**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* or connection 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 *input layer*. As a convention, there is one input layer. In literature, input layer is not counted, which will be explained later.
* *Hidden units* own input and output signals that remain within NN. These units structure *hidden layer*. There can be one or more *hidden layers*.
* *Output units* send data out of the network. These units structure *output layer*. As a convention, there is one output layer.

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

Diagram

Description automatically generated

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

However, the simplest topology has two layers such as input layer and output layer where output layer is also hidden layer. Later on, the NN having such simplest layer is called single layer NN which will be explained later. 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:

* *Feedforward 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 feedforward NN. As a convention, the ordering of layers is counted from left to right, in which the leftmost one is input layer, the middle ones are hidden layers, and the rightmost one is output layer.
* *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 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:

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

The formulation of propagation rule with time points emphasizes the process of changing NN in time series but its meaningfulness is not changed.

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 input layer is not counted for this convention. This convention is reasonable because propagation rule is not applied to input units. The simplest NN is single layer NN owning one input layer and one output layer where the output layer can be considered as hidden 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, states, etc. The actual value *vk* can be equal to or different from the output *vk* with note that *vk* is derived from 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. Besides, clamped units also are ones that are concerned in training process or some special tasks. 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 propagation rule, but 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. Given time point *t*, the propagation rule is rewritten fully as follows:

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*. 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* = *v*(*sk*) (= *v*(*xk*)). 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*,… 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*,… 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*,… 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*,… 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 some typical activation functions:

* *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 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 backpropagation 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* (also *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* which is called learning rate (0 < *γ* ≤ 1) specifies the power of proportional, which relates to speed of learning process. In simplest case, it is 1 as *γ* = 1. Both *yj* and *yk* are results of propagation rule. Let *vk* be the desired output of unit *k* from environment or database, delta rule indicates that Δ*wjk* (also *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 stochastic gradient descent (SGD) method for minimizing squared error with least squares method. Moreover, it is possible to consider delta rule is an improved Hebbian rule and thus, Hebbian is the base for learning NN algorithms.

Recall that the most popular NN algorithm is backpropagation 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. 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. 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, weight update rule of Perceptron is slightly similar to Hebbian rule. 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). 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*)} or summing all squared errors *ε*(*p*)(*wi*, *θ*), it is possible to calculate a minimizer (*wi*\*, *θ*\*) at each pattern *p*, which minimizes the *p*th squared error *ε*(*p*)(*wi*, *θ*).

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

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

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

Note, the gradient of *ε*(*p*)(*wi*, *θ*) denoted ∇*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, 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. Recall that the equation above for weigh deviation and bias deviation above is derived from the squared error function *ε*(*p*)(*wi*, *θ*) at pattern *p* and so, it is easy to extend such equation for the total squared error function over all patterns:

The extension is easy to be asserted because the squared error function *ε*(*p*)(*wi*, *θ*) and the total squared error function *ε*(*wi*, *θ*) are second-order functions so that SGD is applied easily to the two function without loss of generality. As a result, weight update rule and bias update rule of Adaline are:

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

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 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 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 total squared error is the sum of many squared errors over all patterns but here we focus on the squared error without loss of generality because these squared errors are second-order functions which are fed to SGD.

In other words, here we focus on one pattern such that:

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 stochastic gradient descent (SGD) 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:

By SGD, weight deviation Δ*who* and bias deviation Δ*θo* are products of learning rate and descending direction of *ε*(*yo*) which is the opposite of the gradient ∇*ε*(*who*, *θo*).

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*.

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

Recall that the equation above for weigh deviation and bias deviation is derived from the squared error function *ε*(*p*)(*yo*) at pattern *p* and so, it is easy to extend such equation for the total squared error function over all patterns:

The extension is easy to be asserted because the squared error function *ε*(*p*)(*yo*) and the total squared error function *ε*(*yo*) are second-order functions so that SGD is applied easily to the two functions without loss of generality.

