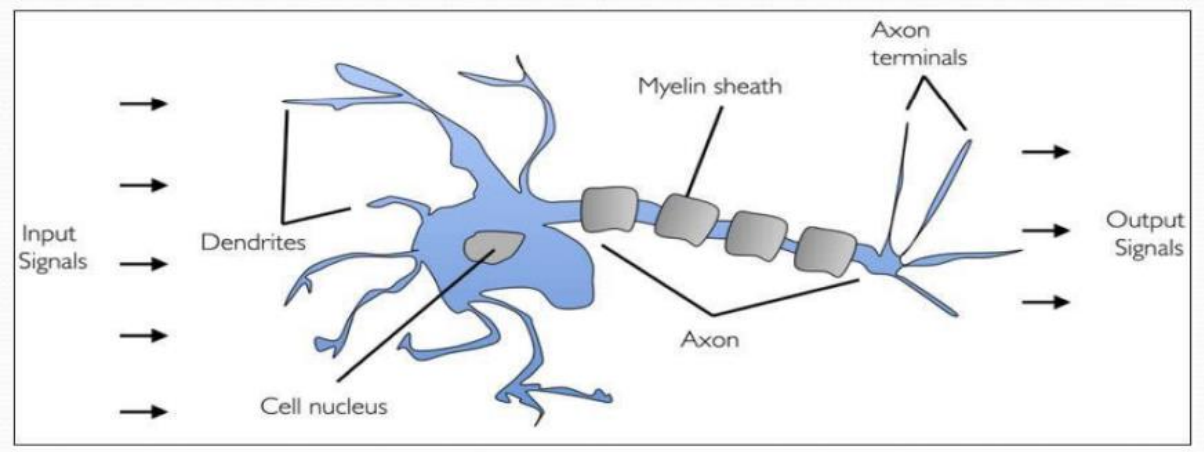
PERCEPTION:

A perceptron is a simple model of a biological neuron in an artificial neural network. The perceptron algorithm was designed to classify visual inputs, categorizing subjects into one of two types and separating groups with a line. Classification is an important part of machine learning and image processing. Perceptron was introduced by Frank Rosenblatt in 1957. Each perceptron sends multiple signals, one signal going to each perceptron in the next layer. For each signal, the perceptron uses different weights. One difference between an MLP and a neural network is that in the classic perceptron, the decision function is a step function and the output is binary. Perceptron is a single layer neural network and a multilayer perceptron is called Neural Networks. Perceptron is a linear classifier (binary). It is used in supervised learning. It helps to classify the given input data.

Biological Neurons:

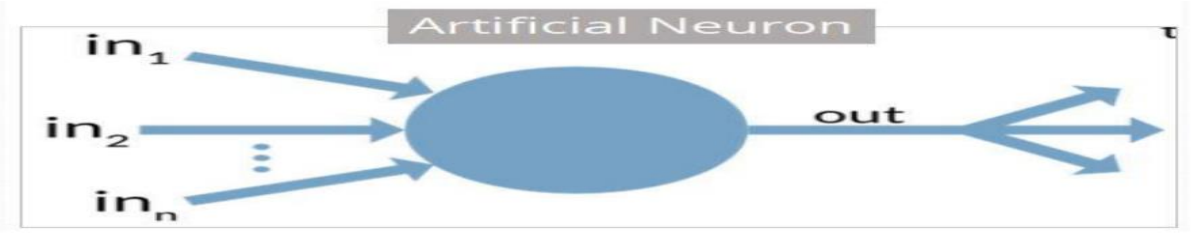
Cell nucleus or soma processes the information received from dendrites. Axon is a cable that is used by neurons to send information. Synapse is the connection between an axon and other neuron dendrites.



Researchers Warren McCullock and Walter Pitts published their first concept of simplified brain cell in 1943. This was called McCullock-Pitts (MCP) neuron. They described such a nerve cell as a simple logic gate with binary outputs. Multiple signals arrive at the dendrites and are then integrated into the cell body. If the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

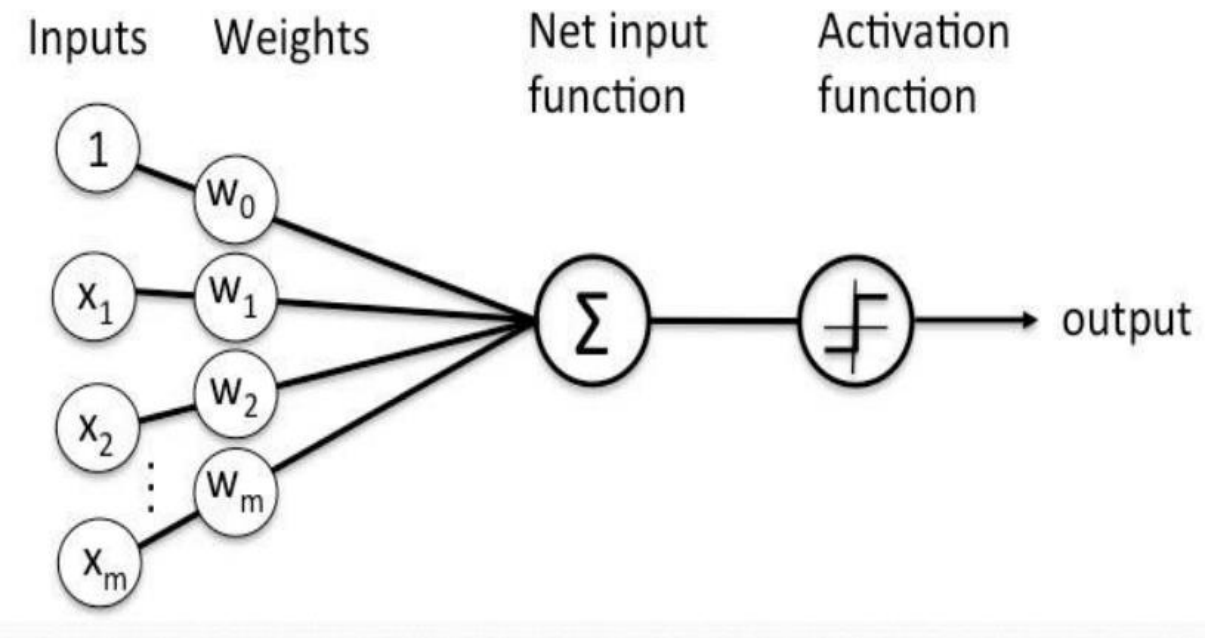
Artificial Neurons:

