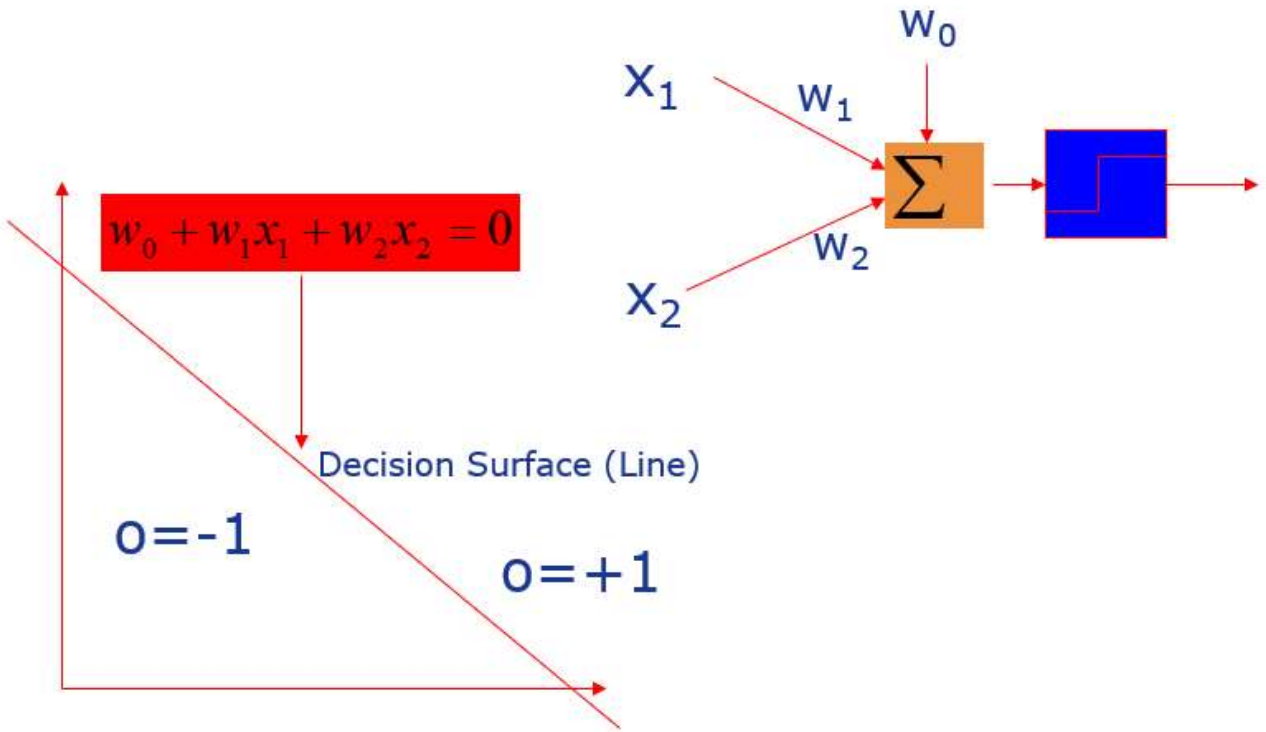
- Perceptron can be regarded as representing a hyperplane decision surface in the n-dimensional **feature space** of instances.

- The perceptron outputs a 1 for instances lying on one side of the hyperplane and a -1 for instances lying on the other side.

- This hyperplane is called the **Decision Surface**

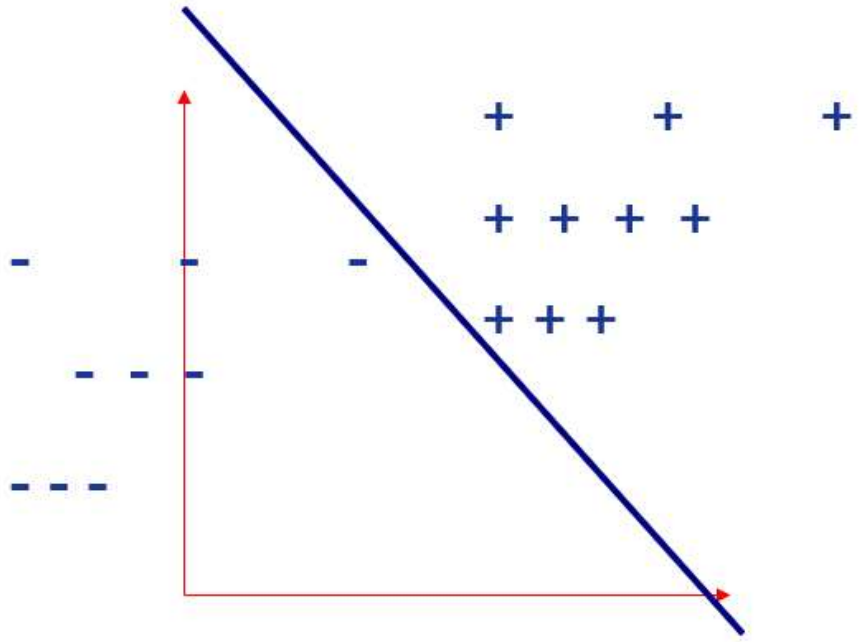
Perceptron – Decision Surface

- In 2-dimensional space



Perceptron – Representation Power

- The Decision Surface is linear
- Perceptron can only solve **Linearly Separable Problems**

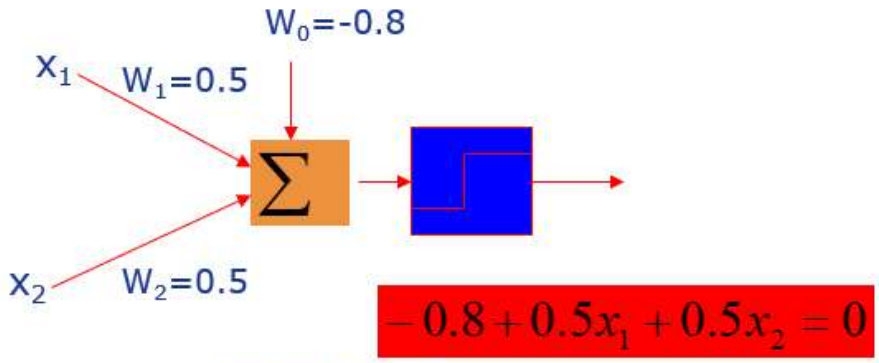


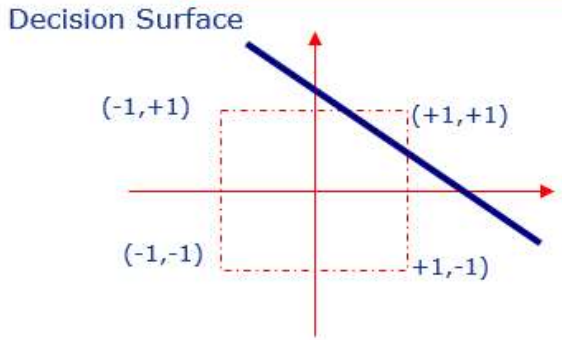
Perceptron – Representation Power

- Can represent many boolean functions: Assume boolean values of 1 (true) and -1 (false)

AND

x1	x2	D
-1	-1	-1
-1	+1	-1
+1	-1	-1
+1	+1	+1



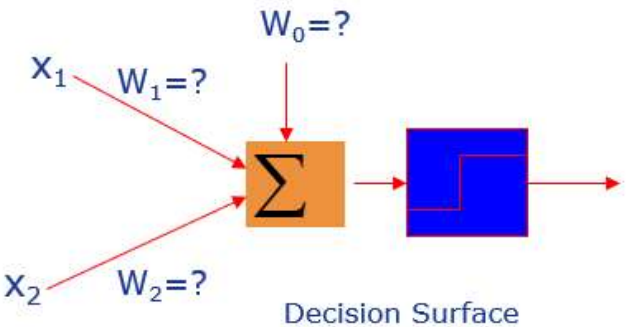


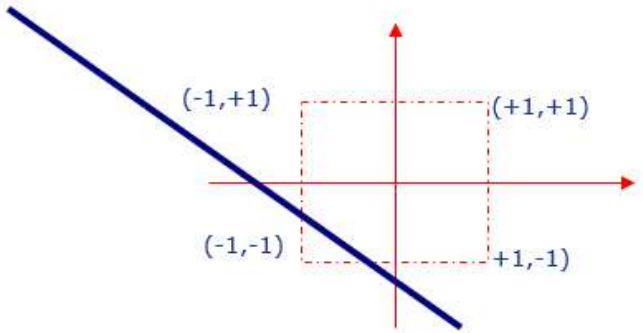
Perceptron – Representation Power

- Can represent many boolean functions: Assume boolean values of 1 (true) and -1 (false)

OR

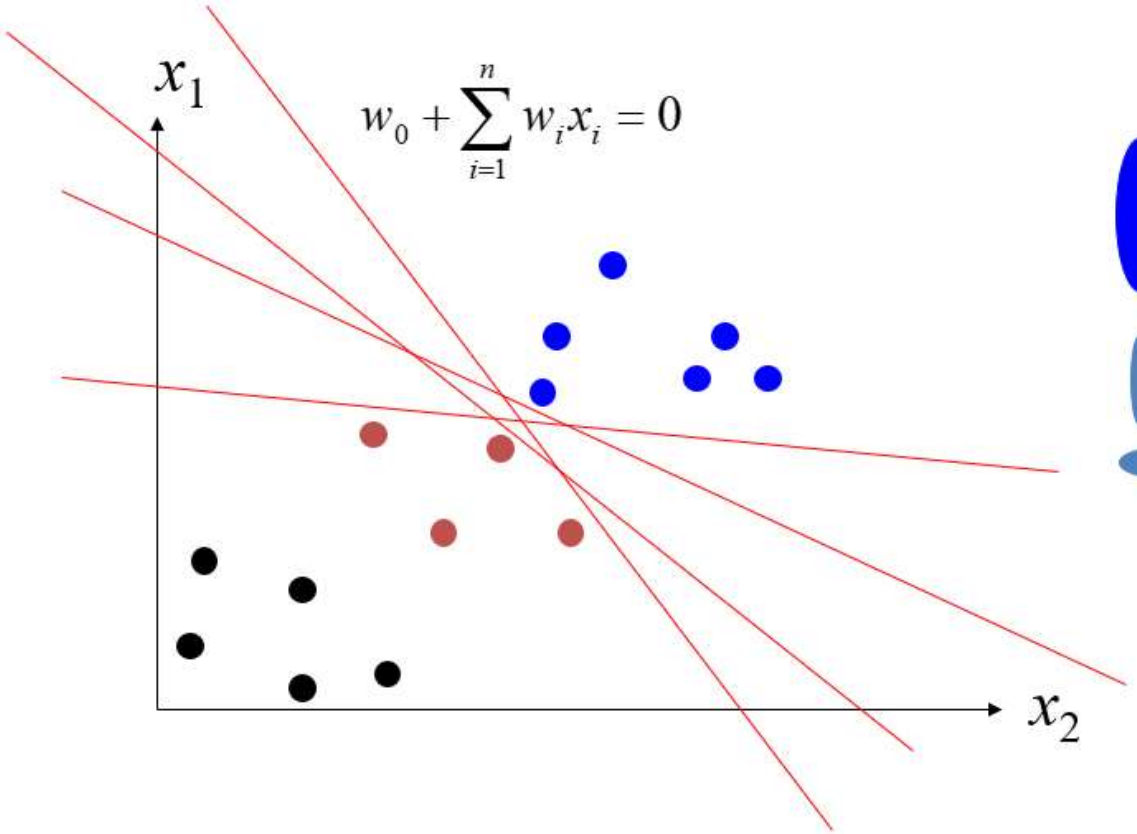
x1	x2	D
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	+1

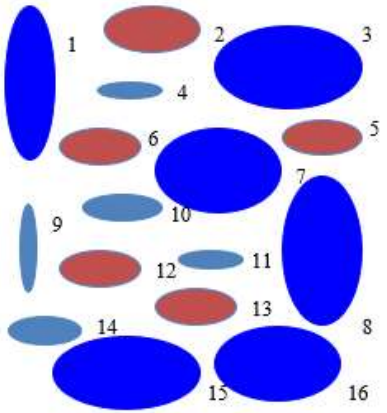




Perceptron – Representation Power

- Separate the  objects from the rest





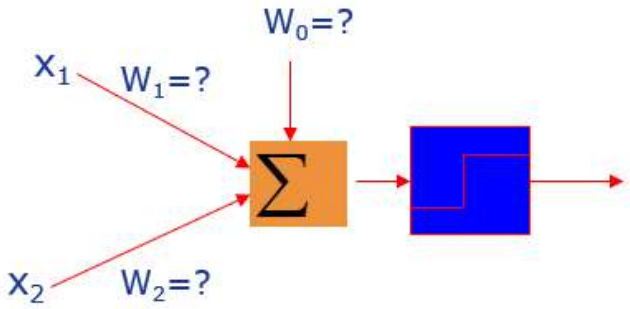
Elliptical blobs (objects)

Perceptron – Representation Power

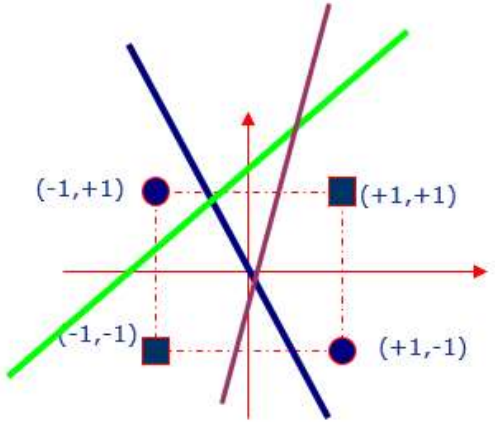
- Some problems are linearly non-separable

XOR

x1	x2	D
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	-1



Decision Surface:
It doesn't matter where you place the line (decision surface), it is impossible to separate the space such that on one side we have D = 1 and on the other we have D = -1



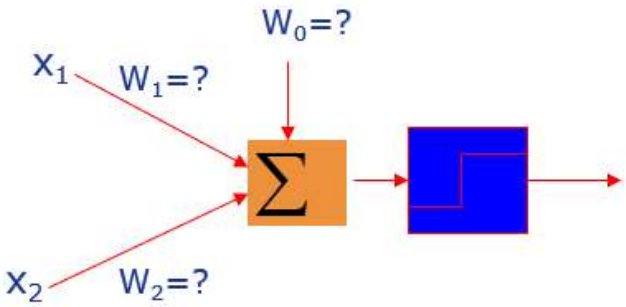
Perceptron Cannot Solve such Problem!

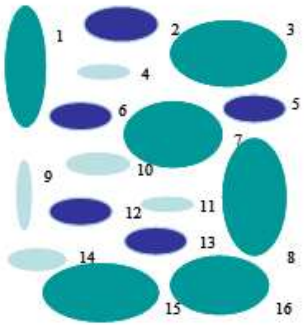
LIMITATION OF PERCEPTRONS

- Not every set of inputs can be divided by a line like this. Those that can be are called *linearly separable*. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly.