Obviously, we determine weight update rule and 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 squared 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 total squared error is the sum of many squared errors over all patterns but here we focus on the squared error without loss of generality because these squared errors are second-order functions which are fed to SGD.

Where,

In other words, we focus on one pattern such that:

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 stochastic gradient descent (SGD) 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*(*wjh*, *θh*). Firstly, we have:

Recall that, according to 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, by SGD, 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 weight update rule and bias update rule for hidden units as follows:

In general, given any output unit *h* and any hidden unit *o*, weight update rule and 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 an extension of delta rule.

Recall that the equation above for weigh deviation and bias deviation is derived from the squared error function *ε*(*p*)(*yh*) at pattern *p* and so, it is easy to extend such equation for the total squared error function over all patterns:

Where,

The extension is easy to be asserted because the squared error function *ε*(*p*)(*yh*) and the total squared error function *ε*(*yh*) are second-order functions so that SGD is applied easily to the two functions without loss of generality.

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, weight update rule and 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 *backpropagation algorithm*. Moreover, such backpropagation algorithm is the representation of traditional learning NN algorithm and so please pay attention to it. 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 backpropagation 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 backpropagation algorithm, 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 backpropagation 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 backpropagation algorithm. Backpropagation algorithm (Han & Kamber, 2006, pp. 330-333) is also a famous supervised learning algorithm for classification, besides learning feedforward NN. Therefore, backpropagation algorithm is a supervised learning algorithm. 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. Backpropagation 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 the backpropagation 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 (0 < *γ* ≤ 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, the number of iterations is large enough. * Or, iterating through all possible training data rows. |

**Table 1.1.** Backpropagation algorithm for learning NN with sigmoid activation

The trained (learned) NN derived from backpropagation 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 feedforward NN in which the weights are initialized arbitrarily and all biases are zero. Note that such feedforward 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. Backpropagation 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 backpropagation 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 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 backpropagation 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 backpropagation 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 backpropagation 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 backpropagation 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. Convergence of learning algorithm

Recall that there are two rules for learning NN such as Hebbian rule and delta rule where Hebbian rule is inspired from Hebbian theory developed by Donald Hebb in his 1949 book “The Organization of Behavior” and delta rule is derived from stochastic gradient descent (SGD) method in solving optimization problem. Moreover, delta rule can be considered as an improved Hebbian rule. Backpropagation algorithm is based on SGD for updating weights and biases. In this section we research convergence of Hebbian rule and delta rule (also SGD). The NN convergence implies that a concrete learning algorithm like propagation algorithm will converge to optimal solutions that are optimal weights after a limit number of iterations. Therefore, the NN convergence is *stability* of learning NN algorithm. Essentially, Hebbian rule and delta rule explain the same meaningfulness. Although weights and biases are the main objects of learning algorithms, other parameters affecting the convergence such as learning rate are discussed too. These parameters are called augmented parameters.

Hebbian theory (Wikipedia, Hebbian theory, 2003) is a neuropsychological theory in which Hebb stated that when two neurons (neural cells) communicate together via a synapsis, activities of the presynaptic cell stimulate the postsynaptic cell. In other words, the synapsis of two neurons will be consolidated if the two neurons are stimulated simultaneously and frequently. This phenomenon is called synaptic plasticity. Therefore, Hebbian rule in machine learning will increase the connection weight of two units proportional to two values of the two units (Wikipedia, Hebbian theory, 2003).

The weight *wjk* represents the synaptic plasticity of the presynaptic unit *j* and the postsynaptic unit *k*. Hebbian rule for learning NN is specified exactly as follows:

Note, the positive constant *γ* which is called learning rate specifies the power of proportional whereas *yj* and *yk* are outputs of unit *j* and unit *k*. Of course, the weight deviation Δ*wjk* represents the synaptic plasticity too. The convergence of Hebbian rule implies that that a concrete learning algorithm that follows Hebbian rule will converge to optimal weights after a limit number of iterations. For easily understandable explanation and without loss of generality, given a single layer NN with output unit (output value) *y* and *n* input units (input values) *xi* like aforementioned Perceptron. Suppose bias is zero, propagation rule is:

We will prove the convergence of the following Hebbian rule for learning weight vector ***w*** = (*w*1, *w*2,…, *wn*)*T* with ***x*** = (*x*1, *x*2,…, *xn*)*T*.

