Artificial Neural Network

Introduction

- The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons
- In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units)
- The human brain, for example, is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^{4} others
- Neuron activity is typically excited or inhibited through connections to other neurons
- The fastest neuron switching times are known to be on the order of 10^{-3} seconds--quite slow compared to computer switching speeds of 10^{-10} seconds
- Humans are able to make surprisingly complex decisions, surprisingly quickly

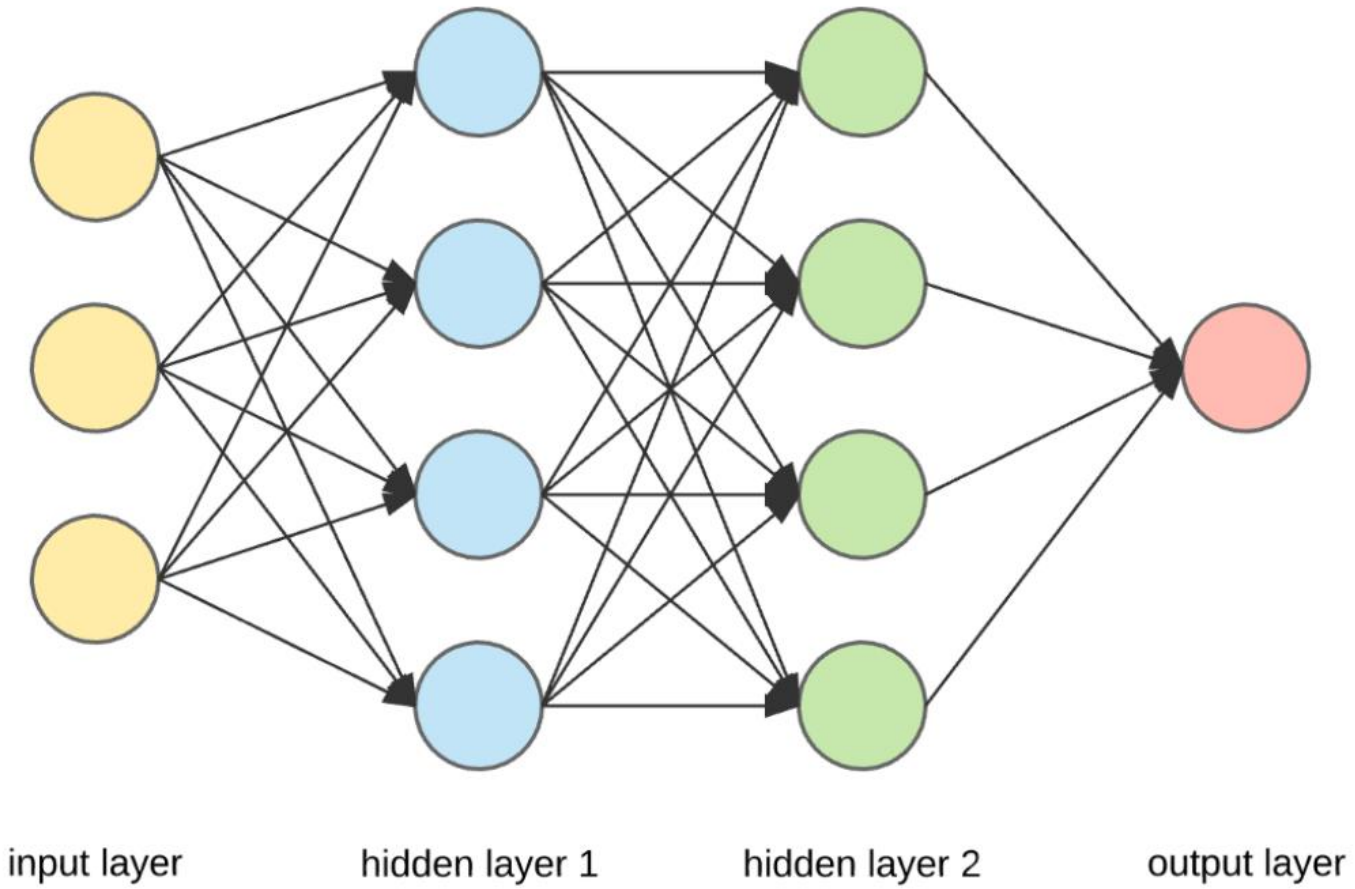
- For example, it requires approximately 10^{-1} seconds to visually recognize your mother
- Notice the sequence of neuron firings that can take place during this 10^{-3} second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons
- This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons
- One motivation for ANN systems is to capture this kind of highly parallel computation based on distributed representations
- While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs
- Many features of the ANNs we discuss here are known to be inconsistent with biological systems
- For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes

- Two groups of researchers have worked with artificial neural networks
- One group has been motivated by the goal of using ANNs to study and model biological learning processes.
- A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes
- As the human brain is a highly complex, non-linear, and parallel computer, therefore, complex tasks (such as pattern recognition, perception, and motor control) can be accomplished much faster than the fastest digital computer
- The ANN resembles the brain in two respect: (i) the network uses the learning process to acquire the knowledge from its environment and (ii) the acquired knowledge is stored by the interconnection strength (also known as synaptic weights)
- Although the idea of ANNs is derived from biological neural architecture, the goal of ANN is to make use of the known functionality of biological networks rather than to reproduce the operations of biological systems

