**Tutorial on artificial neural network**

Loc Nguyen

Loc Nguyen’s Academic Network, Vietnam

Email: ng\_phloc@yahoo.com

Homepage: www.locnguyen.net

# Abstract

**Keywords:** artificial neural network (ANN).

|  |  |
| --- | --- |
|  | () |

# 1. Introduction

Artificial neural network (ANN) is the mathematical model based on biological neural network but neural network (NN) in this research always indicates artificial neural network. NN consists of a set of processing units which communicate together by sending signals to each other over a number of weighted connections (Kröse & Smagt, 1996, p. 15). Each *unit* is also called neuron, cell, node, or variable which is quantified by a real variable. Each weighted connection, which is considered a neural cord, is often quantified by a real number called *weight*. According to Kröse & Smagt, each unit is responsible for receiving input from neighbors or external sources and using this input to compute an output signal which is propagated to other units (Kröse & Smagt, 1996, p. 15). The most important thing here is that the signal propagation is done by the means of weighed connections which are imitated as biological neurotransmission with neurons and neural cords. According to Kröse & Smagt (Kröse & Smagt, 1996, pp. 15-16), there are three types of units:

* *Input units* receive data from outside the network. These units structure the *input layer*.
* *Hidden units* own input and output signals that remain within NN. These units structure the hidden layer. There can be one or more *hidden layers*.
* *Output units* send data out of the network. These units structure the *output layer*.

Units in NN are also considered variables. The figure (Wikipedia, Artificial neural network, 2009) below shows the simplest structure of an NN with three layers such as input layer, hidden layer, and output layer. The structure of NN is often called the topology.

Diagram

Description automatically generated

**Figure 1.1.** Simplest topology of NN with three layers such as input layer, hidden layer, and output layer

Note that the main reference of this research report is the book “An Introduction to Neural Networks” by Ben Kröse and Patrick van der Smagt (Kröse & Smagt, 1996).

According to Daniel Rios (Rios), there are two main topologies (structures) of NN:

* *Feed-forward NN* is directed acyclic graphic in which flow of signal from input units to output units is one-way flow and so, there is no feedback connection. The NN in this section is feed-forward NN.
* *Recurrent NN* is the one whose graph (topology) contains cycles and so, there are feedback connections.

It is necessary to evolve NN by modifying the weights of connections so that they become more accurate. In other words, such weights should not be fixed by experts. NN should be trained by feeding it teaching patterns and letting it change its weights. This is learning process or training process. According to Daniel Rios (Rios), there are three types of learning methods:

* *Supervised learning*: According to Daniel Rios (Rios), the network is trained by matching its input and its output patterns. These patterns are often known as classes which can be represented by binary values, integers for nominal indices, or real numbers.
* *Unsupervised learning*: The network is trained in response to clusters of patterns behind the input. According to Daniel Rios (Rios), there is no a priori set of categories into which the patterns are to be classified.
* *Reinforcement learning*: The learning algorithms receive partially information along with input from environments and then, adjust partially and progressively the weighted connections by adaptive way to such input. Reinforcement learning is the intermediate form between supervised learning and unsupervised learning.

This introduction section focuses on supervised learning in which input and output are realistic quantities (real numbers). For NN, the essence of supervised learning is to improve weighted connections by matching input and output. Learning NN process is also called *training NN* process in NN literature. Given unit *i*, let *xi* and *yi* denote *input* and *output* of unit *i*, which are real numbers. In NN literature, a unit will be activated if its output is determined and so the output *yi* is also called *activation* of unit *i*. If a unit is input unit (in input layer) then its input contributes to input of NN. If a unit is output unit (in output layer) then its output contributes to output of NN. Each connection between two successive units such as unit *i* and unit *j* is defined by the weight *wij* determining effect of unit *i* on unit *j*. In the normal topology, an output unit is the composition of other hidden units which in turn are the compositions of others input units. The composition (aggregation) of a unit is represented as a weighted sum which will be evaluated to determine the output of this unit. The process of computing the output of a unit includes two following steps (Han & Kamber, 2006, p. 331):

* An adder called *summing function* sums up all the inputs multiplied by their respective weights. It is essential to compute the weighted sum. This activity is referred to as linear combination.
* An *activation function* controls the amplitude of the output of the neuron. This activity aims to determine the output. Note that the output of the previous units is the input of current unit.

Figure 1.2 (Han & Kamber, 2006, p. 331) describes the process of computing the output.

**Figure 1.2.** Process of computing output of a unit

For example, as seen in figure 1.2, given a concerned unit *k*, suppose there are previous units whose outputs *yj* (s) are considered as inputs of unit *k*. According to the process of computing the output of a unit, we have following equation (Han & Kamber, 2006, p. 331), (Kröse & Smagt, 1996, pp. 16-17) for computing the output value of a unit.

|  |  |
| --- | --- |
|  | () |

Or shortly:

The equation above for output processing is called *propagation rule*. Note, *wjk* is the weight of the connection from unit *j* to unit *k* and *θj* is the bias of unit *j* while *fj*(.) is the activation function acting on unit *j*. If all units use the same form of activation function, we can denote *f*(.) = *fj*(.).