There is an theorem in (Kröse & Smagt, 1996) stated that if there exists a set of optimal weights {***w***\*} so that propagation rule *y* = (***w***\*)*T****x*** is satisfied then any iterative learning algorithm that converges to an optimal weight (may be or may not be ***w***\*) has a limited number of iterations. Suppose *wi* is initialized 0 and so, after *t* time points over *t* iterations of the iterative learning algorithm, by recurring calculation *wi* at time point *t* as follows:

Where,

So, we have:

Suppose optimal weight of the iterative learning algorithm is denoted as ***w***\*, cosine of ***w***(*t*) and ***w***\* is:

If *t* approaches +∞ then cosine of ***w***(*t*) and ***w***\* approaches +∞, which raises a contradiction.

Therefore, the iterative learning algorithm must stop at some finite *t* iterations with the optimal weight ***w***\*. This proof which is also described in (Kröse & Smagt, 1996, pp. 25-26) only asserts the iterative limitation of any converged algorithm but it does not assert the existence of the optimal solution ***w***\*. So, we need to research the delta rule which is an improved version of Hebbian rule.

Recall that delta rule is derived from stochastic gradient descent (SGD) method which is known as a stochastic approximation of gradient descend method on which the traditional backpropagation algorithm is based. Here, the convergence of delta rule implies the convergence of SGD. Extended delta rule derived from SGD is:

Where,

Essentially, Hebbian rule and delta rule explain the same meaningfulness where the extended delta rule is more general and hence, please pay more attention to the convergence of extended delta rule. Now we skim through SGD. Given target function *f*(***w***), gradient descent method is an iterative algorithm that moves the parameter ***w*** along descending direction which is the opposite of the gradient of *f*(***w***) at every time point (or iteration) *t* until reaching the optimizer ***w***\*.

Note, *γt* is the length of descending direction at time point *t*, which is also called learning rate. Moreover, *f*(***w***) receives some data ***x*** as input.

For learning NN with weight update rule and bias update rule, *f*(***w***) is the squared error function *ε*(.) whose parameters are weights. In general case ***w*** is vector. When *f*(***w***) is averaged sum of a large number of member target functions *fi*(***w****i*) (De Sa, 2021, p. 1):

Where ***w*** is composed of many parts as ***w*** = (***w***1, ***w***2,…, ***w****n*)*T*. However, without loss of generality, we can denote *fi*(***w***) by convention that *fi*(***w***) only acts on its part ***w****i* while considering other parts ***w****j* where *j*≠*i* as constants or ignoring them in its analytic formulation, as follows:

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

Anyhow, an important aspect is that gradient of *f*(***w***) is always averaged sum of gradients of all *fi*(***w***) as follows:

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

If *n* is too large for a very complicated gradient ∇*f*(***w***) to be calculated at one time then, SGD is a replacement of gradient descent method by replacing the whole gradient ∇*f*(***w***) by every member gradient ∇*fi*(***w***). Suppose there is a sample {***x***1, ***x***2,…, ***x****N*,…} where ***x****i* is corresponding to some *fk*(.), SGD will feed these ***x****i* (s) one by one or batch by batch (De Sa, 2021, p. 1) for each time point *t* to learn ***w***.

