An Introduction to Machine Learning

Machine learning is the task of inferring a function, e.g., $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$. This inference has to be made using a series of *examples.* An example is nothing but a pair of the form ($\boldsymbol{x}$, $\boldsymbol{y}$), where $\boldsymbol{x} \in \mathbb{R}^m$ and $\boldsymbol{y} \in \mathbb{R}^n$ and $\boldsymbol{y}=f(\boldsymbol{x})$. The examples constitute the *training data* for machine learning.

Convention: Lowercase bold letters (such as $\boldsymbol{x}$ and $\boldsymbol{y}$) will be used to denote vectors and boldface capital letters (such as $\boldsymbol{A}$) will be used to denote matrices.

A simple example:

Assume that the function f is a degree k polynomial.

$f: \mathbb{R} \rightarrow \mathbb{R}$

$f = a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0$

If we know the value of the polynomial at (k+1) distinct points, we can uniquely determine $f(.)$ by interpolating.

In practice we may not know the form of the function and also there could be errors.
In practice we guess the form of the function. Each such possible function will be called a *model* and characterized by some parameters.

We choose parameter values that will minimize the difference between the model outputs & the true function values.

MITCHELL (1997) :

A computer program is said to be learning to perform a set T of tasks from experience E under some performance measure P, if its performance with respect to the tasks in T improves with E.

There are two kinds of learning: **supervised** & **unsupervised**.
In **supervised learning** we are given a set of examples ($\boldsymbol{x}$, $\boldsymbol{y}$) and the goal is to infer the predicting conditional probability distribution P($\boldsymbol{y}|\boldsymbol{x}$).
In **unsupervised learning** the goal is to predict a data generating distribution P($\boldsymbol{x}$) after observing many random vectors $\boldsymbol{x}$ from this distribution.

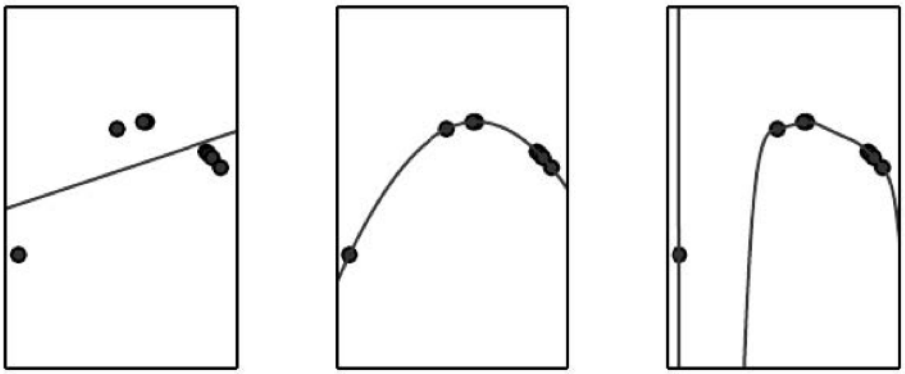
Capacity of a model:

A machine learning algorithm builds a model from the input training data. We can measure the accuracy of the model with respect to the training data. The corresponding error (i.e., the difference between the model and desired outputs) will be called the *training error*.

A machine learning algorithm will also be tested on data points previously unseen. These unseen data points used to test the algorithm will be referred to as the test data. We can define a corresponding *test error*.

A model is said to underfit if its training error is not low enough. A model is said to overfit if the difference between the training and test error is very large. In this case the model memorizes the properties of the training data closely. We can modify the underfitting and overfitting behavior of a learning algorithm by changing the capacity with a low capacity tends to underfit and a model with a high capacity tends to overfit.

(Goodfellow, et al. 2016) have generated data from a quadratic function and used three different models to fit the data. These three models were degree 1, degree 2, and degree 9 polynomials, respectively. The results they got are shown below:



The performance of a model is optimal when its capacity is close to the complexity of the function learned.

No free lunch theorem (WOLPERT 1996):

When averaged over all possible data generating distributions, each classifier has the same error rate on previously unseen data points. We can develop better algorithms if we restrict the functions of interest and/or we have more information on the data generating distribution.