An artificial neuron is a mathematical function based on a model of biological neurons, where each neuron takes inputs, weighs them separately, sums them up and passes this sum through a nonlinear function to produce output.



Artificial Neurons has the following characteristics:

A neuron is a mathematical function modeled on the working of biological neurons.
It is an elementary unit in an artificial neural network.
One or more inputs are separately weighted.
Inputs are summed and passed through a nonlinear function to produce output.
Every neuron holds an internal state called activation signal.
Each connection link carries information about the input signal.
Every neuron is connected to another neuron via connection link.
Training Neural Network :
The neural network is a simple Perceptron, or a much more complicated multi-layer network with special activation functions.
We need to develop a systematic procedure for determining appropriate connection weights.
The network learn the appropriate weights from a representative set of training data.
The simplest cases, however, direct computation of the weights are unmanageable.
A good process is to start off with random initial weights and adjust them in small steps until the required outputs are generated.
A perceptron is a neural network unit that does certain computations to detect features or business intelligence in the input data. A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time. A single artificial neuron that computes its weighted input and uses a threshold activation function.



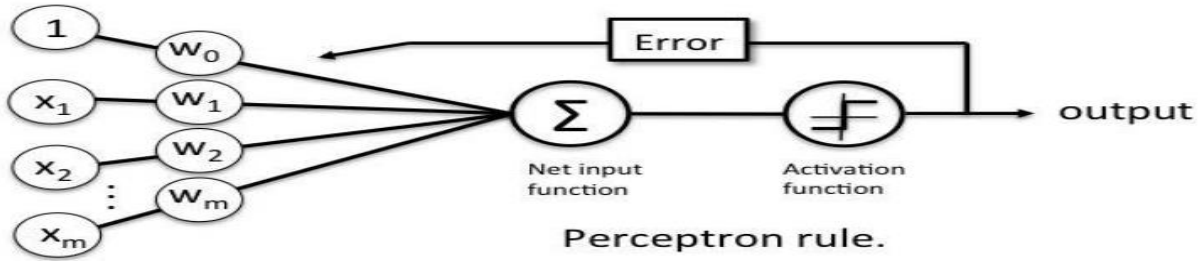
There are two types of perceptrons:

• Single layer Perceptrons can learn only linearly separable patterns.
• Multilayer Perceptrons or feedforward neural networks with two or more layers have the greater processing power.

The Perceptron algorithm learns the weights for the input signals in order to draw a linear decision boundary. • This enables you to distinguish between the two linearly separable classes +1 and -1.

The decision boundaries are hyperplanes, learning as the process of shifting around the hyperplanes, until each training pattern is classified correctly. Somehow, we need to formalise that process of "shifting around" into a systematic algorithm that can easily be implemented on a computer. The "shifting around" can be split up into a number of small steps. If the network weights at time t are wij(t) , then the shifting process corresponds to moving them by a small amount ⊗wij(t) so that at time t+1 we have weights. wij(t +1) = wij(t) + ⊗wij(t).It is convenient to treat the thresholds as weights, so we don't need separate equations for them.

Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients. The input features are then multiplied with these weights to determine if a neuron fires or not. The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. In the context of supervised learning and classification, this can then be used to predict the class of a sample.



Perceptron Function:

It is a function that maps its input "x," which is multiplied with the learned weight coefficient; an output value "f(x)"is generated.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

where

w = vector of real-valued weights.

b = bias (an element that adjusts the boundary away from origin without any dependence on the input value)

x = vector of input x values.

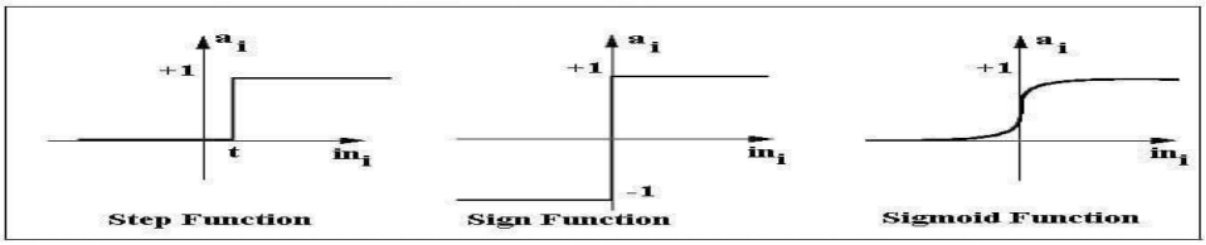
$$\sum_{i=1}^{m} w_i x_i$$

m= number of inputs to the perceptron.