As a convention, the propagation rule can be denoted by succinct way as follows:

|  |  |
| --- | --- |
|  | () |

The parameters of propagation rule are weights *wjk* and biases *θk* in which weights are most important. Conversely, it is possible to consider the propagation rule as a function of variables *wjk* and *θk*. In a distributed environment, NN can be evolved asynchronously when the computing processes on different units can be computed by distributed way. Given time point *t*, the propagation rule at time point *t* + 1 is rewritten as follows:

|  |  |
| --- | --- |
|  | () |

As a convention, input units in input layer are indexed by *i* (for instance, *xi* and *yi*), hidden units in hidden layer are indexed by *h* (for instance, *xh* and *yh*), and output units in output layer are indexed by *o* (for instance, *xo* and *yo*). Therefore, indices *j*, *k*, *l*, etc. indicate normal units having both input and output. However, in some cases, the convention of input indices *i*, hidden indices *h*, and output indices *o* may not be applied, for example, when writing pseudo code for learning NN algorithm. For input units, we assume that *xi* = *yi* and *θi*=0. A NN is valid if it has two or more layers and so there is a convention that a *n*-layer NN has *n*+1 actual layers, which means that the input layer is not counted for this convention. This convention is reasonable because the propagation rule is not applied to input units. The simplest NN is single layer NN owning one input layer and one output layer.

Output values of units are arbitrary, but they should range from 0 to 1 (sometimes –1 to 1 range). In general, every unit has following aspects:

* Each unit *k* has input *xk* and output *yk*. Moreover, let *vk* be the actual value of unit *k* taken from experts, environment, database, etc. The actual value *vk* can be equal to or different from the output *vk* with note that *vk* is derived from the propagation rule. The actual value *vk* is called *desired output* of unit *k*. When a unit *k* is put in NN, which means that it connects to other units via weighted connections, then unit *k* is called clamped in NN. The input of a clamped unit *k* is denoted *sk*. By default, all units are clamped and so, the *clamped input* *sk* is the same to the input *xk* as *sk* = *xk* by default.
* A set of inputs connects to it. Each connection is quantified by a weight *wjk*.
* A bias value *θk* will be added to the weighted sum.
* The weighted sum is computed by summing up all the inputs modified by their respective weights. Summing function or adder is responsible for this summing task.
* Its output *yk* is the outcome of activation function *f*(.) on weighted sum. Activation function is crucial factor in NN. The combination of summing function and activation function constitutes the propagation rule, but the propagation rule can be more complicated with some enhancements.

Given unit *k*, there are many desired outputs of unit *k*, for example, *vk*(1), *vk*(2),…, and hence, given a *pattern* *p* (Kröse & Smagt, 1996, p. 19) there is a desired output *vk*(*p*) corresponding to pattern *p*. For easily understandable explanation, if *vk*(*p*) is taken from a database table, *p* indicates the *p*th row in the table. As a convention, let *xk*(*p*), *yk*(*p*), *vk*(*p*), and *sk*(*p*) be input, output, desired output, clamped input of unit *k* within the *p* pattern, respectively or they can be called the *p*th input, output, desired output, and clamped input of unit *k*, respectively. With pattern *p*, the propagation rule is rewritten exactly as follows:

|  |  |
| --- | --- |
|  | () |

Where *N*(*k*) denotes a set of previous (clamped) units to which the current clamped unit *k* connects.

The propagation rule essentially transforms inputs to outputs but an output *yk* may not totally equal to the desired output *vk* when it is often approximated to *vk*. The propagation rule with optimal weights and optimal bias is a good enough presentation of NN when NN tries its best to approach the desired function *v*(.) that produces desired outputs *vk*. Therefore, in NN literature, *representation power* (Kröse & Smagt, 1996, p. 20) implies the approximation of NN and the desired function *v*(.) and so, the ideology under any learning NN algorithms is to make such approximation.

There are some conventions for learning NN from sample or training dataset. The set of inputs *x*1, *x*2,…, *xk*,… in a layer is denoted as ***x*** = (*x*1, *x*2,…, *xk*,…)*T* which is called *input vector* where the superscript “*T*” denotes transposition operator of vector and matrix. The set of outputs *y*1, *y*2,…, *yk*,… in a layer is denoted as ***y*** = (*y*1, *y*2,…, *yk*,…)*T* which is called *output vector*. The set of desired outputs *v*1, *v*2,…, *vk*,… in a layer is denoted as ***v*** = (*v*1, *v*2,…, *vk*,…)*T* which is called *desired* *output vector*. The set of clamped inputs *s*1, *s*,…, *sk*,… in a layer is denoted as ***s*** = (*s*1, *s*2,…, *sk*,…)*T* which is called *clamped input vector*. Input vector, output vector, desired vector, and clamped input vector with *p* pattern are denoted ***x***(*p*), ***y***(*p*), ***v***(*p*), and ***s***(*p*), respectively. The set of input vector over entire input layer and desired output vector over entire output layer compose a sample or training dataset *D* = {***x***(*p*), ***v***(*p*)} for learning NN where *p* = 1, 2, 3, etc. By default, all units are clamped in NN and so we have *D* = {***x***(*p*), ***v***(*p*)} = {***s***(*p*), ***v***(*p*)} by default.