ARTIFICIAL NEURAL NETWORKS

A model that can represent more complex functions is the ARTIFICAL NEURAL NETWORK (ANN). ANNs have been employed since a long time ago. They were known with different names:

Cybernetics

⇓

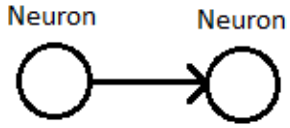
Connectionist models

⇓

DEEP NEURAL NETWORK

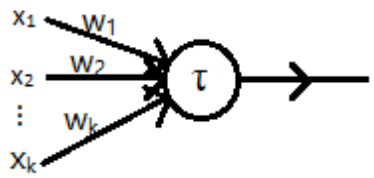
A neural network (NN) is a weighted directed graph $G(V, E)$.

Each node in V corresponds to a neuron. If there is a directed edge from a neuron u to another neuron v, a signal passes from u to v. I.e., v gets an input from u.



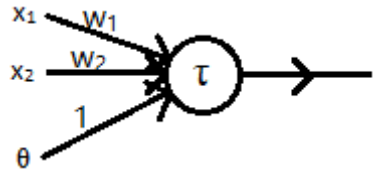
In any NN there are some input nodes and some output nodes. Input flows through the other nodes in the network, gets transformed, and finally reaches the output nodes.

Example: A PERCEPTRON:



output is 1 if $\sum_{i=1}^{k} w_i x_i \geq \tau$; It is zero otherwise.

A PERCEPTRON can be thought of as a BINARY CLASSIFIER. Consider the following perceptron:



output is 1 if $w_1x_1 + w_2x_2 + \theta \geq \tau$

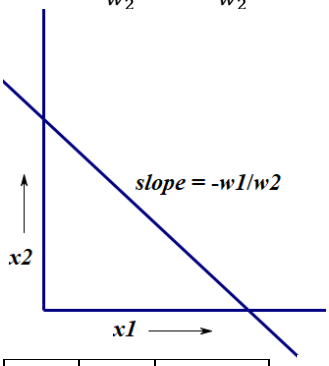
$\Rightarrow w_1x_1 + w_2x_2 \geq u$

$u \rightarrow$ a constant

$\Rightarrow w_2x_2 \geq u - w_1x_1$

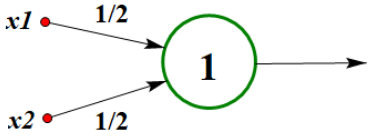
$\Rightarrow x_2 \geq \frac{-w_1}{w_2}x_1 + \frac{u}{w_2}$

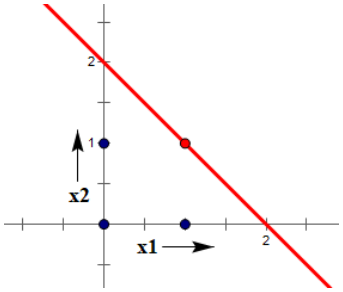
A perceptron is a binary classifier when the two classes can be separated by a straight line.



REALIZING Boolean AND:

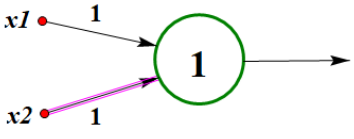
x_2	x_1	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

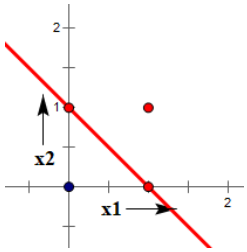




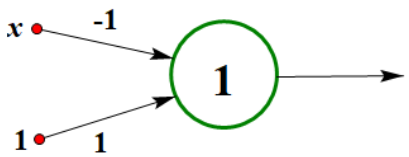
BOOLEAN OR:

