1. **Artificial neural networks ?**

**Artificial neural networks** (**ANNs**) or connectionist**systems** are computing systems vaguely inspired by the biological neural networks that constitute animal brain.Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge about cats, e.g., that they have fur, tails, whiskers and cat-like faces. Instead, they automatically generate identifying characteristics from the learning material that they process.

An ANN is based on a collection of connected units or nodes called artificial neurons which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it.

In common ANN implementations, the signal at a connection between artificial neurons is a real number, and the output of each artificial neuron is computed by some non-linear function of the sum of its inputs. The connections between artificial neurons are called 'edges'. Artificial neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Artificial neurons may have a threshold such that the signal is only sent if the aggregate signal crosses that threshold. Typically, artificial neurons are aggregated into layers. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

The original goal of the ANN approach was to solve problems in the same way that a human brain would. However, over time, attention moved to performing specific tasks, leading to deviations from biology. ANNs have been used on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, playing board and video games and medical diagnosis.

1. **Model**

An *artificial neural network* is a network of simple elements called *artificial neurons*, which receive input, change their internal state (*activation*) according to that input, and produce output depending on the input and activation. The *network* forms by connecting the output of certain neurons to the input of other neurons forming a directed, weighted graph. The weights as well as the functions that compute the activation can be modified by a process called *learning* which is governed by a *learning rule.*

* 1. **Components of an artificial neural network**

1. **Neurons**

A neuron with label receiving an input (t) from predecessor neurons consists of the following components:

* an *activation (t),* depending on a discrete time parameter
* possibly a *threshold ,* which stays fixed unless changed by a learning function
* an *activation function* that computes the new activation at a given time  ***t + 1*** from (t), and the net input (t) giving rise to the relation (t+1) =
* and an *output function* {\displaystyle f\_{out}} computing the output from the activation = .

Often the output function is simply the Identity function.

An *input neuron* has no predecessor but serves as input interface for the whole network. Similarly an *output neuron* has no successor and thus serves as output interface of the whole network.

1. **Connections and weights**

The *network* consists of connections, each connection transferring the output of a neuron to the input of a neuron . In this sense s the predecessor of and is the successor of . Each connection is assigned a weight .

1. **Propagation function**

The *propagation function* computes the *input* to the neuron from the outputs of predecessor neurons and typically has the form.

1. **Learning rule**

The *learning rule* is a rule or an algorithm which modifies the parameters of the neural network, in order for a given input to the network to produce a favored output. This *learning* process typically amounts to modifying the weights and thresholds of the variables within the network.

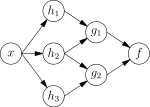
* 1. **Neural networks as functions**

Neural network models can be viewed as simple mathematical models defining a function or a distribution over or both  and . Sometimes models are intimately associated with a particular learning rule. A common use of the phrase "ANN model" is really the definition of a *class* of such functions (where members of the class are obtained by varying parameters, connection weights, or specifics of the architecture such as the number of neurons or their connectivity).

Mathematically, a neuron's network function is defined as a composition of other functions , that can further be decomposed into other functions. This can be conveniently represented as a network structure, with arrows depicting the dependencies between functions. A widely used type of composition is the *nonlinear weighted sum*, where  , where (commonly referred to as the activation function) is some predefined function, such as the hyperbolic tangent or sigmoid function or softmax function or rectifier function. The important characteristic of the activation function is that it provides a smooth transition as input values change, i.e. a small change in input produces a small change in output. The following refers to a collection of functions as vector .

This figure depicts such a decomposition of , with dependencies between variables indicated by arrows. These can be interpreted in two ways.

The first view is the functional view: the input is transformed into a 3-dimensional vector ,  which is then transformed into a 2-dimensional vector , which is finally transformed into . This view is most commonly encountered in the context of optimization.

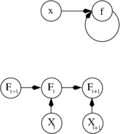


*ANN dependency graph*

The second view is the probabilistic view: the random variable depends upon the random variable , which depends upon  , which depends upon the random variable . This view is most commonly encountered in the context of graphical models.