Perceptron – Training Algorithm

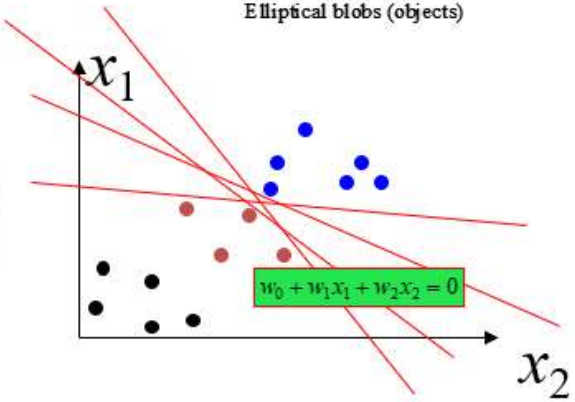
- Separate the  objects from the rest





Elliptical blobs (objects)

We are given the training sample (experience) pairs (X, D), how can we determine the weights that will produce the correct +1 and -1 outputs for the given training samples?



Perceptron – Training Algorithm

- Training sample pairs (X, d), where X is the input vector, d is the input vector's classification (+1 or -1) is iteratively presented to the network for training, *one at a time*, until the process converges

Perceptron – Training Algorithm

- The Procedure is as follows

1. Set the weights to small random values, e.g., in the range (-1, 1)

2. Present X, and calculate

$$R = w_0 + \sum_{i=1}^{n} w_i x_i \quad o = sign\ (R) = \begin{cases} +1; & \text{if } R > 0 \\ -1, & \text{otherwise} \end{cases}$$

3. Update the weights

$$w_i \leftarrow w_i + \eta (d - o) x_i, i = 1, 2, \cdots, n$$

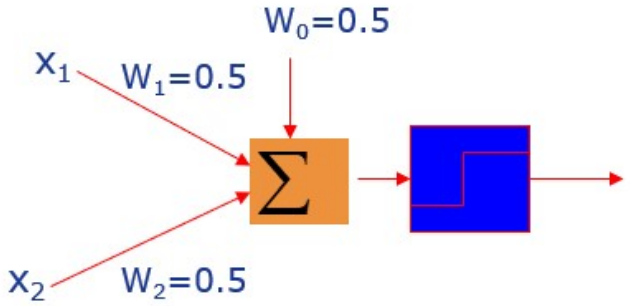
$$0 < \eta < 1 \quad \textit{is the training rate}$$

4. Repeat by going to step 2

Perceptron – Training Algorithm

Example

x1	x2	D
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	+1



$$w_i \leftarrow w_i + \eta (d - o) x_i, i = 1, 2, \cdots, n$$

Training artificial neural network

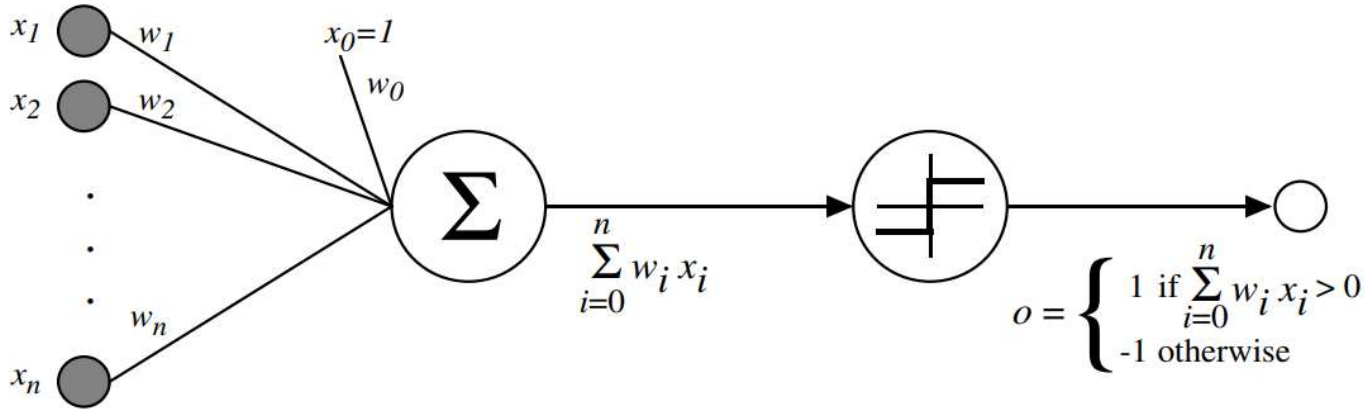
There are three general paradigms of learning:

"The teacher" - neural network has the correct answer (output network) for each input sample. Weights are adjusted so that the network produced a response as possible close to the known correct answers.

"Without a teacher" (self) - requires knowledge of correct answers for each sample training set. In this case reveals the internal structure of the data and the correlation between samples in the training set that allows you to distribute samples by category.

mixed - part weights determined by means learning from the teacher, while the other is determined by means of self-study. LOGO

Perceptron training



$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Weights Adjusting

- After each iteration, weights should be adjusted to minimize the error.
    - All possible weights

Perceptron – Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

Perceptron Learning algorithm

While epoch produces an error

Present network with next inputs from epoch

Error = T – O

If Error <> 0 then

$W_j = W_j + \eta * x_j * \text{Error}$

End If

End While

Error Estimation

- The **root mean square error (RMSE)** is a frequently-used measure of the differences between values predicted by a model or an estimator and the values actually observed from the thing being modeled or estimated

Perceptron – Training Rule

Perceptron training rule guaranteed to succeed if

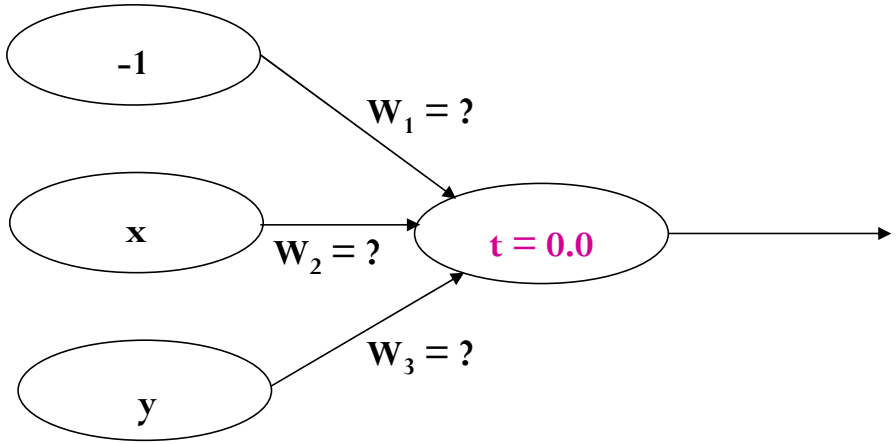
- If training data is linearly separable
- and η is sufficiently small

Perceptron – Training Algorithm

- Convergence Theorem

  - The perceptron training rule will converge (finding a weight vector correctly classifies all training samples) within a finite number of iterations, **provided the training examples are linearly separable** and provided a sufficiently small *h* is used.

Training Perceptrons

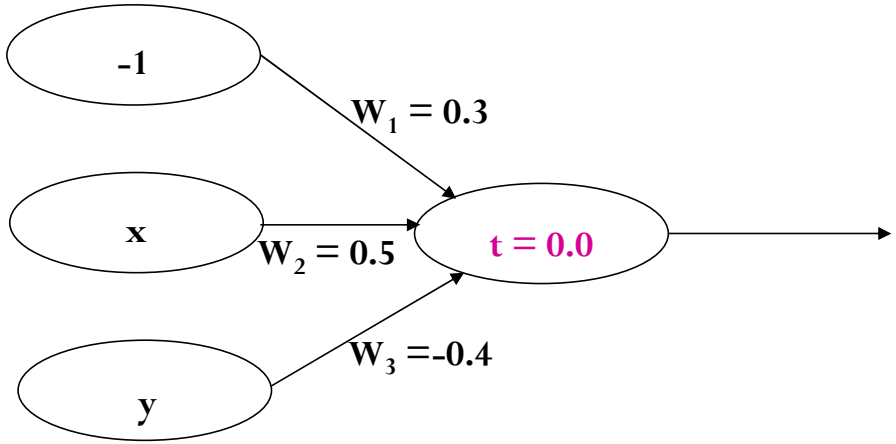


For AND

A B	Output
0 0	0
0 1	0
1 0	0
1 1	1

- What are the weight values?
- Initialize with random weight values

Training Perceptrons



For AND

A B	Output
0 0	0
0 1	0
1 0	0
1 1	1

I_1	I_2	I_3	Summation	Output
-1	0	0	(-1*0.3) + (0*0.5) + (0*-0.4) = -0.3	0
-1	0	1	(-1*0.3) + (0*0.5) + (1*-0.4) = -0.7	0
-1	1	0	(-1*0.3) + (1*0.5) + (0*-0.4) = 0.2	1
-1	1	1	(-1*0.3) + (1*0.5) + (1*-0.4) = -0.2	0

Learning in Neural Networks

- Learn values of weights from I/O pairs
- Start with random weights
- Load training example's input
- Observe computed input
- Modify weights to reduce difference
- Iterate over all training examples
- Terminate when weights stop changing OR when error is very small

Gradient Descent algorithm and its variants

Gradient Descent is an optimization algorithm used for minimizing the cost function in various machine learning algorithms. It is basically used for updating the parameters of the learning model.

Gradient descent algorithm is an iterative process that takes us to the minimum of a function

Types of gradient Descent:

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini Batch gradient descent

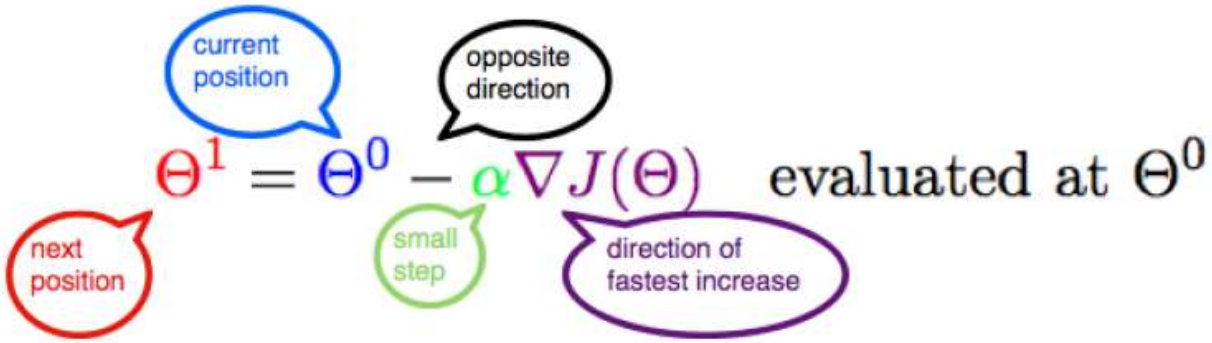
Gradient Descent

Linear unit training rule uses gradient descent

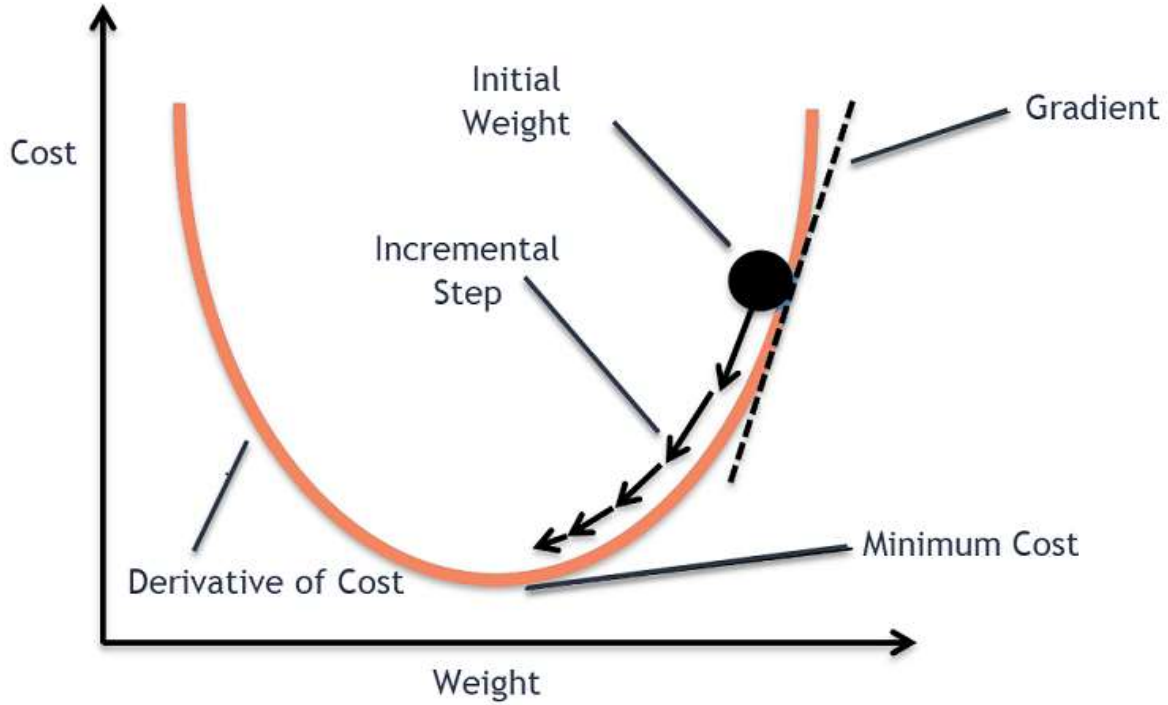
- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Gradient Descent

The formula below sums up the entire Gradient Descent algorithm in a single line.