x_2	x_1	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1



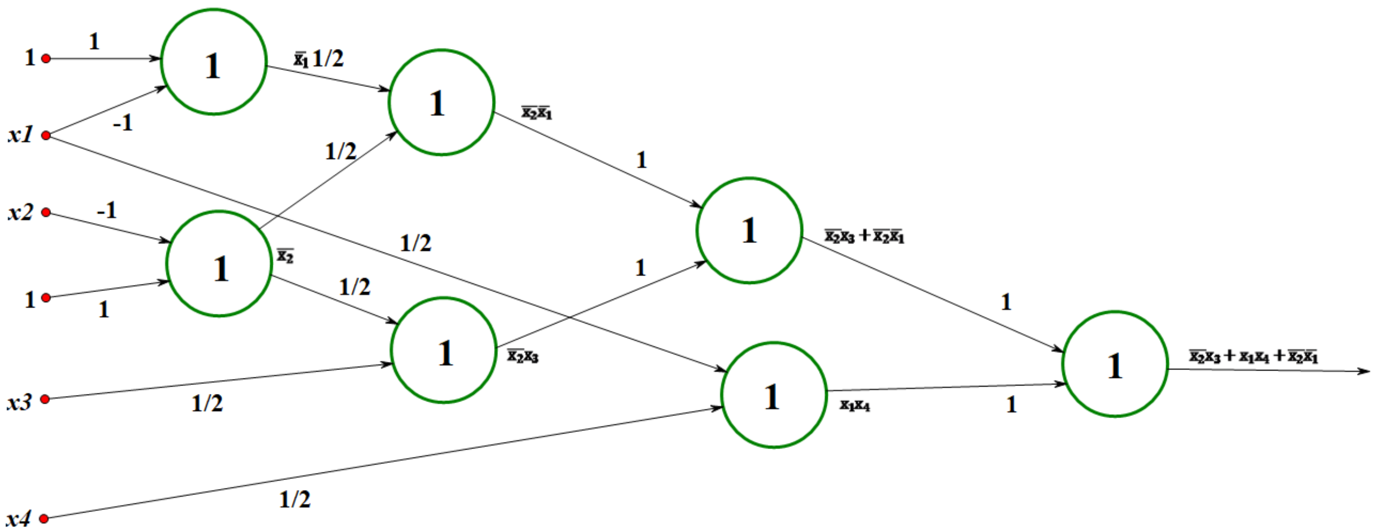


Boolean NOT:

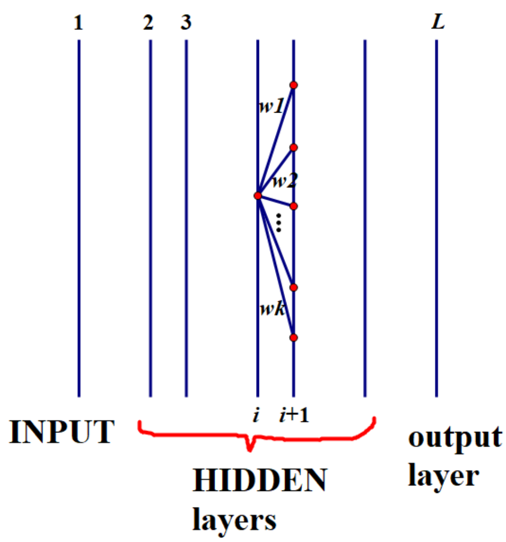


COROLLARY: ANY Boolean function can be realized with a neural network using perceptrons.

Example: F = $\overline{x_2}x_3 + x_1x_4 + \overline{x_2}\ \overline{x_1}$

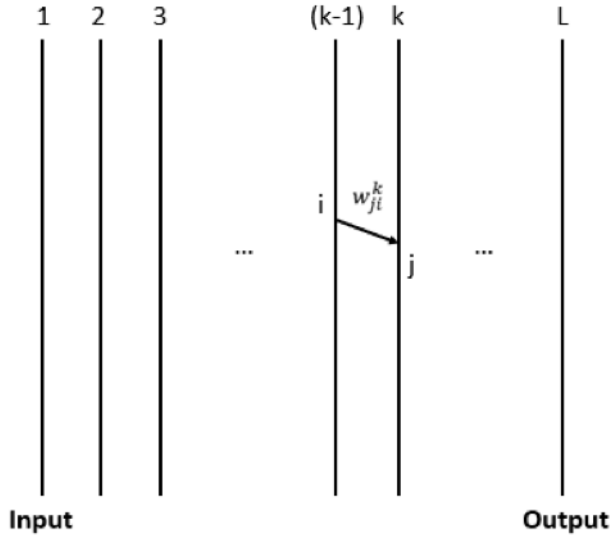


A General NN looks like:



To use GRADIENT DESCENT, we have to make sure that the output from every node is continuous.