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

Where (.) is some *fk*(.) corresponding to the data ***x****i* in the sample. For instance, if = *k* given data point ***x****i* at time point *t* then, ***x****i* will be fed to the member function *fk*(***w****t*) = *fk*(***w****t* | ***x****i*) at time point *t*. Moreover, if ***x****i* is fed to a set of *m* member functions, for example {*f*1(), *f*2(.),…., *fm*(*.*)} at one time then, it is possible to consider that ***x****i* is fed *m* times, each time point for one member function, without loss of generality. Therefore, is selected among *n* member functions *fi*(***w***), the probability distribution of is even as follows (De Sa, 2021, p. 2):

This probability distribution implies is called *selective* *distribution*. It is more important that ***w****t* follows a so-called *stochastic distribution* below:

The stochastic distribution *g*(***w****t*) implies ***w****t* is moved randomly because data ***x****i* is provided randomly for SGD. Shortly, the stochastic process of SGD is represented by both stochastic distribution and selective distribution, but stochastic distribution is more important because data will be provided randomly by format of data stream in real time applications. The iterative feeding process is very important because it makes SGD adaptive to real time applications where large data is provided by series of small packets. Moreover, these packets do not cover all *fi*(***w***) at one providing time. Besides, the iterative feeding process makes SGD feasible to calculate a gradient with some data ***x****i* (or package ***x****i*) at one time.

In order to assure the convergence of SGD, we need to research Lipschitz continuity. Recall that if function *fi*(*.*) is Lipschitz continuous then, given two vector ***u*** and ***v*** we have (Wikipedia, Lipschitz continuity, 2001):

Where *L* is Lipschitz constant. In this research, notation |.| denotes absolute value of scalar, norm of vector (magnitude of vector, module of vector, length of vector), determinant of matrix, and cardinality of set where notation ||.|| denotes only norms. Norm in Euclidean space is denoted ||.||2, which is default norm and so we implies ||.|| = ||.||2 if there is no additional information. If ***v*** is zero vector, we have:

The convergence condition for SGD is that gradient of every member function *fi*(***w***) must be Lipschitz continuous and bounded. This condition is called *bounded Lipschitz continuous gradient* condition, as follows:

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

Where *Li* is a Lipschitz constant and *Gi* is constant. Let *G* be the maximum one among all *Gi*, we have:

The bounded condition of gradient ||∇*fi*(***w***)|| ≤ *G* is not strict because we can restrict magnitude of this gradient when implementing SGD, for example, ∇*fi*(***w***) is normalized as follows:

There is an important property in the theory of Lipschitz continuity which stated that a function is Lipschitz continuous if and only if its derivative is bounded (Wikipedia, Lipschitz continuity, 2001). Note that Lipschitz continuity is stronger than continuously differentiable aspect and so derivative of Lipschitz continuous function is always existent. Because every gradient ∇*fi*(***w***) is Lipschitz continuous, its derivative ∇2*fi*(***w***) which is Hessian matrix (second-order derivative) of *fi*(***w***) is bounded according to the important property, as follows:

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

Where *Hi* is a constant. When ∇2*fi*(***w***) is matrix, please research documents (Wikipedia, Matrix norm, 2003) about norm of matrix which is not determinant of matrix. The bounded condition of ∇2*fi*(***w***) derives (De Sa, 2021, p. 2):

Suppose Hessian matrix ∇2*fi*(***w***) is a set of basic vectors of a vector space that is image of an Euclidean space, hence, ∇2*fi*(***w***) represents a mapping with note that |***w****T*∇2*fi*(***w***)***w*|** is square of the norm of ***w*** in the vector space specified by ∇2*fi*(***w***) whereas |***w***|2 is square of the norm of ***w*** in Euclidean space. Obviously, we also have:

Where *H* is a constant too, due to:

Where let,

Recall that SGD is a iterative algorithm which feeds data ***x****i* (s) one by one or batch by batch (De Sa, 2021, p. 1) for each time point *t* to learn ***w***.