- The properties of ANNs are derived from the characteristic properties of biological systems, such as non-linearity, fault and noise tolerance, massive parallelism, learning, ability to handle imprecise and fuzzy information and powerful generalization capabilities
- These properties have made ANNs very much appealing
- It consists of a large number of neurons connected to each other through the links. The weight of the links can be adjusted on the neural network experience as the ANNs are adaptive to inputs and capable to learn
- ANN has been employed to model highly non-linear systems where the relationship among the variables is unknown or very complex

- It has been widely applied for pattern recognition, classification, and prediction tasks
- The most important benefit of ANN is good classification performance even with the unseen data
- However, there are two important drawbacks of the ANN: (i) difficult to comprehend the symbolic implication of the learned weights of the hidden units in case of MLP, therefore, the explanation for the particular decision made by the network is not clear and (ii) more susceptible to overfitting than other machine learning techniques

Neural Network: Components



PERCEPTRON

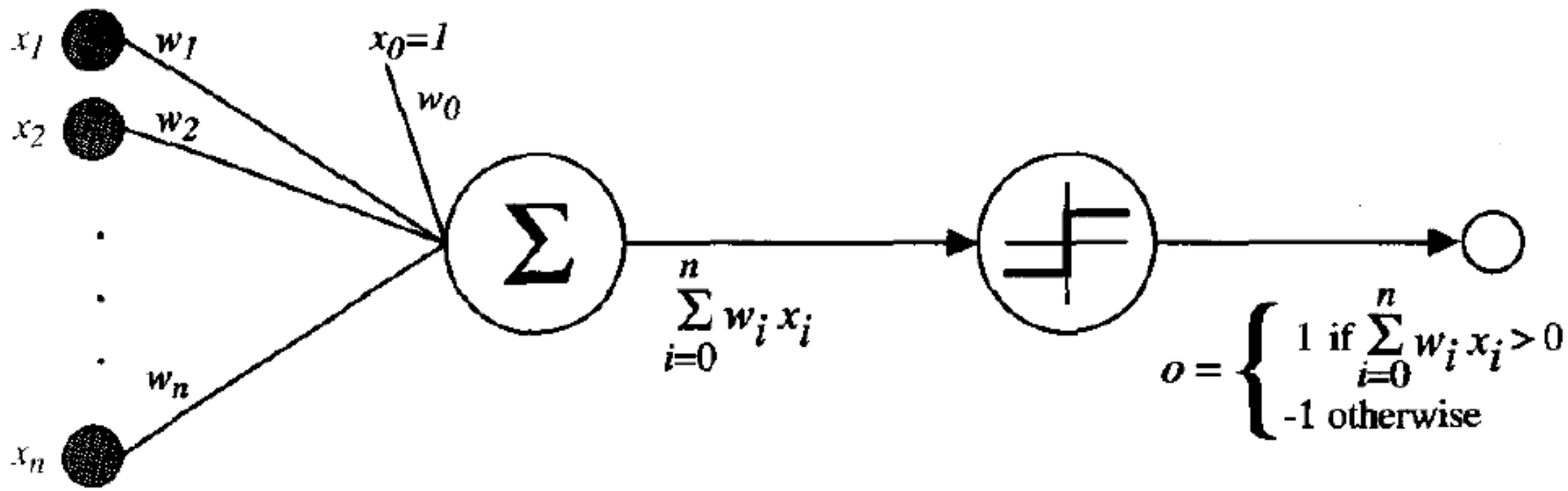
- One type of ANN system is based on a unit called a perceptron
- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise
- More precisely, given inputs x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

- where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output
- The quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$ must surpass in order for the perceptron to output a 1

- To simplify notation, we imagine an additional constant input $\boldsymbol{x_0} = 1$, allowing us to write the above inequality as $\sum_{i=1}^{n} \boldsymbol{w_i}\, \boldsymbol{x_i} > 0$
- Therefore, one can represent the output as $o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$



Representational Power of Perceptron's

- We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side
- The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane
- A single perceptron can be used to represent many boolean functions
- For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights $\boldsymbol{w_0 = -.8}$, and $\boldsymbol{w_1 = w_2 = .5}$
- This perceptron can be made to represent the OR function instead by altering the threshold to $\boldsymbol{w_0 = -.3}$

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

- Perceptrons can represent all of the primitive boolean functions AND, OR, NAND , and NOR
- Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $\boldsymbol{x_1 = x_2}$
- The ability of perceptrons to represent AND, OR, NAND, and NOR is important because ***every*** boolean function can be represented by some network of interconnected units based on these primitives
- In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage
- Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units