The output can be represented as " 1 " or " 0." It can also be represented as " 1 " or "-1 " depending on which activation function is used.

Activation Functions of Perceptron:

The activation function applies a step rule (convert the numerical output into +1 or -1) to check if the output of the weighting function is greater than zero or not. Function gets triggered above a certain value of the neuron output; else it outputs zero. Sign Function outputs +1 or -1 depending on whether neuron output is greater than zero or not. Sigmoid is the S-curve and outputs a value between 0 and 1. For example: If Σwixi > 0 then final output =1 else final output = - 1.



Inputs of a Perceptron:

A Perceptron accepts inputs, moderates them with certain weight values, then applies the transformation function to output the final result. A Boolean output is based on inputs such as salaried, married, age, past credit profile, etc. It has only two values: Yes and No or True and False. The summation function "∑" multiplies all inputs of "x" by weights "w" and then adds them up as follows:

$W_0+W_1X_1+W_2X_2+W_3X_3+\ldots\ldots+W_nX_n$

Output of Perceptron

→Perceptron with a Boolean output:

→Input : $X_1\ldots\ldots X_n$

→Output: $O(X_1\ldots\ldots X_n)$

$$o(x_1,\ldots,x_n)=\begin{cases}1 \text{ if } w_0+w_1x_1+w_2x_2+\cdots+w_nx_n>0\\-1 \text{ otherwise}\end{cases}$$

→Weights Wi = contribution of input Xi to the perceptron output.

→Wo = bias or threshold.

→If ∑w.x > 0, output is +1, else -1. The neuron gets triggered only when weighted input reaches a certain threshold value.

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

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

→An output of +1 specifies that the neuron is triggered. An output of -1 specifies that the neuron did not get triggered.

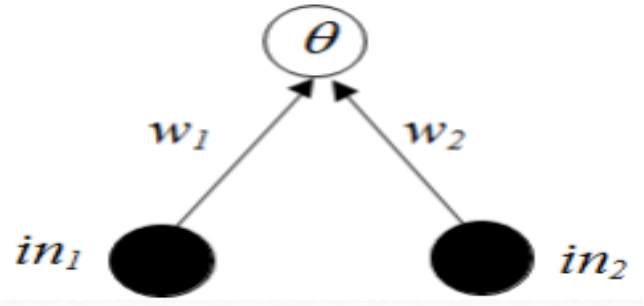
→"sgn" stands for sign function with output +1 or -1

Decision Boundary

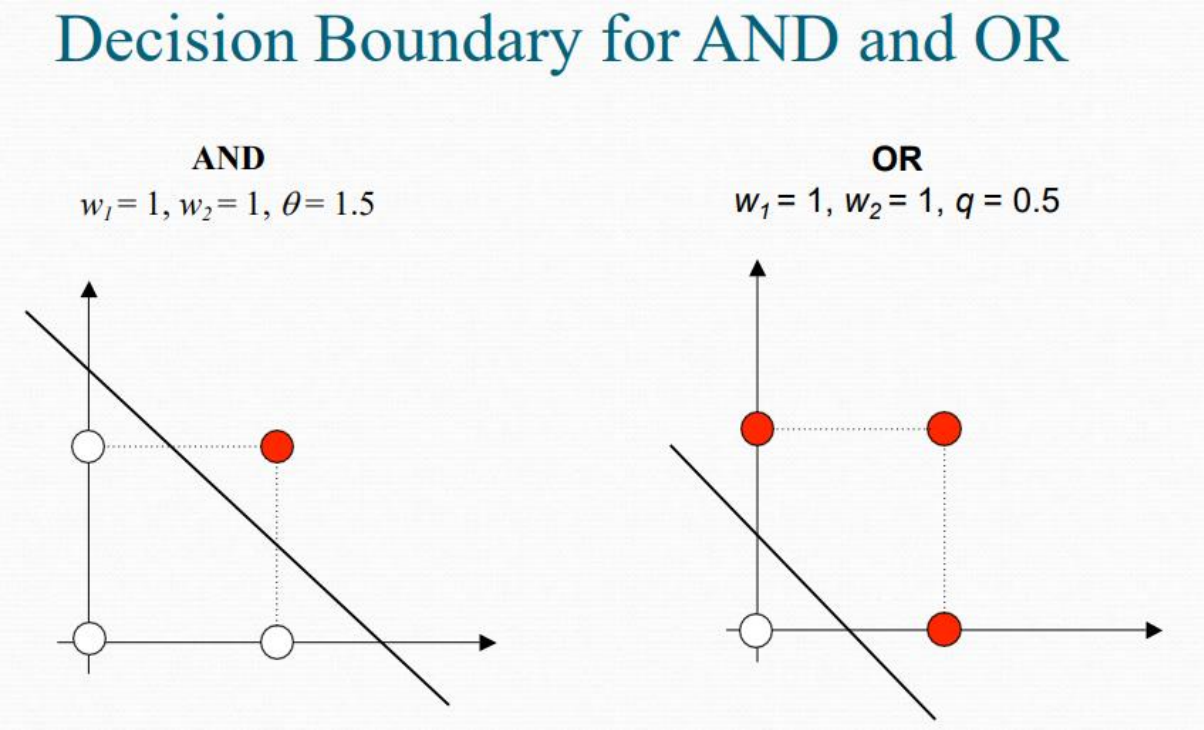
The weight vector is relevant to the decision boundary. The weight vector points in the direction of the vector which should produce an output of 1. Vector with the positive output are on the right side of the decision boundary. If w pointed in the opposite direction, the dot products of

all input vectors would have the opposite sign. Would result in same classification but opposite labels. The bias determines the position of the boundary. $W^TP + b = 0$ using one point on the decision boundary to find b.