In order to prove the convergence of SGD, we need to prove that the expectation of magnitude of the stochastic gradient ∇*f*(***w****t*) approaches 0 when *t* approaches positive infinity because a local optimizer such as minimizer or maximizer which is stable point is the point at which ∇*f*(***w****t*) is zero with note that the expectation is mainly associated with the stochastic distribution *g*(***w****t*). In general, we will prove the equation as follows:

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

Or,

This proof was made, available, and provided by Christopher De Sa (De Sa, 2021) in the course of Principles of Large-Scale Machine Learning Systems, College of Computing and Information Science, Cornell University. By expending *f*(***w****t*+1) at ***w****t* according to Taylor’s theorem, there is a *ξt* between ***w****t* and ***w****t*+1 such that (De Sa, 2021, p. 2):

The inequation above was also proved by Wang (Wang, 2016) in another way. This implies:

Taking expectation on both sides of the inequation above by both stochastic distribution *g*(***w****t*) and selective distribution *P*(), we have:

Please pay attention that *γt* is independent from both stochastic distribution *g*(***w****t*) and selective distribution *P*(). Because *f*(***w****t*) and *f*(***w****t*+1) are independent from the selective distribution *P*(), we have:

Due to (De Sa, 2021, p. 2):

We have:

This implies:

As a convention, *g*(***w****t*) is the default distribution and so it is implied in the expectation and so we denote:

Summing both sides of the equation above via *T* iterations of SGD, we have (De Sa, 2021, p. 2):

Suppose the optimization problem is minimization problem, let *f*\* is the expected optimal value such that *f*\* ≤ *f*(***w****T*) for all *T*, we have (De Sa, 2021, p. 2):

Suppose the probability that SGD runs the *τ* = *t* iteration is (De Sa, 2021, p. 3):

The expected gradient (averaged gradient) over *T* iteration represented at some time point *τ* is (De Sa, 2021, p. 3):

This implies (De Sa, 2021, p. 3):

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

If fixing learning rate such that *γt* = *γ*, we have (De Sa, 2021, p. 3):

Due to:

The convergence of SGD is not proved yet because the problem here is that *γt* (0 < *γ* ≤ 1) is larger than *γt*2 and *γt* is dependent on time points. Therefore, suppose let *γt* is inversely proportional to time point *t* as follows (De Sa, 2021, p. 3):

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

We have (De Sa, 2021, p. 3):

We have:

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

Due to:

We obtain:

As a result, we assert that SGD will converge if gradients of all member functions *fi*(***w***) are bounded and Lipschitz continuous with note that the learning rate which is an augmented important parameter of NN must be inversely proportional to time points (iterations). Obviously, these conditions are satisfied with squared error function with decreased learning rate. The condition of decreased learning rate is not hazard. In other words, the convergence of delta rule is asserted.

# 3. Recurrent network

Default NN is feedforward NN in which there is no circle in the network, which means that there is no backward connection from the next layers back to the previous layers. Conversely, *recurrent neural network* (*RNN*) (Kröse & Smagt, 1996, p. 47) allows such backward connection, which means that an output unit or hidden unit can connect to a previous hidden unit directly or indirectly. Because output layer is fixed or not counted in the network, backward connections exist among only hidden units and output units. In general, there are two types of backward connections:

* An output unit or a hidden unit is connected directly to a previous hidden unit in previous layer.
* An output unit or a hidden unit is connected directly to an immediate unit which in turn connects to a previous hidden unit in previous layer.

Most of traditional RNNs follows the second type of backward connection. Moreover, as usual the immediate units connect to the hidden units of the first hidden layer. In other words, such immediate units play the role of input units and so, they are called *extra input units* which compose an *extra input layer*. Some RNNs can call extra input unit by other names, for example, state unit or context unit. Some RNNs may modify the backpropagation algorithm for learning NN via modifying weight update rule and bias update rule but some others may not change the learning NN algorithm. However, propagation rule is not changed. Now we should skim some traditional RNNs along with their learning algorithms.

*Jordan network* developed by Jordan 1986 (Kröse & Smagt, 1996, p. 48) establishes that outputs (activation values) of output units are fed backward to the so-called *state units* playing the role of input units where state units in turn connect directly to the first hidden units. In other words, Jordan network follows the second type of backward connection and the extra input units are called state units, as follows (Kröse & Smagt, 1996, p. 48):

**Figure 3.1.** Jordan network

In Jordan network, the layer of state units is called *state layer*. The connection weights between output units and state units are fixed by +1 (Kröse & Smagt, 1996, p. 48) and so backpropagation algorithm does not modify these weights.

