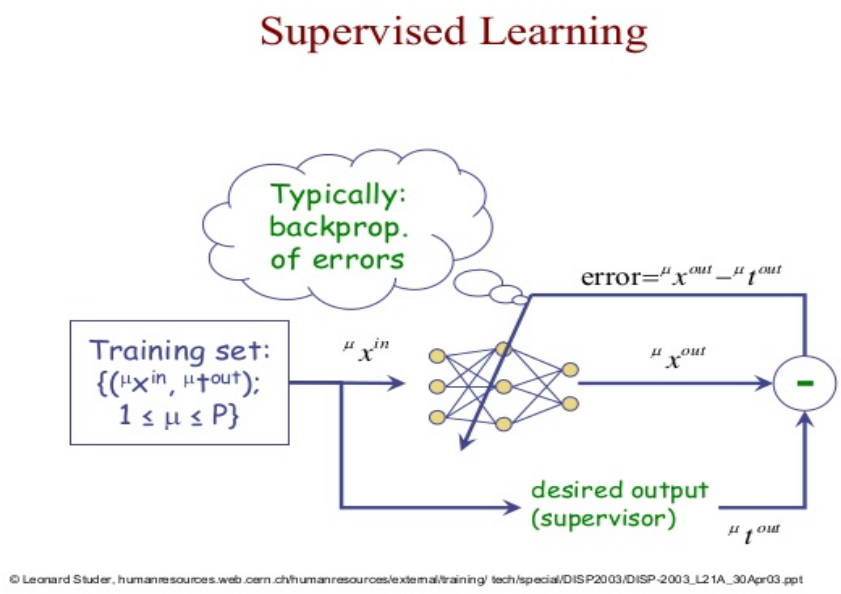
Neural Network - Supervised Learning

Supervised learning is the algorithmic process of approximating the underlying function between labeled data and their corresponding attributes or features. A popular example of supervised learning is that of a machine that is asked to distinguish between apples and pears (labeled data) given a set of features or data attributes such as the fruits' color and size. Initially, the machine learns to classify between apples and pears by seeing a number of available fruit examples—which contain the color and size of each fruit, on one hand, and their corresponding label (apple or pear) on the other. After learning is complete, the machine should ideally be able to tell whether a new and unseen fruit is a pear or an apple based solely on its color and size. Beyond distinguishing between apples and pears supervised learning nowadays is used in a plethora of applications including financial services, medical diagnosis, fraud detection, web page categorization, image and speech recognition, and user modeling (among many).

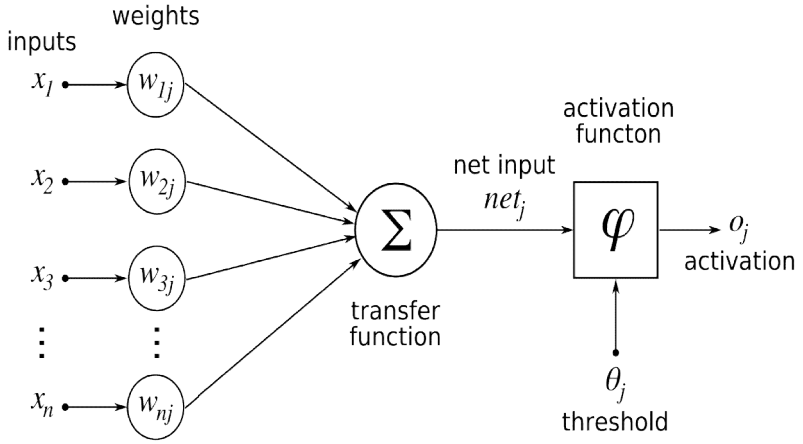


Evidently, supervised learning requires a set of labeled training examples; hence **supervised**. More specifically, the training signal comes as a set of supervised labels on the data (e.g., this is an apple whereas that one is a pear) which acts upon a set of characterizations of these labels (e.g., this apple has red color and medium size).