To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called **Gradient Ascent.**



Gradient Descent

To understand, consider simpler *linear unit*, where

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Batch Gradient Descent

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental(Stochastic) Gradient Descent

Incremental mode Gradient Descent:
Do until satisfied

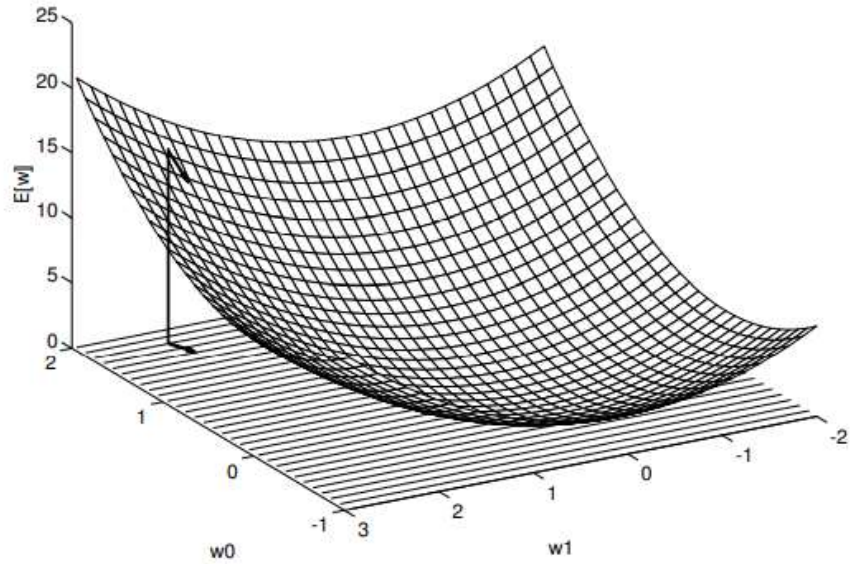
- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Gradient Descent



Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

Gradient Descent

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

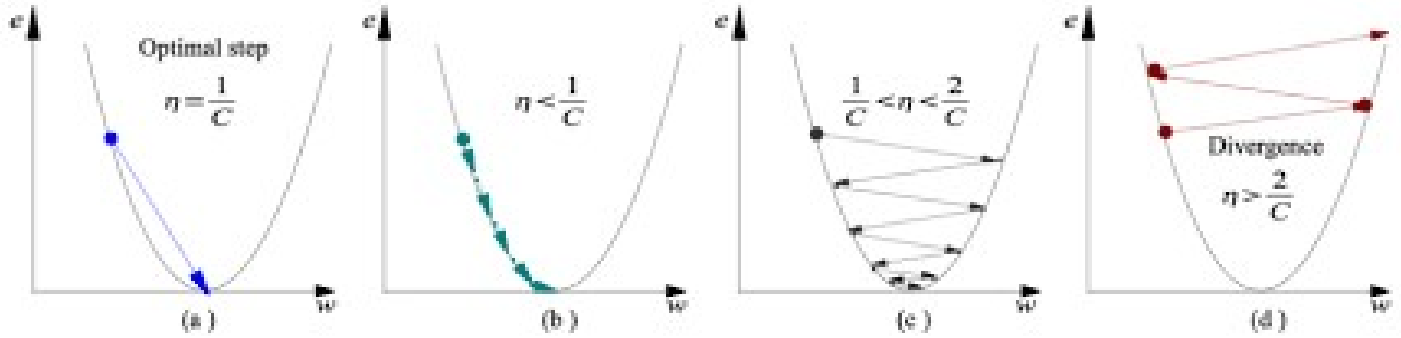
  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

The Learning Rate

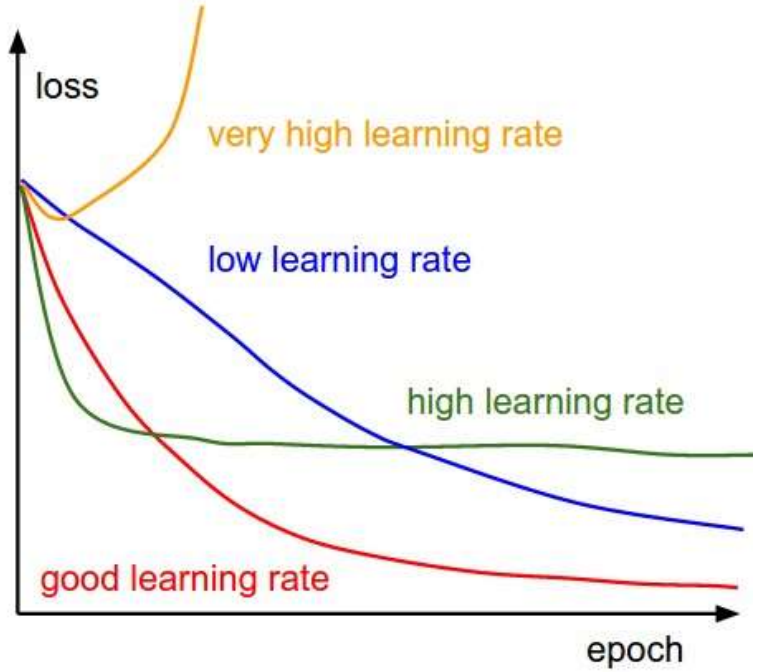
It must be chosen carefully to end up with local minima

- If the learning rate is too high, we might **OVERSHOOT** the minima and keep bouncing, without reaching the minima
- If the learning rate is too small, the training might turn out to be too long



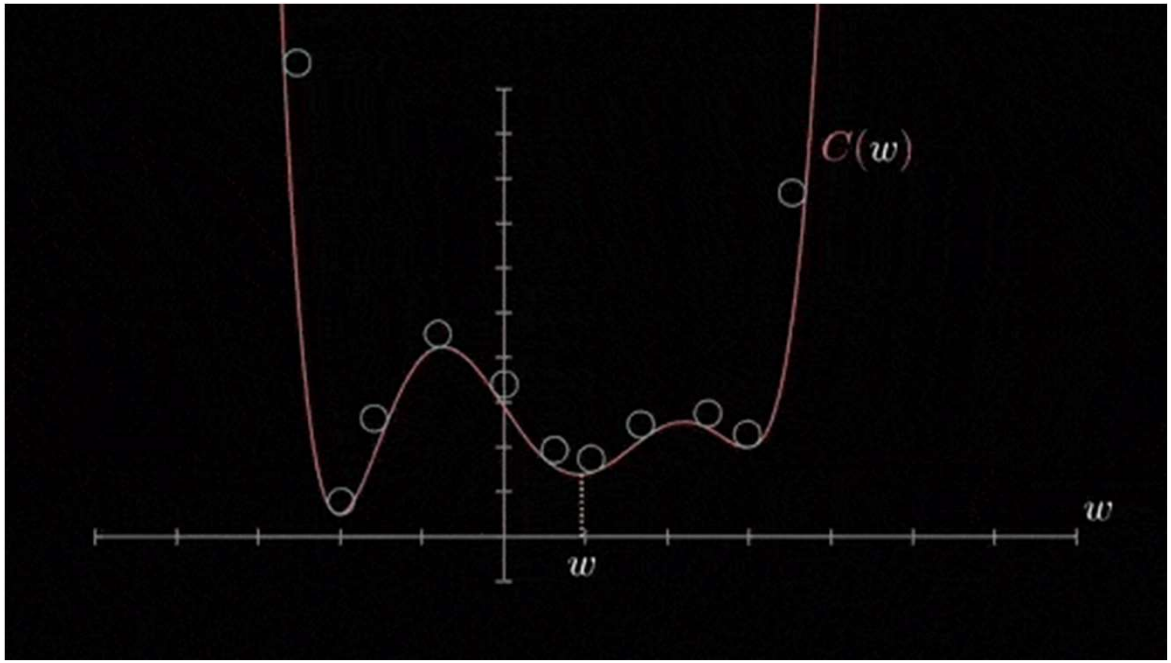
The Learning Rate

1.a) Learning rate is optimal, model converges to the minimum
2.b) Learning rate is too small, it takes more time but converges to the minimum
3.c) Learning rate is higher than the optimal value, it overshoots but converges
4.d) Learning rate is very large, it overshoots and diverges, moves away from the minima, performance decreases on learning



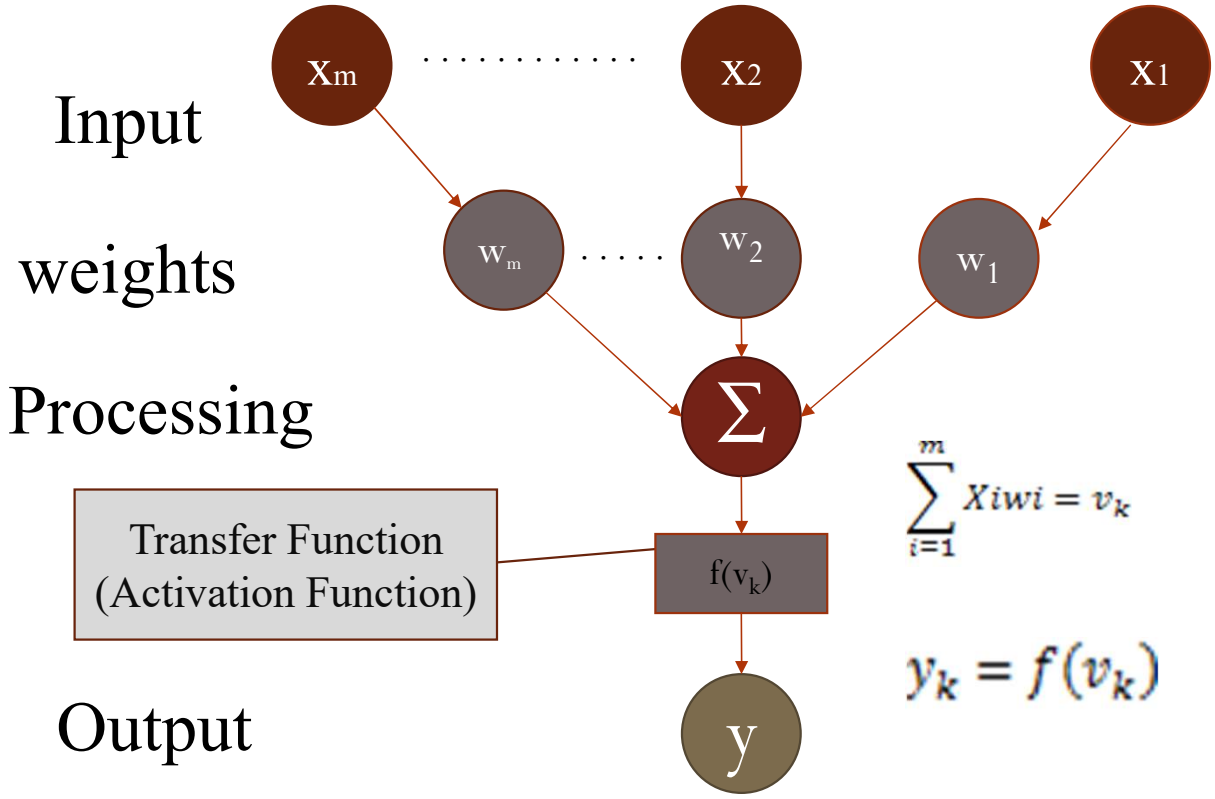
Local Minima

The cost function may consist of many minimum points. The gradient may settle on any one of the minima, which depends on the initial point (i.e initial parameters(theta)) and the learning rate. Therefore, the optimization may converge to different points with different starting points and learning rate.

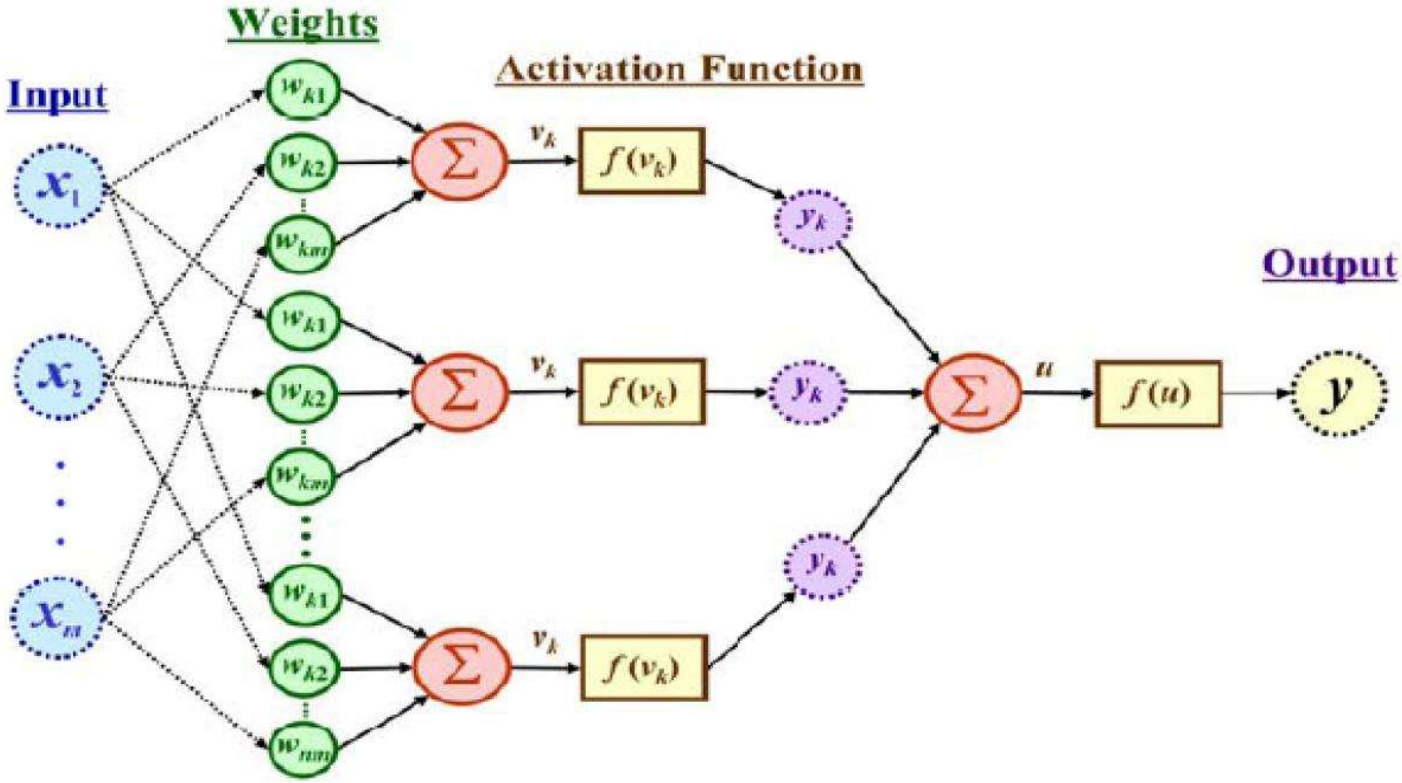


How do ANNs work?

The signal is not passed down to the next neuron verbatim



The output is a function of the input, that is affected by the weights, and the transfer functions



ACTIVATION FUNCTION

- A function that transforms the values or states the conditions for the decision of the output neuron is known as an **activation function**.
- What does an artificial neuron do? Simply, it calculates a “weighted sum” of its input, adds a bias and then decides whether it should be “fired” or not.
- So consider a neuron.

$$Y = \sum (weight * input) + bias$$

ACTIVATION FUNCTION

- The value of Y can be anything ranging from -inf to +inf. The neuron really doesn't know the bounds of the value. So how do we decide whether the neuron should fire or not (why this firing pattern? Because we learnt it from biology that's the way brain works and brain is a working testimony of an awesome and intelligent system).
- We decided to add "activation functions" for this purpose. To check the Y value produced by a neuron and decide whether outside connections should consider this neuron as "fired" or not. Or rather let's say — "activated" or not.

ACTIVATION FUNCTION

- If we do not apply an Activation function, then the output signal would simply be a simple ***linear function***. A *linear function* is just a polynomial of **one degree.**
- A linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data.
- A Neural Network without Activation function would simply be a **Linear Regression Model,** which has limited power and does not performs good most of the times.
- We want our Neural Network to not just learn and compute a linear function but something more complicated than that.
- Also, without activation function our Neural network would not be able to learn and model other complicated kinds of data such as images, videos , audio , speech etc. That is why we use Artificial Neural network techniques such as **Deep learning to make sense of something complicated ,high dimensional, non-linear -big datasets, where the model has lots and lots of hidden layers in between and has a very complicated architecture which helps us to make sense and extract knowledge form such complicated big datasets.**

ACTIVATION FUNCTION

- ***Activation*** *functions* are really important for a Artificial Neural Network to learn and make sense of something really complicated and non-linear complex functional mappings between the inputs and response variable. They introduce non-linear properties to our network.
- **Their main purpose is to convert an input signal of a node in a A-NN to an output signal.** That output signal now is used as a input in the next layer in the stack.