*Elman network* developed by Elman 1990 (Kröse & Smagt, 1996, pp. 48-49)establishes that outputs (activation values) of hidden units are fed backward to the so-called *context units* playing the role of input units where context units in turn connect directly to the first hidden units. In other words, Elman network follows the second type of backward connection and the extra input units are called context units, as follows (Kröse & Smagt, 1996, p. 49):

**Figure 3.2.** Elman network

In Elman network, the layer of context units is called *context layer*. The main difference between Elman network and Jordan network is that Elman network makes the backward connections between hidden units and extra input units whereas Jordan network makes the backward connections between output units and extra input units. However, like Jordan network, the connection weights from hidden units to context units in Elman network are fixed by +1 (Kröse & Smagt, 1996, pp. 48-49). In general, both Jordan network and Elman network can be trained by backpropagation algorithm.

*Hopfield network* developed by Hopfield 1982 (Kröse & Smagt, 1996, pp. 50-53), which is very different from Jordan network and Elman network, establishes connections between all units. In other words, all units in Hopfield network play the role of both input units and output units and so it is a kind to auto-associator network (Kröse & Smagt, 1996, p. 51), which can be considered following the first type of backward connections where each backward connection occurs directly between two units.

**Figure 3.3.** Hopfield network

It is possible to say that auto-associator network is a special NN in which hidden units vanish. Therefore, backpropagation algorithm cannot be applied into learning Hopfield network, which requires another learning algorithm that will be mentioned later. Because Hopfield network leans forward learning processes in time series, its propagation rule should be written in time point *t* as follows (Kröse & Smagt, 1996, p. 51):

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

Where *Uk* is a threshold. It is easy to recognize that units in Hopfield network are binary {1, –1}. If time point is not concerned, Hopfield propagation rule is written as follows:

Suppose there are *n* units, weights in Hopfield network form a square *n*x*n* weight matrix *W* = (*wij*)*n*x*n* with convention that *wii* = 0 which implies that a unit does not connect with itself.

Bias vector of Hopfield is *n*-elements vectors of *n* bias *θk* as follows:

A unit *k* is called stable at time point *t* if its output is not changed at time point *t* as follows:

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

If time point is not concerned, a unit *k* is stable if its *yk* is not changed from the previous value.

At the time Hopfield network was invented, it was used to model associative memory, which mean that after its weights are trained from sample, units can become stable as persistent memory. Delta rule can be used to learn Hopfield network as follows:

|  |
| --- |
| *Input*: *n* units whose inputs are initialized arbitrarily, weight matrix *W* is initialized arbitrarily with suppose *W* is symmetric, and bias vector Θ is initialized as zero vector Θ = **0***T*. A sample is a set of many records ***v*** = (*v*1, *v*2,…., *vn*) where *vk* is desired output of unit *k*.  *Output*: weight matrix *W* is trained and biases vector Θ is trained.  Repeat  For each record ***v*** = (*v*1, *v*2,…., *vn*)  Calculate outputs *yk* for all units according to propagation rule.  Update weights for all units according to delta rule.  Update biases for units as follows (Kröse & Smagt, 1996, p. 52):  End for  Until all units are stable |

**Table 3.1.** Learning Hopfield network

Jordan network, Elman network, and Hopfield network are traditional and typical RNN. In this research, I also propose another RNN called *fishbone neural network* (*FBNN*) in which there are backward connections from output units to extra input units called *memory units* like Jordan network. Besides, each hidden unit can have an outside connection to an outside unit. Such outside connection is called *rib connection* because it attaches from a main unit such as hidden unit and output unit. Such outside unit to which the rib connection attaches is called *rib unit*. Connections from input layer to hidden layers to output layer structure the backbone of FBNN, which are called *backbone connections*. Recall that rib connections cannot attach to input units but they can attach to both hidden units and output units. Following is figure of FBNN.