The two views are largely equivalent. In either case, for this particular architecture, the components of individual layers are independent of each other (e.g., the components of are independent of each other given their input ). This naturally enables a degree of parallelism in the implementation.

Networks such as the previous one are commonly called feedforward, because their graph is a directed acyclic graph. Networks with cycles are commonly called recurrent. Such networks are commonly depicted in the manner shown at the top of the figure, where is shown as being dependent upon itself. However, an implied temporal dependence is not shown.



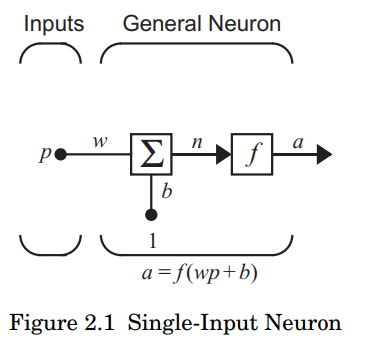
Two separate depictions of the recurrent ANN dependency graph

* 1. **Neuron model**

1. **Single-input neuron**

A single-input neuron is shown in Figure 2.1. The scalar input is multiplied by the scalar *weight to form ,* one of the terms that is sent to the summer. The other input, , is multiplied by a *bias* and then passed to the summer. The summer output , often referred to as the *net input*, goes into a *transfer function ,* which produces the scalar neuron output . (Some authors use the term “activation function” rather than *transfer function* and “offset” rather than *bias*.)

If we relate this simple model back to the biological neuron, the weight corresponds to the strength of a synapse, the cell body is represented by the summation and the transfer function, and the neuron output represents the signal on the axon.



The neuron output is calculated as

If, for instance, , then

The actual output depends on the particular transfer function that is chosen. We will discuss transfer functions in the next section.

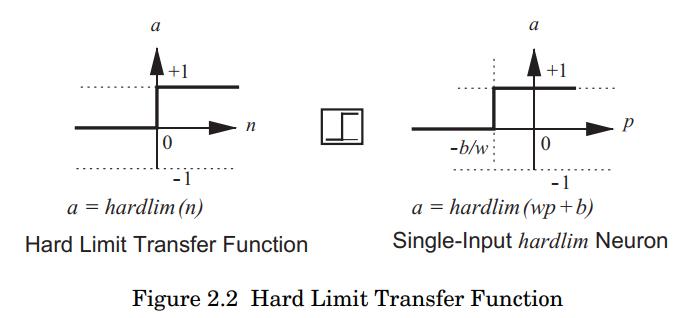
The bias is much like a weight, except that it has a constant input of 1. However, if you do not want to have a bias in a particular neuron, it can be omitted.

Note that and are both *adjustable* scalar parameters of the neuron. Typically the transfer function is chosen by the designer and then the parameters and will be adjusted by some learning rule so that the neuron input/output relationship meets some specific goal. As described in the following section, we have different transfer functions for different purposes.

**Transfer Functions**

The transfer function in Figure 2.1 may be a linear or a nonlinear function of . A particular transfer function is chosen to satisfy some specification of the problem that the neuron is attempting to solve.

The *hard limit transfer function*, shown on the left side of Figure 2.2, sets the output of the neuron to 0 if the function argument is less than 0, or 1 if its argument is greater than or equal to 0. We will use this function to create neurons that classify inputs into two distinct categories.

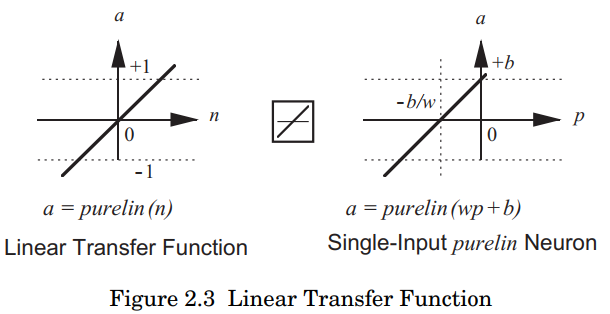


The graph on the right side of Figure 2.2 illustrates the input/output characteristic of a single-input neuron that uses a hard limit transfer function. Here we can see the effect of the weight and the bias. Note that an icon for the hard limit transfer function is shown between the two figures. Such icons will replace the general in network diagrams to show the particular transfer function that is being used.