The activation function *f*(*.*), which is an important factor of NN, is the squashing function which “squashes” a large weighted sum into possible smaller values ranging from 0 to 1 (sometimes –1 to 1 range). According to Daniel Rios (Rios), there are three types of activation function:

* *Threshold function* takes on value 0 if weighted sum is less than 0 and otherwise. The formula of threshold function is:
* *Piecewise-linear function* takes on values according to amplification factor in a certain region of linear operation. The formula of piecewise-linear function is:
* *Sigmoid function* or logistic function takes on values in range [0, 1] or [–1, 1]. The formula of sigmoid function is:

|  |  |
| --- | --- |
|  | () |

Where *e*(.) or exp(.) denotes exponent function. Logistic function is the most popular activation function.

Recall that the essence of learning NN (training NN) is to improve weighted connections by matching input and output. Given a weight *wjk* from unit *j* to unit *k*, a new version of *wjk* after learning process at time point *t* is updated by weight deviation Δ*wjk* as follows:

Or shortly:

|  |  |
| --- | --- |
|  | () |

The equation above is called *weight update rule* and hence, weight update rule focuses on how to calculate the weight deviation Δ*wjk* which is also called the change in weight. Learning NN algorithms also improve biases beside improving weights. Given bias *θk* of unit *k*, a new version of *θk* after learning process at time point *t* is updated by bias deviation Δ*θk* as follows:

Or shortly:

|  |  |
| --- | --- |
|  | () |

The equation above is called *bias update rule* and hence, bias update rule focuses on how to calculate the bias deviation Δ*θk* which is also called the change in bias. In general, a normal learning NN algorithm needs to specify both weight update rule and bias update rule because both of them determine the propagation rule. Because the weight update rule and the bias update rule are based on the weight deviation and the bias deviation, Δ*wjk* and Δ*θk* can be used to represent these rules.

The most popular learning NN algorithm is back-propagation algorithm, but we should skim some simpler learning algorithms first. Two common simpler learning algorithms are Perceptron and Adaline. Both of them are based on *Hebbian rule* and *delta rule*. Hebbian rule indicates that Δ*wjk* is proportional to product of output of unit *j* and output of unit *k* as follows (Kröse & Smagt, 1996, p. 18):

|  |  |
| --- | --- |
|  | () |

Where the positive constant γ is called learning rate specifying the power of proportional, which relates to speed of learning process. Both *yj* and *yk* are results of the propagation rule. Let *vk* be the desired output of unit *k* from environment or database, delta rule indicates that Δ*wjk* is proportional to product of output value of unit *j* and output deviation of unit *k* as follows (Kröse & Smagt, 1996, p. 18):

|  |  |
| --- | --- |
|  | () |

Obviously, Hebbian rule and delta rule are weight update rules. After researching learning NN algorithm, we will recognize that delta rule is derived from gradient descent method for minimizing squared error, known as least squares method.

Recall that the most popular NN algorithm is back-propagation algorithm whereas two simpler learning algorithms are Perceptron and Adaline. Perceptron algorithm is used to train a simple single layer NN called Perceptron. For instance, Perceptron has some input units and one output unit. Without loss of generality, Perceptron has two input units whose (input) values are denoted *x*1 and *x*2 and one output unit whose (output) value is denoted *y* with note that *y* is binary {–1, 1} and bias of the output unit is *θ*, as seen in figure 1.3 (Kröse & Smagt, 1996, p. 23).

**Figure 1.3.** Perceptron topology

As a convention, we can call input unit *x*1, input unit *x*2, output unit *y*, and bias *θ* although they are values. The propagation rule of Perceptron is (Kröse & Smagt, 1996, p. 23):

|  |  |
| --- | --- |
|  | () |

Which is, indeed, a binary classifier for supervised learning whose inputs are *x*1 and *x*2 and whose output is the binary class {–1, 1}. Classification equation from the Perceptron propagation rule is *w*1*x*1 + *w*2*x*2 + *θ* = 0. The weight update rule of Perceptron is:

Let *v* {–1, 1} be the desired value of unit *y* from environment or database, Perceptron learning algorithm calculates the weight deviation Δ*wi* as follows (Kröse & Smagt, 1996, pp. 24-25):

|  |  |
| --- | --- |
|  | () |

Therefore, the weight update rule of Perceptron is slightly similar to Hebbian rule. The bias update rule of Perceptron is:

Perceptron learning algorithm calculates the bias deviation Δ*θi* as follows (Kröse & Smagt, 1996, p. 25):

|  |  |
| --- | --- |
|  | () |

For example, with initialized values *w*1 = 1, *w*2 = 1, and *θ* = 0, given sample *x*1 = 1, *x*2 = 2, and *v* = 1, Perceptron weights and biases are updated as follows:

Adaline developed by Widrow and Hoff (Kröse & Smagt, 1996, p. 27), which is abbreviation of adaptive linear element, is an extension of Perceptron, whose inputs and outputs are real numbers. Of course, Adaline is a single layer NN. Therefore, the output unit *y* is linear combination of input units *xi* (s). The propagation rule of Adaline is (Kröse & Smagt, 1996, p. 28):

|  |  |
| --- | --- |
|  | () |

Obviously, the activation function of Adaline is identical function. Suppose Adaline is learned from the sample {***x***(*p*), ***v***(*p*)} where each *v*(*p*) is the *p*th desired output which is corresponding to the *p*th instance *y*(*p*) at pattern *p*. By default, all units are clamped and so, the *clamped input* *sk* is the same to the input *xk* as *sk* = *xk* by default such that {***x***(*p*), ***v***(*p*)} = {***s***(*p*), ***v***(*p*)}. The total error given this sample is the sum of squared deviations between desired outputs and outputs as follows (Kröse & Smagt, 1996, p. 28):

|  |  |
| --- | --- |
|  | () |

Where (Kröse & Smagt, 1996, p. 28),

|  |  |
| --- | --- |
|  | () |

Note, *ε*(*p*)(*wi*, *θ*), which is function of *wi* and *θ*, is the squared error at pattern *p* or the *p*th squared error in short. According to the least squares method, the optimal (*wi*\*, *θ*\*)*T* is the minimizer of the total error.

By feeding successively each {*x*(*p*), *v*(*p*)}, it is possible to calculate the minimizer (*wi*\*, *θ*\*) at each pattern *p*, which minimizes the *p*th error *ε*(*p*)(*wi*, *θ*).

|  |  |
| --- | --- |
|  | () |

After feeding all patterns, the final minimizer (*wi*\*, *θ*\*)*T* is expected to minimize the total error *ε*(*wi*, *θ*). Gradient descent method is used to search for the maximizer (*wi*\*, *θ*\*)*T* with the target function *ε*(*p*)(*wi*, *θ*). Gradient descent method pushes candidate solution along with a so-called descending direction *d*(*p*) multiplied with length *γ* of such descending direction where *d*(*p*) is the opposite of gradient of *ε*(*p*)(*wi*, *θ*).

|  |  |
| --- | --- |
|  | () |

Note, the gradient of *ε*(*p*)(*wi*, *θ*) is row vector of partial derivatives of *ε*(*p*)(*wi*, *θ*) (Kröse & Smagt, 1996, p. 28). Due to (Kröse & Smagt, 1996, pp. 28-29)

We have:

As a result, the weight deviation and bias deviation are determined based on *γd*(*p*) as follows (Kröse & Smagt, 1996, p. 29):

|  |  |
| --- | --- |
|  | () |

In NN literature, *γ* is called learning rate which implies speed of the learning NN algorithm. The weight update rule and bias update rule of Adaline are:

Or shortly:

|  |  |
| --- | --- |
|  | () |

Where,

Obviously, Adaline learning algorithm follows delta rule.

By extending Adaline we obtain weight update rule and bias update rule for normal NN in general case. Recall that the propagation rule for normal NN is:

Without loss of generality, the pattern *p* is removed from the formulation, but it exists in training sample for learning algorithms. Because the propagation rule is only applied to hidden units and output units and so only weights and biases of hidden units and output units are learned, of course. Because only output units have desired outputs, we estimate weights and bias of output units first and then, turn back to estimate weights and biases of hidden units according to backward direction. Given output unit *o* whose output and desired output are *y*o and *vo*, the squared error function of output unit *o* for normal NN is (Kröse & Smagt, 1996, p. 34):

|  |  |
| --- | --- |
|  | () |

Where,

Note that all previous outputs *yh* were determined. Moreover, by default, all units are clamped and so, the clamped input *so* is the same to the input *xo* as *so* = *xo* by default. Recall that the weight deviation Δ*who* and the bias deviation Δ*θo* are determined based on the gradient of the squared error function *ε*(*yo*) according to gradient descent method for minimizing the squared error function *ε*(*yo*).

Note, the gradient of *ε*(*yo*) with regard to *who* and *θo* is the row vector of partial derivatives of *ε*(*yo*) with regard to *who* and *θo* as follows:

With gradient descent method, the weight deviation Δ*who* and the bias deviation Δ*θo* are products of learning rate and descending direction of *ε*(*yo*).