WHY DO WE NEED NON-LINEARITIES?

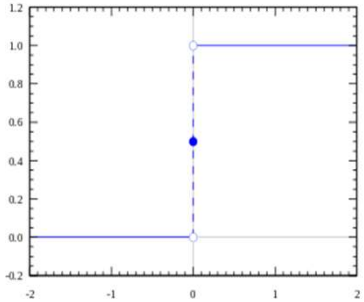
- Non-linear functions are those which have degree more than one and they have a curvature when we plot a Non-Linear function. Now we need a Neural Network Model to learn and represent almost anything and any arbitrary complex function which maps inputs to outputs. Neural-Networks are considered ***Universal Function Approximators***. It means that they can compute and learn any function at all. Almost any process we can think of can be represented as a functional computation in Neural Networks.

- Hence it all comes down to this, we need to apply an Activation function f(x) so as to make the network more powerful and add ability to it to learn something complex and complicated form data and represent non-linear complex arbitrary functional mappings between inputs and outputs. Hence using a non linear Activation, we are able to generate non-linear mappings from inputs to outputs.

TYPES OF ACTIVATION FUNCTIONS*

Step function

- Activation function A = “activated” if Y > threshold else not
- Alternatively, A = 1 if Y > threshold, 0 otherwise
- Well, what we just did is a “step function”, see the below figure.



- **DRAWBACK:** Suppose you are creating a binary classifier. Something which should say a “yes” or “no” (activate or not activate). A Step function could do that for you! That’s exactly what it does, say a 1 or 0. Now, think about the use case where you would want multiple such neurons to be connected to bring in more classes. Class1, class2, class3 etc. What will happen if more than 1 neuron is “activated”. All neurons will output a 1 (from step function). Now what would you decide? Which class is it? Hard, complicated.

* https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f

TYPES OF ACTIVATION FUNCTIONS

Linear function

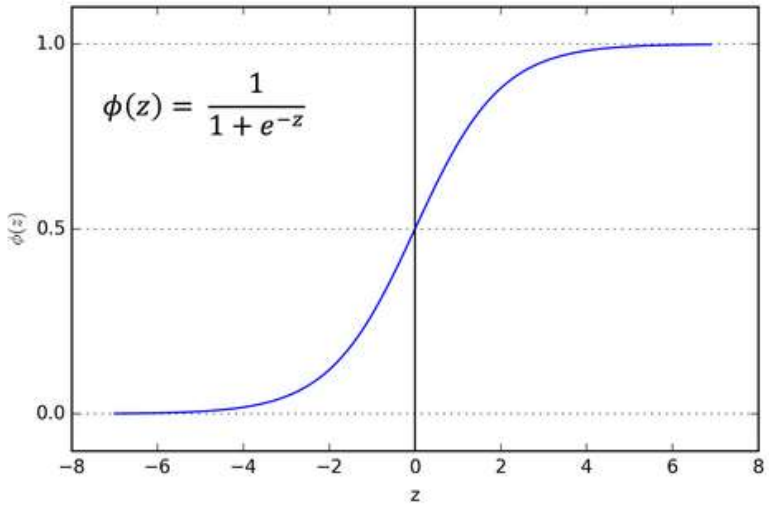
- A = cx
- A straight line function where activation is proportional to input (which is the weighted sum from neuron).
- This way, it gives a range of activations, so it is not binary activation. We can definitely connect a few neurons together and if more than 1 fires, we could take the max and decide based on that. So that is ok too. Then what is the problem with this?
- A = cx, derivative with respect to x is c. That means, the gradient has no relationship with X. It is a constant gradient and the descent is going to be on constant gradient. If there is an error in prediction, the changes made by back propagation is constant and not depending on the change in input.

TYPES OF ACTIVATION FUNCTIONS

Sigmoid function

$$A = \frac{1}{1+e^{-x}}$$

This looks smooth and “step function like”. What are the benefits of this? It is nonlinear in nature. Combinations of this function are also nonlinear! Great. Now we can stack layers. What about non binary activations? Yes, that too! It will give an analog activation unlike step function. It has a smooth gradient too.



And if you notice, between X values -2 to 2, Y values are very steep. Which means, any small changes in the values of X in that region will cause values of Y to change significantly. That means this function has a tendency to bring the Y values to either end of the curve.

- Looks like it's good for a classifier considering its property? Yes ! It tends to bring the activations to either side of the curve (above x = 2 and below x = -2 for example). Making clear distinctions on prediction.
- Another advantage of this activation function is, unlike linear function, the output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in a range. It won't blow up the activations then. This is great.
- Sigmoid functions are one of the most widely used activation functions today. Then what are the problems with this?
- If you notice, towards either end of the sigmoid function, the Y values tend to respond very less to changes in X. What does that mean? The gradient at that region is going to be small. It gives rise to a problem of "vanishing gradients". So what happens when the activations reach near the "near-horizontal" part of the curve on either sides?
- Gradient is small or has vanished (cannot make significant change because of the extremely small value). The network refuses to learn further or is drastically slow. There are ways to work around this problem and sigmoid is still very popular in classification problems.

TYPES OF ACTIVATION FUNCTIONS

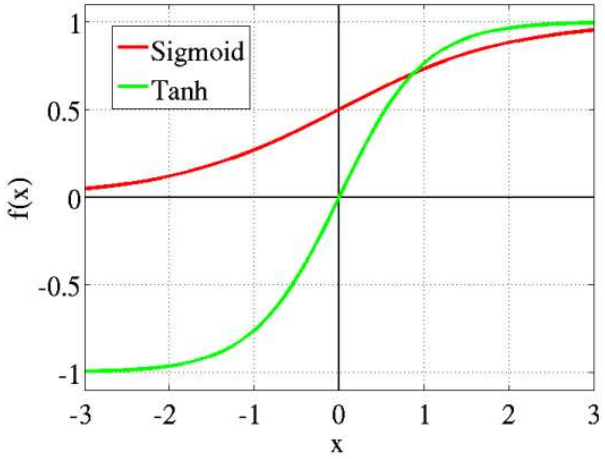
Tanh Function

- Another activation function that is used is the tanh function.

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

This looks very similar to sigmoid. In fact, it is a scaled sigmoid function!

$$tanh(x) = 2\,sigmoid(2x) - 1$$



- This has characteristics similar to sigmoid that we discussed above. It is nonlinear in nature, so great we can stack layers! It is bound to range (-1, 1) so no worries of activations blowing up. One point to mention is that the gradient is stronger for tanh than sigmoid (derivatives are steeper). Deciding between the sigmoid or tanh will depend on your requirement of gradient strength. Like sigmoid, tanh also has the vanishing gradient problem.
- Tanh is also a very popular and widely used activation function. Especially in time series data.

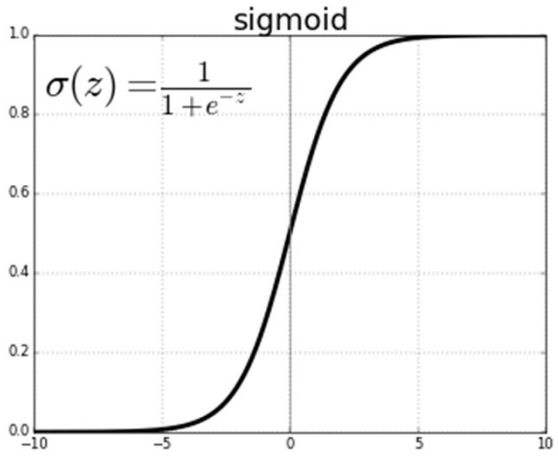
TYPES OF ACTIVATION FUNCTIONS

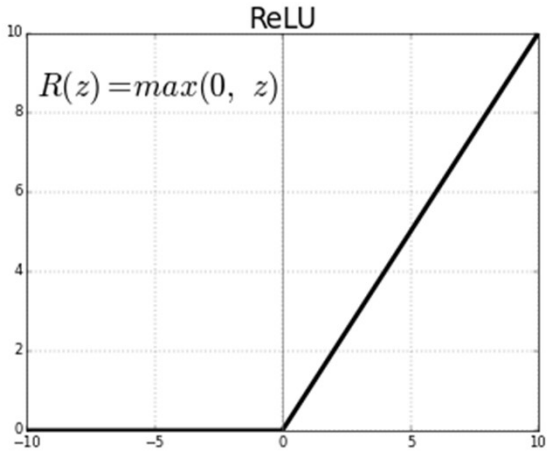
ReLu

- Later, comes the ReLu function,

$$A(x) = \max(0,x)$$

The ReLu function is as shown above. It gives an output x if x is positive and 0 otherwise.





- At first look, this would look like having the same problems of linear function, as it is linear in positive axis. First of all, ReLu is nonlinear in nature. And combinations of ReLu are also non linear! (in fact it is a good approximator. Any function can be approximated with combinations of ReLu). Great, so this means we can stack layers. It is not bound though. The range of ReLu is [0, inf). This means it can blow up the activation.
- Another point to discuss here is the sparsity of the activation. Imagine a big neural network with a lot of neurons. Using a sigmoid or tanh will cause almost all neurons to fire in an analog way. That means almost all activations will be processed to describe the output of a network. In other words, the activation is dense. This is costly. We would ideally want a few neurons in the network to not activate and thereby making the activations sparse and efficient.
- ReLu give us this benefit. Imagine a network with random initialized weights (or normalized) and almost 50% of the network yields 0 activation because of the characteristic of ReLu (output 0 for negative values of x). This means a fewer neurons are firing (sparse activation) and the network is lighter. ReLu seems to be awesome! Yes it is, but nothing is flawless.. Not even ReLu.

- Because of the horizontal line in ReLu (for negative X), the gradient can go towards 0. For activations in that region of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called **dying ReLu problem**. This problem can cause several neurons to just die and not respond making a substantial part of the network passive. There are variations in ReLu to mitigate this issue by simply making the horizontal line into non-horizontal component . For example, $y = 0.01x$ for $x<0$ will make it a slightly inclined line rather than horizontal line. This is leaky ReLu. There are other variations too. The main idea is to let the gradient be non zero and recover during training eventually.
- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. That is a good point to consider when we are designing deep neural nets.

NOW WHICH ONE DO WE USE?

- Does that mean we just use ReLu for everything we do? Or sigmoid or tanh? Well, yes and no.
- When you know the function you are trying to approximate has certain characteristics, you can choose an activation function which will approximate the function faster leading to faster training process. For example, a sigmoid works well for a classifier, because approximating a classifier function as combinations of sigmoid is easier than maybe ReLu, for example. Which will lead to faster training process and convergence. You can use your own custom functions too! If you don't know the nature of the function you are trying to learn, then maybe you can start with ReLu, and then work backwards. ReLu works most of the time as a general approximator!

Tree elements are particularly important in any model of artificial neural networks:

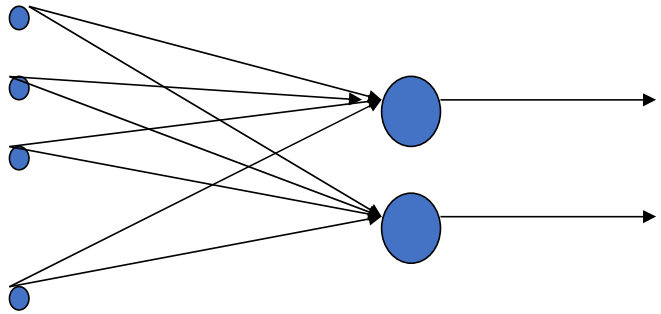
- the structure of the nodes,
- the topology of the network,
- the learning algorithm used to find the weights of the network

Network Topology

- Feedforward Network
  - **Single layer feedforward network**
  - **Multilayer feedforward network**
- Feedback Network
  - **Recurrent networks**
  - **Fully recurrent network**
  - **Jordan network**

Different Network Topologies

- Single layer feed-forward networks
  - Input layer projecting into the output layer



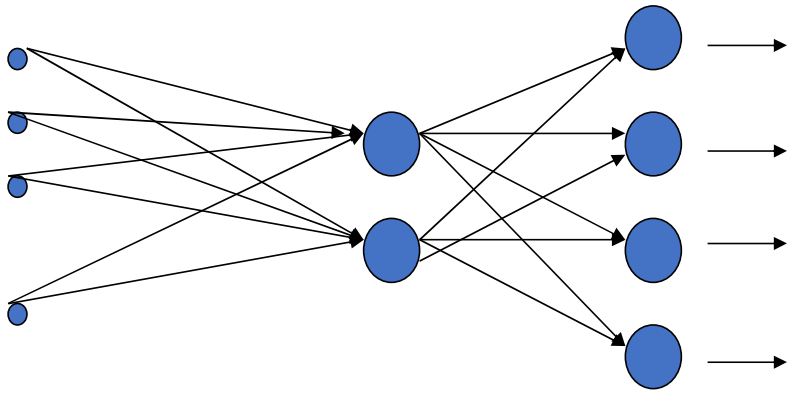
Single layer network

Input layer

Output layer

Different Network Topologies

- Multi-layer feed-forward networks
  - One or more hidden layers. Input projects only from previous layers onto a layer.