The output of a *linear transfer function* is equal to its input:

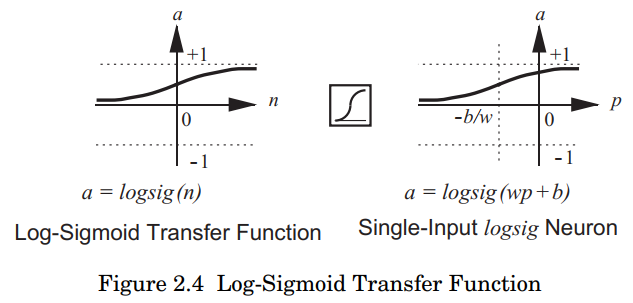
as illustrated in Figure 2.3.

Neurons with this transfer function are used in the ADALINE networks



The output versus input characteristic of a single-input linear neuron with a bias is shown on the right of Figure 2.3.

The *log-sigmoid transfer function* is shown in Figure 2.4.

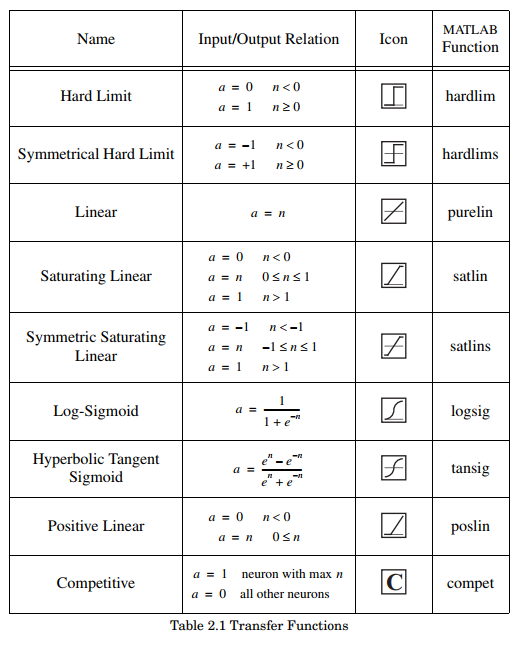


This transfer function takes the input (which may have any value between  
plus and minus infinity) and squashes the output into the range 0 to 1, according to the expression:

The log-sigmoid transfer function is commonly used in multilayer networks that are trained using the backpropagation algorithm, in part because this function is differentiable.

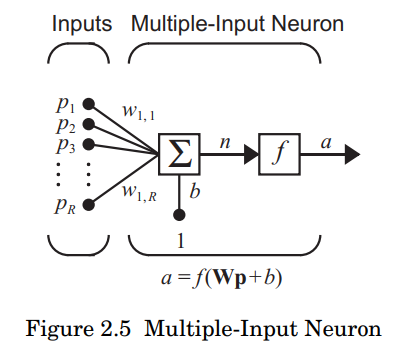
Most of the transfer functions are summarized in Table 2.1. Of course, you can define other transfer functions in addition to those shown in Table 2.1 if you wish.

*To experiment with a single-input neuron, use the Neural Network Design Demonstration One-Input Neuron* **nnd2n1**.