Due to chain rule in derivation:

We obtain the weight deviation Δ*who* and the bias deviation Δ*θo* of any output unit as follows:

|  |  |
| --- | --- |
|  | () |

Where *f*’(*xo*) is derivative of the activation function *f*(.) at *xo*. Obviously,

Let (Kröse & Smagt, 1996, p. 34),

|  |  |
| --- | --- |
|  | () |

The quantity *δo* is called error of output unit in literature. We have the succinct equation of the weight deviation Δ*who* and the bias deviation Δ*θo*.

|  |  |
| --- | --- |
|  | () |

Obviously, we determine the weight update rule and the bias update rule for output units as follows:

Now we turn back to estimate weights and bias of a hidden unit *h* according to backward direction with suppose that unit *h* is connected to a set of output units *o*. Therefore, the squared error function *ε*(*yh*) of unit *h* is the sum of output errors *ε*(*yo*) with regard to such set of output units, as follows:

|  |  |
| --- | --- |
|  | () |

Each output error *ε*(*yo*) were aforementioned:

Note,

By default, all units are clamped and so, the clamped input *sh* is the same to the input *xh* as *sh* = *xh* by default. Recall that the weight deviation Δ*wjh* and the bias deviation Δ*θh* are determined based on the gradient of the squared error function *ε*(*yh*) according to gradient descent method for minimizing the squared error function *ε*(*yh*).

Note, the gradient of *ε*(*yh*) with regard to *wjh* and *θh* is the row vector of partial derivatives of *ε*(*yh*) with regard to *wjh* and *θh* as follows:

It is necessary to calculate the gradient *ε*’(*yh*). Firstly, we have:

Recall that, according to the propagation rule, *xh* is:

It is necessary to calculate the derivative . Indeed, we have:

Due to:

We obtain:

This implies:

As a result, the gradient of the squared error function *ε*(*yh*) with regard to *wjh* and *θh* is:

Where,

Note,

Therefore, with gradient descent method, the weight deviation Δ*wjh* and the bias deviation Δ*θh* are inversely proportional to the gradient of the squared error function *ε*(*yh*) multiplied with learning rate as follows:

|  |  |
| --- | --- |
|  | () |

Obviously, we determine the weight update rule and the bias update rule for hidden units as follows:

In general, given any output unit *h* and any hidden unit *o*, the weight update rule and the bias update rule in the most general case of learning NN are represented as follows:

|  |  |
| --- | --- |
|  | () |

Where,

|  |  |
| --- | --- |
|  | () |

Note,

The quantity *δh* is called error of hidden unit in literature. The equation above is extension of delta rule. For learning any previous unit *j* connecting to unit *k*, the backward estimation is done similarly with note that unit *k* plays the role of output unit for unit *j*. The essence of a learning NN algorithm is back propagation process from the last layer (output layer) backwards the first layer (input layer). The final stage of this common learning NN algorithm is to specify the derivative *f*’(*x*) of activation function, which depends on concrete applications. A popular activation function is sigmoid function *f*(*x*) = 1 / (1 + exp(–*x*) whose derivative is:

Therefore, the weight update rule and the bias update rule for sigmoid function are:

Where,

|  |  |
| --- | --- |
|  | () |

Recall that *δo* and *δh* are also called errors of output unit and hidden unit, respectively.

Now it is easy to implement an iteration algorithm for learning NN with sigmoid function (logistic function), which is called *back-propagation algorithm*. Recall that a learning NN process is also called training NN process in NN literature. For easily understandable explanation, there are some new notations. Given current unit *j* and *n* previous units *i* connecting to unit *j*, let *Oi*, *Ij* and *Oj* be the output of unit *i*, input of unit *j*, and output of unit *j*. Obviously, we have *Oi* = *yi*, *Ij* = *xj* = *sj*, and *Oj* = *yj*. The propagation rule is written according to these notations is (Han & Kamber, 2006, p. 331) for computing the output value of a unit. These notations are necessary for describing pseudo code of back-propagation algorithm because in some cases output units and hidden units are treated similarly in the algorithm. Therefore, the convention of input indices *i*, hidden indices *h*, and output indices *o* may not be applied here.

For back-propagation algorithm, the weight update rule and bias update rule of any unit *j* are represented as follows:

Given actual value (desired value) *Vj* of unit *j* and a set of units *k* to which unit *j* connects, we have:

The back-propagation algorithm is described here along with an example. Suppose the sample consists of many data row and each row has many attributes. There is a so-called class attribute which is used to group (classify) rows. All attributes except class attribute are often represented as input units in NN and class attribute is often represented as output unit in NN. If NN is used to classify document then, rows represent documents and non-class attributes are terms; in this case, the corpus becomes a matrix *n*x*p*, which have *n* rows and *p* columns with respect to *n* document vectors and *p* terms. The sample for document classification is called *corpus*. Tables mentioned later are typical examples of corpus.