Decision Boundary in Two Dimensions :It is easy to visualise what the neural network is doing. It is forming decision boundaries between classes. Out = step (w1 in1 + w2 in2 – θ)



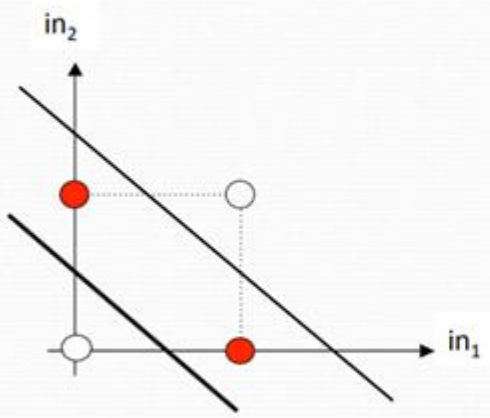
The decision boundary (between out = 0 and out =1) w1 in1 + w2 in2 – θ = 0. Therefore, in two dimensions the decision boundaries are always straight line.

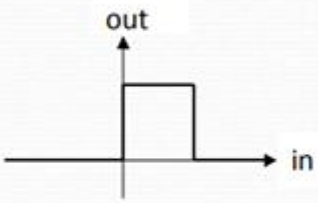


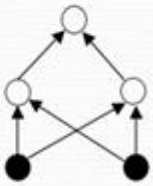
We can change the weights and thresholds without changing the output decisions

Decision Boudaries For XOR

- We need two straight lines to separate the different outputs/decisions:







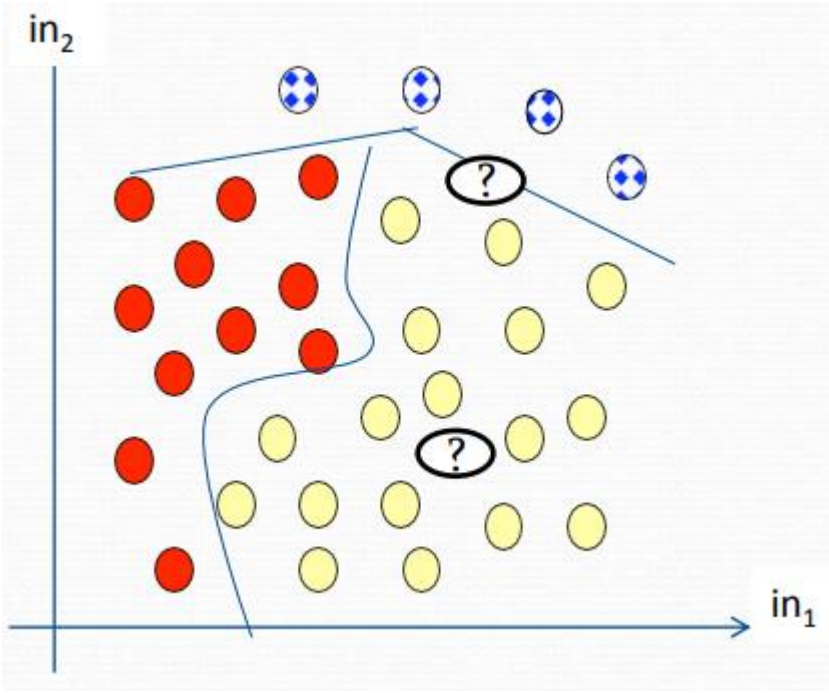
There are two examples:
- Either change the transfer function so that it has more than one decision boundary
- Use a more complex network that is able to generate more complex decision boundaries.

Decision Hyperplanes and Linear Separability : If user have two inputs, then the weights define a decision boundary that is a one dimensional straight line in the two dimensional input space of possible input values.If we have n inputs, the weights define a decision boundary that is an n–1 dimensional hyperplane in the n dimensional input space:

$$w_1 in_1 + w_2 in_2 + w_n in_n - \theta = 0$$

This hyperplane is linear (i.e., straight or flat or non-curved) and can only divide the space into two regions. We still need more complex transfer functions, or more complex networks, to deal with XOR type problems.Problems with input patterns that can be classified using a single hyperplane are said to be linearly separable. Problems (like XOR) which cannot be classified in this way are said to be non-linearly separable.

General Decision Boundaries: To deal with input patterns that are not binary, and expect our neural networks to form complex decision boundaries,Eg:



In figure classes are classified into three categories

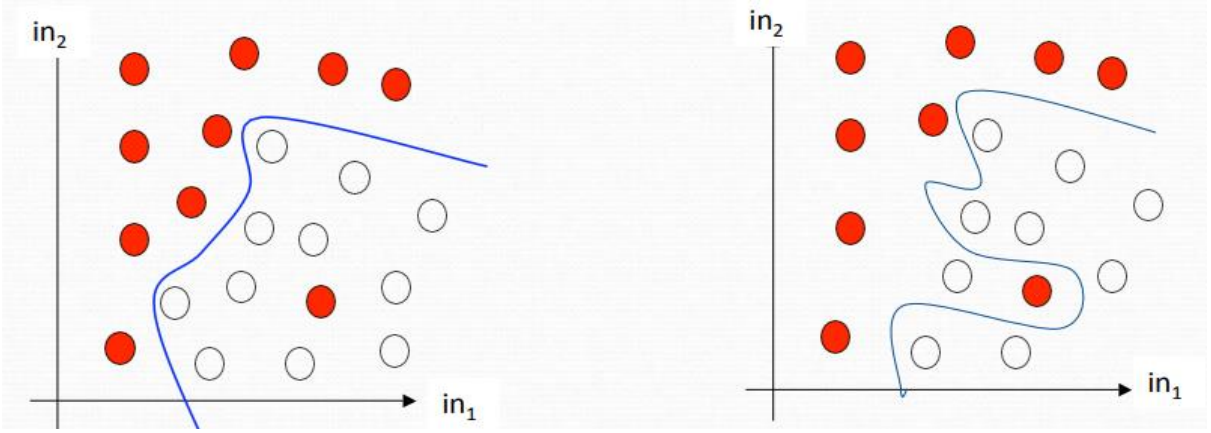
Learning and Generalization: A network produce outputs for input patterns that was not originally set up to classify (shown with question marks), though those classifications may be incorrect There are two important aspects of the network's operation :

Learning: The network must learn decision boundaries from a set of training patterns so that these training patterns are classified correctly.

Generalization: After training, the network must also be able to generalize, i.e. correctly classify test patterns it has never seen before.

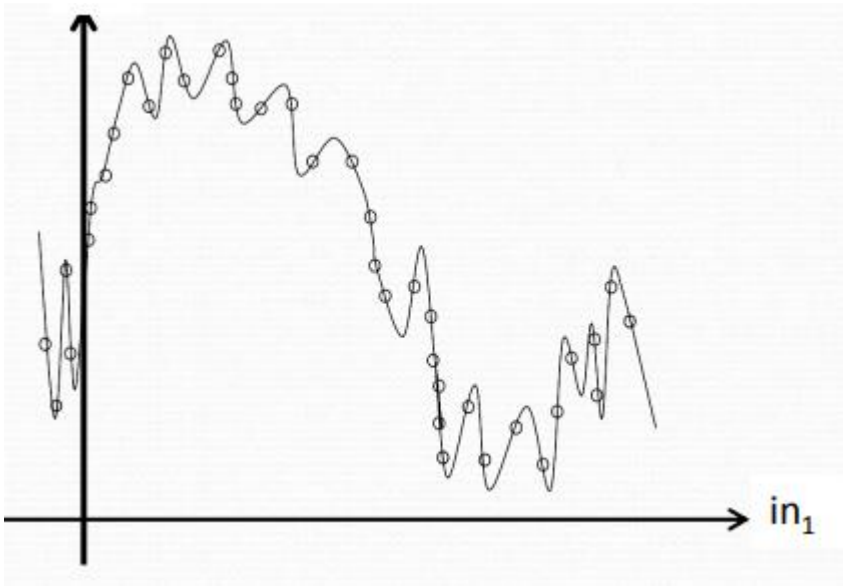
Sometimes, the training data may contain errors (e.g., noise in the experimental determination of the input values, or incorrect classifications). In this case, learning the training data perfectly may make the generalization worse.

Generalization in Classification :The objective of the neural network is to learn a classification decision boundary:



The objective is to get the network to generalize to classify new inputs appropriately. If the training data contains noise, we don't necessarily want the training data to be classified totally accurately, as that is likely to reduce the generalization ability.

Generalization in Function Approximation:
We wish to recover a function for which we only have noisy data samples:



• The neural network output give better representation of the under-lying function if its output curve does not pass through all the data points.
• A larger error on the training data is likely to lead to better generalization.

FEED FORWARD PROCESS

A multilayer feedforward neural network consists of a layer of input units, one or more layers of hidden units, and one output layer of units. A neural network that has no hidden units is called a Perceptron. However, a perceptron can only represent linear functions, so it isn't powerful enough for the kinds of applications we want to solve. On the other hand, a multilayer feedforward neural network can represent a very broad set of nonlinear functions. So, it is very useful in practice.

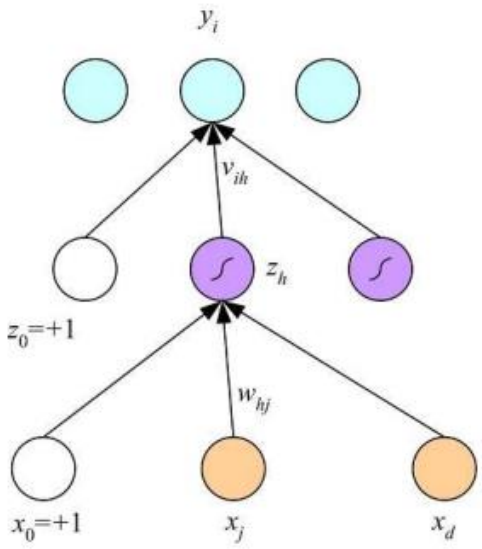


Fig. 1. Typical structure of a multilayer feedfoward artificial neural network. Here, there is one layer of input nodes (shown in the bottom row), one layer of hidden nodes (i.e., the middle row), and one layer of output nodes (at the top). The number of nodes per layer is application-dependent.

The most common network structure we will deal with is a network with one layer of hidden units, so for the rest of these notes, we'll make the assumption that we have exactly one layer of hidden units in addition to one layer of input units and one layer of output units. This structure is called multilayer because it has a layer of processing units (i.e., the hidden units) in addition to the output units. These networks are called feedforward because the output from one layer of neurons feeds forward into the next layer of neurons. There are never any backward connections, and connections never skip a layer.