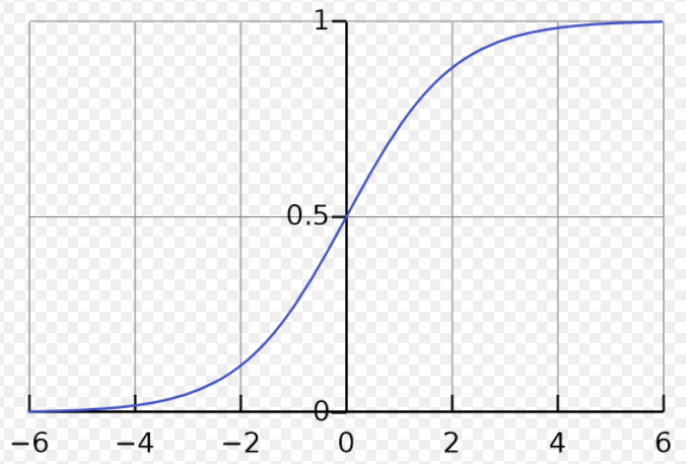
Therefore, we apply an “ACTIVATION FUNCTION” at each node.



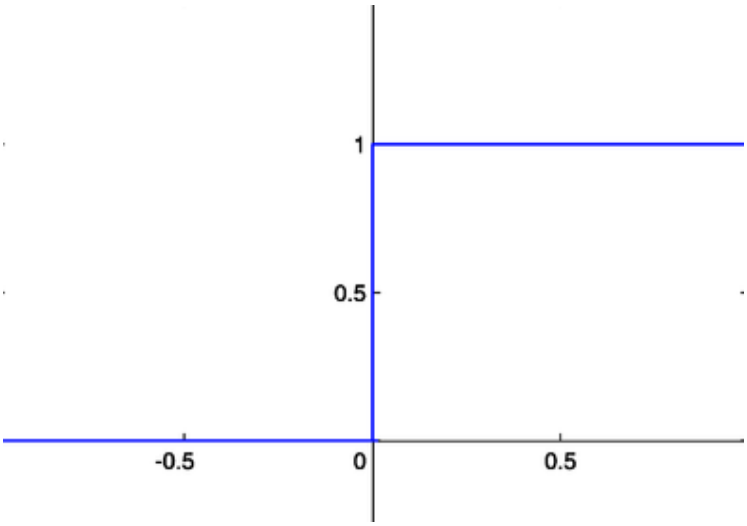
Let n_k be the number of neurons in level k, $1 \leq k \leq L$.
There could be a connection from every node in level $(k-1)$ to every node in level k, $1 \leq k \leq L$.
Let the weight of the edge from node i of level $(k-1)$ to node j of level k be denote as w_{ji}^k.
Let the output of node j in level k be denoted as a_j^k. $a_j^k = \sigma\left(\sum_{i=1}^{n_{l-1}} w_{ji}^k \, a_i^{k-1} + b_j^k\right)$, where $\sigma \rightarrow Activation\ Function$. Let $z_j^k = \sum_{i=1}^{n_{l-1}} w_{ji}^k \, a_i^{k-1} + b_j^k$. z_j^k is the weighted input for the node j in level k.

Possible Activation Functions:

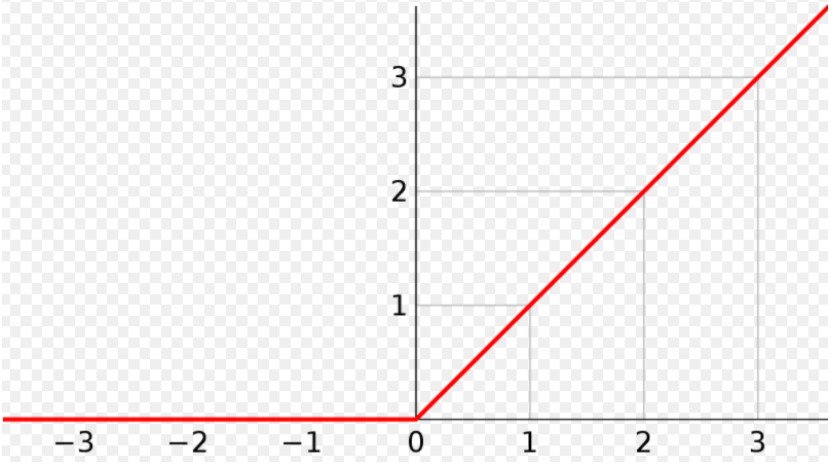
1.Sigmoid Function: $\sigma(x) = \frac{1}{1+e^{-x}}$