Input layer

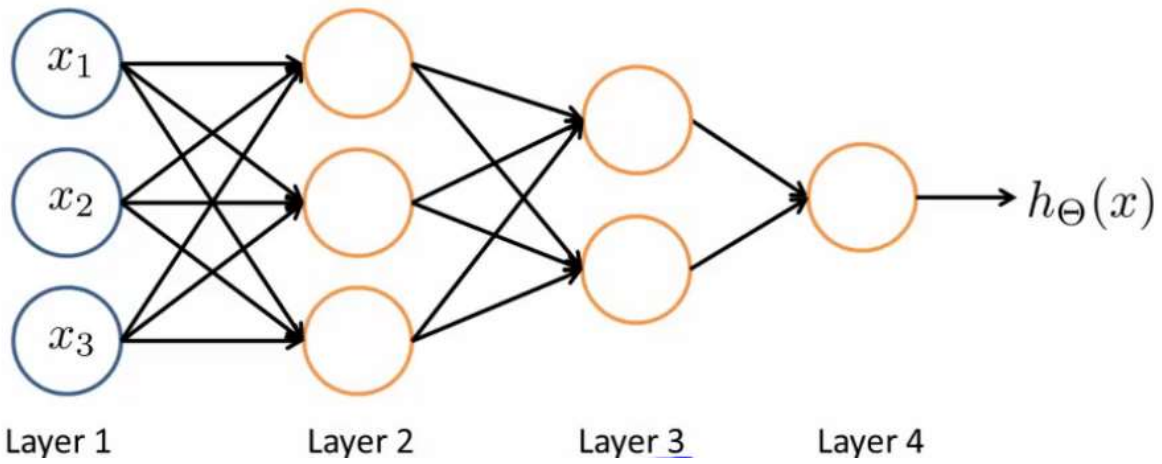
Hidden layer

Output layer

2-layer or
1-hidden layer
fully connected
network

Different Network Topologies

- Multi-layer feed-forward networks

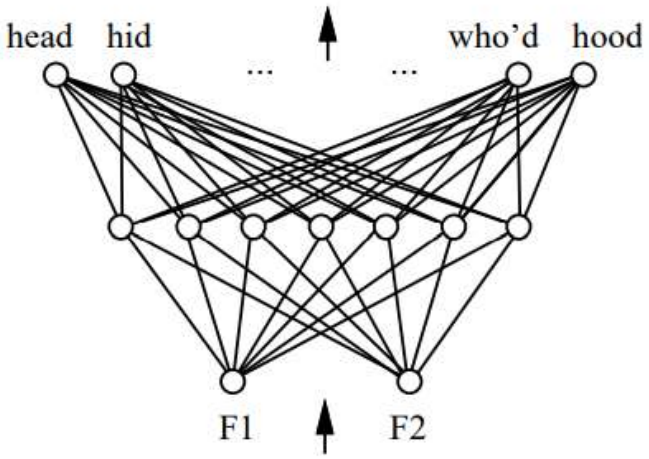


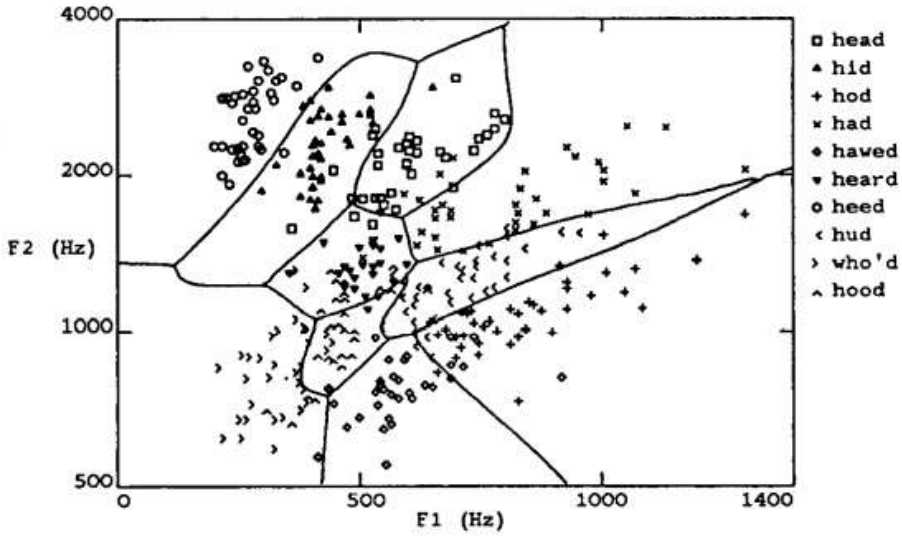
Input layer

Hidden layers

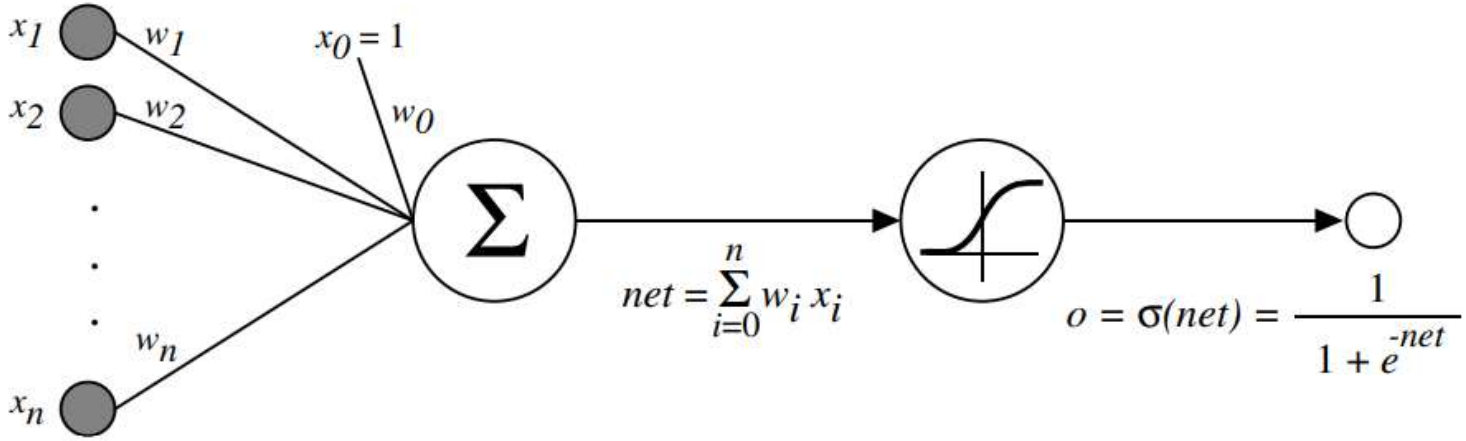
Output layer

Multi-layer networks of sigmoid networks





Multi-layer networks of sigmoid unit



$\sigma(x)$ is the sigmoid function / activation function

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

Multi-layer networks of sigmoid unit

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Error Gradient for a sigmoid unit

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\
&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}
$$

Error Gradient for a sigmoid unit

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D}(t_d - o_d)o_d(1 - o_d)x_{i,d}$$

Backpropagation Algorithm

- The backpropagation algorithm (Rumelhart and McClelland, 1986) is used in layered feed-forward Artificial Neural Networks.
- Back propagation is a multi-layer feed forward, supervised learning network based on gradient descent learning rule.
- we provide the algorithm with examples of the inputs and outputs we want the network to compute, and then the error (difference between actual and expected results) is calculated.
- The idea of the backpropagation algorithm is to reduce this error, until the Artificial Neural Network learns the training data.

Backpropagation Algorithm

- **Purpose:** To compute the weights of a feedforward multilayer neural network adaptatively, given a set of labeled training examples.
- **Method:** By minimizing the following cost function (the sum of square error)

where N is the total number of training examples and K, the total number of output units (useful for multiclass problems) and fk is the function implemented by the neural net

Backpropagation: Overview

- Backpropagation works by applying the ***gradient descent*** rule to a feedforward network.
- The algorithm is composed of two parts that get repeated over and over until a pre-set maximal number of ***epochs***, ***EPmax***.
- Part I, the ***feedforward*** pass: the activation values of the hidden and then output units are computed.
- Part II, the ***backpropagation*** pass: the weights of the network are updated--starting with the hidden to output weights and followed by the input to hidden weights--with respect to the sum of squares error and through a series of weight update rules called the ***Delta Rule***.

Backpropagation Algorithm

Initialize all weights to small random numbers.
Until satisfied, Do

- For each training example, Do
  1. Input the training example to the network and compute the network outputs
  2. For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

  3. For each hidden unit h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

  4. Update each network weight $w_{i,j}$

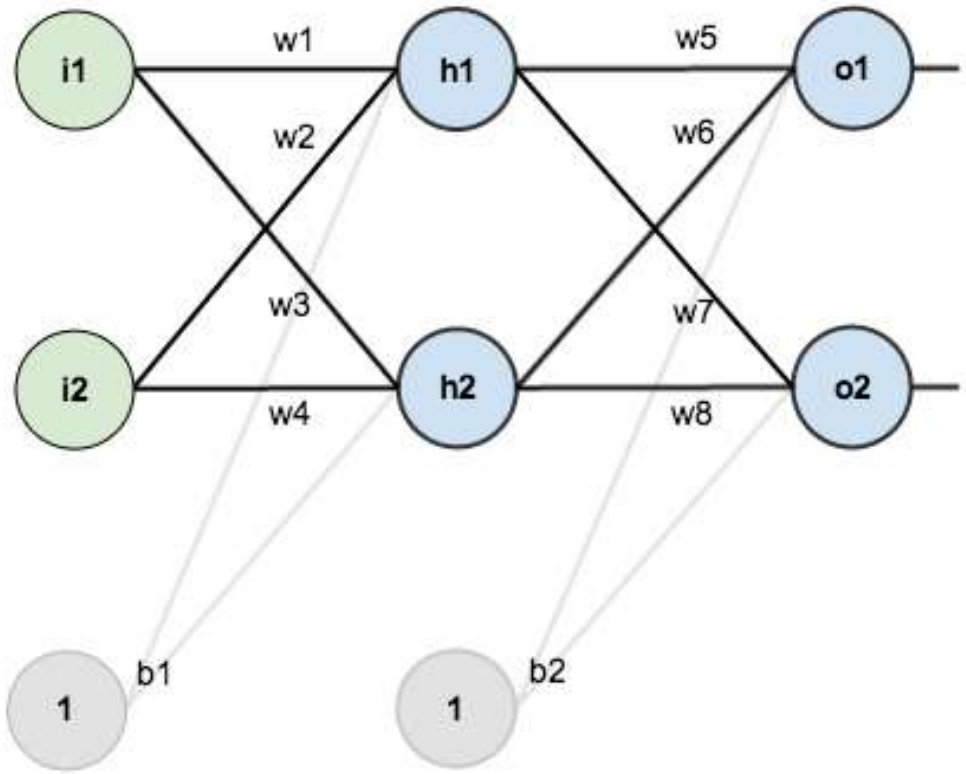
$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

  where

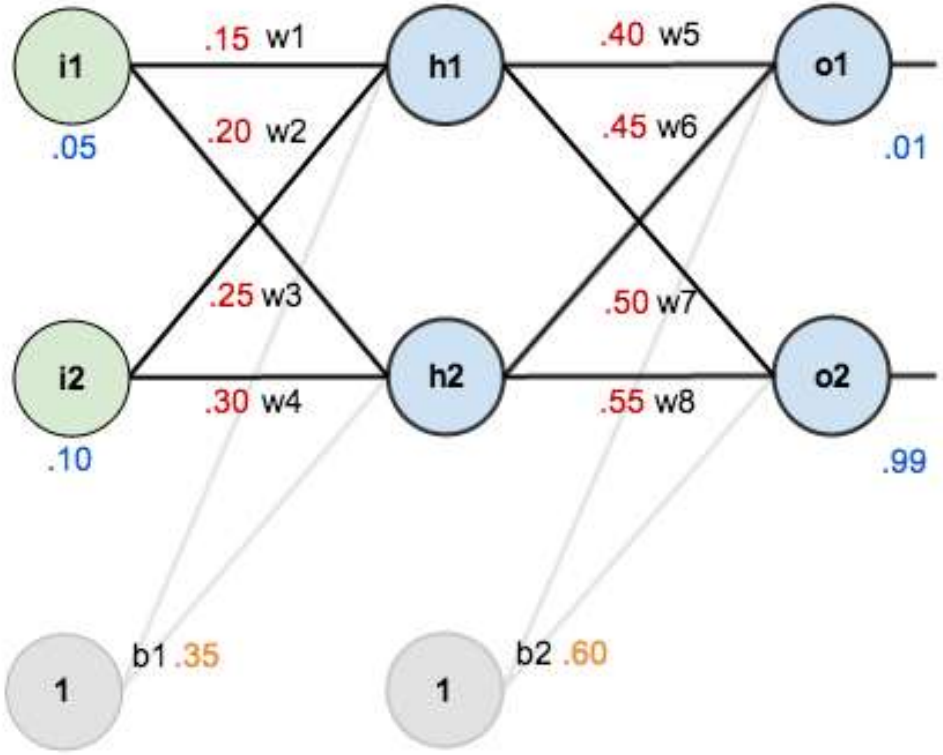
$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

Example of Backpropagation Algorithm

we’re going to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.
Here’s the basic structure:



In order to have some numbers to work with, here are the initial weights, the biases, and training inputs/outputs:



The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.
We're going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

The Forward Pass
To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we'll feed those inputs forward though the network.

We figure out the *total net input* to each hidden layer neuron, *squash* the total net input using an *activation function* (here we use the *logistic function*), then repeat the process with the output layer neurons.

Here's how we calculate the total net input for h_1:

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

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

We then squash it using the logistic function to get the output of h_1:

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

Carrying out the same process for h_2 we get:

$$out_{h2} = 0.596884378$$

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Here's the output for o_1:

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

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

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

And carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$

Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

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

For example, the target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is:

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

Repeating this process for o_2 (remembering that the target is 0.99) we get:

$$E_{o2} = 0.023560026$$

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

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

The Backwards Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

Output Layer

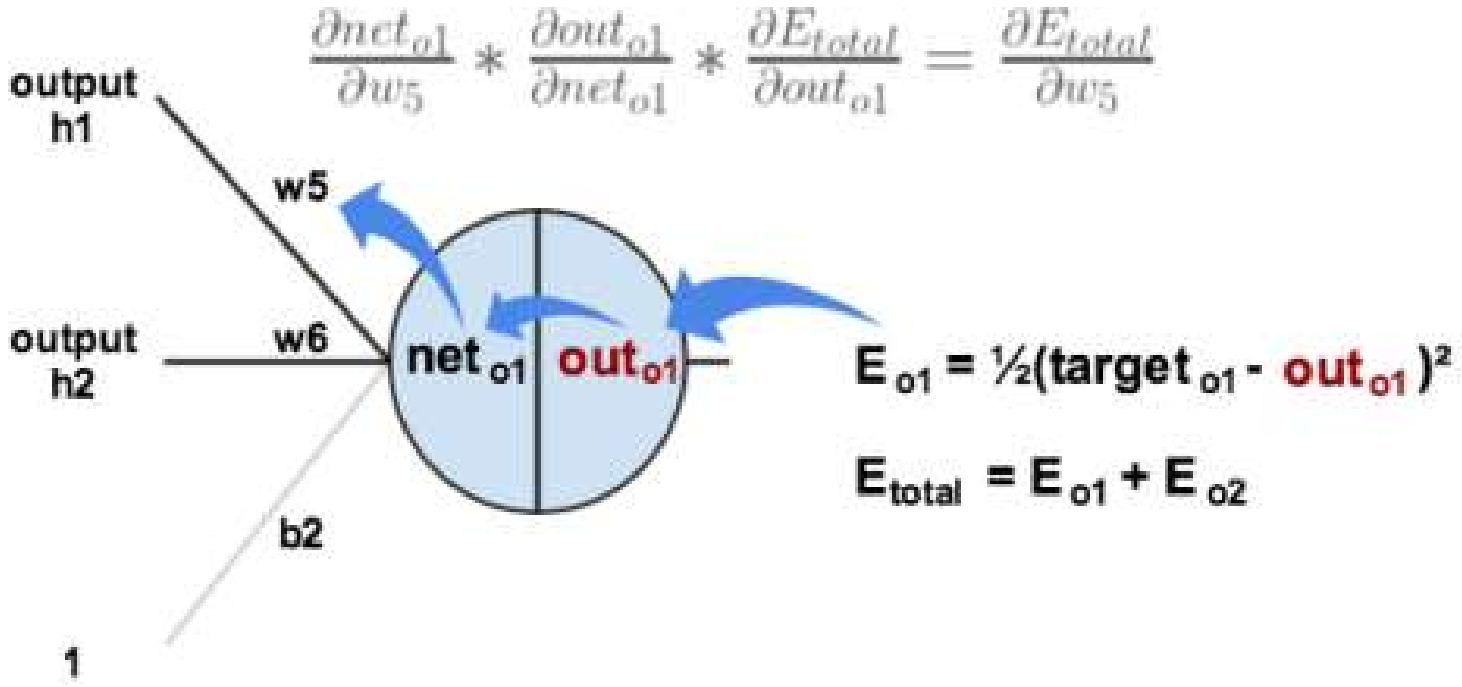
Consider w_5. We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$.