Formally, supervised learning attempts to derive a function $f : X \rightarrow Y$, given a set of N training examples $\{(x_1, y_1), ..., (x_N, y_N)\}$; where X and Y is the input and output space, respectively; x_i is the feature (input) vector of the i-th example and y_i is its corresponding set of labels. A supervised learning task has two core steps:

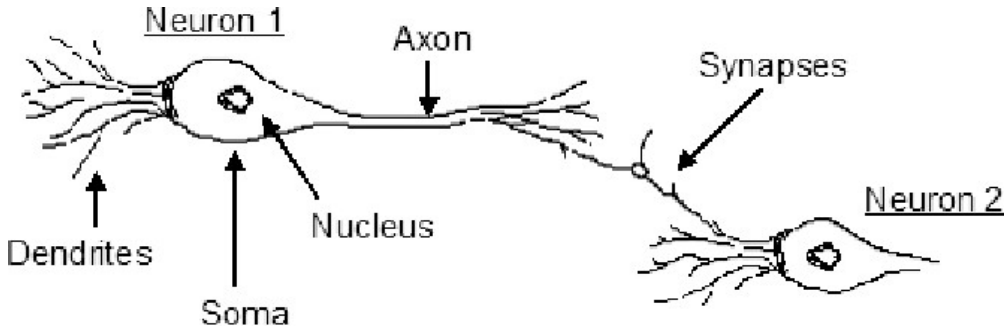
In the first **training** step, the training samples—attributes and corresponding labels are presented and the function f between attributes and labels is derived.

In the second testing step f can be used to **predict** the labels of unknown data given their attributes. To validate the generalizability of f.

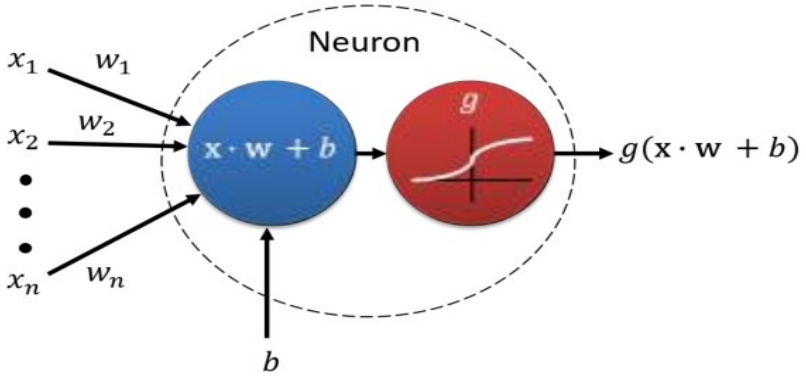


Artificial Neural Networks

Artificial Neural Networks (ANNs) are a bio-inspired approach for computational intelligence and machine learning. An **ANN** is a **set of interconnected processing units (named neurons) which was originally designed to model the way a biological brain**—containing over 10^{11} neurons—processes information, operates, learns and performs in several tasks. Biological neurons have a cell body, a number of dendrites which bring information into the neuron and an axon which transmits electrochemical information outside the neuron, as shown in Figure below.



The artificial neuron (see Figure below) resembles the biological neuron as it has a number of inputs **x** (corresponding to the neuron **dendrites**) each with an associated weight parameter **w** (corresponding to the **synaptic strength**). It also has a processing unit that combines inputs with their corresponding weights via an inner product (weighted sum) and **adds** a bias (or **threshold**) weight **b** to the weighted sum as follows: **x·w+b**. This value is then fed to an activation function **g** (cell body) that yields the output of the neuron (corresponding to an axon terminal). ANNs are essentially simple mathematical models defining a function f : **x** → **y**.

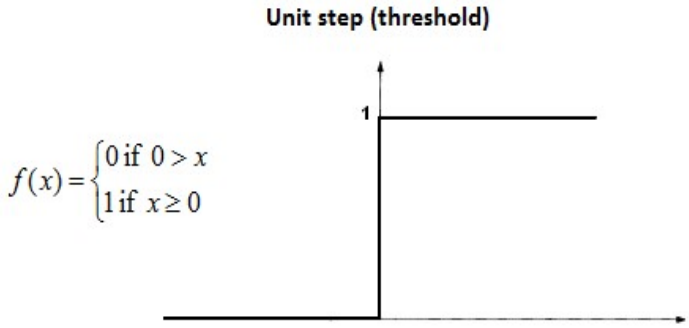


An illustration of an artificial neuron, the neuron is fed with the input vector x through n connections with corresponding weight values w. The neuron processes the input by calculating the weighted sum of inputs and corresponding connection weights and adding a bias weight (b): x·w+b. The resulting formula feeds an activation function (g), the value of which defines the output of the neuron.