The sigmoid function can be thought of as an approximation to the step function：



2.Rectilinear Function: $\sigma(x) = \max\{0, x\}$



3. Softmax Function:

$$\boldsymbol{softmax(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}, where\ z\ is\ a\ vector.}$$

We compute optimal values for the network parameters using gradient descent.
Let C be any cost function on $x_1, x_2, x_3, \ldots x_N$. If we change the parameter values by $\Delta x_1, \Delta x_2, \ldots\ \Delta x_N$, the change in $C = \Delta C = \frac{\partial C}{\partial x_1}\Delta x_1 + \frac{\partial C}{\partial x_2}\Delta x_2 + \cdots \frac{\partial C}{\partial x_N}\Delta x_N$. Let $\boldsymbol{x} = (x_1, x_2, .. x_N)^{\mathrm{T}}$.

$$\nabla_{\boldsymbol{x}} C = \begin{bmatrix} \frac{\partial C}{\partial x_1} \\ \vdots \\ \frac{\partial C}{\partial x_N} \end{bmatrix}, \Delta C = (\Delta x_1\ \Delta x_2\ \ldots\ \Delta x_N)\,.\nabla_{\boldsymbol{x}} C$$

If $\Delta \boldsymbol{x} = -\alpha \nabla_{\boldsymbol{x}} C$, then $\Delta C = -\alpha(\nabla_{\boldsymbol{x}} C)^{\mathrm{T}}(\nabla_{\boldsymbol{x}} C) = -\alpha \|\nabla_{\boldsymbol{x}} C\|_2^2$ which will always be negative. The idea of gradient descent is to choose this value for $\Delta \boldsymbol{x}$.
We can think of an iterative algorithm for minimizing C:

1. Start with some initial value for $\boldsymbol{x}$; Let this value be $\boldsymbol{x_0}$;

2. Compute $C(x_0)$ and $\nabla_x C$, let $\boldsymbol{x_1 = x_0 - \alpha \nabla_x C}$
3. Repeat step 2 until the gradient becomes zero or close to zero.
   α is called the $learning\ rate$;

In the case of a Neural Network, let $w_1, w_2, \ldots, w_N$ be the list of parameters, *C* could be the MSE. We'll use gradient decent to update each parameter, i.e., $w_i = w_i - \alpha \frac{\partial C}{\partial w_i}, \forall\ i$.

Let C_x be the cost for example *x*, we can compute *C* as $\frac{1}{m}\sum_x C_x$. We can also compute ∇C as $\frac{1}{m}\sum_x \nabla C_x$.

If *m* is large, it will take a very long time before the parameters are updated. An alternative is to use Stochastic Gradient Descent. We pick a random sample *S* of examples & the gradient can be computed using the sample. Once the gradient is computed, we can update the parameters. We'll compute *C* as $\frac{1}{|S|}\sum_{x \in S} C_x$ and $\nabla_x C$ as $\frac{1}{|S|}\sum_{x \in S} \nabla C_x$. Make updates using these values. We'll repeat this for other samples from the input.

Each subset is a mini batch.

When we end up using all the examples in the input once, we have completed an epoch. In fact, we can, in any epoch, shuffle the input examples randomly and partition the input into several mini batches. In the epoch, we will process each of the mini batches exactly once.

We may repeat the epochs.

When |*S*| = 1, we call it as online or incremental learning.

Training:

An Epoch:

For every Mini Batch do:

Compute the activation values for each node in each layer starting from the input layer and ending at the output. This is one forward propagation. Followed by this, compute $\frac{\partial C}{\partial w_i}$ for every parameter w_i.

Compute ∇C & C, and update the parameter values.