**

1. Multiple-Input neuron

Typically, a neuron has more than one input. A neuron with inputs is shown in Figure 2.5. The individual inputs are each weighted by corresponding elements of the *weight matrix* ***.***

******

The neuron has a bias , which is summed with the weighted inputs to  
form the net input :

This expression can be written in matrix form:

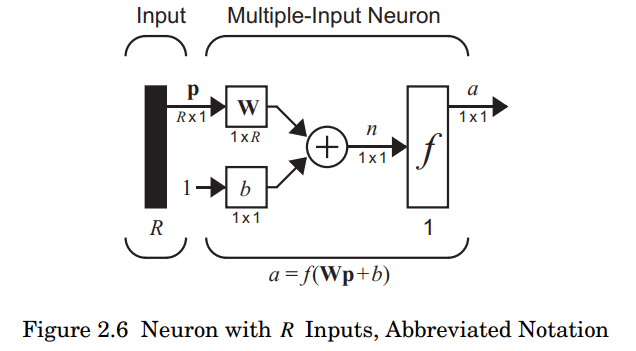
Where the matrix for the single neuron case has only one row.

Now the neuron output can be written as

Fortunately, neural networks can often be described with matrices. Don’t be concerned if you are rusty with matrix and vector operations.

We have adopted a particular convention in assigning the indices of the elements of the weight matrix. The first index indicates the particular neuron destination for that weight. The second index indicates the source of the signal fed to the neuron. Thus, the indices in say that this weight represents the connection *to* the first (and only) neuron *from* the second source. Of course, this convention is more useful if there is more than one neuron.

We would like to draw networks with several neurons, each having several inputs. Further, we would like to have more than one layer of neurons. You can imagine how complex such a network might appear if all the lines were drawn. It would take a lot of ink, could hardly be read, and the mass of detail might obscure the main features. Thus, we will use an *abbreviated notation*. A multiple-input neuron using this notation is shown in Figure 2.6.

As shown in Figure 2.6, the input vector is represented by the solid vertical bar at the left. The dimensions of are displayed below the variable as , indicating that the input is a single vector of elements. These inputs go to the weight matrix  **,** which has columns but only one row in this single neuron case. A constant 1 enters the neuron as an input and is multiplied by a scalar bias . The net input to the transfer function is , which is the sum of the bias and the product **.** The neuron’s output is a scalar in this case. If we had more than one neuron, the network output would be a vector.

The dimensions of the variables in these abbreviated notation figures will always be included, so that you can tell immediately if we are talking about a scalar, a vector or a matrix. You will not have to guess the kind of variable or its dimensions.

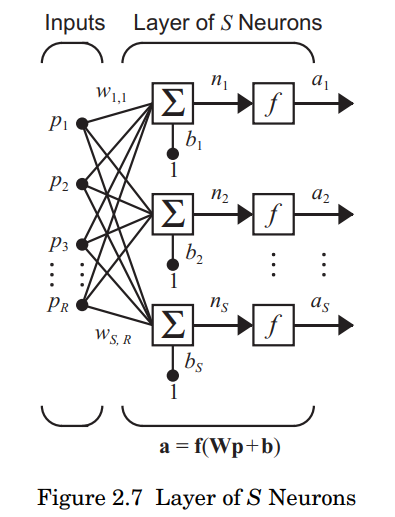
Note that the number of inputs to a network is set by the external specifications of the problem. If, for instance, you want to design a neural network that is to predict kite-flying conditions and the inputs are air temperature, wind velocity and humidity, then there would be three inputs to the network.

*To experiment with a two-input neuron, use the Neural Network Design  
Demonstration Two-Input Neuron* (**nnd2n2**).

* 1. **Network Architectures**

1. **A Layer of Neurons**

A single-*layer* network of neurons is shown in Figure 2.7. Note that each of the inputs is connected to each of the neurons and that the weight matrix now has rows.



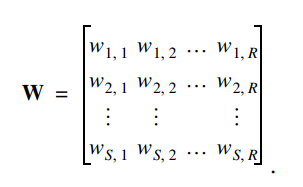
The layer includes the weight matrix, the summers, the bias vector  **,** the transfer function boxes and the output vector . Some authors refer to the inputs as another layer, but we will not do that here.

Each element of the input vector is connected to each neuron through the weight  **.** Each neuron has a bias , a summer, a transfer function and an output . Taken together, the output form the output vector .

It is common for the number of inputs to a layer to be different from the number of neurons (i.e. , .