Core application areas include pattern recognition, robot and agent control, game-playing, decision making, gesture, speech and text recognition, medical and financial applications, affective modeling, and image recognition

Activation Functions

Which activation function should one use in an ANN? The original model of a neuron by McCulloch and Pitts in 1943 featured a Heaviside step activation function (see Figure below) which either allows the neuron to fire or not. When such neurons are employed and connected to a multi-layered ANN the resulting network can merely solve linearly separable problems.



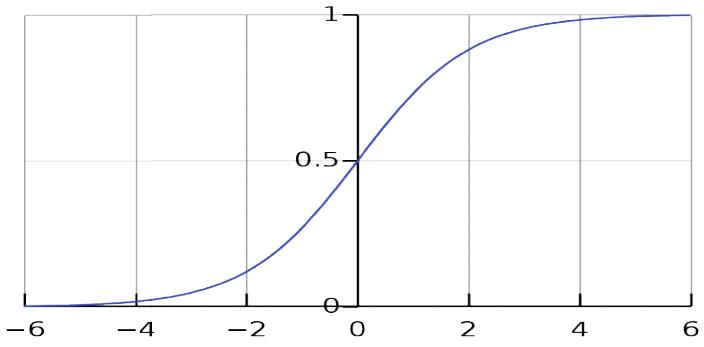
The algorithm that trains such ANNs was invented in 1958 and is known as the **Rosenblatt's perceptron algorithm**. Non linearly separable problems such as the **exclusive OR** gate could only be solved after the invention of the **backpropagation algorithm** in 1975.

Nowadays, there are several activation functions used in conjunction with ANNs and their training. The use of the activation function, in turn, yields different types of ANNs. Examples include **Gaussian activation function.** The most common function used for ANN training is the sigmoid-shaped logistic function $\mathbf{g(x) = 1/(1+e^{-x})}$ for the following properties (see Figure):

1) It is bounded, monotonic and non-linear;

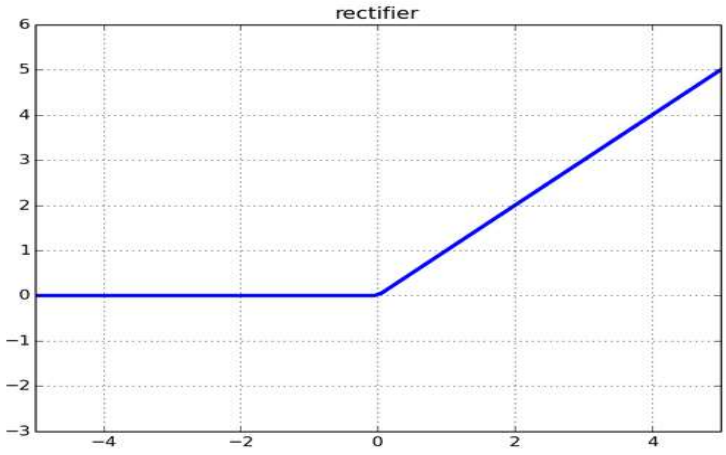
2) It is continuous and smooth and

3) Its **derivative** is calculated **trivially** as $g'(x) = g(x)(1-g(x))$. Given the properties above the logistic function can be used in conjunction with gradient-based optimization algorithms such as back-propagation which is described below.



Other popular activation functions for training deep architectures of neural networks include the rectifier—named **rectified linear unit** (**ReLU**) when employed to a neuron—and its smooth approximation, the soft plus function. Compared to sigmoid-shaped activation functions, **ReLUs**

allow for faster and (empirically) more effective training of deep ANNs, which are generally trained on large datasets.