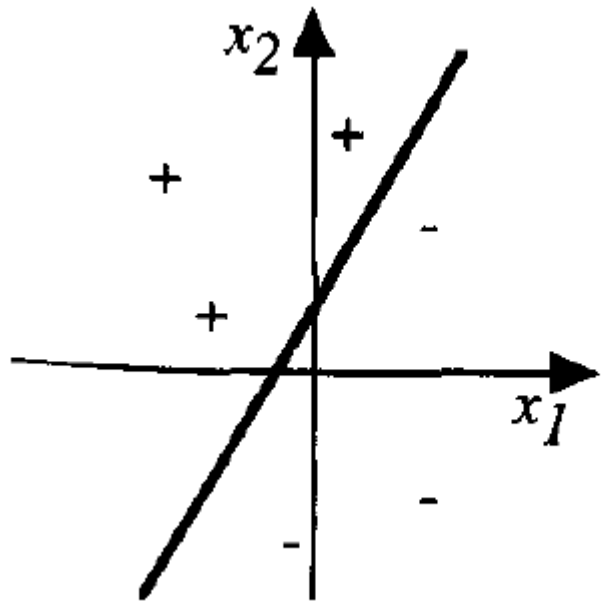
The Perceptron Training Rule

- Let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct output for each of the given training examples
- Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule
- One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly
- Weights are modified at each step according to the ***perceptron training rule,*** which revises the weight $\boldsymbol{w_i}$ associated with input $\boldsymbol{x_i}$ according to the rule

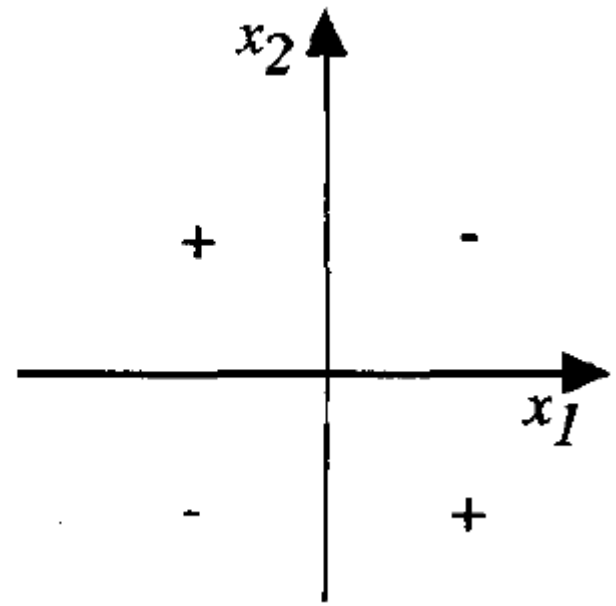
$$w_i \leftarrow w_i + \Delta w_i$$

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

- Here ***t*** is the target output for the current training example, ***o*** is the output generated by the perceptron, and η is a positive constant called the ***learning rate***
- The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases
- Suppose the training example is correctly classified already by the perceptron. In this case, ***(t - o)*** is zero, making Δw_i zero, so that no weights are updated
- Suppose the perceptron outputs a -1, when the target output is +1. The training rule will increase $\boldsymbol{w_i}$ **,** in this case, because ***(t - o),*** η , and $\boldsymbol{x_i}$ are all positive
- For example, if $\boldsymbol{x_i}$ ***= .8,*** η ***= 0.1, t = 1,*** and ***o = - 1,*** then the weight update will be Δw_i= η***(t - o)*** $\boldsymbol{x_i}$ ***= O.1(1 - (-1))0.8 = 0.16.*** On the other hand, if ***t = -1*** and ***o =*** 1, then weights associated with positive $\boldsymbol{x_i}$ will be decreased rather than increased
- In fact, the above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, ***provided the training examples are linearly separable*** and provided a sufficiently small η is used (see Minsky and Papert ***1969).*** If the data are not linearly separable, convergence is not assured



Training examples are linearly separable



Training examples are not linearly separable

Gradient Descent and the Delta Rule

- Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable
- **A** second training rule, called the ***delta rule,*** is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept
- The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples
- This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units
- The delta training rule is best understood by considering the task of training an ***unthresholded*** perceptron; that is, a ***linear unit*** for which the output ***o*** is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

- Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold

- In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples

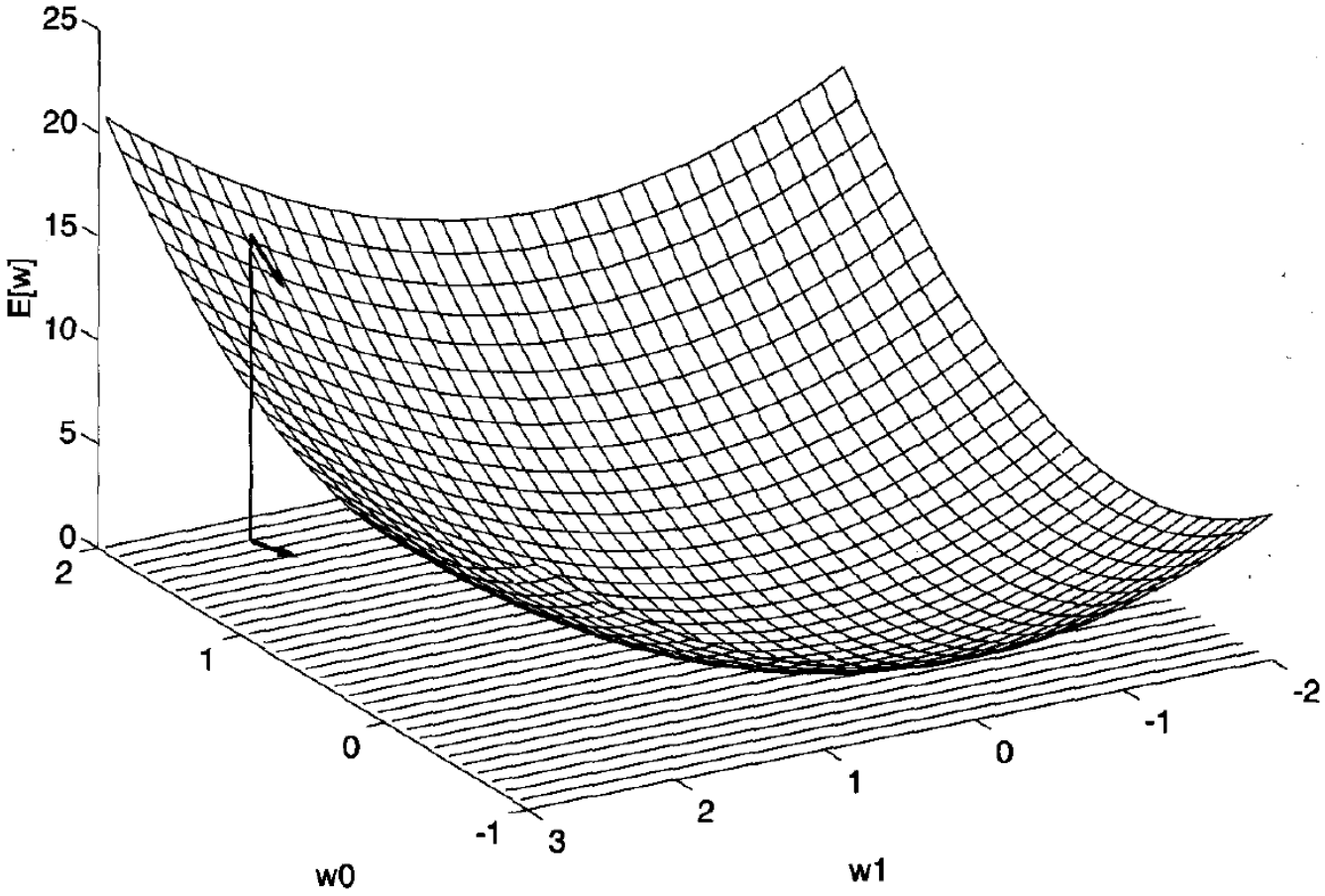
$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \qquad \text{----------(1)}$$

- where D is the set of training examples, $\boldsymbol{t_d}$ is the target output for training example d, and $\boldsymbol{o_d}$ is the output of the linear unit for training example d
- Here we characterize ***E*** as a function of $\vec{w}$**,** because the linear unit output ***o*** depends on this weight vector
- Of course ***E*** also depends on the particular **set** of training examples, but we assume these are fixed during training, so we do not bother to write E as an explicit function of these

VISUALIZING THE HYPOTHESIS SPACE

- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values
- Here the axes w_0 and w_1 represent possible values for the two weights of a simple linear unit
- The w_0, w_1 plane therefore represents the entire hypothesis space
- The vertical axis indicates the error E relative to some fixed set of training examples

- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error)
- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum. The specific parabola will depend, of course, on the particular set of training examples
- Gradient descent search determines a weight vector that minimizes ***E*** by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps
- At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface
- This process continues until the global minimum error is reached



The error surface

DERIVATION OF THE GRADIENT DESCENT RULE

- How can we calculate the direction of steepest descent along the error surface?
- This direction can be found by computing the derivative of ***E*** with respect to each component of the vector $\vec{w}$
- This vector derivative is called the ***gradient*** of ***E*** with respect to $\vec{w}$, written $\nabla \mathrm{E}(\vec{w})$

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

- Notice $\nabla \mathrm{E}(\vec{w})$ is itself a vector, whose components are the partial derivatives of ***E*** with respect to each of the w_i
- When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E
- The negative of this vector therefore gives the direction of steepest decrease
- For example, the arrow in the previous figure shows the negated gradient $-\nabla \mathrm{E}(\vec{w})$ for a particular point in the w_0, w_1 plane

- Since the gradient specifies the direction of steepest increase of ***E,*** the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

Where $\Delta\vec{w} = -\eta\nabla E(\vec{w})$

- Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search
- The negative sign is present because we want to move the weight vector in the direction that ***decreases*** E
- This training rule can also be written in its component form

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

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \qquad \text{----------(2)}$$

- Steepest descent is achieved by altering each component w_i of $\vec{w}$ in proportion to $\dfrac{\partial E}{\partial w_i}$

- To construct a practical algorithm for iteratively updating weights according to Equation ***(2).*** The vector of derivatives that form the gradient can be obtained by differentiating E from Equation (1), as

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\ &= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\ \frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id}) \qquad \text{----(3)}\end{aligned}$$

- Where x_{id} denotes the single input component x_i for training example d. We now have an equation that gives in terms of the linear unit inputs x_{id}, outputs o_d, and target values t_d associated with the training examples

- From equation (2) and equation (3),

$$\Delta w_i = \eta \sum_{d\in D}(t_d - o_d)\, x_{id} \qquad \text{---(4)}$$

Summary of gradient descent algorithm for training linear units

- Pick an initial random weight vector. Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (4)
- Update each weight w_i by adding Δw_i , then repeat this process
- Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate η is used
- If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows

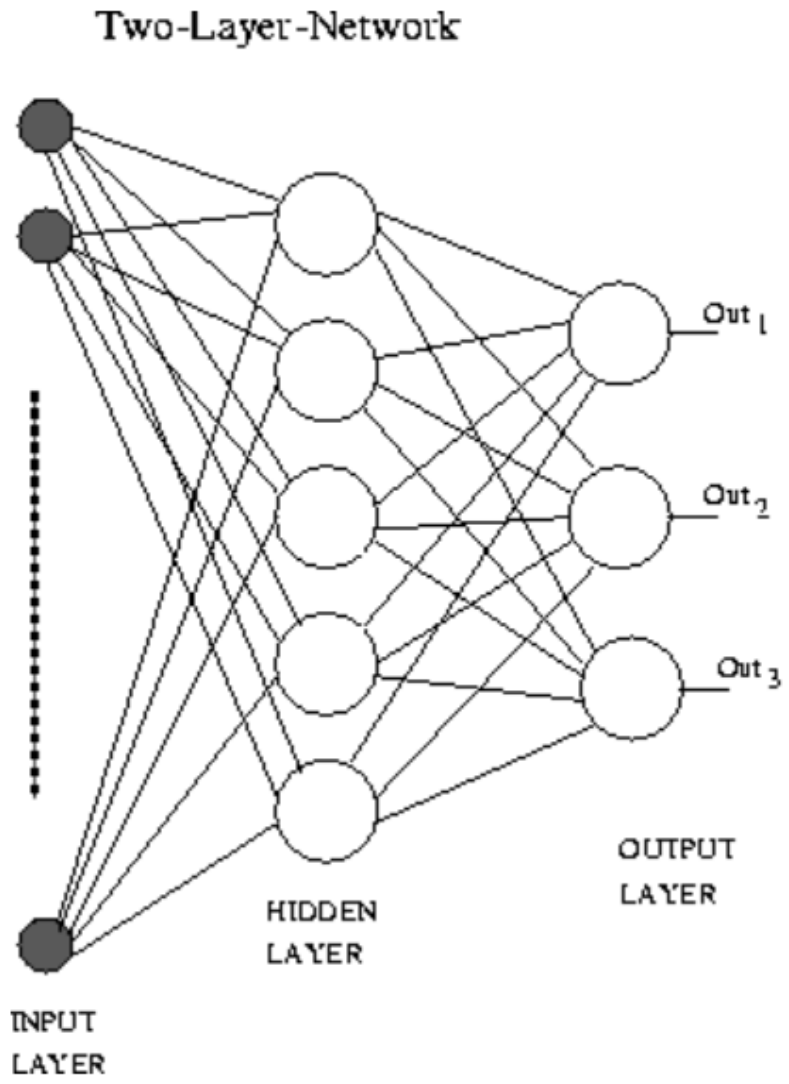
- Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters
- The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum

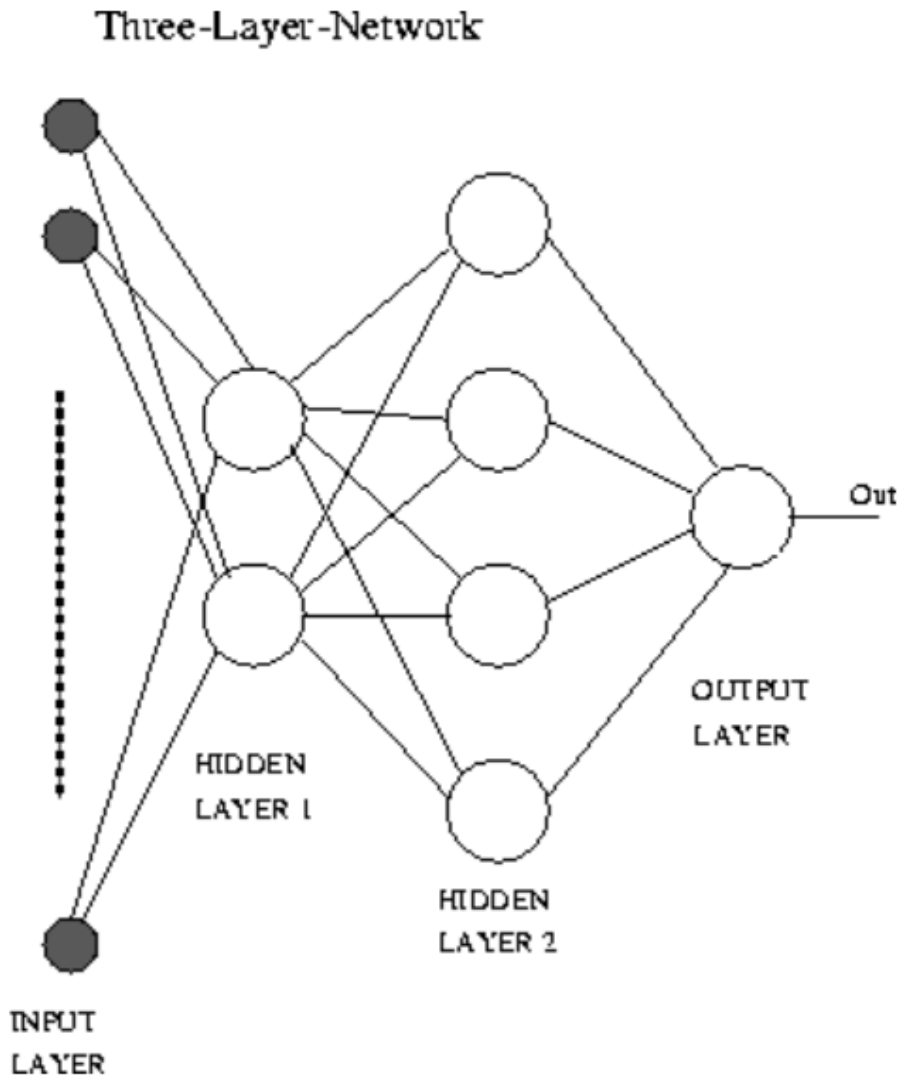
MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