Typically, the layers are fully connected, meaning that all units at one layer are connected with all units at the next layer. So, this means that all input units are connected to all the units in the layer of hidden units, and all the units in the hidden layer are connected to all the output units. Usually, determining the number of input units and output units is clear from your application. However, determining the number of hidden units is a bit of an art form, and requires experimentation to determine the best number of hidden units.

Too few hidden units will prevent the network from being able to learn the required function, because it will have too few degrees of freedom. Too many hidden units may cause the network to tend to overfit the training data, thus reducing generalization accuracy. In many applications, some minimum number of hidden units is needed to learn the target function accurately, but extra hidden units above this number do not significantly affect the generalization accuracy, as long as cross validation techniques are used (described later).

Too many hidden units can also significantly increase the training time. Each connection between nodes has a weight associated with it. In addition, there is a special weight (called w0) that feeds into every node at the hidden layer and a special weight (called z0) that feeds into every node at the output layer. These weights are called the bias, and set the thresholding values for the nodes. We'll come back to this later. Initially, all of the weights are set to some small random values near zero. The training of our network will adjust these weights (using the Backpropagation algorithm that we'll describe later) so that the output generated by the network matches the correct output.

Every node in the hidden layer and in the output layer processes its weighted input to produce an output. This can be done slightly differently at the hidden layer, compared to the output layer. Here's how it works.

(i) Input units The input data you provide your network comes through the input units. No processing takes place in an input unit – it simply feeds data into the system. For example, if you are inputting a grayscale image (e.g., a grayscale picture of your pet Fido) to your network, your picture will be divided into pixels (say, 120 x 128 pixels), each of which is represented by a number (typically in the range from 0 to 255) that says what the grayscale value is for that piece of the image. One pixel (i.e., a number from 0 to 255) will be fed into each input unit. So, if you have an image of 120 x 128 pixels, you'll have 15360 input units2 . The value coming out of an input unit is labeled xj , for j going from 1 to d, representing d input units. There is also a special input unit labeled x0, which always has the value of 1. This is used to provide the bias to the hidden nodes.

(ii)Hidden units The connections coming out of an input unit have weights associated with them. A weight going to hidden unit zh from input unit xj would be labeled whj . The bias input node, x0, has a weight of w0. In the training, this bias weight, w0, is treated like all other weights, and is updated according to the backpropagation algorithm we'll discuss later. Remember, the value coming out of x0 is always 1. Each hidden node calculates the weighted sum of its inputs and applies a thresholding function to determine the output of the hidden node. The weighted sum of the inputs for hidden node zh is calculated as: $\sum_{j=0}^{d} w_{hj} x_j$

The thresholding function applied at the hidden node is typically either a step function or a sigmoid function. For our purposes, we'll stick with the sigmoid function. The general form of the sigmoid function is:

$$\text{sigmoid}(a) = \frac{1}{1+e^{-a}}$$

C. Output units Now, we can do a similar computation for the output nodes. The difference is that the exact way we compute our output depends on the type of problem we're solving – either a regression problem or a classification problem. And, the calculation also depends on whether we have 1 output unit or multiple output units. We start out the same as we did with the hidden units, calculating the weighted sum. We label the weights going into output unit i from hidden unit h as vih. Just like the input layer, we also have a bias at the hidden layer. So, each output unit has a bias input from hidden unit z0, where the input from z0 is always 1 and the weight associated with that input is trained just like all the other weights. So, output unit i computes the weighted sum of its inputs as:

$$o_i = \sum_{h=0}^{H} v_{ih} z_h$$

If there is just one output unit, then we omit the i subscripts, and we have:

$$o = \sum_{h=0}^{H} v_h z_h$$

Now, we have to decide what function we're going to apply to this weighted sum to generate yi , which is the output of unit i. We'll look at four cases:

1) Regression with a single output. This is the problem of learning a function, where the single output corresponds to the value of the function for the given input. Here, because we are learning a function, we do not want our output to be "squashed" to be between 0 and 1 (which is what the sigmoid function does). Instead, we just want the regular, unthresholded output. So, in this case, we calculate the output unit value of y as simply the weighted sum of its inputs:

$$y = o = \sum_{h=0}^{H} v_h z_h$$

Note here that we call this y instead of yi because we only have 1 output unit.

2) Regression with multiple (i.e., K) outputs. This is the regression problem applied to several functions at once; that is, we learn to approximate several functions at once, and each output corresponds to the output of one of those functions. Similar to the previous case, we don't want to squash the output. However, here, we have multiple output nodes. So, we calculate the output value of unit yi as:

$$y_i = o_i = \sum_{h=0}^{H} v_{ih} z_h$$

3) Classification for 2 classes. This is the problem of discriminating between two classes. Since one node can output either a 0 or a 1, we can have one class correspond to a 0 output, and the other class correspond to an output of 1. Here, we do want our output to be squashed between 0 and 1. So, we apply the sigmoid function at our output unit, y, to get the following output:

$$y = \text{sigmoid}(o) = \text{sigmoid}\left(\sum_{h=0}^{H} v_h z_h\right) = \frac{1}{1+e^{-\sum_{h=0}^{H} v_h z_h}}$$