From a Neuron to a Network

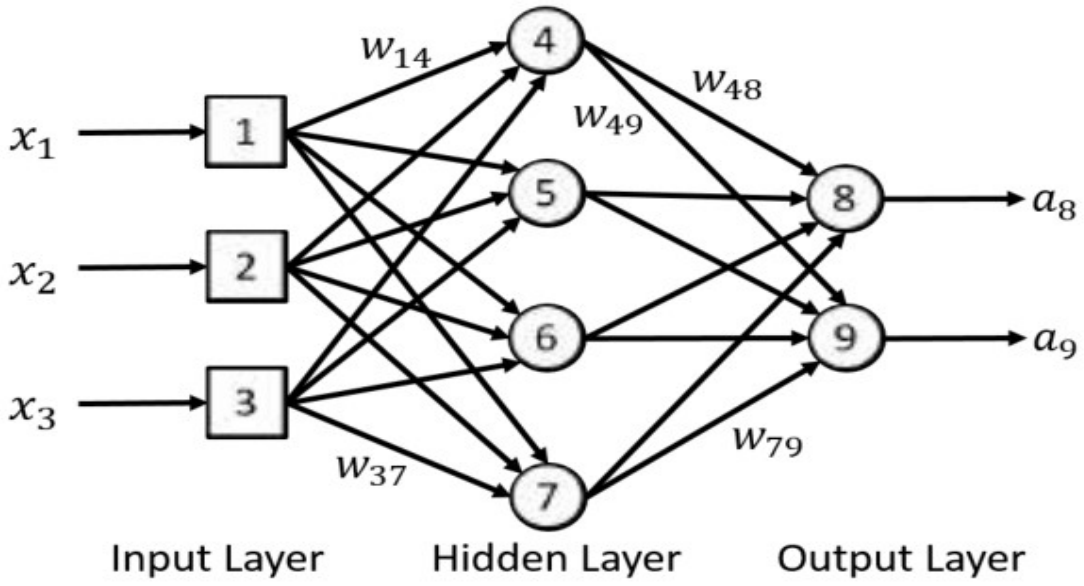
To form an ANN a number of neurons need to be structured and connected. While numerous ways have been proposed in the literature the most common of them all is to structure neurons in layers. In its simplest form, known as the **multi-layer perceptron (MLP),** neurons in an ANN are layered across one or more layers but not connected to other neurons in the same layer (see Figure below for a typical MLP structure).

The output of each neuron in each layer is connected to all the neurons in the next layer. Note that a neuron's output value feeds merely the neurons of the next layer and, thereby, becomes their input. Consequently, **the outputs of the neurons in the last layer are the outputs of the ANN**. The last layer of the ANN is also known as the **output layer** whereas all intermediate layers between the output and the input are the **hidden layers**. It is important to note that the inputs of the ANN, **x,** are connected to all the neurons of the **first hidden layer**. We illustrate this with an additional layer we call the **input layer**. **The input layer does not contain neurons as it only distributes the inputs to the first layer of neurons**. In summary, MLPs are

1) Layered because they are grouped in layers;

2) Feed-forward because their connections are unidirectional and always forward (from a previous layer to the next); and

3) Fully connected because every neuron is connected to all neurons of the next layer.



An MLP example with three inputs, one hidden layer containing four hidden neurons and two outputs. The ANN has labeled and ordered neurons and example connection weight labels. Bias weights b_j are not illustrated in this example but are connected to each neuron j of the ANN.

Forward Operation

In the previous section we defined the core components of an ANN whereas in this section we will see how we compute the output of the ANN when an input pattern is presented. The process is called **forward operation** and **propagates the inputs of the ANN throughout its consecutive layers to yield the outputs**. The basic steps of the forward operation are as follows:

1- Label and order neurons. We typically start numbering at the input layer and increment the numbers towards the output layer (see above Figure). Note that the input layer does not contain neurons, nevertheless is treated as such for numbering purposes only.

2- Label connection weights assuming that $\mathbf{w_{ij}}$ is the connection weight from neuron i (pre-synaptic neuron) to neuron j (post-synaptic neuron). Label bias weights that connect to neuron j as $\mathbf{b_j}$

3- Present an input pattern $\mathbf{x}$.

4- For each neuron j compute its output as follows: $\alpha_j = g(\sum_i \{w_{ij}\ \alpha_i\} + b_j)$, where α_j and α_i are, respectively, the **output** of and the **inputs** to neuron **j** (n.b. $\alpha_i = x_i$ in the input layer); **g** is the activation function (usually the logistic **sigmoid function**).

5- The outputs of the neurons of the output layer are the outputs of the ANN.

How Does an ANN Learn?

How do we approximate **f(x, w, b)** so that the outputs of the ANN match the desired outputs (labels) of our dataset, y? We will need a **training algorithm that adjusts the weights (w and b)** so that **f : x → y**. A training algorithm as such requires two components:

First, it requires a cost function to evaluate the quality of any set of weights.

Second, it requires a search strategy within the space of possible solutions (i.e., the weight space).