NN is applied into classifying corpus and such supervised learning algorithm used in this chapter is back-propagation algorithm. The back-propagation algorithm (Han & Kamber, 2006, pp. 330-333) is a famous supervised learning algorithm for classification, which is used in feed-forward NN. It processes iteratively data row in training corpus and compares the network’s prediction for each row to the actual class of the row. For each time it feeds a training row, the weights are modified in order to minimize the error between network’s prediction and actual class. The modifications are made in backward direction, from output layer through hidden layer down to input layer. Back-propagation algorithm includes four main steps such as initializing the weights, propagating input values forward, propagating errors backward, and updating weights and biases (Han & Kamber, 2006, pp. 330-333). The following table describes back-propagation algorithm for learning NN by pseudo-code like programming language.

|  |
| --- |
| **1. Initializing the weights**: The weights *wij* of connections between units are initialized as random real numbers which should be in space [0, 1]. Each bias *θi* associated to each unit is also initialized, which is 0 as usual.  *While terminating condition is not satisfied*  *For each data row in corpus*  **2. Propagating input values forward**: Training data row is fed to input layer.  *For each input unit i*, its input value denoted *Ii* and its output value denoted *Oi* are the same.  *End for each input unit i*  *For each hidden unit j or output unit j*, its input value *Ij* is the weighted sum of all output values of units from previous layer. The bias is also added to this weighted sum.  Where *wij* is the weight of connection from unit *i* in previous layer to unit *j*, *Oi* is the output value of unit *i* from previous layer and *θj* is the bias of unit *j*. The output value of hidden unit or output unit *Oj* is computed by applying activation function to its input value (weighted sum). Suppose activation function is sigmoid function. We have:  *End for each hidden unit j or output unit j*  **3. Propagating errors backward**: The error is propagated backward by updating the weights and biases to reflect the error of network’s prediction.  *For each output unit j*, its error *Errj* is computed as below:  Where *Vj* is the real value of unit *j* in training corpus; in other words, *Vj* is the actual class. This error is the *δo* aforementioned.  *End for each output unit j*  *For each hidden unit j* from the last hidden layer to the first hidden layer, the weighted sum of the errors of other units connected to it in the next higher layer is considered when its error is computed. So the error of hidden unit *j* is computed as below:  Where *wjk* is the weight of the connection from hidden unit *j* to a unit *k* in next higher layer and *Errk* is the error of unit *k*. This error is the *δh* aforementioned.  *End for each hidden unit j*  **4. Updating weights and biases** is based on the errors.  *For each weight wij* over the whole NN. The weights are updated so as to minimize the errors. Given Δ*wij* is the change in weight *wij*, the weight *wij* is updated as below:  Where *γ* is learning rate ranging from 0 to 1. Learning rate helps to avoid getting stuck at a local minimum in decision space and helps to approach to a global minimum (Han & Kamber, 2006, pp. 332-333).  *End for each weight wij* in the whole NN  *For each bias θj*over the whole NN. The bias *θj* of hidden or output unit *j* is updated as below:  Where *γ* is learning rate ranging from 0 to 1.  *End for each bias θj*  *End for each data row in corpus*  *End while terminating condition is not satisfied* with note that there are two common terminating conditions:   * All Δ*wij* in some iteration are smaller than given threshold. * Or iterating through all possible training data rows. |

**Table 1.1.** Back-propagation algorithm for learning NN with sigmoid activation

The trained (learned) NN derived from back-propagation algorithm is the classifier of NN. Now the application of NN into document classification is described right here.

Given a corpus (sample), in which there are a set of classes *C* = {*computer science*, *math*}, and a set of terms *T* = {*computer*, *programming language*, *algorithm*, *derivative*}. Every document (vector) is represented as a set of input variables. Each term is mapped to an input variable whose value is term frequency (*tf*). So the input layer consists of four input units: “*computer*”,“*programming language*”,“*algorithm*”and “*derivative*”.

The hidden layer is constituted of two hidden units: “*computer science*”,“*math*”. Values of these hidden units range in interval [0, 1]. The output layer has only one unit named “*document* *class*” whose value also ranges in interval [0, 1] where value 1 denotes that document belongs totally to “*computer science*” class and value 0 denotes that document belongs totally to “*math*” class. The evaluation function used in network is sigmoid function. Suppose our original topology is feed-forward NN in which the weights are initialized arbitrarily and all biases are zero. Note that such feed-forward NN shown in following figure is the one that has no cycle in its model.

**Figure 1.4.** The NN for document classification

Note that units *C*, *P*, *A* and *D* denote terms “*computer*”,“*programming language*”,“*algorithm*”and “*derivative*”, respectively. Units *S* and *M* denote “*computer science*”classand “*math*” class, respectively. Unit *L* denotes “*document* *class*”. It is easy to infer that if output value of unit *L* is greater than 0.5 then, it is likely that document belongs to “*computer science*” class.

Suppose the given corpus = {*doc*1*.txt*, *doc*2*.txt*, *doc*3*.txt*, *doc*4*.txt*, *doc*5*.txt*, *doc*6*.txt*}. The training corpus (training data) is shown in following table in which cell (*i, j*) indicates the number of times that term *j* (column *j*) occurs in document *i* (row *i*); in other words, each cell represents a term frequency and each row represents a document vector.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *computer* | *programming*  *language* | *algorithm* | *derivative* | **class** |
| *doc*1*.txt* | 5 | 3 | 1 | 1 | 1 |
| *doc*2*.txt* | 5 | 5 | 40 | 50 | 0 |
| *doc*3*.txt* | 20 | 5 | 20 | 55 | 0 |
| *doc*4*.txt* | 20 | 55 | 5 | 20 | 1 |
| *doc*5*.txt* | 15 | 15 | 40 | 30 | 0 |
| *doc*6*.txt* | 35 | 10 | 45 | 10 | 1 |

**Table 1.2.** Training corpus – Term frequencies of documents

Note that the “class” column has binary values where value 1 expresses “*computer science*” class and value 0 expresses “*math*” class.