> $\frac{\partial E_{total}}{\partial w_5}$ is read as "the partial derivative of E_{total} with respect to w_5". You can also say "the gradient with respect to w_5".

By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

Visually, here's what we're doing:



We need to figure out each piece in this equation.

First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

> $-(target - out)$ is sometimes expressed as $out - target$

Next, how much does the output of o_1 change with respect to its total net input?

The partial derivative of the logistic function is the output multiplied by 1 minus the output:

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

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Finally, how much does the total net input of $_o1$ change with respect to w_5?

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

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

You'll often see this calculation combined in the form of the delta rule:

$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

Alternatively, we have $\frac{\partial E_{total}}{\partial out_{o1}}$ and $\frac{\partial out_{o1}}{\partial net_{o1}}$ which can be written as $\frac{\partial E_{total}}{\partial net_{o1}}$, aka δ_{o1} (the Greek letter delta) aka the *node delta*. We can use this to rewrite the calculation above:

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

Therefore:

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

Some sources extract the negative sign from δ so it would be written as:

$$\frac{\partial E_{total}}{\partial w_5} = -\delta_{o1} out_{h1}$$

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

We can repeat this process to get the new weights w_6, w_7, and w_8:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

We perform the actual updates in the neural network *after* we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

Hidden Layer

Next, we'll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.

Big picture, here's what we need to figure out:

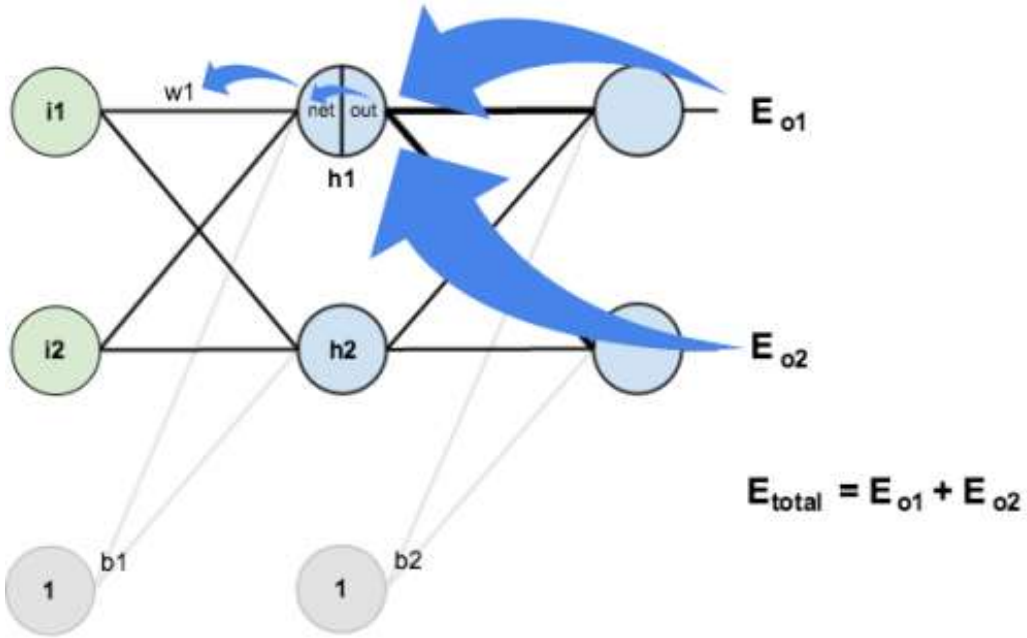
$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

Visually:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h1} affects both out_{o1} and out_{o2} therefore the $\frac{\partial E_{total}}{\partial out_{h1}}$ needs to take into consideration its effect on the both output neurons:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

We can calculate $\frac{\partial E_{o1}}{\partial net_{o1}}$ using values we calculated earlier:

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

And $\frac{\partial net_{o1}}{\partial out_{h1}}$ is equal to w_5:

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

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

Plugging them in:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

Following the same process for $\frac{\partial E_{o2}}{\partial out_{h1}}$, we get:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

Therefore:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

Now that we have $\frac{\partial E_{total}}{\partial out_{h1}}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then $\frac{\partial net_{h1}}{\partial w}$ for each weight:

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

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

We calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron:

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

You might also see this written as:

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \delta_o * w_{ho}) * out_{h1}(1 - out_{h1}) * i_1$$

$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} i_1$$

We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Repeating this for w_2, w_3, and w_4

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Finally, we've updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Backpropagation Algorithm

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

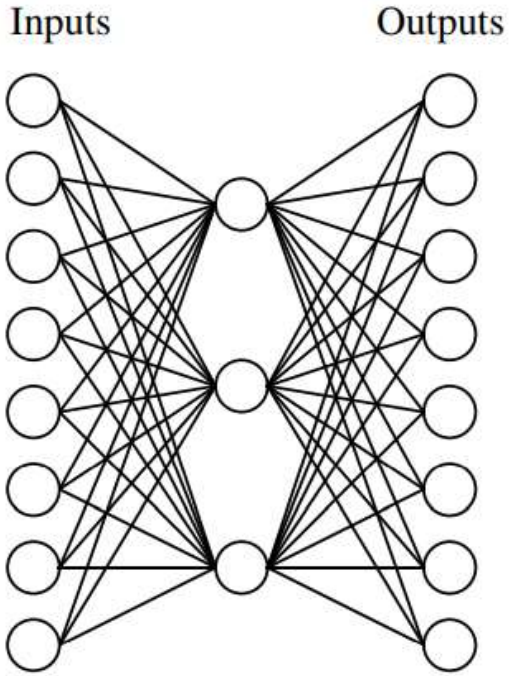
$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

Backpropagation Algorithm

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → slow!
- Using network after training is very fast

Hidden Layer Representations

One intriguing property of BACKPROPAGATION is its ability to discover useful intermediate representations at the hidden unit layers inside the network

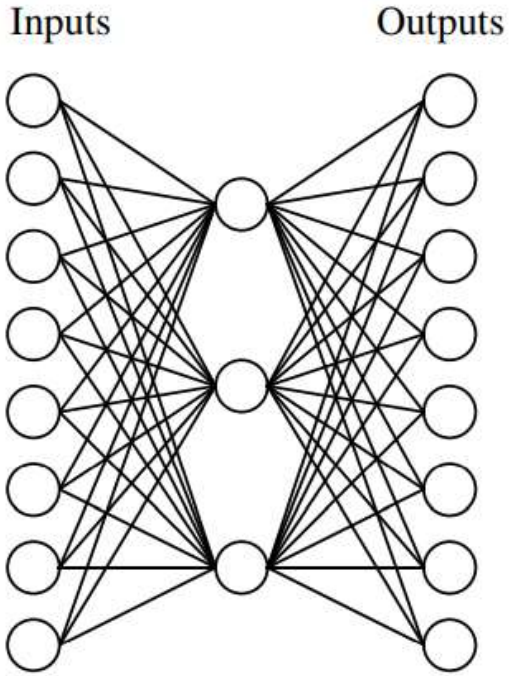


A target function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned??

Here, the eight network inputs are connected to three hidden units, which are in turn connected to the eight output units. Because of this structure, the three hidden units will be forced to re-represent the eight input values in some way that captures their relevant features, so that this hidden layer representation can be used by the output units to compute the correct target values.

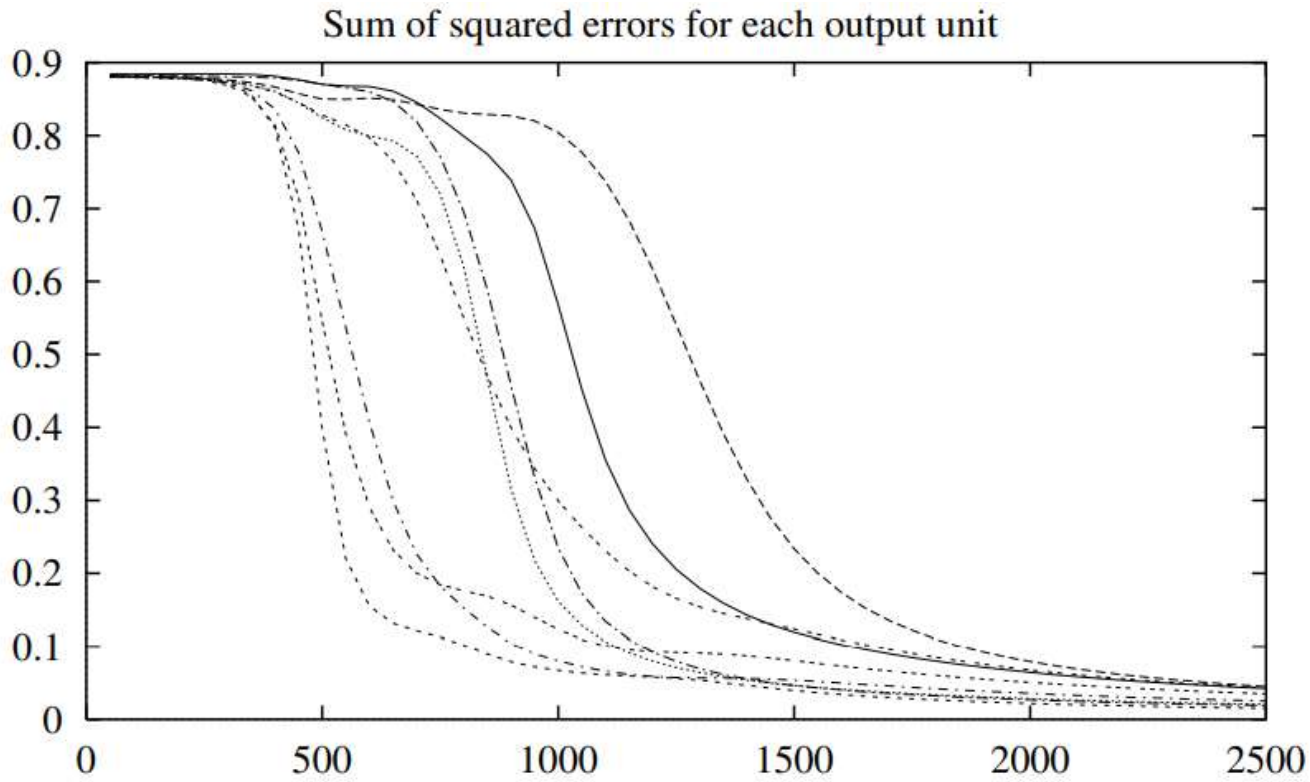


Learned hidden layer representation:

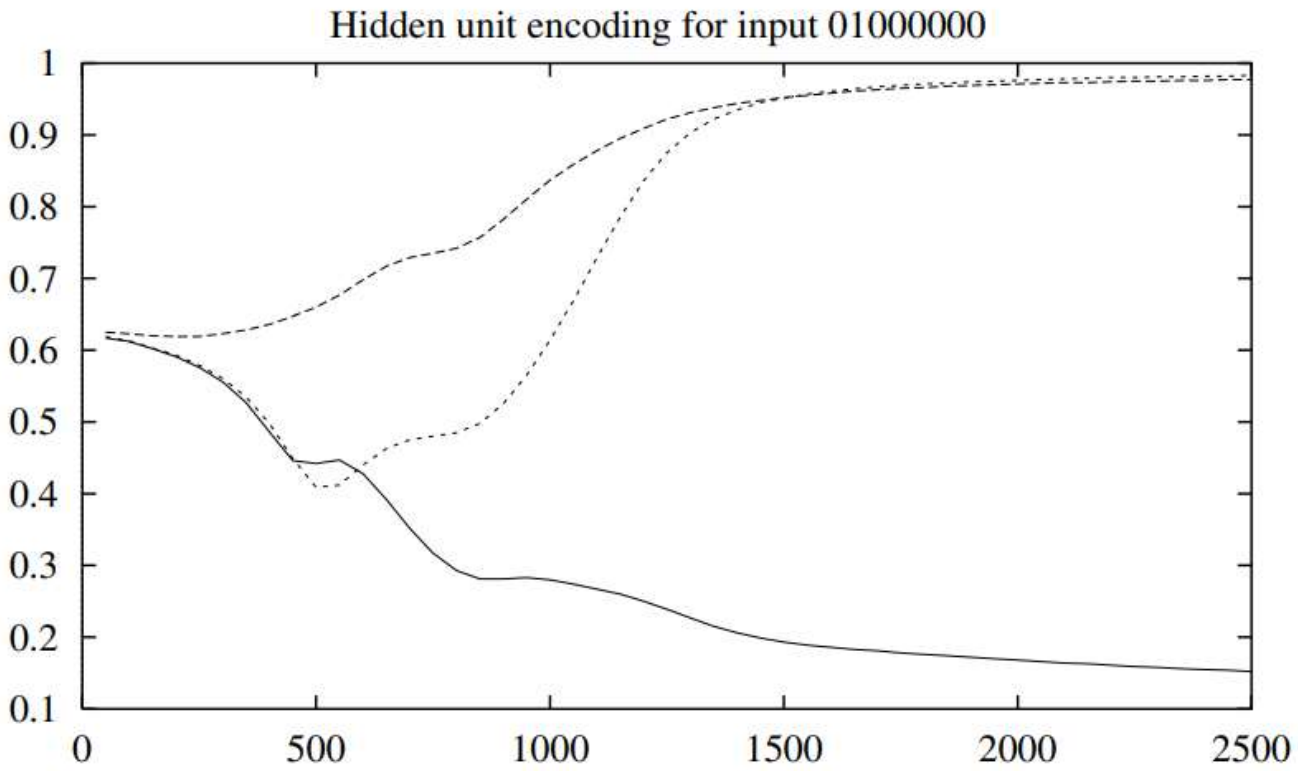
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning.

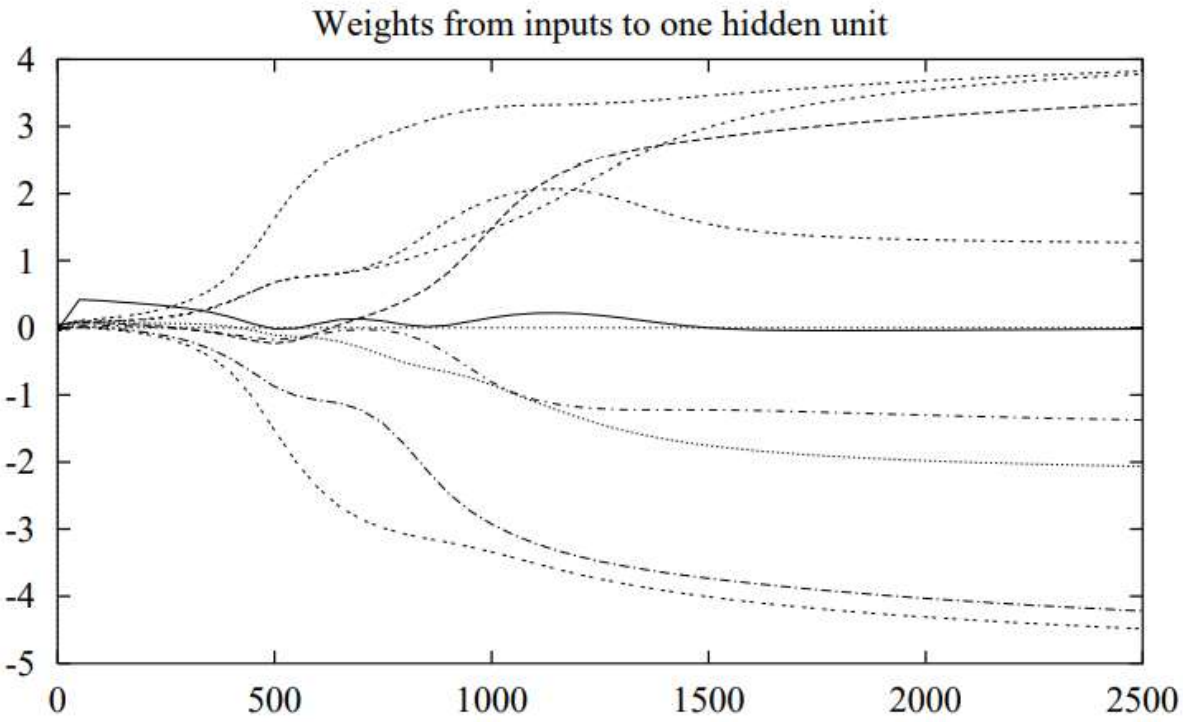
We can directly observe the effect of BACKPROPAGATION'S gradient descent search by plotting the squared output error as a function of the number of gradient descent search steps



Learning the 8 x 3 x 8 Network. This plot shows the evolving sum of squared errors for each of the eight output units, as the number of training iterations (epochs) increases.