4) Classification for K > 2 classes. Here, we typically have K output nodes, for K classes. We usually have one ouput node per class, instead of having log2(K) output nodes for a couple of reasons. First, our network will have a more expressive space to find a function when we have more weights to learn. Second, with this structure, we can get information on the second choice

of the network (e.g., if the first output node gives value of 0.9 and a second node gives a value of 0.8, we know that the network doesn't have strong "confidence" that the first class is the correct class, since 0.9 and 0.8 are both high values. We would know the network had high confidence if only one of the outputs were close to 1, and the rest were close to 0.) In this case, we do want our output values to be between 0 and 1, so we could apply the sigmoid function at each of the output nodes. However, we still have one more step to generate the ultimate answer of the network, which is to determine which of the output nodes has the largest value. Whichever node generates the largest value tells us which class the network believes the input belongs to. We could do this by applying the max function to the outputs. However, a nicer way of doing this is to apply the "softmax" function to the weighted sum. The softmax function has the effect of making the maximum value of the outputs to be close to 1 and the rest to be close to 0. An added bonus is that it is differentiable, which is nice for some theoretical proofs (but which we'll skip!). So, we'll calculate the output of node yi as:

$$y_i = \text{softmax}(o_i) = \frac{e^{o_i}}{\sum_{i=1}^{K} e^{o_i}} = \frac{e^{\sum_{h=0}^{H} v_{ih} z_h}}{\sum_{i=1}^{K} e^{\sum_{h=0}^{H} v_{ih} z_h}}$$

Training your neural network to produce the correct outputs for the given inputs is an iterative process, in which you repeatedly present the network with an example, compare the output on this example (sometimes called the actual output) with the desired output (sometimes called the target output), and adjust the weights in the network to (hopefully) generate better output the next time (i.e., output that is closer to the correct answer). By training the network over and over with various examples, and using the Backpropagation algorithm (which we'll talk about in a minute) to adjust the weights, the network should learn to produce the correct answer. Ideally, the "correct answer" is not just the right answer for the data that you train your network on, but also for generalizations of that data. (For example, if you train your network on pictures of all the cats in your neighborhood, you still want it to recognize Morris (the 9 Lives cat, remember?) as a cat, not a dog. Hmmm ... I'm assuming Morris doesn't live in your neighborhood.) You train your network using a data set of examples (called training data). For each example, you know the correct answer, and you tell the network this correct answer. We will call the process of running 1 example through your network (and training your network on that 1 example) a weight update iteration. Training your network once on each example of your training set is called an epoch. Typically, you have to train your network for many epochs before it converges, meaning that the network has settled in on a function that it thinks is the best predictor of your input data. More about convergence later.

A. Backpropagation Algorithm The algorithm we'll use to train the network is the Backpropagation Algorithm. The general idea with the backpropagation algorithm is to use gradient descent to update the weights so as to minimize the squared error between the network output values and the target output values. The update rules are derived by taking the partial derivative of the error function with respect to the weights to determine each weight's contribution to the error. Then, each weight is adjusted, using gradient descent, according to its contribution to the error.

We won't go into the actual derivations here – you can find that in your text and in other sources. This process occurs iteratively for each layer of the network, starting with the last set of weights, and working back towards the input layer (hence the name backpropagation). 1) Offline versus Online Learning: Before we get to the details of the algorithm, though, we need to make clear a distinction between "offline" and "online" learning. "Offline" learning, in the context of this discussion, occurs when you compute the weight updates after summing over all of the training examples. "Online" learning is when you update the weights after each training example. The theoretical difference between the two approaches is that offline learning

implements what is called Gradient Descent, whereas online learning implements Stochastic Gradient Descent (also called Incremental Gradient Descent).
The general approach for the weight updates is the same, whether online or offline learning is used. The only difference is that offline learning will sum the error over all inputs, while the online learning will compute the error for each input (one at a time). Keep this in mind when reading Chapter 11 of the Alpaydin Machine Learning textbook; whenever you see P t , this means the calculation is over all t input examples, not just one example. You can convert almost any equation in this chapter from offline to online by just eliminating P t from the equation. 2) Online Weight Updates: As with the output calculation, the weight update calculation depends on the type of problem we're trying to solve. Remember, we're looking at 4 cases, and we're calculating the weight updates given a single instance (x t , rt), where x t is the input, r t is the target output, and y t is the actual output of the network. Here, the t superscript just means the current example that the network is training on. In these weight updates, we also use a positive constant learning rate, η, that moderates 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. Here are the 4 situations:

1) **Regression with a single output.** The weight updates for this case are:

$$\begin{aligned} \Delta v_h &= \eta(r^t - y^t)z_h^t \\ \Delta w_{hj} &= \eta(r^t - y^t)v_h z_h^t(1 - z_h^t)x_j^t \end{aligned}$$

2) **Regression with multiple (i.e., K) outputs.** The weight updates for this case are:

$$\begin{aligned} \Delta v_{ih} &= \eta(r_i^t - y_i^t)z_h^t \\ \Delta w_{hj} &= \eta\left(\sum_{i=1}^{K}(r_i^t - y_i^t)v_{ih}\right)z_h^t(1 - z_h^t)x_j^t \end{aligned}$$

3) **Classification for 2 classes.** The weight updates for this case are:

$$\begin{aligned} \Delta v_h &= \eta(r^t - y^t)z_h^t \\ \Delta w_{hj} &= \eta(r^t - y^t)v_h z_h^t(1 - z_h^t)x_j^t \end{aligned}$$

4) **Classification for $K > 2$ classes.** The weight updates for this case are:

$$\begin{aligned} \Delta v_{ih} &= \eta(r_i^t - y_i^t)z_h^t \\ \Delta w_{hj} &= \eta\left(\sum_{i=1}^{K}(r_i^t - y_i^t)v_{ih}\right)z_h^t(1 - z_h^t)x_j^t \end{aligned}$$

The Algorithm: Let's pick the classification problem for K > 2 cases as our example; then the algorithm for online backpropagation for this problem is given as Algorithm 1. This equation implements equation 9 for the output calculation, and equations 16 and 17 for the weight updates. You can change this algorithm to handle the other types of problems (i.e., regression or classification with 2 classes) by replacing the algorithm's output calculations on lines 1.8 - 1.12 with one of the equations 6, 7, or 8, as well as the weight updates, replacing the weight updates on lines 1.15 and 1.20 with the appropriate equations from subsection IV-A.2. Note that Figure 11.1 in the Alpaydin text implements the backpropagation algorithm for regression with multiple outputs, which is different from the algorithm given here.