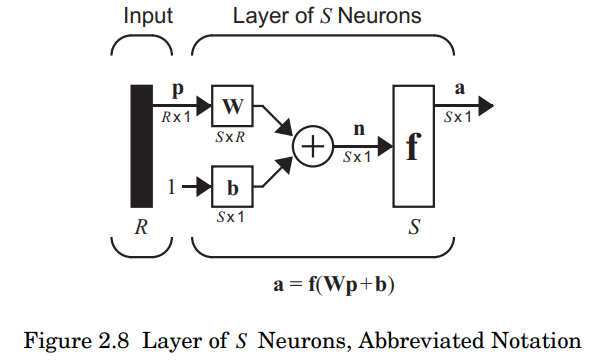
You might ask if all the neurons in a layer must have the same transfer function. The answer is no; you can define a single (composite) layer of neurons having different transfer functions by combining two of the networks shown above in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **:**



As noted previously, the row indices of the elements of matrix indicate the destination neuron associated with that weight, while the column indices indicate the source of the input for that weight. Thus, the indices in say that this weight represents the connection *to* the third neuron *from* the second source.

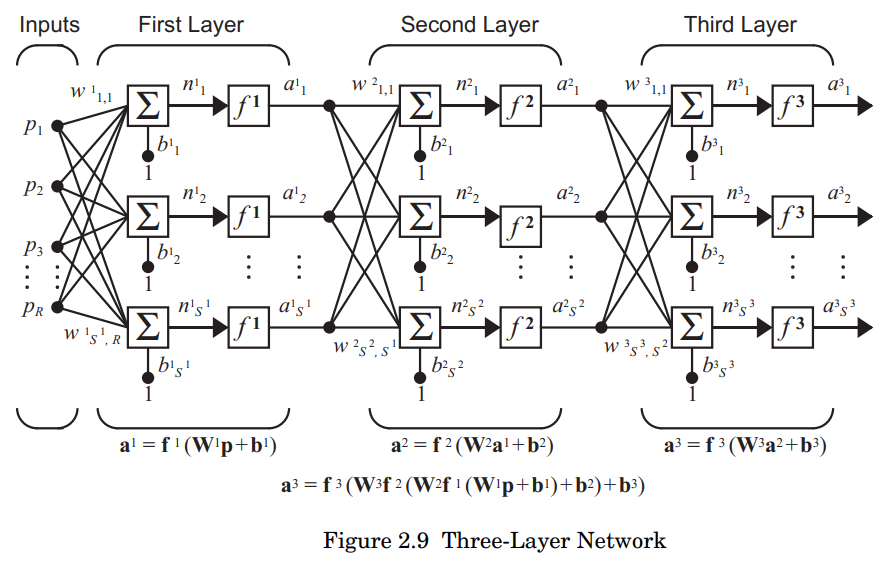
Fortunately, the *S*-neuron, *R*-input, one-layer network also can be drawn in abbreviated notation, as shown in Figure 2.8.



Here again, the symbols below the variables tell you that for this layer, is a vector of length is an matrix, and and are vectors of length . As defined previously, the layer includes the weight matrix, the summation and multiplication operations, the bias vector  **,** the transfer function boxes and the output vector.

1. **Multiple Layers of Neurons**

Now consider a network with several layers. Each layer has its own weight matrix **,** its own bias vector , a net input vector and an output vector  **.** We need to introduce some additional notation to distinguish between these layers. We will use superscripts to identify the layers. Specifically, we append the number of the layer as a *superscript* to the names for each of these variables. Thus, the weight matrix for the first layer is written as , and the weight matrix for the second layer is written as . This notation is used in the three-layer network shown in Figure 2.9.