Learning the 8 x 3 x 8 Network. This plot shows the evolving hidden layer representation for the input string "01000000



Learning the 8 x 3 x 8 Network. This bottom plot shows the evolving weights for one of the three hidden units.

Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

Convergence of Backpropagation

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Expressive Capabilities of ANNs

Boolean functions:

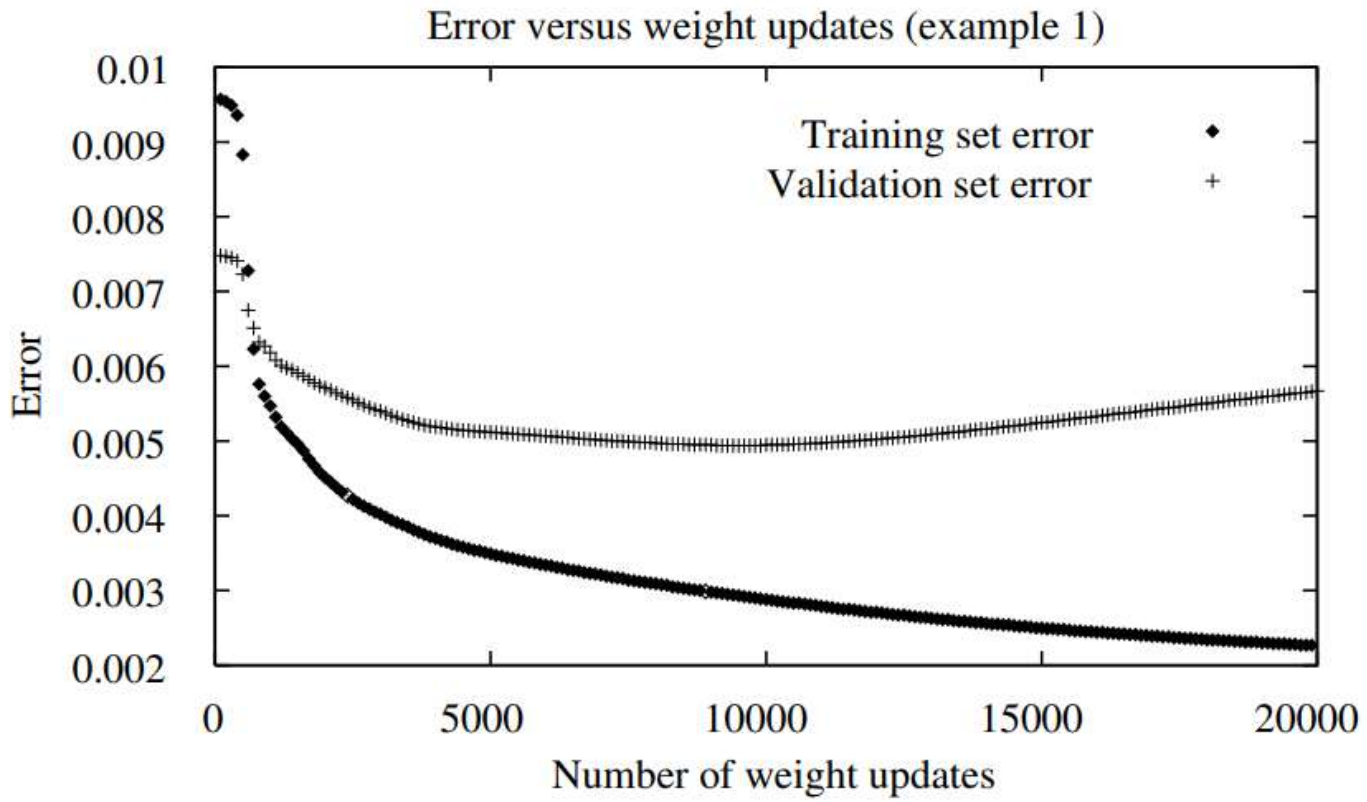
- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

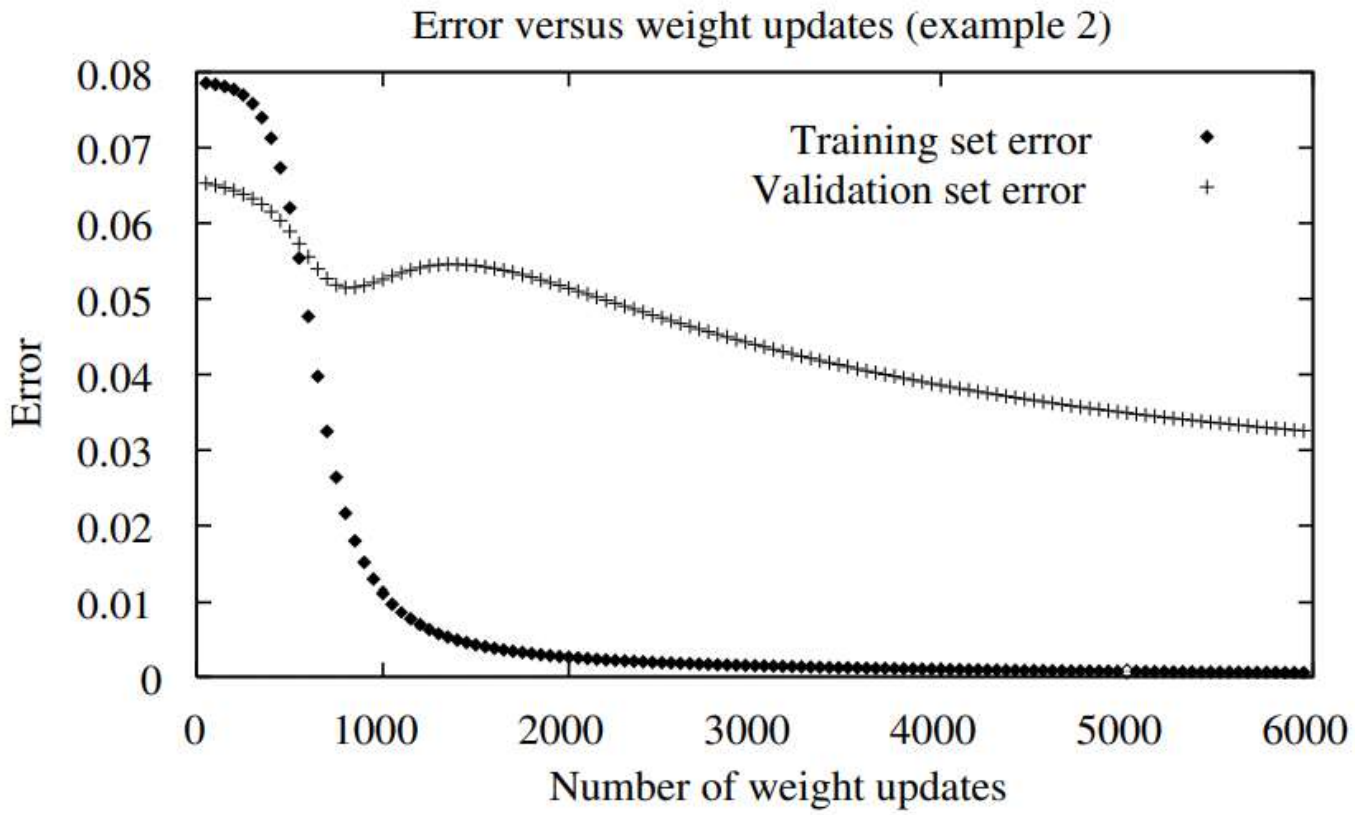
Expressive Capabilities of ANNs

Continuous functions:

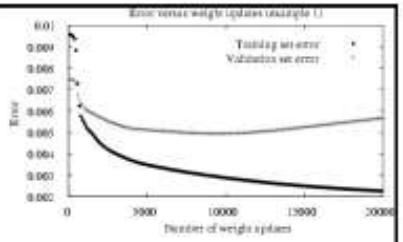
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Overfitting ANN




Error versus weight updates (example 2)
Training set error
Validation set error
Error
Number of weight updates
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
1000
2000
3000
4000
5000
6000

Dealing with Overfitting



Our learning algorithm involves a parameter
n=number of gradient descent iterations
How do we choose n to optimize future error?

- Separate available data into training and validation set
- Use training to perform gradient descent
- n ← number of iterations that optimizes validation set error

→ *gives unbiased estimate of optimal n*
(but a biased estimate of true error)

K-Fold Cross Validation

Idea: train multiple times, leaving out a disjoint subset of data each time for test. Average the test set accuracies.

Partition data into K disjoint subsets

For k=1 to K

testData = kth subset

h ← classifier trained* on all data except for testData

accuracy(k) = accuracy of h on testData

end

FinalAccuracy = mean of the K recorded testset accuracies

* might withhold some of this to choose number of gradient decent steps

Leave-One-Out Cross Validation

This is just k-fold cross validation leaving out one example each iteration

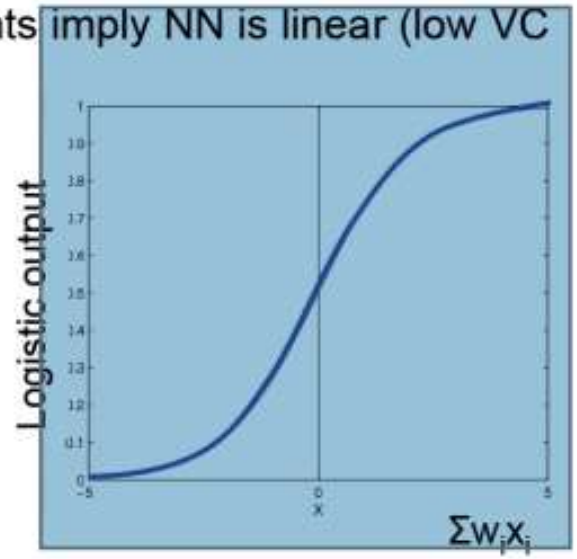
---

Partition data into K disjoint subsets, *each containing one example*
For k=1 to K
    testData = kth subset
    h ← classifier trained* on all data except for testData
    accuracy(k) = accuracy of h on testData
end
FinalAccuracy = mean of the K recorded testset accuracies

* might withhold some of this to choose number of gradient decent steps

Dealing with Overfitting

- Cross-validation
- Regularization – small weights imply NN is linear (low VC dimension)



- Control number of hidden units – low complexity

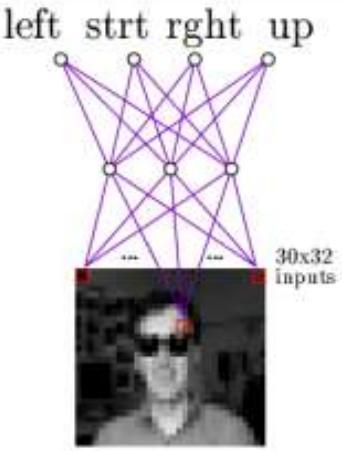
AN ILLUSTRATIVE EXAMPLE: FACE RECOGNITION

The learning task here involves classifying camera images of faces of various people in various poses. Images of 20 different people were collected, including approximately 32 images per person, varying the person's expression (happy, sad, angry, neutral), the direction in which they were looking (left, right, straight ahead, up), and whether or not they were wearing sunglasses. As can be seen from the example images in Figure, here is also variation in the background behind the person, the clothing worn by the person, and the position of the person's face within the image. In total, 624 greyscale images were collected, each with a resolution of 120 x 128, with each image pixel described by a greyscale intensity value between 0 (black) and 255 (white).

30 × 32 resolution input images

target function: learning the direction in which the person is facing (to their left, right, straight ahead, or upward).