B. Calculating the Error Determining how well your network is approximating the desired output requires that you measure the error of the network. Many error functions are possible, but the most common error function used is the sum of squared errors. We can measure this error for one output unit for one training example (x t , rt) as:

$$E(\mathbf{W},\mathbf{v}|x^t, r^t) = \frac{1}{2}(r^t - y^t)^2$$

This equation says, “the error for a set of weights W and v, given a particular example (x t , rt) is one-half the sum of the squared difference between the desired output and the actual output.” When we have multiple output units, we simply sum the error over all the output units, as follows (again, for just 1 training example):

$$E(\mathbf{W},\mathbf{v}|x^t, r^t) = \frac{1}{2}\sum_{i=1}^{K}(r_i^t - y_i^t)^2$$

If we want to calculate the error for one output node (say, output yi) for 1 complete epoch (i.e., for one pass of all the training examples, we simply sum the error for that output unit over all training examples, as follows:

$$E(\mathbf{W},\mathbf{v}|\mathcal{X}) = \frac{1}{2}\sum_{(x^t,r^t)\in\mathcal{X}}(r_i^t - y_i^t)^2$$

If we want to calculate the error for the entire network for 1 complete epoch (i.e., for one pass of all the training examples), we simply sum the error over all output units over all training examples, as follows:

$$E(\mathbf{W},\mathbf{v}|\mathcal{X}) = \frac{1}{2}\sum_{(x^t,r^t)\in\mathcal{X}}\left(\sum_{i=1}(r_i^t - y_i^t)^2\right)$$

When the error during validation is to be calculated the above equation must be used for determining the error pf network for all validation sets.

C. Determining Convergence

The best way to determine whether your network has reached the best set of weights for your training data is to validate the results using a validation set of data. This is a separate data set that you do not use during training. Instead, you use the validation data to detect when your network is beginning to overfit to the training data. Remember, you want your network to generalize its inputs, so that it can correctly answer regression or classification queries not only for your training data, but also for other examples. If you train your network too long, it will overfit to the training data, which means that it will correctly answer only examples that are in your training data set. So, to help ensure that your network does not overfit, you can use a cross validation procedure. This involves training your network for a while (i.e., several epochs), and then presenting your network with the validation data. Again, the validation data is a set of data that your network has never seen before. You keep track of the error of your network as it is presented with the validation data using equation 21, but you DO NOT update the network weights during this procedure. You repeat the process of training and validating, keeping track of the errors in both cases, and you declare convergence when your validation error is consistently growing. As you go, you need to save the weights of your network when the validation error is decreasing. This way, once the validation errors begin going up, you’ll have the weights to go back to that give you the best performance. Keep in mind that it is possible for the validation error to grow for a while, but then begin decreasing again (before perhaps going up again). So, you have to analyze the trend of your validation error to convince yourself that it is consistently growing before you halt the training of your network. Once you’ve declared convergence, you use your saved weights (i.e., from when the validation error was the lowest) for your final network. If you have enough data, one easy way to handle the validation is to divide your dataset into 3 parts. The first part is used for training, the second part is used for validation, and the third part is used at the very end to see how well your network has truly learned, on examples it has never seen before. However, there are other ways of validating your network, too, such as k−fold cross validation. This approach is especially good when you have a small data set. In this approach, you divide your data set into k pieces, and save one of these pieces for validation while you train on the other k − 1 pieces. You keep track of how many iterations it takes to converge. Then, you repeat this process for all k pieces (selecting a

different piece for validation each time), keeping track of the number of iterations to convergence each time. After all k training stages are complete, you compute the average number of iterations to converge over the k stages, which we'll call i. Finally, you train one last time on all the data for i iterations, and this is your final network.

D. Momentum Gradient descent is generally a slow process, taking a long time to converge. One easy way to speed the learning process is to use momentum. Momentum takes into account the previous weight update when making a current weight update. So, you must save the updates made for each weight for 1 time step. Then, on the next iteration of weight updates, you make use of this previous update information. Recall that our old weight updates were as follows:

$$\begin{aligned} v_{ih} &= v_{ih} + \Delta v_{ih} \\ w_{hj} &= w_{hj} + \Delta w_{hj} \end{aligned}$$

So, to add momentum, your new weight update equations become:

$$\begin{aligned} v_{ih}^{t} &= v_{ih}^{t} + \Delta v_{ih}^{t} + \alpha \Delta v_{ih}^{t-1} \\ w_{hj}^{t} &= w_{hj}^{t} + \Delta w_{hj}^{t} + \alpha \Delta w_{hj}^{t-1} \end{aligned}$$

Here, the superscript t refers to the current training example and $t - 1$ refers to the previous training example. So, with momentum, you just add α times the previous update when adjusting your weights. Here, α is a constant called momentum, with $0 \leq \alpha < 1$.