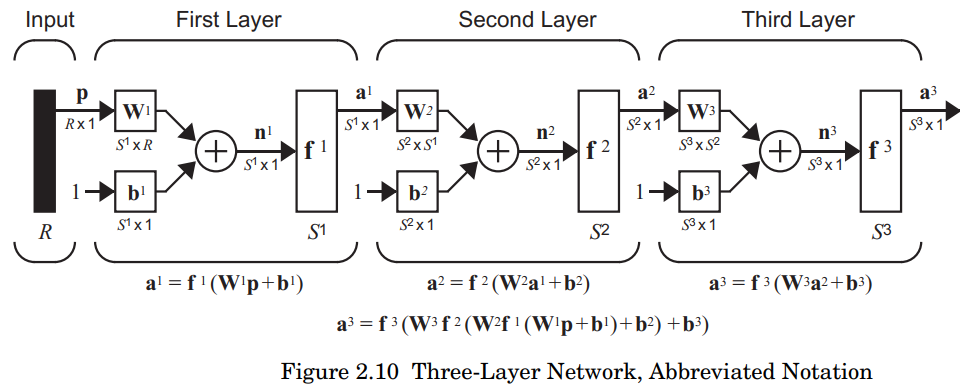


As shown, there are inputs, neurons in the first layer, neurons in the second layer, etc. As noted, different layers can have different numbers of neurons.

The outputs of layers one and two are the inputs for layers two and three. Thus layer 2 can be viewed as a one-layer network with inputs , neurons, and an weight matrix . The input to layer 2 is , and the output is .

A layer whose output is the network output is called an *output layer*. The other layers are called *hidden layers*. The network shown above has an output layer (layer 3) and two hidden layers (layers 1 and 2).

The same three-layer network discussed previously also can be drawn using our abbreviated notation, as shown in Figure 2.10.



Multilayer networks are more powerful than single-layer networks. For instance, a two layer network having a sigmoid first layer and a linear second layer can be trained to approximate most functions arbitrarily well. Singlelayer networks cannot do this.

At this point the number of choices to be made in specifying a network may look overwhelming, so let us consider this topic. The problem is not as bad as it looks. First, recall that the number of inputs to the network and the number of outputs from the network are defined by external problem specifications. So if there are four external variables to be used as inputs, there are four inputs to the network. Similarly, if there are to be seven outputs from the network, there must be seven neurons in the output layer. Finally, the desired characteristics of the output signal also help to select the transfer function for the output layer. If an output is to be either or , then a symmetrical hard limit transfer function should be used. Thus, the architecture of a single-layer network is almost completely determined by problem specifications, including the specific number of inputs and outputs and the particular output signal characteristic.

Now, what if we have more than two layers? Here the external problem does not tell you directly the number of neurons required in the hidden layers. In fact, there are few problems for which one can predict the optimal number of neurons needed in a hidden layer. This problem is an active area of research.

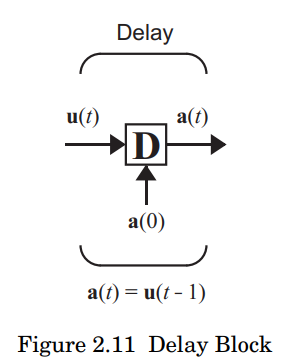
As for the number of layers, most practical neural networks have just two or three layers. Four or more layers are used rarely.

We should say something about the use of biases. One can choose neurons with or without biases. The bias gives the network an extra variable, and so you might expect that networks with biases would be more powerful than those without, and that is true. Note, for instance, that a neuron without a bias will always have a net input of zero when the network inputs are zero. This may not be desirable and can be avoided by the use of a  
bias.

In later chapters we will omit a bias in some examples or demonstrations. In some cases this is done simply to reduce the number of network parameters. With just two variables, we can plot system convergence in a two-dimensional plane. Three or more variables are difficult to display.