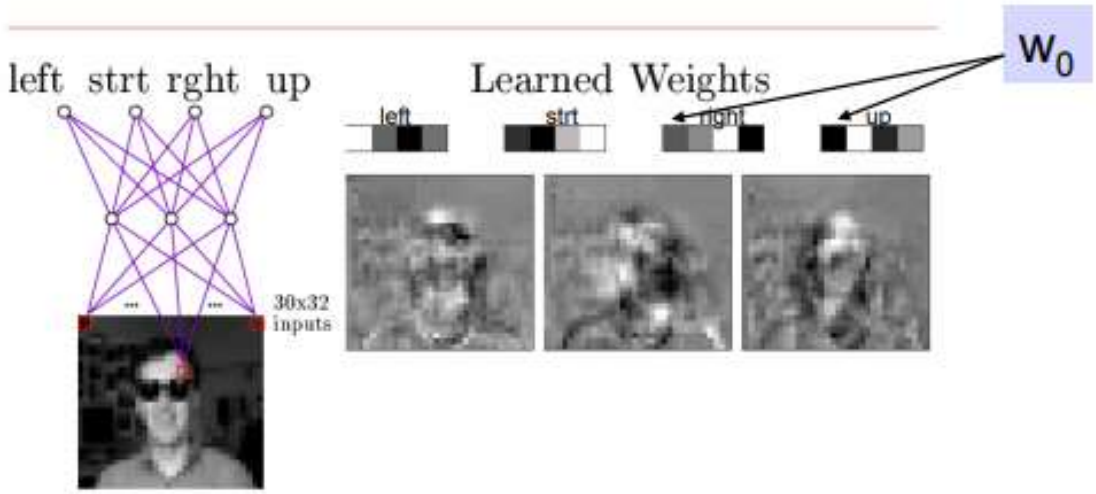
Neural Nets for Face Recognition



Typical input images

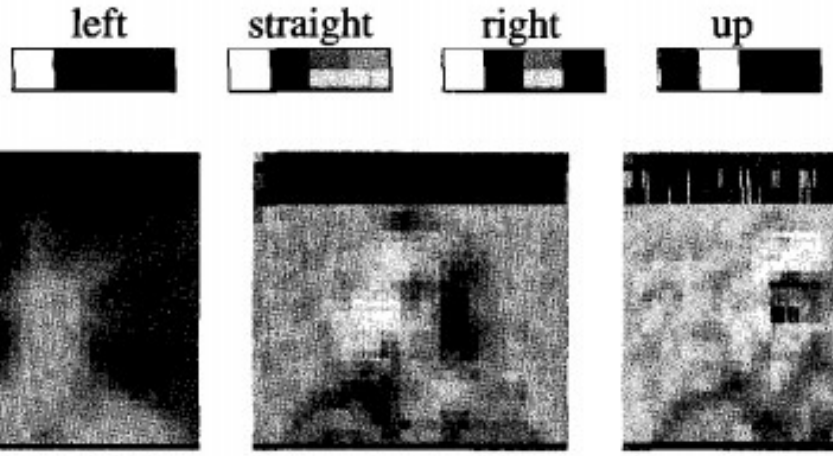
90% accurate learning head pose, and recognizing 1-of-20 faces

Learned Hidden Unit Weights

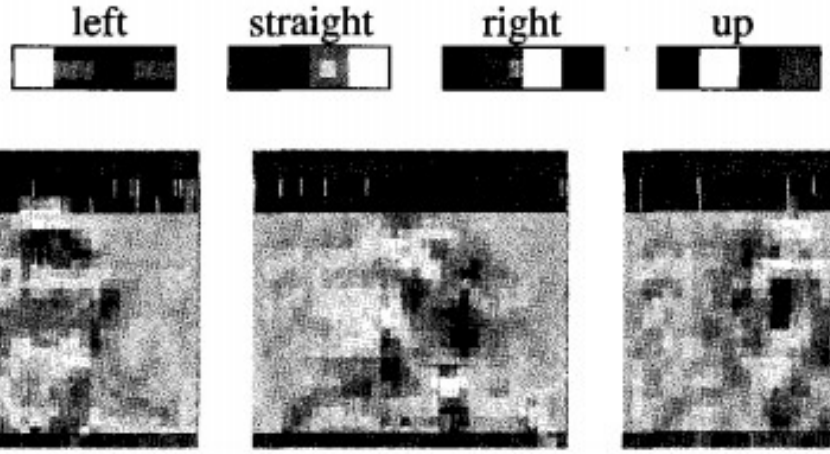


Typical input images

http://www.cs.cmu.edu/~tom/faces.html



Network weights after 1 iteration through each training example



Network weights after 100 iterations through each training example

Learning an artificial neural network to recognize face pose. Here a 960 x 3 x 4 network is trained on grey-level images of faces (see top), to predict whether a person is looking to their left, right, ahead, or up. After training on 260 such images, the network achieves an accuracy of 90% over a separate test set. The learned network weights are shown after one weight-tuning iteration through the training examples and after 100 iterations. Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks. The leftmost block corresponds to the weight wg, which determines the unit threshold, and the three blocks to the right correspond to weights on inputs from the three hidden units. The weights from the image pixels into each hidden unit are also shown, with each weight plotted in the position of the corresponding image pixel.

Design Choice

- Input encoding
- Output encoding
- Network graph structure
- Other learning algorithm parameters

ADVANCED TOPICS
IN
ARTIFICIAL NEURAL NETWORKS

Alternative Error Functions

Adding a penalty term for weight magnitude

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Alternative Error Functions

Adding a term for errors in the slope, or derivative of the target function

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Alternative Error Functions

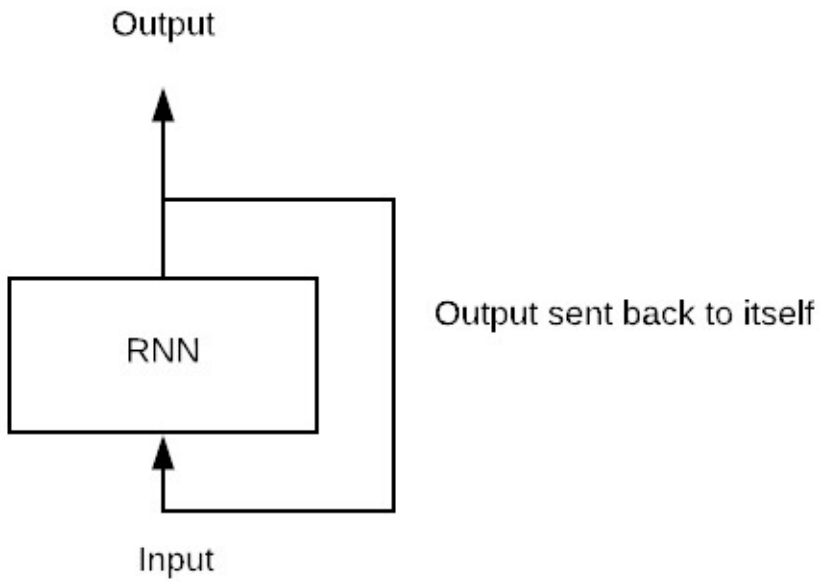
Minimizing the cross entropy of the network with respect to the target values

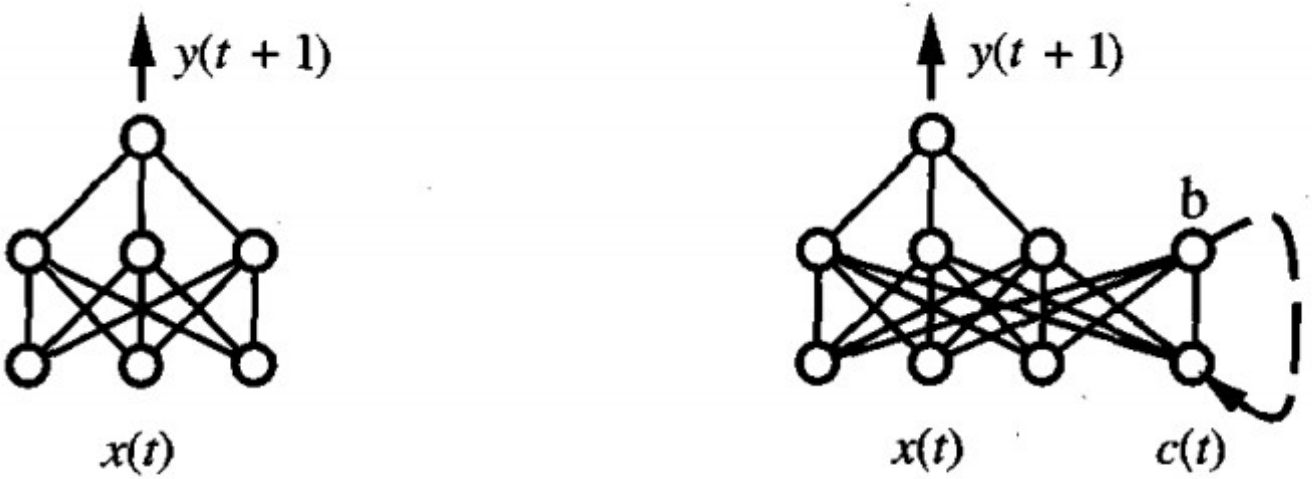
$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

Recurrent Networks

A **Recurrent Neural Network (RNN)** is a class of Artificial Neural Network in which the connection between different nodes forms a directed graph to give a temporal dynamic behavior. It helps to model sequential data that are derived from feedforward networks. It works similarly to human brains to deliver predictive results.

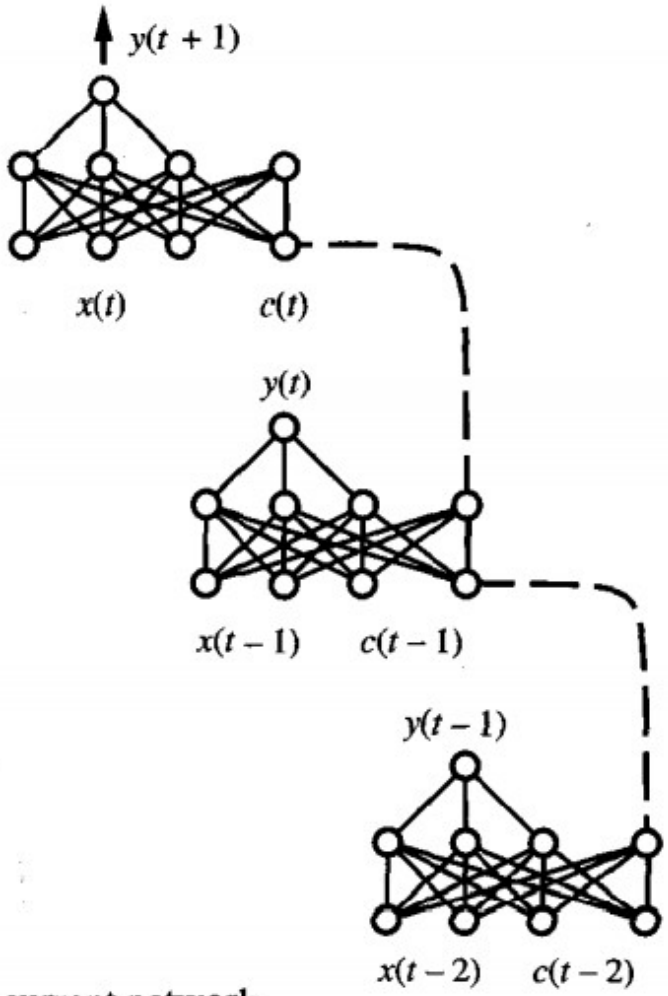
They support a form of directed cycles in the network





(*a*) Feedforward network (*b*) Recurrent network

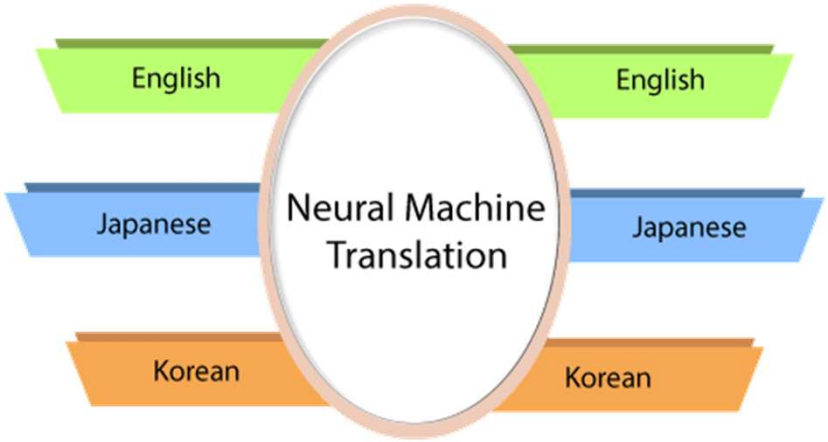
we have added a new unit b to the hidden layer, and new input unit c(t). The input value c(t) to the network at one time step is simply copied from the value of unit b on the previous time step.



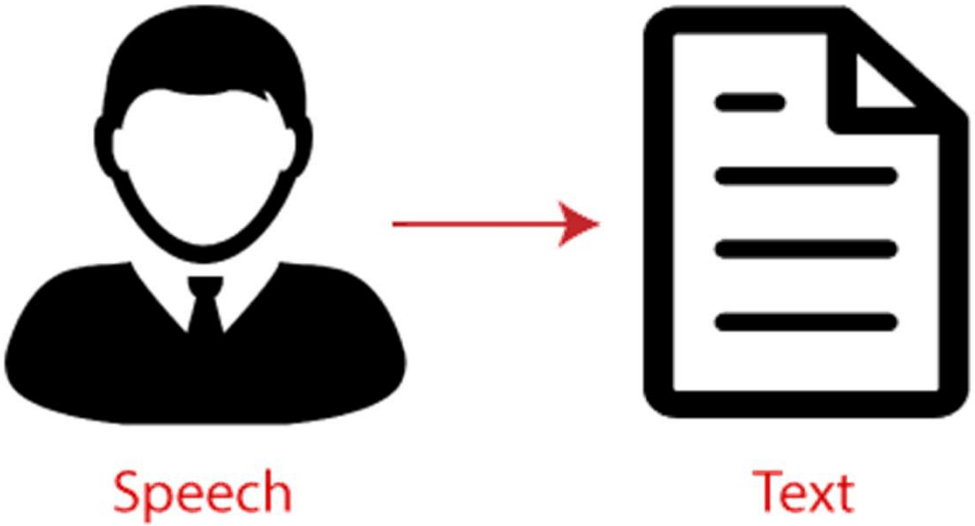
(c) Recurrent network unfolded in time

Following are the application of RNN:

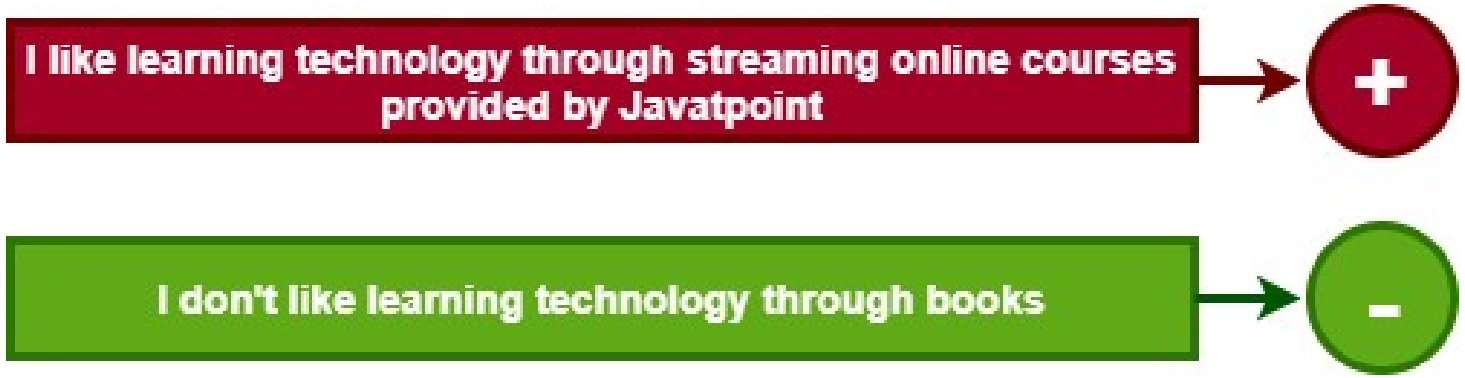
1. Machine Translation



2. Speech Recognition



3. Sentiment Analysis



4. Automatic Image Tagger

A White Dog Jumping into the water

Dynamically Modifying Network Structure

Dynamically Modifying Network Structure

Up to this point we have considered neural network learning as a problem of adjusting weights within a fixed graph structure. A variety of methods have been proposed to dynamically grow or shrink the number of network units and interconnections in an attempt to improve generalization accuracy and training efficiency.

One idea is to begin with a network containing no hidden units, then grow the network as needed by adding hidden units until the training error is reduced to some acceptable level.

CASCADE-CORRELATION algorithm can be used for **Dynamically Modifying Network Structure**

A second idea for dynamically altering network structure is to take the opposite approach. Instead of beginning with the simplest possible network and adding complexity, we begin with a complex network and prune it as we find that certain connections are inessential.

One way to decide whether a particular weight is inessential is to see whether its value is close to zero. A second way, which appears to be more successful in practice, is to consider the effect that a small variation in the weight has on the error E