- The Backpropagation neural network is a multilayered, feedforward neural network and is by far the most extensively used
- It is also considered one of the simplest and most general methods used for supervised training of multilayered neural networks
- Backpropagation works by approximating the non-linear relationship between the input and the output by adjusting the weight values internally
- It can further be generalized for the input that is not included in the training patterns

- Single perceptrons can only express linear decision surfaces
- In contrast, the kind of multilayer networks learned by the BACKPROPACATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces

Examples of multi-layered networks





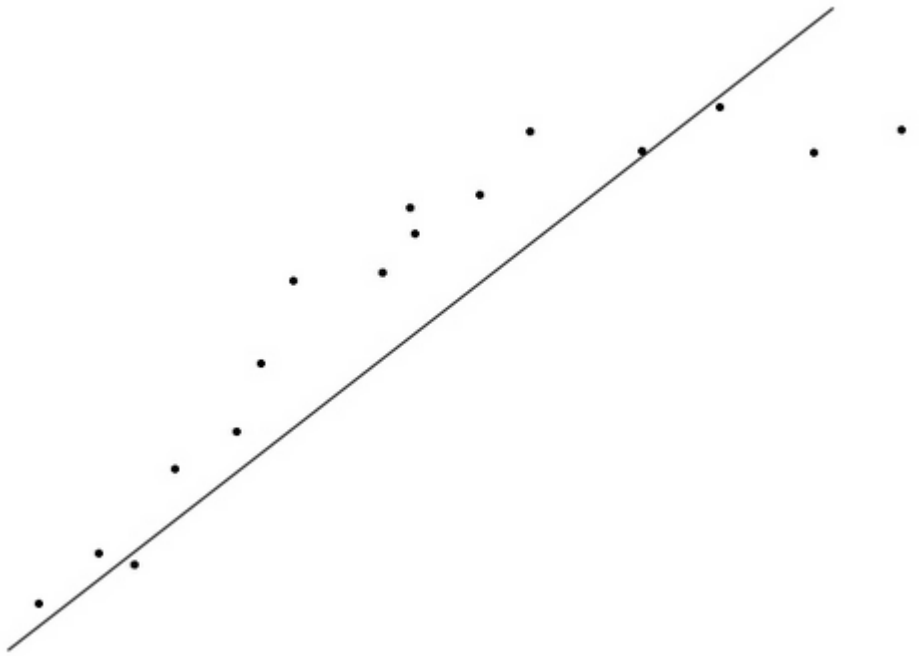
Theoretical background (Backpropagation)

- The operations of the Backpropagation neural networks can be divided into two steps: feedforward and Backpropagation
- In the feedforward step, an input pattern is applied to the input layer and its effect propagates, layer by layer, through the network until an output is produced
- The network's actual output value is then compared to the expected output, and an error signal is computed for each of the output nodes
- Since all the hidden nodes have, to some degree, contributed to the errors evident in the output layer, the output error signals are transmitted backwards from the output layer to each node in the hidden layer that immediately contributed to the output layer
- This process is then repeated, layer by layer, until each node in the network has received an error signal that describes its relative contribution to the overall error