1. **Recurrent Networks**

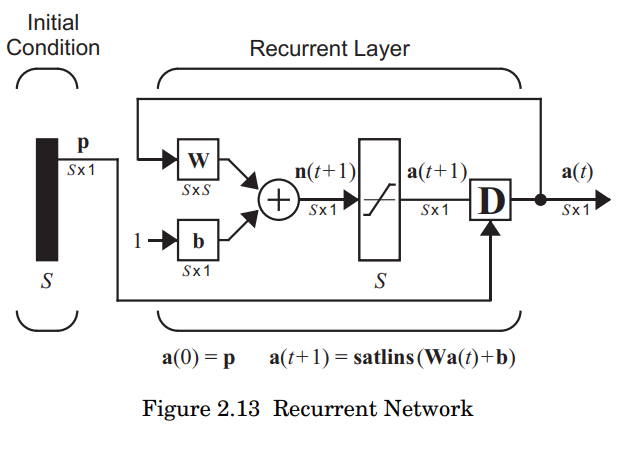
Before we discuss recurrent networks, we need to introduce some simple building blocks. The first is the *delay* block, which is illustrated in Figure 2.11.



The integrator output is computed from its input according to

The initial condition is indicated by the arrow coming into the bottom of the integrator block.

We are now ready to introduce recurrent networks. A *recurrent network* is a network with feedback; some of its outputs are connected to its inputs. This is quite different from the networks that we have studied thus far, which were strictly feedforward with no backward connections. One type of discrete-time recurrent network is shown in Figure 2.13.



In this particular network the vector supplies the initial conditions (i.e., **).** Then future outputs of the network are computed from previous outputs:

Recurrent networks are potentially more powerful than feedforward networks and can exhibit temporal behavior.

* 1. **Learning**

1. **Choosing a cost function**

While it is possible to define an ad hoc cost function, frequently a particular cost (function) is used, either because it has desirable properties (such as convexity) or because it arises naturally from a particular formulation of the problem (e.g., in a probabilistic formulation the posterior probability of the model can be used as an inverse cost). Ultimately, the cost function depends on the task.

1. **Backpropagation**

A DNN can be discriminatively trained with the standard backpropagation algorithm. Backpropagation is a method to calculate the gradient of the loss function (produces the cost associated with a given state) with respect to the weights in an ANN.

The basics of continuous backpropagationwere derived in the context of control theory by Kelley in 1960 and by Bryson in 1961, using principles of dynamic programming. In 1962, Dreyfus published a simpler derivation based only on the chain rule. Bryson and Ho described it as a multi-stage dynamic system optimization method in 1969. In 1970, Linnainmaa finally published the general method for automatic differentiation(AD) of discrete connected networks of nested differentiable functions.This corresponds to the modern version of backpropagation which is efficient even when the networks are sparse. In 1973, Dreyfus used backpropagation to adapt parameters of controllers in proportion to error gradients. In 1974, Werbos mentioned the possibility of applying this principle to ANNs, and in 1982, he applied Linnainmaa's AD method to neural networks in the way that is widely used today.In 1986, Rumelhart, Hinton and Williams noted that this method can generate useful internal representations of incoming data in hidden layers of neural networks.In 1993, Wan was the first to win an international pattern recognition contest through backpropagation.

The weight updates of backpropagation can be done via stochastic gradient descent using the following equation:

where, is the learning rate, is the cost (loss) function and a stochastic term. The choice of the cost function depends on factors such as the learning type (supervised, unsupervised, reinforcement, etc.) and the activation function. For example, when performing supervised learning on a multiclass classification problem, common choices for the activation function and cost function are the softmax function and cross entropy function, respectively. The softmax function is defined as  where represents the class probability (output of the unit ) and and represent the total input to units and of the same level respectively. Cross entropy is defined as where epresents the target probability for output unit and is the probability output for after applying the activation function.

These can be used to output object bounding boxes in the form of a binary mask. They are also used for multi-scale regression to increase localization precision. DNN-based regression can learn features that capture geometric information in addition to serving as a good classifier. They remove the requirement to explicitly model parts and their relations. This helps to broaden the variety of objects that can be learned. The model consists of multiple layers, each of which has a rectified linear unit as its activation function for non-linear transformation. Some layers are convolutional, while others are fully connected. Every convolutional layer has an additional max pooling. The network is trained to minimize ***L*2 error** for predicting the mask ranging over the entire training set containing bounding boxes represented as masks.

Alternatives to backpropagation include Extreme Learning Machines, "No-prop" networks, training without backtracking, "weightless" networks, and non-connectionist neural networks.