**Figure 3.4.** Fishbone neural network (FBNN)

An important aspect is that a rib connection is forward connection from a main unit (hidden unit or output unit) to a rib unit so that propagation rule can move right direction. Rib connections are associated with *rib weights* and backbone connections are associated with *backbone weights*. Backpropagation algorithm is applied into learning FBNN as usual with note that the algorithm does not go beyond rib units even though rib units connect with other FBNNs. The purpose of rib connection is that, for solving some problems, a set of many FBNNs are created and communicated together via rib connections. In other words, a FBNN connects with another FBNN via rib unit and rib connection. The set of many FBNNs is considered as a fish school and each FBNN is considered as a fish. The following figure depicts the connection between two FBNNs via rib unit and rib connection.

**Figure 3.5.** Two FBNNs connect together

Note, by rib connection mechanism, a FBNN can connect with many FBNNs. In other words, a fish can communicate with many ones. Recall that, for solving a concrete problem, a set of many FBNNs are created and communicated together via rib connections. Every FBNN solves the problem by itself and then shares results or information with other FBNNs by propagation rule so that the other FBNNs can improve solutions of the concrete problem. The mechanism of social intelligence can improve the capacity of NN in solving complex problems where solutions of many FBNN can converge to an optimal solution.

# 4. Self-organizing network

Standard feedforward NN as well as RNN need both inputs and desired outputs in sample for matching in training. In other words, feedforward NN and RNN focus on supervised learning where outputs like attributes, classes, etc. play the role of supervisors who direct the training process. Backpropagation algorithm is a well-known supervised learning algorithm, especially for learning feedforward NN. Given the input ***x***, supervised learning algorithms improve weights and biases in order to make an approximation to the desired output function *v*(***x***) = ***v***. However, in case that there is no desired outputs ***v*** as supervisors, learning algorithms must process only inputs ***x***, which raises a domain of unsupervised learning. There are many applications as well as algorithms for unsupervised learning like clustering, vector quantization, dimensionality reduction, and feature extraction where clustering and feature extraction are very popular in computer science. Especially, feature extraction is crucial to any recognition applications. Self-organizing network (SON) is designed to solve the problem of unsupervised learning without desired outputs. This section focuses on SON along with unsupervised learning algorithms. The term “self-organizing” in SON implies that SON controls its topology as well as weights and biases by itself without desired outputs.

The most popular SON is competitive SON with competitive learning which is similar to clustering in which competitive learning will select output unit (s) appropriate to inputs of input units. In other words, competitive learning aims to divide inputs into clusters and each cluster is represented by a selected output unit. All inputs in the same cluster share the same output unit.

# References

De Sa, C. (2021). *Lecture 5: Stochastic Gradient Descent.* Cornell University, College of Computing and Information Science. Cornell University. Retrieved from https://www.cs.cornell.edu/courses/cs4787/2021sp/lectures/Lecture5.pdf

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

Wang, C. (2016). *Notes on Convex Optimization Gradient Descent.* GitHub. Chunpai's Blog. Retrieved from https://chunpai.github.io/assets/note/1\_\_Gradient\_Descent\_and\_Line\_Search.pdf

Wikipedia. (2001, August 30). *Lipschitz continuity*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Lipschitz\_continuity

Wikipedia. (2003, December 16). *Hebbian theory*. (Wikimedia Foundation) Retrieved April 5, 2023, from Wikipedia website: https://en.wikipedia.org/wiki/Hebbian\_theory

Wikipedia. (2003, April 25). *Matrix norm*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Matrix\_norm

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](#_Toc131835246)

[1. Introduction 1](#_Toc131835247)

[2. Convergence of learning algorithm 19](#_Toc131835248)

[3. Recurrent network 25](#_Toc131835249)

[4. Self-organizing network 28](#_Toc131835250)

[References 29](#_Toc131835251)