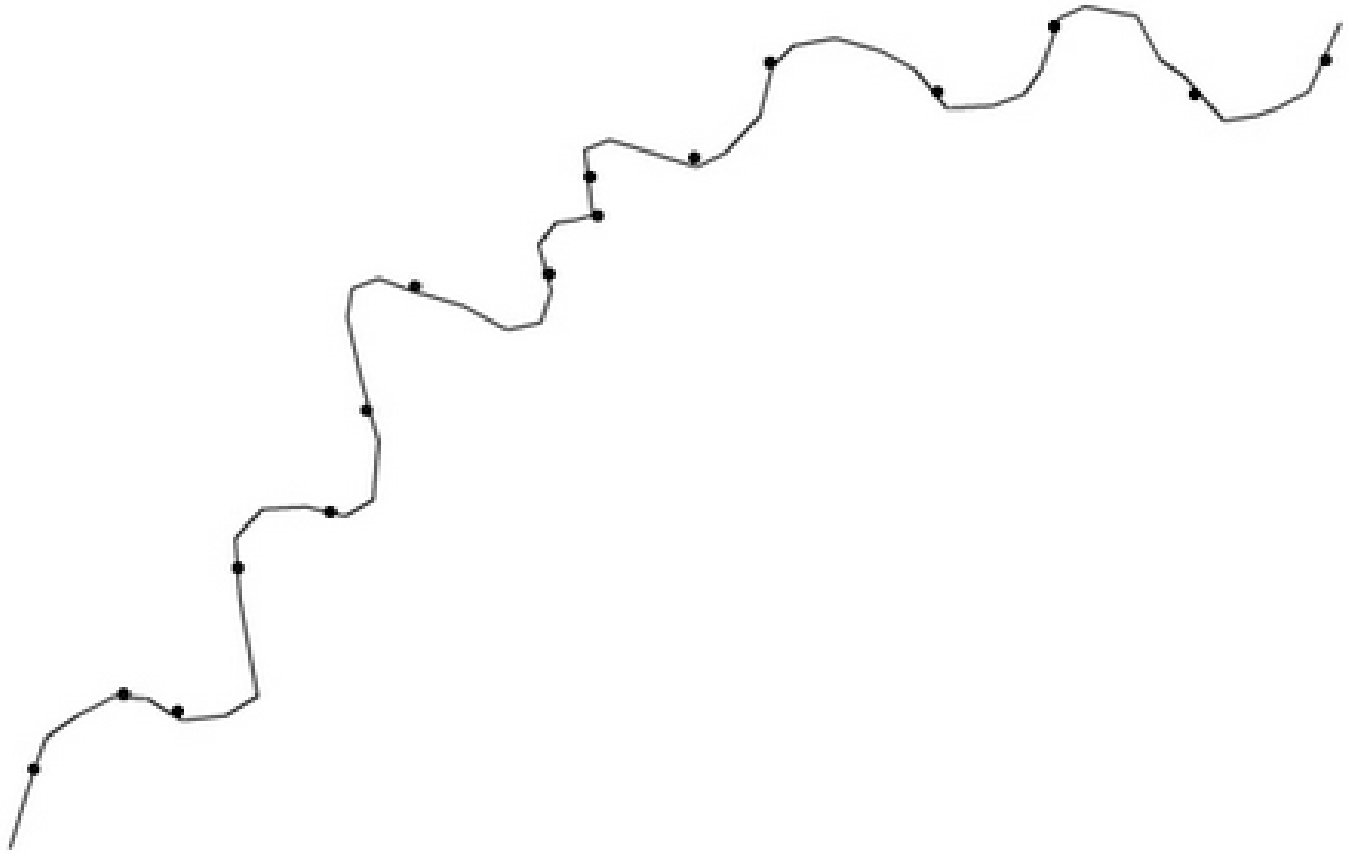
- Once the error signal for each node has been determined, the errors are then used by the nodes to update the values for each connection weights until the network converges to a state that allows all the training patterns to be encoded
- The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent
- The weights that minimize the error function is then considered to be a solution to the learning problem

- There are also issues regarding generalizing a neural network
- Issues to consider are problems associated with under-training and over-training data
- Under-training can occur when the neural network is not complex enough to detect a pattern in a complicated data set
- This is usually the result of networks with so few hidden nodes that it cannot accurately represent the solution, therefore under-fitting the data
- On the other hand, over-training can result in a network that is too complex, resulting in predictions that are far beyond the range of the training data. Networks with too many hidden nodes will tend to over-fit the solution
- The aim is to create a neural network with the "right" number of hidden nodes that will lead to a good solution to the problem