* 1. **Learning paradigms**

1. **Supervised learning**

Supervised learning uses a set of example pairs and the aim is to find a function  in the allowed class of functions that matches the examples. In other words, we wish to infer the mapping implied by the data; the cost function is related to the mismatch between our mapping and the data and it implicitly contains prior knowledge about the problem domain.

A commonly used cost is the mean-squared error, which tries to minimize the average squared error between the network's output,  and the target value  over all the example pairs. Minimizing this cost using gradient descent for the class of neural networks called multilayer perceptrons (MLP), produces the backpropagation algorithm for training neural networks.

Tasks that fall within the paradigm of supervised learning are pattern recognition(also known as classification) and regression (also known as function approximation). The supervised learning paradigm is also applicable to sequential data (e.g., for hand writing, speech and gesture recognition). This can be thought of as learning with a "teacher", in the form of a function that provides continuous feedback on the quality of solutions obtained thus far.

{\displaystyle f(x)}

1. **Unsupervised learning**

In unsupervised learning, some data  is given and the cost function to be minimized, that can be any function of the data and the network's output, **.**

The cost function is dependent on the task (the model domain) and any *a priori* assumptions (the implicit properties of the model, its parameters and the observed variables).

As a trivial example, consider the model where is a constant and the cost  . Minimizing this cost produces a value of that is equal to the mean of the data. The cost function can be much more complicated. Its form depends on the application: for example, in compression it could be related to the mutual information between and {\displaystyle \textstyle x}{\displaystyle \textstyle C=E[(x-f(x))^{2}]}{\displaystyle \textstyle f(x)=a}whereas in statistical modeling, it could be related to the posterior probability of the model given the data (note that in both of those examples those quantities would be maximized rather than minimized).

Tasks that fall within the paradigm of unsupervised learning are in general estimation problems; the applications include clustering, the estimation of statistical distributions, compression and filtering.

1. **Reinforcement learning**

In reinforcement learning, data are usually not given, but generated by an agent's interactions with the environment. At each point in time  the agent performs an action and the environment generates an observation and an instantaneous cost according to some (usually unknown) dynamics. The aim is to discover a policy for selecting actions that minimizes some measure of a long-term cost, e.g., the expected cumulative cost. The environment's dynamics and the long-term cost for each policy are usually unknown, but can be estimated.

More formally the environment is modeled as a Markov decision process (MDP) with states and actions with the following probability distributions: the instantaneous cost distribution , the observation distribution and the transition **,** while a policy is defined as the conditional distribution over actions given the observations. Taken together, the two then define a Markov chain (MC). The aim is to discover the policy (i.e., the MC) that minimizes the cost.

ANNs are frequently used in reinforcement learning as part of the overall algorithm. Dynamic programming was coupled with ANNs (giving neurodynamic programming) by Bertsekas and Tsitsiklisand applied to multi-dimensional nonlinear problems such as those involved in vehicle routin, natural resources management or medicine because of the ability of ANNs to mitigate losses of accuracy even when reducing the discretization grid density for numerically approximating the solution of the original control problems.

Tasks that fall within the paradigm of reinforcement learning are control problems, games and other sequential decision making tasks.{\displaystyle \textstyle P(c\_{t}|s\_{t})}{\displaystyle \textstyle c\_{t}}

1. **Convergent recursive learning algorithm**

This is a learning method specially designed for cerebellar model articulation controller (CMAC) neural networks. In 2004 a recursive least squares algorithm was introduced to train CMAC neural network online. This algorithm can converge in one step and update all weights in one step with any new input data. Initially, this algorithm had computational complexity of *O*(*N*3). Based on QR decomposition, this recursive learning algorithm was simplified to be *O*(*N*).

**Sources Document**

1. Artificial neural networks, Wikipedia, <https://en.wikipedia.org/wiki/Artificial_neural_network>
2. **Neural Network Design** (**2nd Edition**) [Martin T Hagan, Howard B Demuth, Mark H Beale, Orlando De Jesús]