It is required to normalize term frequencies. Let *tf*11=5, *tf*12=3, *tf*13=1, and *tf*14=1 be the frequencies of terms “*computer*”, “*programming language*”, “*algorithm*”, and “*derivative*”, respectively of document “*doc*1*.txt*”, for example, these terms are normalized as follows:

Following table shows normalized term frequencies in corpus .

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *computer* | *programming*  *language* | *algorithm* | *derivative* | **class** |
| *D*1 | 0.5 | 0.3 | 0.1 | 0.1 | 1 |
| *D*2 | 0.05 | 0.05 | 0.4 | 0.5 | 0 |
| *D*3 | 0.2 | 0.05 | 0.2 | 0.55 | 0 |
| *D*4 | 0.2 | 0.55 | 0.05 | 0.2 | 1 |
| *D*5 | 0.15 | 0.15 | 0.4 | 0.3 | 0 |
| *D*6 | 0.35 | 0.1 | 0.45 | 0.1 | 1 |

**Table 1.3.** Training corpus – Normalized term frequencies

Data rows in the table above representing normalized document vectors are fed to our original NN in the aforementioned figure for supervised learning. The back-propagation algorithm is used to train network, as described in the aforementioned table.

Let *IC*, *IP*, *IA*, *ID*, *IS*, *IM*, and *IL* be input values of units *C*, *P*, *A*, *D*, *S*, *M*, and *L*. Let *OC*, *OP*, *OA*, *OD*, *OS*, *OM*, and *OL* be output values of units *C*, *P*, *A*, *D*, *S*, *M*, and *L*. Let *θS*, *θM*, and *θL* be biases of units *S*, *M*, and *L*. Suppose all biases are initialized by zero, we have *θS*=*θM*=*θL*=0. Let *wCS*, *wCM*, *wPS*, *wPM*, *wAS*, *wAM*, *wDS*, *wDM*, *wSL*, and *wML* be weights of connections (arcs) from *C* to *S*, from *C* to *M*, from *P* to *S*, from *P* to *M*, from *A* to *S*, from *A* to *M*, from *D* to *S*, from *D* to *M*, from *S* to *L*, and from *M* to *L*. According to the origin neural network depicted in figure V.2.2.3.3, we have *wCS=*0.7, *wCM=*0.3, *wPS=*0.6, *wPM=*0.4, *wAS=*0.4, *wAM=*0.6, *wDS=*0.3, *wDM=*0.7, *wSL=*0.8, and *wML=*0.2.

From the corpus shown in table above, the first document *D*1=(0.5, 0.3, 0.1, 0.1) is fed into the back-propagation algorithm. It is required to compute output values *OS*, *OM*, *OL* and update connection weights. For simplicity, the evaluation function is sigmoid function . According to the propagation rule (Han & Kamber, 2006, p. 331) for computing the output value of a unit, we have:

*OC*=*IC*=0.5

*OP*=*IP*=0.3

*OA*=*IA*=0.1

*OD*=*ID*=0.1

Let *VL* be the value of output unit *L*. Because *D*1 belongs to “*computer science*” class, we have:

Let *ErrL*, *ErrS*, and *ErrM* be the errors of units *L*, *S*, and *M*, respectively. According to the equation for updating error of output unit, we have:

According to the equation for updating error of hidden units, we have:

According to the equation for updating connection weights given learning rate *γ*=1, we have:

According to the equation for updating biases *θS*, *θM*, and *θL*, we have:

In similar way, remaining documents *D*2=(0.05, 0.05, 0.4, 0.5), *D*3=(0.05, 0.05, 0.4, 0.5) , *D*4=(0.2, 0.05, 0.2, 0.55), *D*5=(0.15, 0.15, 0.4, 0.3), and *D*6=(0.35, 0.1, 0.45, 0.1) are fed into the back-propagation algorithm so as to calculate the final output values *OS*, *OM*, *OL* and update final connection weights. The following table shows results from this training process based on back-propagation algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Inputs | Outputs | Weights | Biases |
| *D*1 | *IC=*0.5  *IP=*0.3  *IA=*0.1  *ID=*0.1 | *OS=*0.65  *OM=*0.60  *OL=*0.65 | *wCS=*0.70  *wCM=*0.30  *wPS=*0.60  *wPM=*0.40  *wAS=*0.40  *wAM=*0.60  *wDS=*0.30  *wDM=*0.70  *wSL=*0.85  *wML=*0.25 | *θS=*0.01  *θM=*0.00  *θL=*0.08 |
| *D*2 | *IC=*0.05  *IP=*0.05  *IA=*0.40  *ID=*0.50 | *OS=*0.60  *OM=*0.65  *OL=*0.71 | *wCS=*0.70  *wCM=*0.30  *wPS=*0.60  *wPM=*0.40  *wAS=*0.39  *wAM=*0.59  *wDS=*0.29  *wDM=*0.69  *wSL=*0.76  *wML=*0.40 | *θS=*–0.02  *θM=*–0.01  *θL=*–0.07 |
| *D*3 | *IC=*0.05  *IP=*0.05  *IA=*0.40  *ID=*0.50 | *OS=*0.60  *OM=*0.64  *OL=*0.67 | *wCS=*0.70  *wCM=*0.30  *wPS=*0.60  *wPM=*0.40  *wAS=*0.38  *wAM=*0.59  *wDS=*0.27  *wDM=*0.68  *wSL=*0.68  *wML=*0.41 | *θS=*–0.04  *θM=*–0.03  *θL=*–0.22 |
| *D*4 | *IC=*0.20  *IP=*0.05  *IA=*0.20  *ID=*0.55 | *OS=*0.62  *OM=*0.60  *OL=*0.62 | *wCS=*0.70  *wCM=*0.30  *wPS=*0.61  *wPM=*0.41  *wAS=*0.38  *wAM=*0.59  *wDS=*0.27  *wDM=*0.68  *wSL=*0.73  *wML=*0.55 | *θS=*–0.03  *θM=*–0.02  *θL=*–0.13 |
| *D*5 | *IC=*0.15  *IP=*0.15  *IA=*0.40  *ID=*0.30 | *OS=*0.60  *OM=*0.63  *OL=*0.65 | *wCS=*0.70  *wCM=*0.30  *wPS=*0.61  *wPM=*0.40  *wAS=*0.37  *wAM=*0.58  *wDS=*0.27  *wDM=*0.68  *wSL=*0.64  *wML=*0.41 | *θS=*–0.05  *θM=*–0.04  *θL=*–0.28 |
| *D*6 | *IC=*0.35  *IP=*0.10  *IA=*0.45  *ID=*0.10 | *OS=*0.61  *OM=*0.61  *OL=*0.60 | *wCS=*0.70  *wCM=*0.30  *wPS=*0.61  *wPM=*0.40  *wAS=*0.38  *wAM=*0.59  *wDS=*0.27  *wDM=*0.68  *wSL=*0.70  *wML=*0.56 | *θS=*–0.04  *θM=*–0.03  *θL=*–0.18 |

**Table 1.4.** Results from training process based on back-propagation algorithm

According to the training results shown in the table above, the weights and biases of origin NN are changed. It means that NN is already trained. Thus, the following figure expresses the NN learned by back-propagation algorithm.

**Figure 1.5.** Trained neural network

The trained NN depicted in the figure above is the typical classifier of classification method based on neural work.

Suppose the numbers of times that terms “*computer*”,“*programming language*”,“*algorithm*” and“*derivative*” occur in document *D* are 40, 30, 10, and 20, respectively. We need to determine which class document *D* is belongs to. *D* is normalized as term frequency vector.

*D =* (0.4, 0.3, 0.1, 0.2)

Recall that the trained neural network depicted in the figure above has connection weights *wCS=*0.7, *wCM=*0.3, *wPS=*0.61, *wPM=*0.4, *wAS=*0.38, *wAM=*0.59, *wDS=*0.27, *wDM=*0.68, *wSL=*0.7, *wML=*0.56 and biases *θS*=–0.04, *θM*=–0.03, *θL*=–0.18. It is required to compute output values *OS*, *OM*, and *OL*. For simplicity, the evaluation function is sigmoid function . According to the equation (Han & Kamber, 2006, p. 331) for computing the output value of a unit, we have:

Because *OL* is greater than 0.5, it is more likely that document *D =* (0.4, 0.3, 0.1, 0.2) belongs to class “*computer science*”.

# 2. Recurrent networks

# References

Han, J., & Kamber, M. (2006). *Data Mining: Concepts and Techniques* (2nd Edition ed.). (J. Gray, Ed.) San Francisco, CA, USA: Morgan Kaufmann Publishers, Elsevier.

Kröse, B., & Smagt, P. v. (1996). *An Introduction to Neural Networks* (8th Edition ed.). Amsterdam, The Netherlands: University of Amsterdam.

Rios, D. (n.d.). *Introduction to Neural Networks.* Retrieved 2009, from Neuro AI website: http://www.learnartificialneuralnetworks.com/introduction-to-neural-networks.html

Wikipedia. (2009, January 4). *Artificial neural network*. (Wikimedia Foundation) Retrieved 2009, from Wikipedia website: http://en.wikipedia.org/wiki/Artificial\_neural\_network

**Contents**

[Abstract 1](#_Toc131410428)

[1. Introduction 1](#_Toc131410429)

[2. Recurrent networks 18](#_Toc131410430)

[References 18](#_Toc131410431)