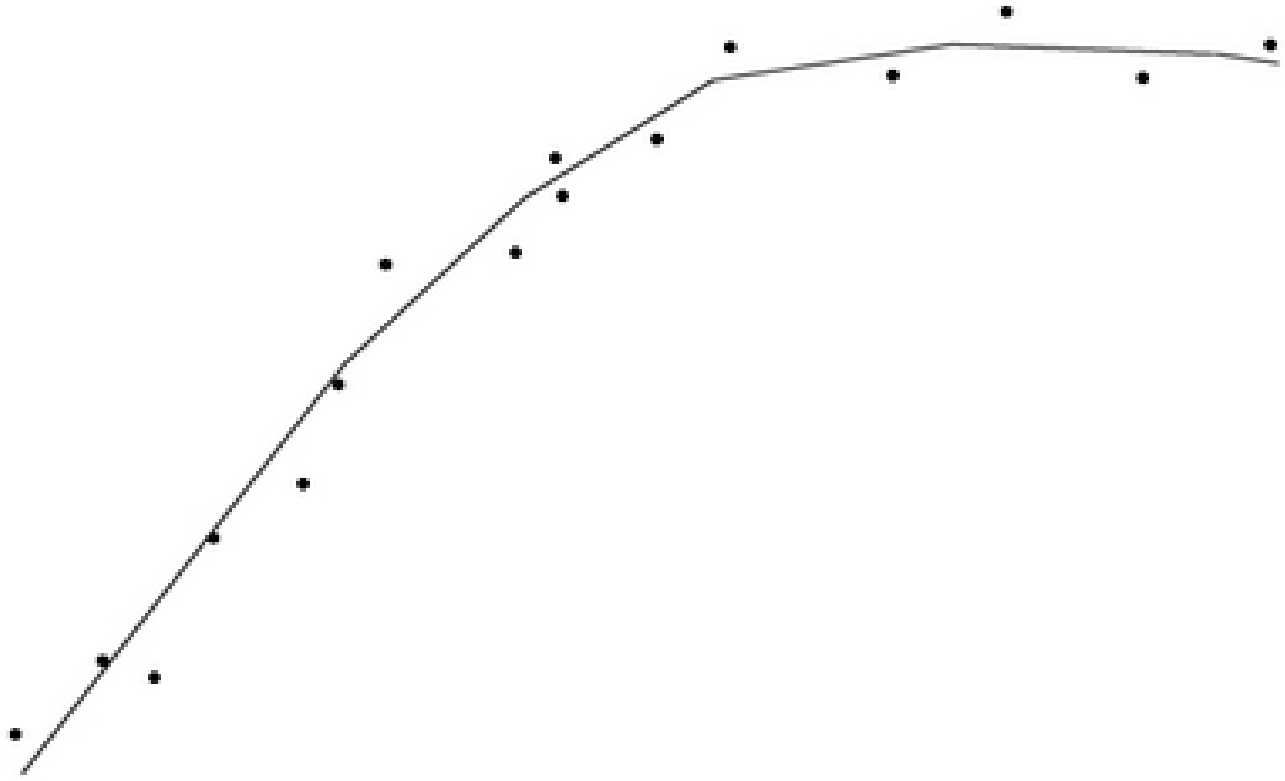
Under-fitting data



Over-fitting data



Good fit of the data



The BACKPROPAGATION Algorithm

- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnection
- It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs
- Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

- where outputs is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the kth output unit and training example d

- The learning problem faced by BACKPROPAGATION Algorithm is to search a large hypothesis space defined by all possible weight values for all the units in the network
- The situation can be visualized in terms of an error surface similar to that shown for linear units
- The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network
- As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E

Stochastic gradient descent version of the BACKPROPAGATION Algorithm for feedforward networks containing two layers of sigmoid units

- BACKPROPAGATION(training_examples, η, n_{in}, n_{out}, n_{hidden})
- Each training example is a pair of the form ($\vec{x}$, $\vec{t}$), where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values
- η is the learning rate (e.g., .O5). n_{in}, is the number of network inputs, n_{hidden}, is the number of units in the hidden layer, and n_{out}, the number of output units
- The input from unit i into unit j is denoted x_{ji} , and the weight from unit i to unit j is denoted w_{ji}

- Create a feed-forward network with n_{in}, inputs, n_{hidden} hidden units, and n_{out} output units
- Initialize all network weights to small random numbers (e.g., between -.05 and .05)

Until the termination condition is met, Do

For each ($\vec{x}$, $\vec{t}$) in *training examples,* **Do**

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u, of every unit u in the network

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

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

3. For each hidden unit h, calculate its error term δ_h

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

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

- One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in the previous error surface
- Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error
- Despite this obstacle, in practice BACKPROPAGATION Algorithm has been found to produce excellent results in many real-world applications

Activation functions in Neural Networks

- In the process of building a neural network, one of the choices you get to make is what activation function to use in the hidden layer as well as at the output layer of the network
- Input Layer :- This layer accepts input features. It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information(features) to the hidden layer
- Hidden Layer :- Nodes of this layer are not exposed to the outer world, they are the part of the abstraction provided by any neural network. Hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer
- Output Layer :- This layer bring up the information learned by the network to the outer world

- Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to **introduce non-linearity** into the output of a neuron
- The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks

VARIANTS OF ACTIVATION FUNCTION

Linear Function

- Linear function has the equation similar to as of a straight line i.e. **y = ax**
- No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer
- **Range :** -inf to +inf
- **Linear activation function** is used at just one place i.e. output layer
- **Issues :** If we will differentiate linear function to bring non-linearity, result will no more depend on *input "x"* and function will become constant, it won't introduce any ground-breaking behavior to our algorithm

Sigmoid Function

- It is a function which is plotted as **‘S’** shaped graph
- **Equation:**
  $Y = 1/(1 + e^{-x})$
- **Nature :** Non-linear
- **Value Range :** 0 to 1
- **Uses :** Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be ***1*** if value is greater than **0.5** and ***0*** otherwise

Tanh Function

- The activation that works almost always better than sigmoid function is Tanh function also knows as **Tangent Hyperbolic function**. It's actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other
- **Equation:** $f(x) = \tanh(x) = 2/(1 + e^{-2x}) - 1 = 2 * \text{sigmoid}(2x) - 1$
- **Value Range:** -1 to +1
- **Nature:** non-linear
- **Uses:** Usually used in hidden layers of a neural network as it's values lies between -1 to 1 hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier

RELU (Rectified linear unit)

- It is the most widely used activation function. Often used in the *hidden layers* of Neural network
- **Equation:** ***A(x) = max(0,x)***. It gives an output x if x is positive and 0 otherwise
- **Value Range:** [0, inf)
- **Nature:** non-linear
- **Uses:** ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation
- RELU learns *much faster* than sigmoid and Tanh function

Softmax Function

- The softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems
- **Nature:** non-linear
- **Uses:** Usually used when trying to handle multiple classes. The softmax function would squeeze the outputs for each class between 0 and 1
- **Output:** The softmax function is ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input

CHOOSING THE RIGHT ACTIVATION FUNCTION

- The basic rule of thumb is if you really don't know what activation function to use, then simply use *RELU* as it is a general activation function and is used in most cases these days
- If your output is for binary classification then, *sigmoid function* is very natural choice for output layer