Cost (Error) Function

Before we attempt to adjust the weights to approximate f, we need some measure of MLP performance. The most common performance measure for training ANNs in a supervised manner is the **squared Euclidean distance (error)** between the vectors of the **actual output of the ANN** (**α**) and the **desired labeled output** (**y**).

$$E = \frac{1}{2}\sum_j (y_j - \alpha_j)^2$$

where the sum is taken over all the output neurons (the neurons in the final layer). Note that the y_j labels are **constant values** and more importantly, also note that **E** is a function of all the **weights** of the ANN since the actual outputs depend on them. As we will see below, ANN training algorithms build strongly upon this relationship between **error** and **weights**.

Backpropagation

The **backpropagation** (or backprop) **algorithm** is based on **gradient descent optimization** and is arguably the **most common algorithm for training ANNs**. Backpropagation **stands for backward propagation of errors as it calculates weight updates that minimize the error function from the output to the input layer**.

In a nutshell, backpropagation computes the partial derivative (gradient) of the error function E with respect to each weight of the ANN and adjusts the weights of the ANN following the (opposite direction of the) gradient that minimizes E.

As mentioned earlier, the squared Euclidean error depends on the weights as the ANN output which is essentially the f (x; w, b) function. As such we can calculate the gradient of E with respect to any weight $\frac{\theta E}{\theta wij}$ and any bias weight $\frac{\theta E}{\theta bij}$ in the ANN, which in turn will determine the degree to which the error will change if we change the weight values. We can then determine how much of such change we desire through a parameter $\eta \in [0,1]$ called **learning rate**.

In the absence of any information about the general shape of the function between the error and the weights but the existence of information about its gradient it appears that a gradient descent approach would seem to be a good fit for attempting to find the **global minimum** of the E function. Given the lack of information about the E function, the search can **start from some random point in the weight space** (i.e., random initial weight values) and follow the gradient towards lower E values. This process is repeated **iteratively** until we reach E values we are happy with or we run out of computational resources. More formally, the basic steps of the backpropagation algorithm are as follows:

1. Initialize **w** and **b** to random (commonly small) values.
2. For each training pattern (input-output pair):
   (a) Present input pattern **x**, ideally normalized to a range (e.g., $[0, 1]$).
   (b) Compute ANN actual outputs α_j using the forward operation.
   (c) Compute E
   (d) Compute error derivatives with respect to each weight $\frac{\theta E}{\theta w_{ij}}$ and bias weight $\frac{\theta E}{\theta b_j}$ of the ANN from the output all the way to the input layer.
   (e) Update weights and bias weights as $\Delta w_{ij} = -\eta \frac{\theta E}{\theta w_{ij}}$ and $\Delta b_j - \eta \frac{\theta E}{\theta b_j}$, respectively.
3. If E is *small* or you are out of *computational budget*, stop! Otherwise go to step 2.

Limitations and Solutions

It is worth noting that backpropagation is not **guaranteed** to find the **global minimum** of E given its local search (hill-climbing) property. Further, given its gradient-based (local) search nature, the algorithm fails to overcome potential plateaux areas in the error function landscape. As these are areas with near-zero gradient, crossing them results in near-zero weight updates and further in premature convergence of the algorithm. Typical solutions and enhancements of the algorithm to overcome convergence to local minima include:

• Random restarts: One can rerun the algorithm with new random connection weight values in the hope that the ANN is not too dependent on luck. No ANN model is good if it depends too much on luck—for instance, if it performs well only in one or two out of ten runs.

• Dynamic learning rate: One can either modify the learning rate parameter and observe changes in the performance of the ANN or introduce a dynamic learning rate parameter that increases when convergence is slow whereas it decreases when convergence to lower E values is fast.

• Momentum: Alternatively, one may add a momentum amount to the weight up-date rule as follows:

$$\Delta w_{ij}^{(t)} = m\Delta w_{ij}^{(t-1)} - \eta \frac{\theta E}{\theta w_{ij}}$$

where $\mathbf{m \in [0,1]}$ is the momentum parameter and $\mathbf{t}$ is the iteration step of the weight update. The addition of a momentum value of the previous weight up-date ($\mathbf{a\Delta w_{ij}^{(t-1)}}$) attempts to help backpropagation to overcome a potential local **minimum**. While the above solutions are directly applicable to ANNs of small size, practical wisdom and empirical evidence with modern (deep) ANN architectures, however, suggests that the above drawbacks are largely eliminated.